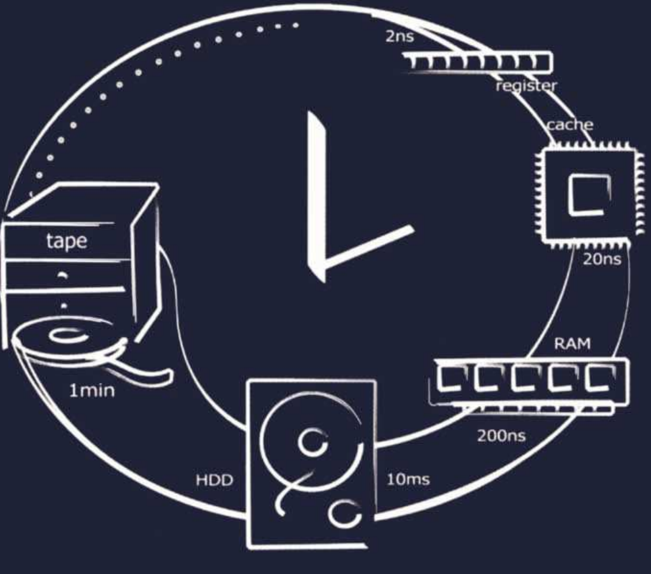
Tutorial

LNCS 2625

Ulrich Meyer
Peter Sanders
Jop Sibeyn (Eds.)

Algorithms for Memory Hierarchies

Advanced Lectures



Springer

Lecture Notes in Computer Science 2625
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Ulrich Meyer   Peter Sanders   Jop Sibeyn (Eds.)

Algorithms for Memory Hierarchies

Advanced Lectures

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Ulrich Meyer
Peter Sanders
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
E-mail: {umeyer,sanders}@mpi-sb.mpg.de

Jop Sibeyn
Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik
Von-Seckendorff-Platz 1, 06120 Halle, Germany
E-mail:jopsi@informatik.uni-halle.de

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek.
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): F.2, E.5, E.1, E.2, D.2, D.4, C.2, G.2, H.2, I.2, I.3.5

ISSN 0302-9743
ISBN 3-540-00883-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

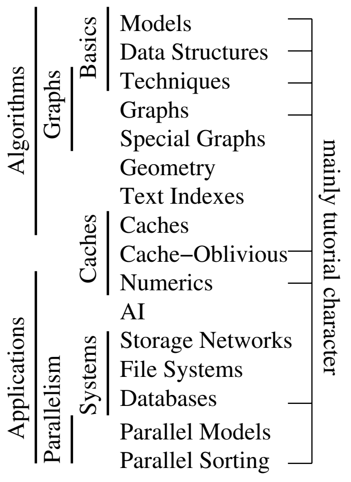
Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10873015 06/3142 5 4 3 2 1 0

Preface

Algorithms that process large data sets have to take into account that the cost of memory accesses depends on where the accessed data is stored. Traditional algorithm design is based on the von Neumann model which assumes uniform memory access costs. Actual machines increasingly deviate from this model. While waiting for a memory access, modern microprocessors can execute 1000 additions of registers. For hard disk accesses this factor can reach seven orders of magnitude. The 16 chapters of this volume introduce and survey algorithmic techniques used to achieve high performance on memory hierarchies. The focus is on methods that are interesting both from a practical and from a theoretical point of view.

This volume is the result of a *GI-Dagstuhl Research Seminar*. The Gesellschaft für Informatik (GI) has organized such seminars since 1997. They can be described as "self-taught" summer schools where graduate students in cooperation with a few more experienced researchers have an opportunity to acquire knowledge about a current topic of computer science. The seminar was organized as Dagstuhl Seminar 02112 from March 10, 2002 to March 14, 2002 in the International Conference and Research Center for Computer Science at Schloss Dagstuhl.

Chapter 1 gives a more detailed motivation for the importance of algorithm design for memory hierarchies and introduces the models used in this volume. Interestingly, the simplest model variant — two levels of memory with a single processor — is sufficient for most algorithms in this book. Chapters 1–7 represent much of the algorithmic core of external memory algorithms and almost exclusively rely on this simple model. Among these, Chaps. 1–3 lay the foundations by describing techniques used in more specific applications. Rasmus Pagh discusses data structures like search trees, hash tables, and priority queues in Chap. 2. Anil Maheshwari and Norbert Zeh explain generic algorithmic approaches in Chap. 3. Many of these techniques such as time-forward processing, Euler tours, or list ranking can be formulated in terms of graph theoretic concepts. Together with Chaps. 4 and 5 this offers a comprehensive review of external graph algorithms. Irit Katriel and Ulrich Meyer discuss fundamental algorithms for graph traversal, shortest paths, and spanning trees that work for many types of graphs. Since even simple graph problems can be difficult to solve in external memory, it



makes sense to look for better algorithms for frequently occurring special types of graphs. Laura Toma and Norbert Zeh present a number of astonishing techniques that work well for planar graphs and graphs with bounded tree width.

In Chap. 6 Christian Breimann and Jan Vahrenhold give a comprehensive overview of algorithms and data structures handling geometric objects like points and lines — an area that is at least as rich as graph algorithms. A third area of again quite different algorithmic techniques are string problems discussed by Juha Kärkäinen and Srinivasa Rao in Chap. 7.

Chapters 8–10 then turn to more detailed models with particular emphasis on the complications introduced by hardware caches. Beyond this common motivation, these chapters are quite diverse. Naila Rahman uses sorting as an example for these issues in Chap. 8 and puts particular emphasis on the often neglected issue of TLB misses. Piyush Kumar introduces *cache-oblivious algorithms* in Chap. 9 that promise to grasp multilevel hierarchies within a very simple model. Markus Kowarschik and Christian Weiß give a practical introduction into cache-efficient programs using numerical algorithms as an example. Numerical applications are particularly important because they allow significant instruction-level parallelism so that slow memory accesses can dramatically slow down processing.

Stefan Edelkamp introduces an application area of very different character in Chap. 11. In artificial intelligence, search programs have to handle huge state spaces that require sophisticated techniques for representing and traversing them.

Chapters 12–14 give a system-oriented view of advanced memory hierarchies. On the lowest level we have storage networks connecting a large number of inhomogeneous disks. Kay Salzwedel discusses this area with particular emphasis on the aspect of inhomogeneity. File systems give a more abstract view of these devices on the operating system level. Florin Isaila explains the organization of modern file systems in Chap. 13. An even higher level view is offered by relational database systems. Josep Larriba-Pey explains their organization in Chap. 14. Both in file systems and databases, basic algorithmic techniques like sorting and search trees turn out to be relevant.

Finally, Chaps. 15 and 16 give a glimpse on memory hierarchies with multiple processors. Massimo Coppola and Martin Schmollinger introduce abstract and concrete programming models like BSP and MPI in Chap. 15. Dani Jimenez, Josep-L. Larriba, and Juan J. Navarro present a concrete case study of sorting algorithms on shared memory machines in Chap. 16. He studies programming techniques that avoid pitfalls like true and false sharing of cache contents.

Most chapters in this volume have partly tutorial character and are partly more dense overviews. At a minimum Chaps. 1, 2, 3, 4, 9, 10, 14, and 16 are tutorial chapters suitable for beginning graduate-level students. They are sufficiently self-contained to be used for the core of a course on external memory algorithms. Augmented with the other chapters and additional papers it should be possible to shape various advanced courses. Chapters 1–3 lay the basis for the remaining chapters that are largely independent.

We are indebted to many people and institutions. We name a few in alphabetical order. Ulrik Brandes helped with sources from a tutorial volume on graph drawing that was our model in several aspects. The International Conference and Research Center for Computer Science in Dagstuhl provided its affordable conference facilities and its unique atmosphere. Springer-Verlag, and in particular Alfred Hofmann, made it possible to smoothly publish the volume in the LNCS series. Kurt Mehlhorn's group at MPI Informatik provided funding for several (also external) participants. Dorothea Wagner came up with the idea for the seminar and advised us in many ways. This volume was also partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

January 2003

Ulrich Meyer
Peter Sanders
Jop Sibeyn

Table of Contents

List of Contributors

Editors

Ulrich Meyer
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`uli@uli-meyer.de`

Peter Sanders
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`sanders@mpi-sb.mpg.de`

Jop F. Sibeyn
Martin-Luther Universität
Halle-Wittenberg
Institut für Informatik
Von-Seckendorff-Platz 1
06120 Halle, Germany
`jopsi@informatik.uni-halle.de`

Authors

Christian Breimann
Westfälische Wilhelms-Universität
Institut für Informatik
Einsteinstr. 62
48149 Münster, Germany
`chr@math.uni-muenster.de`

Massimo Coppola
University of Pisa
Department of Computer Science
Via F. Buonarroti 2
56127 Pisa, Italy
`coppola@di.unipi.it`

Stefan Edelkamp
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg, Germany
`edelkamp@informatik.uni-freiburg.de`

Florin Isaila
University of Karlsruhe
Department of Computer Science
PO-Box 6980
76128 Karlsruhe, Germany
`florin@ipd.uni-karlsruhe.de`

Dani Jiménez-González
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
`djimenez@ac.upc.es`

Irit Katriel
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`irit@mpi-sb.mpg.de`

Juha Kärkkäinen
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`juha@mpi-sb.mpg.de`

Markus Kowarschik
Friedrich–Alexander–Universität
Erlangen–Nürnberg
Lehrstuhl für Informatik 10
Cauerstraße 6,
91058 Erlangen, Germany
`Markus.Kowarschik@cs.fau.de`

Piyush Kumar
State University of New York
at Stony Brook
Department of Computer Science
Stony Brook, NY 11790, USA
`piyush@acm.org`

Josep-L. Larriba-Pey
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
`larri@ac.upc.es`

Anil Maheshwari
Carleton University
School of Computer Science
1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6, Canada
`maheshwa@scs.carleton.ca`

Ulrich Meyer
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
`uli@uli-meyer.de`

Juan J. Navarro
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
`juanjo@ac.upc.es`

Rasmus Pagh
The IT University of Copenhagen
Glentevej 67
2400 København NV, Denmark
`pagh@it-c.dk`

Naila Rahman
University of Leicester
Department of Mathematics
and Computer Science
University Road
Leicester, LE1 7RH, U. K.
`naila@mcs.le.ac.uk`

S. Srinivasa Rao
University of Waterloo
School of Computer Science
200 University Avenue West
Waterloo, Ontario, N2L 3G1, Canada
`ssrao@monod.uwaterloo.ca`

Kay A. Salzwedel
Universität Paderborn
Heinz Nixdorf Institut
Fürstenallee 11
33102 Paderborn, Germany
nkz@upb.de

Peter Sanders
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Martin Schmollinger
Universität Tübingen
Wilhelm-Schickard Institut
für Informatik, Sand 14
72076 Tübingen, Germany
martin.schmollinger
@informatik.uni-tuebingen.de

Laura Toma
Duke University
Department of Computer Science
Durham, NC 27708, USA
laura@cs.duke.edu

Jan Vahrenhold
Westfälische Wilhelms-Universität
Institut für Informatik
Einsteinstr. 62
48149 Münster, Germany
jan@math.uni-muenster.de

Christian Weiß
Technische Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Boltzmannstr. 3
85748 München, Germany
weissc@in.tum.de

Norbert Zeh
Duke University
Department of Computer Science
Durham, NC 27708, USA
nzeh@cs.duke.edu

1. Memory Hierarchies — Models and Lower Bounds

Peter Sanders*

The purpose of this introductory chapter is twofold. On the one hand, it serves the rather prosaic purpose of introducing the basic models and notations used in the subsequent chapters. On the other hand, it explains why these simple abstract models can be used to develop better algorithms for complex real world hardware.

Section 1.1 starts with a basic motivation for memory hierarchies and Section 1.2 gives a glimpse on their current and future technological realizations. More theoretically inclined readers can skip or skim this section and directly proceed to the introduction of the much simpler abstract models in Section 1.3. Then we have all the terminology in place to explain the guiding principles behind algorithm design for memory hierarchies in Section 1.4. A further issue permeating most external memory algorithms is the existence of fundamental lower bounds on I/O complexity described in Section 1.5. Less theoretically inclined readers can skip the proofs but might want to remember these bounds because they show up again and again in later chapters.

Parallelism is another important approach to high performance computing that has many interactions with memory hierarchy issues. We describe parallelism issues in subsections that can be skipped by readers only interested in sequential memory hierarchies.

1.1 Why Memory Hierarchies

There is a wide spectrum of computer applications that can make use of arbitrarily large amounts of memory. For example, consider geographic information systems. NASA measures the data volumes from satellite images in petabytes (10^{15} bytes). Similar figures are given by climate research centers and particle physicists [555].

Although it is unlikely that all these informations will be needed in a single application, we can easily come to huge data sets. For example, consider a map of the world that associates 32 bits with each square meter of a continent — something technologically quite feasible with modern satellite imagery. We would get a data set of about 600 terabytes.

Other examples of huge data sets are *data warehouses* of large companies that keep track of every single transaction, digital libraries for books, images, and movies (a single image frame of high quality movie takes several

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

megabytes), or large scale numerical simulations. Even if the input and output of an application are small, it might be necessary to store huge intermediate data structures. For example, some of the state space search algorithms in Chapter 11 are of this type.

How should a machine for processing such large inputs look? Data should be cheap to store but we also want fast processing. Unfortunately, there are fundamental reasons why we cannot get memory that is at the same time cheap, compact, and fast. For example, no signal can propagate faster than light. Hence, given a storage technology and a desired access latency, there is only a finite amount of data reachable within this time limit. Furthermore, in a cheap and compact storage technology there is no room for wires reaching every single memory cell. It is more economical to use a small number of devices that can be moved to access a given bit.

There are several approaches to escape this so called *memory wall* problem. The simplest and most widely used compromise is a *memory hierarchy.* There are several categories of memory in a computer ranging from small and fast to large, cheap, and slow. Even in a memory hierarchy, we can process huge data sets efficiently. The reason is that although *access latencies* to the huge data sets are large, we can still achieve large bandwidths by accessing many close-by bits together and by using several memory units in parallel. Both approaches can be modeled using the same abstract view: Access to large blocks of memory is almost as fast as access to a single bit. The algorithmic challenge following from this principle is to design algorithms that perform well on systems with blocked memory access. This is the main subject of this volume.

1.2 Current Technology

Although we view memory hierarchies as something fundamental, it is instructive to look at the way memory hierarchies are currently designed and how they are expected to change in the near future. More details and explanations can be found in the still reasonably up to date textbook [392]. A valuable and up-to-date introductory source is the web page on PC Technology `http://www.pctechguide.com/`.

Currently, a high performance microprocessor has a file of *registers* that have multiple ports so that several accesses can be made in parallel. For example, twelve parallel accesses must be supported by a superscalar machine that executes up to four instructions per clock cycle each of which addresses three registers.

Since multiple ports require too much chip area per bit, the *first level (L1) cache* supports only one or two accesses per clock. Each such access already incurs a delay of a few clock cycles since additional stages of the instruction processing pipelines have to be traversed. L1 cache is usually only a few kilobytes large because a larger area would incur longer connections and hence even larger access latencies [399]. Often there are separate L1 caches for instructions and data.

The *second level (L2) cache* is on the same chip as the first level cache but it has quite different properties. The L2 cache is as large as the technology allows because applications that fit most of their data into this cache can execute very fast. The L2 cache has access latencies around ten clock cycles. Communication between L1 and L2 cache uses block sizes of 16–32 bytes. For accessing off-chip data, larger blocks are used. For example, the Pentium 4 uses 128 byte blocks [399].

Some processors have a *third level (L3) cache* that is on a separate set of chips. This cache is made out of fast static[1] RAM cells. The L3 cache can be very large in principle, but this is not always cost effective because static RAMs are rather expensive.

The *main memory* is made out of high density cheap dynamic RAM cells. Since the access speeds of dynamic RAMs have lagged behind processor speeds, dynamic RAMs have developed into devices optimized for block access. For example, RAMBUS RDRAM[2] chips allow blocks of up to 16 bytes to be accessed in only twice the time to access a single byte.

The programmer is not required to know about the details of the hierarchy between caches and main memory. The hardware cuts the main memory into blocks of fixed size and automatically maps a subset of the memory blocks to L3 cache. Furthermore, it automatically maps a subset of the blocks in L3 cache to L2 cache and from L2 cache to L1 cache. Although this automatic cache administration is convenient and often works well, one is up to unpleasant surprises. In Chapter 8 we will see that sometimes a careful manual mapping of data to the memory hierarchy would work much better.

The backbone of current data storage are magnetic *hard disks* because they offer cheap non volatile memory [643]. In the last years, extremely high densities have been achieved for magnetic surfaces that allow several gigabytes to be stored on the area of a postage stamp. The data is accessed by tiny magnetic devices that hover as low as 20 nm over the surface of the rotating disk. It takes very long to move the access head to a particular track of the disk and to wait until the disk rotates into the correct position. With up to 10 ms, disk access can be 10^7 times slower than an access to a register. However, once the head starts reading or writing, data can be transferred at a rate of about 50 megabytes per second. Hence, accessing hundreds of KB takes only about twice as long as accessing a single byte. Clearly, it makes sense to process data in large chunks.

Hard disks are also used as a way to virtually enlarge the main memory. Logical blocks that are currently not in use are swapped to disk. This mechanism is partially supported by the processor hardware that is able to

1 Static RAM needs six transistors per bit which makes it more area consuming but faster than *dynamic* RAM that needs only one transistor per bit.

2 `http://www.rambus.com`

automatically translate between logical memory addresses and physical memory addresses. This translation uses yet another small cache, the *translation lookaside buffer (TLB)*

There is a final level of memory hierarchy used for backups and archiving of data. Magnetic tapes and optical disks allow even cheaper storage of data but have a very high access latency ranging from seconds to minutes because the media have to be retrieved from a shelf and mounted on some access device.

Current and Future Developments

There are too many possible developments to explain or even perceive all of them in detail but a few basic trends should be noted. The memory hierarchy might become even deeper. Third level caches will become more common. Intel has even integrated it on the Itanium 2 processor. In such a system, an off-chip 4th level cache makes sense. There is also a growing gap between the access latencies and capacities of disks and main memory. Therefore, magnetic storage devices with smaller capacity but also lower access latency have been proposed [669].

While storage density in CMOS-RAMs and magnetic disks will keep increasing for quite some time, it is conceivable that different technologies will get their chance in a longer time frame. There are some ideas available that would allow memory cells consisting of single molecules [780]. Furthermore, even with current densities, astronomically large amounts of data could be stored using three-dimensional storage devices. The main difficulty is how to write and read such memories. One approach uses holographic images storing large blocks of data in small three-dimensional regions of a transparent material [716].

Regardless of the technology, it seems likely that block-wise access and the use of parallelism will remain necessary to achieve high performance processing of large volumes of data.

Parallelism

A more radical change in the model is explicit parallel processing. Although this idea is not so new, there are several reasons why it might have increased impact in the near future. Microprocessors like the Intel Xeon first delivered in 2002 have multiple register sets and are able to execute a corresponding number of threads of activity in parallel. These threads share the same execution pipeline. Their accumulated performance can be significantly higher than the performance of a single thread with exclusive access to the processing resources. One main reason is that while one thread is waiting for a memory access to finish, another thread can use the processor. Parallelism spreads in many other respects. Several processors on the same chip can share a main memory and a second level cache. The IBM Power 4 processor already implements this technology. Several processors on different chips can share main memory. Several processor boards can share the same network of disks. Servers usually have many disk drives. In such systems, it becomes more and more important that memory devices on all levels of the memory hierarchy can work on multiple memory accesses in parallel.

On parallel machines, some levels of the memory hierarchy may be shared whereas others are distributed between the processors. Local caches may hold copies of shared or remote data. Thus, a read access to shared data may be as fast as a local access. However, writing shared data invalidates all the copies that are not in the cache of the writing processor. This can cause severe overhead for sending the invalidations and for reloading the data at subsequent remote accesses.

1.3 Modeling

We have seen that real memory hierarchies are very complex. We have multiple levels, all with their own idiosyncrasies. Hardware caches have replacement strategies that vary between simplistic and strange [294], disks have position dependent access delays, etc. It might seem that the best models are those that are as accurate as possible. However, for algorithm design, this leads the wrong way. Complicated models make algorithms difficult to design and analyze. Even if we overcome these differences, it would be very difficult to interpret the results because complicated models have a lot of parameters that vary from machine to machine.

Attractive models for algorithm design are very simple, so that it is easy to develop algorithms. They have few parameters so that it is easy to compare the performance of algorithms. The main issue in model design is to find simple models that grasp the essence of the real situation so that algorithms that are good in the model are also good in reality.

In this volume, we build on the most widely used nonhierarchical model. In the *random access machine (RAM) model* or *von Neumann model* [579], we have a "sufficiently" large uniform memory storing *words* of size $\mathcal{O}(\log n)$ bits where n is the size of our input. Accessing any word in memory takes constant time. Arithmetics and bitwise operations with words can be performed in constant time. For numerical and geometric algorithms, it is sometimes also assumed that words can represent real numbers accurately. Storage consumption is measured in words if not otherwise mentioned.

Most chapters of this volume use a minimalistic extension that we will simply call the *external memory model*. We use the notation introduced by Aggarwal, Vitter, and Shriver [17, 755]. Processing works almost as in the RAM model, except that there are only M words of *internal memory* that can be accessed quickly. The remaining memory can only be accessed using *I/Os* that move B contiguous words between internal and *external memory*.

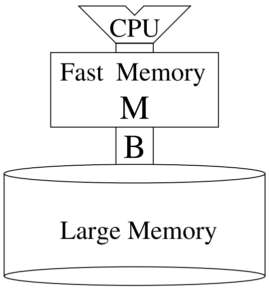


Fig. 1.1. The external memory model.

Figure 1.1 depicts this arrangement. To analyze an external memory algorithm, we count the number of I/Os needed in addition to the time that would be needed on a RAM machine.

Why is such a simple model adequate to describe something as complex as memory hierarchies? The easiest justification would be to lean on authority. Hundreds of papers using this model have been published, many of them in top conferences and journals. Many external memory algorithms developed are successfully used in practice. Vitter [754] gives an extensive overview. But why is this model so successful? Although the word "I/O" suggests that external memory should be identified with disk memory, we are free to choose any two levels of the memory hierarchy for internal and external memory in the model. Inaccuracies of the model are usually limited by reasonable constant factors. This claim needs further explanation. The main problem with hardware caches is that they use a fixed simplistic strategy for deciding which blocks are kept whereas the external memory model gives the programmer full control over the content of internal memory. Although this difference can have devastating effects, it rarely happens in practice. Mehlhorn and Sanders [543] give an explanation of this effect for a large class of cache access patterns. Sen and Chatterjee [685] and Frigo et al. [321] have observed that in principle we can even circumvent hardware replacement schemes and take explicit control of cache content.

Hard disks are even more complicated than caches [643] but again inaccuracies of the external memory model are not as big as one might think: Disks have their own local caches. But these are so small that for algorithms that process really large data sets they do not make a big difference. Roughly speaking, the disk access time consists of a latency needed to move the disk head to the appropriate position and a transfer time that is proportional to the amount of data transmitted. We are more or less free to choose this amount of data and hence it is not accurate to only count the number of accesses. However, if we fix the block size so that the transfer time is about the same as the latency, we only make a small error. Let us explain this for the (oversimplified) case that time $t_0 + B$ is needed to access B words of data.

Then a good choice of the block size is $B = t_0$. When we access less data we are at most a factor two off by accessing an entire block of size B. When we access $L > B$ words, we are at most a factor two off by counting $\lceil L/B \rceil$ block I/Os.

In reality, the access latency depends on the current position of the disk mechanism and on the position of the block to be accessed on the disk. Although exploiting this effect can make a big difference, programs that optimize access latencies are rare since the details depend on the actual disk used and are usually not published by the disk vendors. If other applications or the operating system make additional unpredictable accesses to the same disk, even sophisticated optimizations can be in vain. In summary, by picking an appropriate block size, we can model the most important aspects of disk drives.

Parallelism

Although we mostly use the sequential variant of the external memory model, it also has an option to express parallelism. External memory is partitioned into D parts (e.g. disks) so that in each I/O step, one block can be accessed on *each* of the parts.

With respect to parallel disks, the model of Vitter and Shriver [755] deviates from an earlier model by Aggarwal and Vitter [17] where D *arbitrary* blocks can be accessed in parallel. A hardware realization could have D reading/writing devices that access a single disk or a D-ported memory. This model is more powerful because algorithms need not care about the mapping of data to disks. However, there are efficient (randomized) algorithms for emulating the Aggarwal-Vitter model on the Vitter-Shriver model [656]. Hence, one approach to developing parallel disk external memory algorithms is to start with an algorithm for the Aggarwal-Vitter model and then add an appropriate load balancing algorithm (also called *declustering*).

Vitter and Shriver also make provisions for parallel processing. There are P identical processors that can work in parallel. Each has fast memory M/P and is equipped with D/P disks. In the external memory model there are no additional parameters expressing the communication capabilities of the processors. Although this is an oversimplification, this is already enough to distinguish many algorithms with respect to their ability to be executed on parallel machines. The model seems suitable for parallel machines with shared memory.

For discussing parallel external memory on machines with distributed memory we need a model for communication cost. The *BSP model* [742] that is widely accepted for parallel (internal) processing fits well here: The P processors work in *supersteps*. During a superstep, the processors can perform local communications and post messages to other processors to the communication subsystem. At the end of a superstep, all processors synchronize and exchange all the messages that have been posted during the superstep. This synchronous communication takes time $\ell + gh$ where ℓ is the *latency*, g the *gap* and h the maximum number of words a processor sends or receives in this communication phase. The parameter ℓ models the overhead for synchronizing the processors and the latency of messages traveling through the network. If we assume our unit of time to be the time needed to execute one instruction, the parameter g is the ratio between communication speed and computation speed.

1.3.1 More Models

In Chapter 8 we will see more refined models for the fastest levels of the memory hierarchy, including replacements strategies used by the hardware and the role of the TLB. Chapter 10 contributes additional practical examples from numeric computing. Chapter 15 will explain parallel models in more detail. In particular, we will see models that take multiple levels of hierarchy into account.

There are also alternative models for the simple sequential memory hierarchy. For example, instead of counting block I/Os with respect to a block size B, we could allow variable block sizes and count the *number* of I/Os k and the total I/O *volume* h. The total I/O cost could then be accounted as $\ell_{\mathrm{I/O}}k + g_{\mathrm{I/O}}v$ where — in analogy to the BSP model — $\ell_{\mathrm{I/O}}$ stands for the I/O latency and $g_{\mathrm{I/O}}$ for the ratio between I/O speed and computation speed. This model is largely equivalent to the block based model but it might be more elegant when used together with the BSP model and it is more adequate to explain differences between algorithms with regular and irregular access patterns [227].

Another interesting variant is the *cache oblivious* model discussed in Chapter 9. This model is identical to the external memory model except that the algorithm is not told the values of B and M. The consequence of this seemingly innocent variant is that an I/O efficient cache oblivious algorithm works well not only on any machine but also on all levels of the memory hierarchy at the same time. Cache oblivious algorithms can be very simple, i.e., we do not need to know B and M to scan an array. But even cache oblivious sorting is quite difficult.

Finally, there are interesting approaches to *eliminate* memory hierarchies. Blocked access is only one way to hide access latency. Another approach is *pipelining* where many independent accesses are executed in parallel. This approach is more powerful but also more difficult to support in hardware. Vector computers such as the NEC SX-6 support pipelined memory access even to nonadjacent cells at full memory bandwidth. Several experimental machines [2, 38] use massive pipelined memory access by the hardware to run many parallel threads on a single processor. While one thread waits for a memory access, the other threads can do useful work. Modern mainstream processors also support pipelined memory access to a certain extend [399].

1.4 Issues in External Memory Algorithm Design

Before we look at particular algorithms in the rest of this volume, let us first discuss the goals we should achieve by an external memory algorithm. Ideally, the user should not notice the difference between external memory and internal memory at all, i.e., the program should run as fast as if all the memory would be internal memory. The following principles help:

Internal efficiency: The internal work done by the algorithm should be comparable to the best internal memory algorithms.
Spatial locality: When a block is accessed, it should contain as much useful data as possible.
Temporal locality: Once data is in the internal memory, as much useful work as possible should be done on it before it is written back to external memory.

Which of these criteria is most important, depends a lot on the application and on the hardware used. As usual in computer science, the overall performance is mostly determined by the weakest link. Let us consider a prototypical scenario. Assume we have a good internal memory algorithm for some application. Now it turns out that we want to run it on much larger inputs and internal memory will not suffice any more. The first try could be to ignore this problem and see how the virtual memory capability of the operating system deals with it. When this works, we are exceptionally lucky. If we see very bad performance, this usually means that the existing algorithm has poor locality. We may then apply the algorithmic techniques developed in this volume to improve locality.

Several outcomes are possible. It may be that despite our effort, locality remains the limiting factor. When discussing further improvements we will then focus on locality and might even accept an increase of internal work.

But we should keep in mind that many algorithms do some useful work for every word accessed, i.e., locality is quite good. If the application nevertheless remains *I/O-bound*, this means that the I/O bandwidth of our system is low. This is a common observation when researchers run their external memory algorithms on workstations with a single disk and I/O interfaces not build for high performance I/O. However, we should expect that serious applications of external memory algorithms will run on hardware and software build for high I/O performance. Let us consider a machine recently configured by Roman Dementiev and the author as an example. The parts for this system cost about 3000 Euro in July 2002, i.e., the price is in the range of an ordinary workstation. The STREAM[3] benchmark achieves a main memory bandwidth of 1445MB/s on one of two 2.0 GHz Intel Xeon processors. Using eight disks and four IDE controllers, we achieve an I/O bandwidth of up to 375 MB/s, i.e., the bandwidth gap between main memory and disks is not very large.

[3] `http://www.streambench.org/`

For example, our first implementation of external memory sorting on this machine used internal quicksort as a subroutine. For more than two disks this internal sorting was the main bottleneck. Hence, internal efficiency is a really important aspect of good external memory algorithms.

Parallelism

In parallel models, internal efficiency and locality is as important as in the sequential case. In particular, temporal and spatial locality with respect to the local memory of a processor is an issue.

An new issue is *load balancing* or *declustering* . All disks should access useful data in most parallel I/O steps. All processors should have about the same amount of work during a superstep in the BSP model, and no processor should have to send or receive too much data at the end of a superstep.

When programming shared memory machines, the caching policies described above must be taken into account. *True sharing* occurs when several processors write to the same memory location. Such writes are expensive since they amount to invalidation and reloading of entire cache blocks by all other processors reading this location. Hence, parallel algorithms should avoid frequent write accesses to shared data. Moreover, even write accesses to different memory cells might lead to the same sharing effect if the cells are located on the same block of memory. This phenomenon is called *false sharing*. Chapter 16 studies true and false sharing in detail using sorting as an example.

1.5 Lower Bounds

A large number of external memory algorithms can be assembled from the three ingredients scanning, sorting, and searching. There are essentially matching upper and lower bounds for the number of I/Os needed to perform these operations:
Scanning: Look at the input once in the order it is stored. If N is the amount of data to be inspected, we obviously need

$$\mathrm{scan}(N) = \Theta(N/B) \text{ I/Os.} \tag{1.1}$$

Permuting and Sorting: Too often, the data is not arranged in a way that scanning helps. Then we can rearrange the data into an order where scanning is useful. When we already know where to place each elements, this means *permuting* the data. When the permutation is defined implicitly via a total ordering "<" of the elements, we have to sort with respect to "<". Chapter 3 gives an upper bound of

$$\mathrm{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \text{ I/Os} \tag{1.2}$$

for sorting. In Section 1.5.1, we will see an almost identical lower bound for permuting that is also a lower bound for the more difficult problem of sorting.
Searching: Any pointer based data structure indexing N elements needs access time

$$\mathrm{search}(N) = \Omega\left(\log_B N/M\right) \ \text{I/Os.} \tag{1.3}$$

This lower bound is explained in Section 1.5.2. In Chapter 2 we see a matching upper bound for the simple case of a linear order. High dimensional problems such as the geometric data structures explained in Chapter 6 can be more difficult.

Arge and Bro Miltersen [59] give a more detailed account of lower bounds for external memory algorithms.

1.5.1 Permuting and Sorting

We analyse the following problem. How many I/O operations are necessary to generate a permutation of the input? A lower bound on permuting implies a lower bound for sorting because for every permutation of a set of elements, there is a set of keys that forces sorting to produce this permutation. The lower bound was established in a seminal paper by Aggarwal and Vitter [17]. Here we report a simplified proof based on unpublished lecture notes by Albers, Crauser, and Mehlhorn [24].

To establish a lower bound, we need to specify precisely what a permutation algorithm can do. We make some restrictions but most of them can be lifted without changing the lower bound significantly. We view the internal memory as a bag being able to hold up to M elements. External memory is an array of elements. Reading and writing external memory is always *aligned* to block boundaries, i.e., if the cells of external memory are numbered, access is always to cells $i, \dots, i + B - 1$ such that i is a multiple of B. At the beginning, the first N/B blocks of external memory contain the input. The internal memory and the remaining external memory contain no elements. At the end, the output is again in the first N/B blocks of the external memory. We view our elements as abstract objects, i.e., the only operation available on them is to move them around. They cannot be duplicated, split, or modified in any way. A read step moves B elements from a block of external memory to internal memory. A write step moves any B elements from internal memory into a block of external memory. In this model, the following theorem holds:

Theorem 1.1. *Permuting N elements takes at least*

$$t \geq 2\,\frac{N}{B} \cdot \frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B} \ \textit{I/Os.}$$

For $N = \mathcal{O}\big((eM/B)^{B/2}\big)$ the term $\log(eM/B)$ dominates the denominator and we get a lower bound for sorting of

$$2\frac{N}{B} \cdot \frac{\log(N/eB)}{\mathcal{O}(\log(eM/B))} = \Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) .$$

which is the same as the upper bound from Chapter 3.

The basic approach for establishing Theorem 1.1 is simple. We find an upper bound c_t for the number of different permutations generated after t I/O steps looking at all possible sequences of t I/O steps. Since there are $N!$ possible permutations of N elements, t must be large enough such that $c_t \geq N!$ because otherwise there are permutations that cannot be generated using t I/Os. Solving for t yields the desired lower bound.

A state of the algorithm can be described abstractly as follows:

1. the set of elements in the main memory;
2. the set of elements in each nonempty block of external memory;
3. the permutation in which the elements in each nonempty block of external memory are stored.

We call two states *equivalent* if they agree in the first two components (they may differ in the third).

In the final state, the elements are stored in N/B blocks of B elements each. Each equivalence class of final states therefore consists of $(B!)^{N/B}$ states. Hence, it suffices for our lower bound to find out when the number of equivalence classes of final states C_t reachable after t I/Os exceeds $N!/(B!)^{N/B}$.

We estimate C_t inductively. Clearly $C_0 = 1$.

Lemma 1.2. $C_{t+1} \leq \begin{cases} C_t N/B & \textit{if the I/O-operation is a read} \\ C_t N/B \cdot \binom{M}{B} & \textit{if the I/O-operation is a write.} \end{cases}$

Proof. A read specifies which out of N/B nonempty blocks is to be read. A write additionally specifies which elements are to be written and in which permutation. If i elements are written, there are $\binom{M}{i} \leq \binom{M}{B}$ choices for the elements to be written and their permutation is irrelevant as far as equivalence of states is concerned. The inequality $\binom{M}{i} \leq \binom{M}{B}$ assumes that $B \leq M/2$. ■

Lemma 1.3. *In any algorithm that produces a permutation in our model, the number of reads equals the number of writes.*

Proof. A read increments the number of empty blocks. A write decrements the number of empty blocks. At the beginning and at the end there are exactly N/B nonempty blocks. Hence, the number of increases equals the number of decreases. ■

Combining Lemmas 1.2 and 1.3 we see that for even t,

$$\frac{N!}{(B!)^{N/B}} \leq C_t \leq \left(\frac{N}{B}\right)^t \cdot \binom{M}{B}^{t/2} \tag{1.4}$$

We can simplify this relation using the the well-known bounds $(m/e)^m \leq m! \leq m^m$. We get $\binom{M}{B} \leq M^B/B! \leq (eM/B)^B$ and $N!/(B!)^{N/B} \geq (N/e)^N/B^N$. Relation 1.4 therefore implies

$$\left(\frac{N}{B}\right)^t \cdot \left(\frac{eM}{B}\right)^{Bt/2} \geq \left(\frac{N}{eB}\right)^N ,$$

or, after taking logarithms and solving for t,

$$t \cdot \left(2\log\left(\frac{N}{B}\right) + B\log\left(\frac{eM}{B}\right)\right) \geq 2N\log\left(\frac{N}{eB}\right) \quad \text{or}$$

$$t \geq 2\,\frac{N}{B} \cdot \frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B} .$$

■

1.5.2 Pointer Based Searching

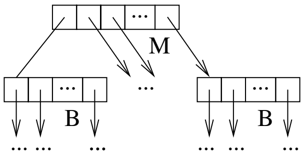


Fig. 1.2. Pointer based searching.

Consider a data structure storing a set of N elements in external memory. Access to blocks of external memory is *pointer based*, i.e., we are only allowed to access a block i if its address is actually stored somewhere. We play a similar game as for the sorting bound and count the number of different blocks C_t that can be accessed after i I/O operations. This count has to exceed N/B to make all elements accessible. Initially, the fast memory could be full of pointers so that we have $C_0 = M$. Each additional block read gives us a factor B more possibilities. Hence, $C_t = MB^t$. Figure 1.2 illustrates this situation. Solving $C_t \geq N/B$ yields $N \geq \log_B \frac{N}{MB}$ since we need an additional access for actually retrieving an element we get a lower bound of $\log_B N/M$ I/Os for being able to reach each of the N elements. ■

2. Basic External Memory Data Structures

Rasmus Pagh*

This chapter is a tutorial on basic data structures that perform well in memory hierarchies. These data structures have a large number of applications and furthermore serve as an introduction to the basic principles of designing data structures for memory hierarchies.

We will assume the reader to have a background in computer science that includes a course in basic (internal memory) algorithms and data structures. In particular, we assume that the reader knows about queues, stacks, and linked lists, and is familiar with the basics of hashing, balanced search trees, and priority queues. Knowledge of amortized and expected case analysis will also be assumed. For readers with no such background we refer to one of the many textbooks covering basic data structures in internal memory, e.g., [216].

The model we use is a simple one that focuses on just two levels of the memory hierarchy, assuming the movement of data between these levels to be the main performance bottleneck. (More precise models and a model that considers all memory levels at the same time are discussed in Chapter 8 and Chapter 9.) Specifically, we consider the *external memory* model described in Chapter 1.

Our notation is summarized in Fig. 2. The parameters M, w and B describe the model. The size of the problem instance is denoted by N, where $N \leq 2^w$. The parameter Z is query dependent, and is used to state output sensitive I/O bounds. To reduce notational overhead we take logarithms to always be at least 1, i.e., $\log_a b$ should be read "$\max(\log_a b, 1)$".

N – number of data items
M – number of data items that can be stored in internal memory
B – number of data items that can be stored in an external memory block
Z – number of data items reported in a query
w – word length of processor and size of data items in bits

Fig. 2.1. Summary of notation

We will not go into details of external memory management, but simply assume that we can allocate a chunk of contiguous external memory of any size we desire, such that access to any block in the chunk costs one I/O. (However, as described in Section 2.2.1 we may use a dictionary to simulate virtual memory using just one large chunk of memory, incurring a constant factor I/O overhead).

* Part of this work was done while the author was at BRICS, University of Aarhus, where he was partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

2.1 Elementary Data Structures

We start by going through some of the most elementary data structures. These are used extensively in algorithms and as building blocks when implementing other data structures. This will also highlight some of the main differences between internal and external memory data structures.

2.1.1 Stacks and Queues

Stacks and queues represent dynamic sets of data elements, and support operations for adding and removing elements. They differ in the way elements are removed. In a *stack*, a remove operation deletes and returns the set element most recently inserted (last-in-first-out), whereas in a *queue* it deletes and returns the set element that was first inserted (first-in-first-out).

Recall that both stacks and queues for sets of size at most N can be implemented efficiently in internal memory using an array of length N and a few pointers. Using this implementation on external memory gives a data structure that, in the worst case, uses one I/O per insert and delete operation. However, since we can read or write B elements in one I/O, we could hope to do considerably better. Indeed this is possible, using the well-known technique of a *buffer*.

An External Stack. In the case of a stack, the buffer is just an internal memory array of $2B$ elements that at any time contains the k set elements most recently inserted, where $k \leq 2B$. Remove operations can now be implemented using no I/Os, except for the case where the buffer has run empty. In this case a single I/O is used to retrieve the block of B elements most recently written to external memory.

One way of looking at this is that external memory is used to implement a stack with *blocks* as data elements. In other words: The "macroscopic view" in external memory is the same as the "microscopic view" in internal memory. This is a phenomenon that occurs quite often – other examples will be the search trees in Section 2.3 and the hash tables in Section 2.4.

Returning to external stacks, the above means that at least B remove operations are made for each I/O reading a block. Insertions use no I/Os except when the buffer runs full. In this case a single I/O is used to write the B least recent elements to a block in external memory. Summing up, both insertions and deletions are done in $1/B$ I/O, in the amortized sense. This is the best performance we could hope for when storing or retrieving a sequence of data items much larger than internal memory, since no more that B items can be read or written in one I/O. A desired goal in many external memory data structures is that when reporting a sequence of elements, only $O(1/B)$ I/O is used per element. We return to this in Section 2.3.

Exercise 2.1. Why does the stack not use a buffer of size B?

An External Queue. To implement an efficient queue we use two buffers of size B, a read buffer and a write buffer. Remove operations work on the read buffer until it is empty, in which case the least recently written external memory block is read into the buffer. (If there are no elements in external memory, the contents of the write buffer is transfered to the read buffer.) Insertions are done to the read buffer which when full is written to external memory. Similar to before, we get at most $1/B$ I/O per operation.

Problem 2.2. Above we saw how to implement stacks and queues having a fixed bound on the maximum number of elements. Show how to efficiently implement external stacks and queues with *no* bound on the number of elements.

2.1.2 Linked Lists

Linked lists provide an efficient implementation of ordered lists of elements, supporting sequential search, deletions, and insertion in arbitrary locations of the list. Again, a direct implementation of the internal memory data structure could behave poorly in external memory: When traversing the list, the algorithm may need to perform one I/O every time a pointer is followed. (The task of traversing an *entire* linked list on external memory can be performed more efficiently. It is essentially *list ranking*, described in Chapter 3.)

Again, the solution is to maintain *locality*, i.e., elements that are near each other in the list must tend to be stored in the same block. An immediate idea would be to put chunks of B consecutive elements together in each block and link these blocks together. This would certainly mean that a list of length N could be traversed in $\lceil N/B \rceil$ I/Os. However, this invariant is hard to maintain when inserting and deleting elements.

Exercise 2.3. Argue that certain insertions and deletions will require $\lceil N/B \rceil$ I/Os if we insist on exactly B consecutive elements in every block (except possibly the last).

To allow for efficient updates, we relax the invariant to require that, e.g., there are more than $\frac{2}{3}B$ elements in every pair of consecutive blocks. This increases the number of I/Os needed for a sequential scan by at most a factor of three. Insertions can be done in a single I/O except for the case where the block supposed to hold the new element is full. If either neighbor of the block has spare capacity, we may push an element to this block. In case both neighbors are full, we *split* the block into two blocks of about $B/2$ elements each. Clearly this maintains the invariant (in fact, at least $B/6$ deletions will be needed before the invariant is violated in this place again). When deleting an element we check whether the total number of elements in the block and one of its neighbors is $\frac{2}{3}B$ or less. If this is the case we *merge* the two blocks. It is not hard to see that this reestablishes the invariant: Each of the two pairs involving the new block now have more elements than the corresponding pairs had before.

To sum up, a constant number of I/Os suffice to update a linked list. In general this is the best we can hope for when updates may affect any part of the data structure, and we want queries in an (eager) on-line fashion. In the data structures of Section 2.1.1, updates concerned very local parts of the data structure (the top of the stack and the ends of the queue), and we were able to to better. Section 2.3.5 will show that a similar improvement is possible in some cases where we can afford to wait for an answer of a query to arrive.

Exercise 2.4. Show that insertion of N consecutive elements in a linked list can be done in $O(1 + N/B)$ I/Os.

Exercise 2.5. Show how to implement concatenation of two lists and splitting of a list into two parts in $O(1)$ I/Os.

Problem 2.6. Show how to increase space utilization from $1/3$ to $1 - \epsilon$, where $\epsilon > 0$ is a constant, with no asymptotic increase in update time. (*Hint:* Maintain an invariant on the number of elements in any $\Theta(1/\epsilon)$ consecutive blocks.)

Pointers. In internal memory one often has pointers to elements of linked lists. Since memory for each element is allocated separately, a fixed pointer suffices to identify the element in the list. In external memory elements may be moved around to ensure locality after updates in other parts of the list, so a fixed pointer will not suffice. One solution is to maintain a list of pointers to the pointers, which allows them to be updated whenever we move an element. If the number of pointers to each element is constant, the task of maintaining the pointers does not increase the amortized cost of updates by more than a constant factor, and the space utilization drops only by a constant factor, assuming that each update costs $\Omega(1)$ I/Os. (This need not be the case, as we saw in Exercise 2.4.) A solution that allows an arbitrary number of pointers to each list element is to use a dictionary to maintain the pointers, as described in Section 2.2.1.

2.2 Dictionaries

A *dictionary* is an abstract data structure that supports *lookup* queries: Given a *key* k from some finite set K, return any information in the dictionary associated with k. For example, if we take K to be the set of social security numbers, a dictionary might associate with each valid social security number the tax information of its owner. A dictionary may support dynamic updates in the form of insertions and deletion of keys (with associated information).

Recall that N denotes the number of keys in the dictionary, and that B keys (with associated information) can reside in each block of external memory.

There are two basic approaches to implementing dictionaries: Search trees and hashing. Search trees assume that there is some total ordering on the key set. They offer the highest flexibility towards extending the dictionary to support more types of queries. We consider search trees in Section 2.3. Hashing based dictionaries, described in Section 2.4, support the basic dictionary operations in an expected constant number of I/Os (usually one or two). Before describing these two approaches in detail, we give some applications of external memory dictionaries.

2.2.1 Applications of Dictionaries

Dictionaries can be used for simple database retrieval as in the example above. Furthermore, they are useful components of other external memory data structures. Two such applications are implementations of *virtual memory* and *robust pointers.*

Virtual Memory. External memory algorithms often do allocation and deallocation of arrays of blocks in external memory. As in internal memory this can result in problems with fragmentation and poor utilization of external memory. For almost any given data structure it can be argued that fragmentation can be avoided, but this is often a cumbersome task.

A general solution that gives a constant factor increase in the number of I/Os performed is to implement virtual memory using a dictionary. The key space is $K = \{1, \dots, C\} \times \{1, \dots, L\}$, where C is an upper bound of the number of arrays we will ever use and L is an upper bound on the length of any array. We wish the ith block of array c to be returned from the dictionary when looking up the key (c, i). In case the block has never been written to, the key will not be present, and some standard block content may be returned. Allocation of an array consists of choosing $c \in \{1, \dots, C\}$ not used for any other array (using a counter, say), and associating a linked list of length 0 with the key $(c, 0)$. When writing to block i of array c in virtual memory, we associate the block with the key (c, i) in the dictionary and add the number i to the linked list of key $(c, 0)$. For deallocation of the array we simply traverse the linked list of $(c, 0)$ to remove all keys associated with that array.

In case the dictionary uses $O(1)$ I/Os per operation (amortized expected) the overhead of virtual memory accesses is expected to be a constant factor. Note that the cost of allocation is constant and that the amortized cost of deallocation is constant. If the dictionary uses linear space, the amount of external memory used is bounded by a constant times the amount of virtual memory in use.

Robust Pointers into Data Structures. Pointers into external memory data structures pose some problems, as we saw in Section 2.1.2. It is often possible to deal with such problems in specific cases (e.g., level-balanced B-trees described in Section 2.3.4), but as we will see now there is a general solution that, at the cost of a constant factor overhead, enables pointers to be maintained at the cost of $O(1)$ I/O (expected) each time an element is moved. The solution is to use a hashing based dictionary with constant lookup time and expected constant update time to map "pointers" to disk blocks. In this context a pointer is any kind of unique identifier for a data element. Whenever an element is moved we simply update the information associated with its pointer accordingly. Assuming that pointers are succinct (not much larger than ordinary pointers) the space used for implementing robust pointers increases total space usage by at most a constant factor.

2.3 B-trees

This section considers search trees in external memory. Like the hashing based dictionaries covered in Section 2.4, search trees store a set of keys along with associated information. Though not as efficient as hashing schemes for lookup of keys, we will see that search trees, as in internal memory, can be used as the basis for a wide range of efficient queries on sets (see, e.g., Chapter 6 and Chapter 7). We use N to denote the size of the key set, and B to denote the number of keys or pointers that fit in one block.

B-trees are a generalization of balanced binary search trees to balanced trees of degree $\Theta(B)$ [96, 207, 416, 460]. The intuitive reason why we should change to search trees of large degree in external memory is that we would like to use all the information we get when reading a block to guide the search. In a naïve implementation of binary search trees there would be no guarantee that the nodes on a search path did not reside in distinct blocks, incurring $O(\log N)$ I/Os for a search. As we shall see, it is possible to do significantly better. In this section it is assumed that $B/8$ is an integer greater than or equal to 4.

The following is a modification of the original description of B-trees, with the essential properties preserved or strengthened. In a B-tree all leaves have the same distance to the root (the height h of the tree). The *level* of a B-tree node is the distance to its descendant leaves. Rather than having a single key in each internal node to guide searches to one of two subtrees, a B-tree node guides searches to one of $\Theta(B)$ subtrees. In particular, the number of leaves below a node (called its *weight*) decreases by a factor of $\Theta(B)$ when going one level down the tree. We use a *weight balance* invariant, first described for B-trees by Arge and Vitter [71]: Every node at level $i < h$ has weight at least $(B/8)^i$, and every node at level $i \leq h$ has weight at most $4(B/8)^i$. As shown in the following exercise, the weight balance invariant implies that the degree of any non-root node is $\Theta(B)$ (this was the invariant in the original description of B-trees [96]).

Exercise 2.7. Show that the weight balance invariant implies the following:

1. Any node has at most $B/2$ children.
2. The height of the B-tree is at most $1 + \lceil \log_{B/8} N \rceil$.
3. Any non-root node has at least $B/32$ children.

Note that $B/2$ pointers to subtrees, $B/2 - 1$ keys and a counter of the number of keys in the subtree all fit in one external memory block of size B. All keys and their associated information are stored in the leaves of the tree, represented by a linked list containing the sorted key sequence. Note that there may be fewer than $\Theta(B)$ elements in each block of the linked list if the associated information takes up more space than the keys.

2.3.1 Searching a B-tree

In a binary search tree the key in a node splits the key set into those keys that are larger or equal and those that are smaller, and these two sets are stored separately in the subtrees of the node. In B-trees this is generalized as follows: In a node v storing keys $k_1, \ldots, k_{d_v - 1}$ the ith subtree stores keys k with $k_{i-1} \leq k < k_i$ (defining $k_0 = -\infty$ and $k_{d_v} = \infty$). This means that the information in a node suffices to determine in which subtree to continue a search.

The worst-case number of I/Os needed for searching a B-tree equals the worst-case height of a B-tree, found in Exercise 2.7 to be at most $1 + \lceil \log_{B/8} N \rceil$. Compared to an external binary search tree, we save roughly a factor $\log B$ on the number of I/Os.

Example 2.8. If external memory is a disk, the number $1 + \lceil \log_{B/8} N \rceil$ is quite small for realistic values of N and B. For example, if $B = 2^{12}$ and $N \leq 2^{27}$ the depth of the tree is bounded by 4. Of course, the root could be stored in internal memory, meaning that a search would require three I/Os.

Exercise 2.9. Show a lower bound of $\Omega(\log_B N)$ on the height of a B-tree.

Problem 2.10. Consider the situation where we have no associated information, i.e., we wish to store only the keys. Show that the maximum height of a B-tree can be reduced to $1 + \lceil \log_{B/8}(2N/B) \rceil$ by abandoning the linked list and grouping adjacent leaves together in blocks of at least $B/2$. What consequence does this improvement have in the above example?

Range Reporting. An indication of the strength of tree structures is that B-trees immediately can be seen to support *range queries*, i.e., queries of the form "report all keys in the range $[a; b]$" (we consider the case where there is no associated information). This can be done by first searching for the key a, which will lead to the smallest key $x \geq a$. We then traverse the linked list starting with x and report all keys smaller than b (whenever we encounter a block with a key larger than b the search is over). The number of I/Os used for reporting Z keys from the linked list is $O(Z/B)$, where $\lceil Z/B \rceil$ is the minimum number of I/Os we could hope for. The feature that the number of I/Os used for a query depends on the size of the result is called *output sensitivity.* To sum up, Z elements in a given range can be reported by a B-tree in $O(\log_B N + Z/B)$ I/Os. Many other reporting problems can be solved within this bound.

It should be noted that there exists an optimal size (static) data structure based on hashing that performs range queries in $O(1 + Z/B)$ I/Os [35]. However, a slight change in the query to "report the *first* Z keys in the range $[a; b]$" makes the approach used for this result fail to have optimal output sensitivity (in fact, this query provably has a time complexity that grows with N [98]). Tree structures, on the other hand, tend to easily adapt to such changes.

2.3.2 Inserting and Deleting Keys in a B-tree

Insertions and deletions are performed as in binary search trees except for the case where the weight balance invariant would be violated by doing so.

Inserting. When inserting a key x we search for x in the tree to find the internal node that should be the parent of the leaf node for x. If the weight constraint is not violated on the search path for x we can immediately insert x, and a pointer to the leaf containing x and its associated information. If the weight constraint is violated in one or more nodes, we rebalance it by performing *split* operations in overweight nodes, starting from the bottom and going up. To split a node v at level $i > 0$, we divide its children into two consecutive groups, each of weight between $2(B/8)^i - 2(B/8)^{i-1}$ and $2(B/8)^i + 2(B/8)^{i-1}$. This is possible as the maximum weight of each child is $4(B/8)^{i-1}$. Node v is replaced by two nodes having these groups as children (this requires an update of the parent node, or the creation of a new root if v is the root). Since $B/8 \geq 4$ the weight of each of these new nodes is between $\frac{3}{2}(B/8)^i$ and $\frac{5}{2}(B/8)^i$, which is $\Omega((B/8)^i)$ away from the limits.

Deleting. Deletions can be handled in a manner symmetric to insertions. Whenever deleting a leaf would violate the lower bound on the weight of a node v, we perform a rebalancing operation on v and a sibling w. If several nodes become underweight we start the rebalancing at the bottom and move up the tree.

Suppose v is an underweight node at level i, and that w is (one of) its nearest sibling(s). In case the combined weight of v and w is less than $\frac{7}{2}(B/8)^i$ we *fuse* them into one node having all the children of v and w as children. In case v and w were the only children of the root, this node becomes the new root. The other case to consider is when the combined weight is more than $\frac{7}{2}(B/8)^i$, but at most $5(B/8)^i$ (since v is underweight). In this case we make w *share* some children with v by dividing all the children into two consecutive groups, each of weight between $\frac{7}{4}(B/8)^i - 2(B/8)^{i-1}$ and $\frac{5}{2}(B/8)^i + 2(B/8)^{i-1}$. These groups are then made the children of v and w, respectively. In both cases, the weight of all changed nodes is $\Omega((B/8)^i)$ away from the limits.

An alternative to doing deletions in this way is to perform periodical *global rebuilding*, a technique described in Section 2.5.2.

Analysis. The cost of rebalancing a node is $O(1)$ I/Os, as it involves a constant number of B-tree nodes. This shows that B-tree insertions and deletions can be done in $O(\log_B N)$ I/Os.

However, we have in fact shown something stronger. Suppose that whenever a level i node v of weight $W = \Theta((B/8)^i)$ is rebalanced we spend $f(W)$ I/Os to compute an auxiliary data structure used when searching in the subtree with root v. The above weight balancing arguments show that $\Omega(W)$ insertions and deletions in v's subtree are needed for each rebalancing operation. Thus, the amortized cost of maintaining the auxiliary data structures is $O(f(W)/W)$ I/Os per node on the search path of an update, or $O(\frac{f(W)}{W} \log_B N)$ I/Os per update in total. As an example, if the auxiliary data structure can be constructed by scanning the entire subtree in $O(W/B)$ I/Os, the amortized cost per update is $O(\frac{1}{B} \log_B N)$ I/Os, which is negligible.

Problem 2.11. Modify the rebalancing scheme to support the following type of weight balance condition: A B-tree node at level $i < h$ is the root of a subtree having $\Theta((B/(2+\epsilon))^i)$ leaves, where $\epsilon > 0$ is a constant. What consequence does this have for the height of the B-tree?

2.3.3 On the Optimality of B-trees

As seen in Chapter 1 the bound of $O(\log_B N)$ I/Os for searching is the best we can hope for if we consider algorithms that use only comparisons of keys to guide searches. If we have a large amount of internal memory and are willing to use it to store the top M/B nodes of the B-tree, the number of I/Os for searches and updates drops to $O(\log_B(N/M))$.

Exercise 2.12. How large should internal memory be to make $O(\log_B(N/M))$ asymptotically smaller than $O(\log_B N)$?

There are *non-comparison-based* data structures that break the above bound. For example, the predecessor dictionary mentioned in Section 2.4.2 uses linear space and time $O(\log w)$ to search for a key, where w denotes the number of bits in a key (below we call the amount of storage for a key a *word*). This is faster than a B-tree if N is much larger than B and w. Note that a predecessor dictionary also supports the range queries discussed in Section 2.3.1. There are also predecessor data structures whose search time *improves* with the number of bits that can be read in one step (in our case Bw bits). When translated to external memory, these results (see [48, 98, 373] and the references therein) can be summarized as follows:

Theorem 2.13. *There is an external memory data structure for N keys of w bits that supports deterministic predecessor queries, insertions and deletions in the following worst-case number of I/Os:*

$$O\left(\min\left(\sqrt{\log\left(\tfrac{N}{BM}\right)/\log\log\left(\tfrac{N}{BM}\right)},\ \log_{Bw}\left(\tfrac{N}{M}\right),\ \tfrac{\log w}{\log\log w}\log\log_{Bw}\left(\tfrac{N}{M}\right)\right)\right)$$

where internal space usage is $O(M)$ words and external space usage is $O(N/B)$ blocks of B words.

Using randomization it is also possible to perform all operations in expected $O(\log w)$ time, $O(B)$ words of internal space, and $O(N/B)$ blocks of external space [765]. If main memory size is close to the block size, the upper bounds on predecessor queries are close to optimal, as shown by Beame and Fich [98] in the following general lower bounds:

Theorem 2.14. *Suppose there is a (static) dictionary for w bit keys using $N^{O(1)}$ blocks of memory that supports predecessor queries in t I/Os, worst-case, using $O(B)$ words of internal memory. Then the following bounds hold:*

1. $t = \Omega(\min(\log w/\log\log w,\ \log_{Bw} N))$.
2. If w is a suitable function of N then $t = \Omega(\min(\log_B N, \sqrt{\log N/\log\log N}))$, *i.e., no better bound independent of w can be achieved.*

Exercise 2.15. For what parameters are the upper bounds of Theorem 2.13 within a constant factor of the lower bounds of Theorem 2.14?

Though the above results show that it is possible to improve slightly asymptotically upon the comparison-based upper bounds, the possible savings are so small ($\log_B N$ tends to be a small number already) that it has been common to stick to the comparison-based model. Another reason is that much of the development of external memory algorithms has been driven by computational geometry applications. Geometric problems are usually studied in a model where numbers have infinite precision and can only be manipulated using arithmetic operations and comparisons.

2.3.4 B-tree Variants

There are many variants of B-trees that add or enhance properties of basic B-trees. The weight balance invariant we considered above was introduced in the context of B-trees only recently, making it possible to associate expensive auxiliary data structures with B-tree nodes at small amortized cost. Below we summarize the properties of some other useful B-tree variants and extensions.

Parent Pointers and Level Links. It is simple to extend basic B-trees to maintain a pointer to the parent of each node at no additional cost. A similarly simple extension is to maintain that all nodes at each level are connected in a doubly linked list. One application of these pointers is a *finger search*: Given a leaf v in the B-tree, search for another leaf w. We go up the tree from v until the current node or one of its level-linked neighbors has w below it, and then search down the tree for w. The number of I/Os is $O(\log_B Q)$, where Q is the number of leaves between v and w. When searching for nearby leaves this is a significant improvement over searching for w from the root.

Divide and Merge Operations. In some applications it is useful to be able to divide a B-tree into two parts, with keys smaller than and larger than some splitting element, respectively. Conversely, if we have two B-trees where all keys in one is smaller than all keys in the other, we may wish to efficiently "glue" these trees together into one. In normal B-trees these operations can be supported in $O(\log_B N))$ I/Os. However, it is not easy to simultaneously maintain parent pointers. Level-balanced B-trees [4] maintain parent pointers and support divide, merge, and usual B-tree operations in $O(\log_B N))$ I/Os. If there is no guarantee that keys in one tree are smaller than keys in the other, merging is much harder, as shown in the following problem.

Problem 2.16. Show that, in the lower bound model of Aggarwal and Vitter [17], merging two B-trees with $\Theta(N)$ keys requires $\Theta(N/B)$ I/Os in the worst case.

Partially Persistent B-trees. In *partially persistent B-trees* (sometimes called *multiversion B-trees*) each update conceptually results in a new version of the tree. Queries can be made in any version of the tree, which is useful when the history of the data structure needs to be stored and queried. Persistence is also useful in many geometric algorithms based on the sweepline paradigm (see Chapter 6).

Partially persistent B-trees can be implemented as efficiently as one could hope for, using standard internal memory persistence techniques [258, 661] A sequence of N updates results in a data structure using $O(N/B)$ external memory blocks, where any version of the tree can be queried in $O(\log_B N)$ I/Os. Range queries, etc., are also supported. For details we refer to [99, 661, 749].

String B-trees. We have assumed that the keys stored in a B-tree have fixed length. In some applications this is not the case. Most notably, in *String B-trees* [296] the keys are strings of unbounded length. It turns out that all the usual B-tree operations, as well as a number of operations specific to strings, can be efficiently supported in this setting. String B-trees are presented in Chapter 7.

2.3.5 Batched Dynamic Problems and Buffer Trees

B-trees answer queries in an *on-line* fashion, i.e., the answer to a query is provided immediately after the query is issued. In some applications we can afford to wait for an answer to a query. For example, in *batched dynamic problems* a "batch" of updates and queries is provided to the data structure, and only at the end of the batch is the data structure expected to deliver the answers that would have been returned immediately by the corresponding on-line data structure.

There are many examples of batched dynamic problems in, e.g., computational geometry. As an example, consider the *batched range searching* problem: Given a sequence of insertions and deletions of integers, interleaved with queries for integer intervals, report for each interval the integers contained in it. A data structure for this problem can, using the sweepline technique, be used to solve the *orthogonal line segment intersection* problem: Given a set of horizontal and vertical lines in the plane, report all intersections. We refer to Chapter 6 for details.

Buffer Trees. The *buffer tree* technique [52] has been used for I/O optimal algorithms for a number of problems. In this section we illustrate the basic technique by demonstrating how a buffer tree can be used to handle batched dictionary operations. For simplicity we will assume that the information associated with keys has the same size as the keys.

A buffer tree is similar to a B-tree, but has degree $\Theta(M/B)$. Its name refers to the fact that each internal node has an associated *buffer* which is a queue that contains a sequence of up to M updates and queries to be performed in the subtree where the node is root. New updates and queries are not performed right away, but "lazily" written to the root buffer in $O(1/B)$ I/Os per operation, as described in Section 2.1.1. Non-root buffers reside entirely on external memory, and writing K elements to them requires $O(1 + K/B)$ I/Os.

Whenever a buffer gets full, it is *flushed*: Its content is loaded into internal memory, where the updates and queries are sorted according to the subtree where they have to be performed. These operations are then written to the buffers of the $\Theta(M/B)$ children, in the order they were originally carried out. This may result in buffers of children flushing, and so forth. Leaves contain $\Theta(B)$ keys. When the buffer of a node v just above the leaves is flushed, the updates and queries are performed directly on its M/B children, whose elements fit in main memory. This results in a sorted list of blocks of elements that form the new children of v. If there are too few or too many children, rebalancing operations are performed, similar to the ones described for B-trees (see [52] for details). Each node involved in a rebalancing operation has its buffer flushed before the rebalancing is done. In this way, the content of the buffers need not be considered when splitting, fusing, and sharing.

The cost of flushing a buffer is $O(M/B)$ I/Os for reading the buffer, and $O(M/B)$ I/Os for writing the operations to the buffers of the children. Note that there is a cost of a constant number of I/Os for each child – this is the reason for making the number of children equal to the I/O-cost of reading the buffer. Thus, flushing costs $O(1/B)$ I/Os per operation in the buffer, and since the depth of the tree is $O(\log_{\frac{M}{B}}(\frac{N}{B}))$, the total cost of all flushes is $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os per operation.

The cost of performing a rebalancing operation on a node is $O(M/B)$ I/Os, as we may need to flush the buffer of one of its siblings. However, the number of rebalancing operations during N updates is $O(N/M)$ (see [416]), so the total cost of rebalancing is $O(N/B)$ I/Os.

Problem 2.17. What is the I/O complexity of operations in a "buffer tree" of degree Q?

2.3.6 Priority Queues

The *priority queue* is an abstract data structure of fundamental importance, primarily due to its use in graph algorithms (see Chapter 4 and Chapter 5). A priority queue stores an ordered set of keys, along with associated information (assumed in this section to be of the same size as keys). The basic operations are: insertion of a key, finding the smallest key, and deleting the smallest key. (Since only the smallest key can be inspected, the key can be thought of a *priority*, with small keys being "more important".) Sometimes additional operations are supported, such as deleting an arbitrary key and decreasing the value of a key. The motivation for the decrease-key operation is that it can sometimes be implemented more efficiently than by deleting the old key and inserting the new one.

There are several ways of implementing efficient external memory priority queues. Like for queues and stacks (which are both special cases of priority queues), the technique of *buffering* is the key. We show how to use the buffer tree data structure described in Section 2.3.5 to implement a priority queue using internal memory $O(M)$, supporting insertion, deletion and delete-minimum in $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os, amortized, while keeping the minimum element in internal memory.

The entire buffer of the root node is always kept in internal memory. Also present in memory are the $O(M/B)$ leftmost leaves, more precisely the leaves of the leftmost internal node. The invariant is kept that all buffers on the path from the root to the leftmost leaf are empty. This is done in the obvious fashion: Whenever the root is flushed we also flush all buffers down the leftmost path, at a total cost of $O(\frac{M}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. Since there are $O(M/B)$ operations between each flush of the root buffer, the amortized cost of these extra flushes is $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os per operation. The analysis is completed by the following exercise.

Exercise 2.18. Show that the current minimum can be maintained internally using only the root buffer and the set of $O(M)$ elements in the leftmost leaves. Conclude that find-minimum queries can be answered on-line without using any I/Os.

Optimality. It is not hard to see that the above complexities are, in a certain sense, the best possible.

Exercise 2.19. Show that it is impossible to perform insertion *and* delete-minimums in time $o(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ (*Hint:* Reduce from sorting, and use the sorting lower bound – more information on this reduction technique can be found in Chapter 6).

In internal memory it is in fact possible to improve the complexity of insertion to constant time, while preserving $O(\log N)$ time for delete-minimum (see [216, Chapter 20] and [154]). It appears to be an open problem whether it is possible to implement constant time insertions in external memory.

One way of improving the performance the priority queue described is to provide "worst case" rather than amortized I/O bounds. Of course, it is not possible for every operation to have a cost of less than one I/O. The best one can hope for is that any subsequence of k operations uses $O(1 + \frac{k}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. Brodal and Katajainen [157] have achieved this for subsequences of length $k \geq B$. Their data structure does not support deletions.

A main open problem in external memory priority queues is the complexity of the decrease-key operation (when the other operations have complexity as above). Internally, this operation can be supported in constant time (see [216, Chapter 20] and [154]), and the open problem is whether a corresponding bound of $O(1/B)$ I/Os per decrease-key can be achieved. The currently best complexity is achieved by "tournament trees", described in Chapter 4, where decrease-key operations, as well as the other priority queue operations, cost $O(\frac{1}{B}\log(\frac{N}{B}))$ I/Os.

2.4 Hashing Based Dictionaries

We now consider hashing techniques, which offer the highest performance for the basic dictionary operations. One aspect that we will not discuss here, is how to implement appropriate classes of hash functions. We will simply assume to have access to hash functions that behave like truly random functions, independent of the sequence of dictionary operations. This means that any hash function value $h(x)$ is uniformly random and independent of hash function values on elements other than x. In practice, using easily implementable "pseudorandom" hash functions that try to imitate truly random functions, the behavior of hashing algorithms is quite close to that of this idealized model. We refer the reader to [251] and the references therein for more information on practical hash functions.

2.4.1 Lookup with Good Expected Performance

Several classic hashing schemes (see [460, Section 6.4] for a survey) perform well in the expected sense in external memory. We will consider *linear probing* and *chaining with separate lists*. These schemes need nothing but a single hash function h in internal memory (in practice a few machine words suffice for a good pseudorandom hash function). For both schemes the analysis is beyond the scope of this chapter, but we provide some intuition and state results on their performance.

Linear Probing. In external memory linear probing, a search for the key x starts at block $h(x)$ in a hash table, and proceeds linearly through the table until either x is found or we encounter a block that is not full (indicating that x is not present in the table). Insertions proceed in the same manner as lookups, except that we insert x if we encounter a non-full block. Deletion of a key x requires some rearrangement of the keys in the blocks scanned when looking up x, see [460, Section 6.4] for details. A deletion leaves the table in the state it would have been in if the deleted element had never been inserted.

The intuitive reason that linear probing gives good average behavior is that the pseudorandom function distributes the keys almost evenly to the blocks. In the rare event that a block overflows, it will be unlikely that the next block is not able to accommodate the overflow elements. More precisely, if the load factor of our hash table is α, where $0 < \alpha < 1$ (i.e., the size of the hash table is $N/(\alpha B)$ blocks), we have that the expected average number of I/Os for a lookup is $1 + (1 - \alpha)^{-2} \cdot 2^{-\Omega(B)}$ [460]. If α is bounded away from 1 (i.e., $\alpha \leq 1 - \epsilon$ for some constant $\epsilon > 0$) and if B is not too small, the expected average is very close to 1. In fact, the asymptotic probability of having to use $k > 1$ I/Os for a lookup is $2^{-\Omega(B(k-1))}$. In Section 2.4.4 we will consider the problem of keeping the load factor in a certain range, shrinking and expanding the hash table according to the size of the set.

Chaining with Separate Lists. In chaining with separate lists we again hash to a table of size approximately $N/(\alpha B)$ to achieve load factor α. Each block in the hash table is the start of a linked list of keys hashing to that block. Insertion, deletion, and lookups proceed in the obvious manner. As the pseudorandom function distributes keys approximately evenly to the blocks, almost all lists will consist of just a single block. In fact, the probability that more than kB keys hash to a certain block, for $k \geq 1$, is at most $e^{-\alpha B(k/\alpha-1)^2/3}$ by Chernoff bounds (see, e.g., [375, Eq. 6]).

As can be seen, the probabilities decrease faster with k than in linear probing. On the other hand, chaining may be slightly more complicated to implement as one has to manage $2^{-\Omega(B)}n$ blocks in chained lists (expected). Of course, if B is large and the load factor is not too high, overflows will be very rare. This can be exploited, as discussed in the next section.

2.4.2 Lookup Using One External Memory Access

In the previous section we looked at hashing schemes with good *expected* lookup behavior. Of course, an expected bound may not be good enough for some applications where a firm guarantee on throughput is needed. In this and the following section we investigate how added resources may provide dictionaries in which lookups take just the time of a single I/O in the worst case. In particular, we consider dictionaries using more internal memory, and dictionaries using external memory that allows two I/Os to be performed in parallel.

Making Use of Internal Memory. An important design principle in external memory algorithms is to make full use of internal memory for data structures that reduce the number of external memory accesses. Typically such an internal data structure holds part of the external data structure that will be needed in the future (e.g., the buffers used in Section 2.1), or it holds information that allows the proper data to be found efficiently in external memory.

If sufficient internal memory is available, searching in a dictionary can be done in a single I/O. There are at least two approaches to achieving this.

Overflow area.. When internal memory for $2^{-\Omega(B)}N$ keys and associated information is available internally, there is a very simple strategy that provides lookups in a single I/O, for constant load factor $\alpha < 1$. The idea is to store the keys that cannot be accommodated externally (because of block overflows) in an internal memory dictionary. For some constant $c(\alpha) = \Omega(1-\alpha)$ the probability that there are more than $2^{-c(\alpha)B}N$ such keys is so small (by the Chernoff bound) that we can afford to rehash, i.e., choose a new hash function to replace h, if this should happen.

Alternatively, the overflow area can reside in external memory (this idea appeared in other forms in [341, 623]). To guarantee single I/O lookups this requires internal memory data structures that:

– Identify blocks that have overflown.
– Facilitate single I/O lookup of the elements hashing to these blocks.

The first task can be solved by maintaining a dictionary of overflowing blocks. The probability of a block overflowing is $O(2^{-c(\alpha)B})$, so we expect to store the indices of $O(2^{-c(\alpha)B}N)$ blocks. This requires $O(2^{-c(\alpha)B}N\log N)$ bits of internal space. If we simply discard the external memory blocks that have overflown, the second task can be solved recursively by a dictionary supporting single I/O lookups, storing a set that with high probability has size $O(2^{-c(\alpha)B}N)$.

Perfect hashing.. Mairson [525] considered implementing a *B-perfect hash function* $p : K \to \{1, \dots, \lceil N/B \rceil\}$ that maps at most B keys to each block. Note that if we store key k in block $p(k)$ and the B-perfect hash function resides in internal memory, we need only a single I/O to look up k. Mairson showed that such a function can be implemented using $O(N \log(B)/B)$ bits of internal memory. (In the interest of simplicity, we ignore an extra term that only shows up when the key set K has size $2^{B^{\omega(N)}}$.) If the number of external blocks is only $\lceil N/B \rceil$ and we want to be able to handle every possible key set, this is also the best possible [525]. Unfortunately, the time and space needed to evaluate Mairson's hash functions is extremely high, and it seems very difficult to obtain a dynamic version. The rest of this section deals with more practical ways of implementing (dynamic) B-perfect hashing.

Extendible Hashing. A popular B-perfect hashing method that comes close to Mairson's bound is *extendible hashing* by Fagin et al. [285]. The expected space utilization in external memory is about 69% rather than the 100% achieved by Mairson's scheme.

Extendible hashing employs an internal structure called a *directory* to determine which external block to search. The directory is an array of 2^d pointers to external memory blocks, for some parameter d. Let $h : K \to \{0,1\}^r$ be a truly random hash function, where $r \geq d$. Lookup of a key k is performed by using $h(k)_d$, the function returning the d least significant bits of $h(k)$, to determine an entry in the directory, which in turn specifies the external block to be searched. The parameter d is chosen to be the smallest number for which at most B dictionary keys map to the same value under $h(k)_d$. If $r \geq 3 \log N$, say, such a d exists with high probability. In case it does not we simply rehash. Many pointers in the directory may point to the same block. Specifically, if no more than B dictionary keys map to the same value v under $h_{d'}$, for some $d' < d$, all directory entries with indices having v in their d' least significant bits point to the same external memory block.

Clearly, extendible hashing provides lookups using a single I/O and constant internal processing time. Analyzing its space usage is beyond the scope of this chapter, but we mention some results. Flajolet [305] has shown that the expected number of entries in the directory is approximately $4\frac{N}{B}\sqrt[B]{N}$. If B is just moderately large, this is close to optimal, e.g., in case $B \geq \log N$ the number of bits used is less than $8N \log(N)/B$. In comparison, the optimal space bound for perfect hashing to exactly N/B external memory blocks is $\frac{1}{2}N \log(B)/B + \Theta(N/B)$ bits. The expected external space usage can be shown to be around $N/(B \ln 2)$ blocks, which means that about 69% of the space is utilized [285, 545].

Extendible hashing is named after the way in which it adapts to changes of the key set. The *level* of a block is the largest $d' \leq d$ for which all its keys map to the same value under $h_{d'}$. Whenever a block at level d' has run full, it is split into two blocks at level $d' + 1$ using $h_{d'+1}$. In case $d' = d$ we first need to double the size of the directory. Conversely, if two blocks at level d', with keys having the same function value under $h_{d'-1}$, contain less than B keys in total, these blocks are merged. If no blocks are left at level d, the size of the directory is halved.

Using a Predecessor Dictionary. If one is willing to increase internal computation from a constant to expected $O(\log\log N)$ time per dictionary operation, both internal and external space usage can be made better than that of extendible hashing. The idea is to replace the directory with a dictionary supporting *predecessor queries* in a key set $P \subseteq \{0,1\}^r$: For any $x \in \{0,1\}^r$ it reports the largest key $y \in P$ such that $y \leq x$, along with some information associated with this key. In our application the set P will be the hash values of a small subset of the set of keys in the dictionary.

We will keep the keys of the dictionary stored in a linked list, sorted according to their hash values (interpreted as nonnegative integers). For each block in the linked list we keep the smallest hash value in the predecessor dictionary, and associate with it a pointer to the block. This means that lookup of $x \in K$ can be done by searching the block pointed to by the predecessor of $h(x)$. Insertions and deletions can be done by inserting or deleting the key in the linked list, and possibly making a constant number of updates to the predecessor dictionary.

We saw in Problem 2.6 that a linked list with space utilization $1-\epsilon$ can be maintained in $O(1)$ I/O per update, for any constant $\epsilon > 0$. The internal predecessor data structure then contains at most $\lceil N/((1-\epsilon)B)\rceil$ keys. We choose the range of the hash function such that $3\log N \leq r = O(\log N)$. Since the hash function values are only $O(\log N)$ bits long, one can implement a very efficient linear space predecessor dictionary based on van Emde Boas trees [745, 746]. This data structure [765] allows predecessor queries to be answered in time $O(\log\log N)$, and updates to be made in expected $O(\log\log N)$ time. The space usage is linear in the number of elements stored.

In conclusion, we get a dictionary supporting updates in $O(1)$ I/Os and time $O(\log\log N)$, expected, and lookups in 1 I/O and time $O(\log\log N)$. For most practical purposes the internal processing time is negligible. The external space usage can be made arbitrarily close to optimal, and the internal space usage is $O(N/B)$.

2.4.3 Lookup Using Two Parallel External Memory Accesses

We now consider a scenario in which we may perform two I/Os in parallel, in two separate parts of external memory. This is realistic, for example, if two disks are available or when RAM is divided into independent banks. It turns out that, with high probability, all dictionary operations can be performed accessing just a single block in each part of memory, assuming that the load factor α is bounded away from 1 and that blocks are not too small.

The hashing scheme achieving this is called two-way chaining, and was introduced by Azar et al. [77]. It can be thought of as two chained hashing data structures with pseudorandom hash functions h_1 and h_2. Key x may reside in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two. New keys are always inserted in the block having the smallest number of keys, with ties broken such that keys go to table one (the advantages of this tie-breaking rule were discovered by Vöcking [756]). It can be shown that the probability of an insertion causing an overflow is $N/2^{2^{\Omega((1-\alpha)B)}}$ [115]. That is, the failure probability decreases *doubly exponentially* with the average number of free spaces in each block. The constant factor in the Ω is larger than 1, and it has been shown experimentally that even for very small amounts of free space in each block, the probability of an overflow (causing a rehash) is very small [159]. The effect of deletions in two-way chaining does not appear to have been analyzed.

2.4.4 Resizing Hash Tables

In the above we several times assumed that the load factor of our hash table is at most some constant $\alpha < 1$. Of course, to keep the load factor below α we may have to increase the size of the hash table employed when the size of the set increases. On the other hand we wish to keep α above a certain threshold to have a good external memory utilization, so shrinking the hash table is also occasionally necessary. The challenge is to rehash to the new table without having to do an expensive reorganization of the old hash table. Simply choosing a new hash function would require a random permutation of the keys, a task shown in [17] to require $\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. When $N = (M/B)^{O(B)}$, i.e, when N is not extremely large, this is $O(N)$ I/Os. Since one usually has $\Theta(N)$ updates between two rehashes, the reorganization cost can be amortized over the cost of updates. However, more efficient ways of reorganizing the hash table are important in practice to keep constant factors down. The basic idea is to introduce more "gentle" ways of changing the hash function.

Linear Hashing. Litwin [508] proposed a way of gradually increasing and decreasing the range of hash functions with the size of the set. The basic idea for hashing to a range of size r is to extract $b = \lceil \log r \rceil$ bits from a "mother" hash function. If the extracted bits encode an integer k less than r, this is used as the hash value. Otherwise the hash function value $k - 2^{b-1}$ is returned. When expanding the size of the hash table by one block (increasing r by one), all keys that may hash to the new block $r+1$ previously hashed to block $r+1-2^{b-1}$. This makes it easy to update the hash table. Decreasing the size of the hash table is done in a symmetric manner.

The main problem with linear hashing is that when r is not a power of 2, the keys are not mapped uniformly to the range. For example, if r is 1.5 times a power of two, the expected number of collisions between keys is 12.5% higher than that expected for a uniform hash function. Even worse, the expected maximum number of keys hashing to a single bucket can be up to twice as high as in the uniform case. Some attempts have been made to alleviate these problems, but all have the property that the hash functions used are not completely uniform, see [497] and the references therein. Another problem lies in the analysis, which for many hashing schemes is complicated by nonuniform hash functions. Below we look at a way of doing efficient rehashing in a uniform way.

Uniform Rehashing. We now describe an alternative to linear hashing that yields uniform hash functions [597]. To achieve both uniformity and efficient rehashing we do not allow the hash table size to increase/decrease in increments of 1, but rather support that its size is increased/decreased by a *factor* of around $1 + \epsilon$ for some $\epsilon > 0$. This means that we are not able to control exactly the relative sizes of the set and hash table. On the other hand, uniformity means that we will be able to achieve the performance of linear hashing using a smaller hash table.

As in linear hashing we extract the hash function value for all ranges from a "mother" hash function $\phi : U \to \{0, \ldots, R-1\}$. The factor between consecutive hash table sizes will be between $1+\epsilon_1$ and $1+\epsilon_2$, where $\epsilon_2 > \epsilon_1 > 0$ are arbitrary constants. The size R of the range of ϕ is chosen as follows. Take a sequence of positive integers $i_1, \ldots, i_k$ such that $i_k = 2^p \cdot i_1$ for some positive integer p, and $1 + \epsilon_1 < i_{j+1}/i_j < 1 + \epsilon_2$ for $j = 1, \ldots, k-1$.

Exercise 2.20. Show that $i_1, \ldots, i_k$ can be chosen to satisfy the above requirements, and such that $I = \prod_{j=1}^{k} i_j$ is a constant (depending only on ϵ_1 and ϵ_2).

We let $R = 2^b \cdot I$, where I is defined in Exercise 2.20 and b is chosen such that no hash function with range larger than $2^b i_k$ will be needed. Whenever r divides R we have the uniformly random hash function with range of size r: $h_r = \phi(r)$ div (R/r), where div denotes integer division. The possible range sizes are $2^q i_j$ for $q = 0, \ldots, b$, $j = 1, \ldots, k$. If the current range size is $r = 2^q i_j$ and we wish to hash to a larger table, we choose new range $r' = 2^q i_{j+1}$ if $j < k$ and $r' = 2^{q+p} i_2$ if $j = k$. By the way we have chosen $i_1, \ldots, i_k$ it holds that $1 + \epsilon_1 < r'/r < 1 + \epsilon_2$. The case where we wish to hash to a smaller table is symmetric.

The following property of our hash functions means that, for many hashing schemes, rehashing can be performed by a single scan through the hash table (i.e., in $O(N/B)$ I/Os): If $\phi(x) \leq \phi(y)$ then $h_r(x) \leq h_r(y)$ for any r. In other words, our hash functions never change the ordering of hash values given by ϕ.

2.5 Dynamization Techniques

This section presents two general techniques for obtaining dynamic data structures for sets.

2.5.1 The Logarithmic Method

In many cases it is considerably simpler to come up with an efficient way of constructing a *static* data structure than achieving a correspondingly efficient dynamic data structure. The *logarithmic method* is a technique for obtaining data structures with efficient (though often not optimal) insertion and query operations in some of these cases. More specifically, the problem must be *decomposable*: If we split the set S of elements into disjoint subsets $S_1, \dots, S_k$ and create a (static) data structure for each of them, then queries on the whole set can be answered by querying each of these data structures. Examples of decomposable problems are dictionaries and priority queues.

The basic idea of the logarithmic method is to maintain a collection of data structures of different sizes, and periodically merge a number data structures into one, in order to keep the number of data structures to be queried low. In internal memory, the number of data structures for a set of size N is typically $O(\log N)$, explaining the name of the method. We refer to [109, 110, 600] and the references therein for more background.

In the external memory version of the logarithmic method that we describe [69], the number of data structures used is decreased to $O(\log_B N)$. Insertions are done by rebuilding the first static data structure such that it contains the new element. The invariant is that the ith data structure should have size no more than B^i. If this size is reached, it is merged with the $i+1$st data structure (which may be empty). Merging is done by rebuilding a static data structure containing all the elements of the two data structures.

Exercise 2.21. Show that when inserting N elements, each element will be part of a rebuilding $O(B \log_B N)$ times.

Suppose that building a static data structure for N elements uses $O(\frac{N}{B} \log_B^k N)$ I/Os. Then by the exercise, the total amortized cost of inserting an element is $O(\log_B^{k+1} N)$ I/Os. Queries take $O(\log_B N)$ times more I/Os than queries in the corresponding static data structures.

2.5.2 Global Rebuilding

Some data structures for sets support deletions, but do not recover the space occupied by deleted elements. For example, deletions in a static dictionary can be done by *marking* deleted elements (this is called a *weak delete*). A general technique for keeping the number of deleted elements at some fraction of the total number of elements is *global rebuilding*: In a data structure of N elements (present and deleted), whenever αN elements have been deleted, for some constant $\alpha > 0$, the entire data structure is rebuilt. The cost of rebuilding is at most a constant factor higher than the cost of inserting αN elements, so the amortized cost of global rebuilding can be charged to the insertions of the deleted elements.

Exercise 2.22. Discuss pros and cons of using global rebuilding for B-trees instead of the deletion method described in Section 2.3.2.

2.6 Summary

This chapter has surveyed some of the most important external memory data structures for sets and lists: Elementary abstract data structures (queues, stacks, linked lists), B-trees, buffer trees (including their use for priority queues), and hashing based dictionaries. Along the way, several important design principles for memory hierarchy aware algorithms and data structures have been touched upon: Using buffers, blocking and locality, making use of internal memory, output sensitivity, data structures for batched dynamic problems, the logarithmic method, and global rebuilding. In the following chapters of this volume, the reader who wants to know more can find a wealth of information on virtually all aspects of algorithms and data structures for memory hierarchies.

Since the data structure problems discussed in this chapter are fundamental they are well-studied. Some problems have resisted the efforts of achieving external memory results "equally good" as the corresponding internal memory results. In particular, the problems of supporting fast insertion and decrease-key in priority queues (or show that this is not possible) have remained challenging open research problems.

Acknowledgements. The surveys by Arge [55], Enbody and Du [280], and Vitter [753, 754] were a big help in writing this chapter. I would also like to acknowledge the help of Gerth Stølting Brodal, Ulrich Meyer, Anna Östlin, Jan Vahrenhold, Berthold Vöcking, and last but not least the participants of the GI-Dagstuhl-Forschungsseminar "Algorithms for Memory Hierarchies".

3. A Survey of Techniques for Designing I/O-Efficient Algorithms*

Anil Maheshwari and Norbert Zeh

3.1 Introduction

This survey is meant to give an introduction to elementary techniques used for designing I/O-efficient algorithms. We do not intend to give a complete survey of all state-of-the-art techniques; but rather we aim to provide the reader with a good understanding of the most elementary techniques. Our focus is on general techniques and on techniques used in the design of I/O-efficient graph algorithms. We include the latter because many abstract data structuring problems can be translated into classical graph problems. While this fact is of mostly philosophical interest in general, it gains importance in I/O-efficient algorithms because random access is penalized in external memory algorithms and standard techniques to extract information from graphs can help when trying to extract information from pointer-based data structures.

For the analysis of the I/O-complexity of the algorithms, we adopt the *Parallel Disk Model* (PDM) (see Chapter 1) as the model of computation. We restrict our discussion to the single-disk case ($D = 1$) and refer the reader to appropriate references for the case of multiple disks. In order to improve the readability of the text, we do not worry too much about the integrality of parameters that arise in the discussion. That is, we write x/y to denote $\lfloor x/y \rfloor$ or $\lceil x/y \rceil$, as appropriate. The same applies to expressions such as $\log x$, $\sqrt{x}$, etc.

We begin our discussion in Section 3.2 with an introduction to two general techniques that are applied in virtually all I/O-efficient algorithms: sorting and scanning. In Section 3.3 we describe a general technique to derive I/O-efficient algorithms from efficient parallel algorithms. Using this technique, the huge repository of efficient parallel algorithms can be exploited to obtain I/O-efficient algorithms for a wide range of problems. Sections 3.4 through 3.7 are dedicated to the discussion of techniques used in I/O-efficient algorithms for fundamental graph problems. The choice of the graph problems we consider is based on the importance of these problems as tools for solving other problems that are not of a graph-theoretic nature.

* Research supported by Natural Sciences and Engineering Research Council of Canada. Part of this work was done while the second author was a Ph.D. student at the School of Computer Science of Carleton University.

3.2 Basic Techniques

3.2.1 Scanning

Scanning is the simplest of all paradigms applied in I/O-efficient algorithms. The idea is that reading and writing data in sequential order is less expensive than accessing data at random. In particular, N data items, when read in sequential order, can be accessed in $\mathcal{O}(N/B)$ I/Os, while accessing N data items at random costs $\Omega(N)$ I/Os in the worst case. We illustrate this paradigm using a simple example and then derive the general, formal definition of the paradigm from the discussion of the example.

The example we consider is that of computing the prefix sums of the elements stored in an array A and storing them in an array A'. The straightforward algorithm for solving this problem accesses the elements of A one by one in their order of appearance and adds them to a running sum s, which is initially set to 0. After reading each element and adding it to s, the current value of s is appended to array A'.

In internal memory, this simple algorithm takes linear time. It scans array A, i.e., reads the elements of A in their order of appearance and writes the elements of array A' in sequential order. Whenever data is accessed in sequential order like this, we speak of an application of the *scanning paradigm*. Looking at it from a different angle, we can consider array A as an input stream whose elements are processed by the algorithm as they arrive, and array A' is an output stream produced by the algorithm.

In external memory, the algorithm takes $\mathcal{O}(N/B)$ I/Os after making the following simple modifications: At the beginning of the algorithm, instead of accessing only the first element of A, the first B elements are read into an input buffer associated with input stream A. This transfer of the first B elements from disk into main memory takes a single I/O. After that, instead of reading the next element to be processed directly from input stream A, they are read from the input buffer, which does not incur any I/Os. As soon as all elements in the input buffer have been processed, the next B elements are read from the input stream, which takes another I/O, and then the elements to be processed are again retrieved from the input buffer. Applying this strategy until all elements of A are processed, the algorithm performs N/B I/O-operations to read its input elements from input stream A. The writing of the output can be "blocked" in a similar fashion: That is, we associate an output buffer of size B with output stream A'. Instead of writing the elements of A' directly to disk as they are produced, we append these elements to the output buffer until the buffer is full. When this happens, we write the content of the buffer to disk, which takes one I/O. Then there is room in the buffer for the next B elements to be appended to A'. Repeating this process until all elements of A' have been written to disk, the algorithm performs N/B I/Os to write A'. Thus, in total the computation of

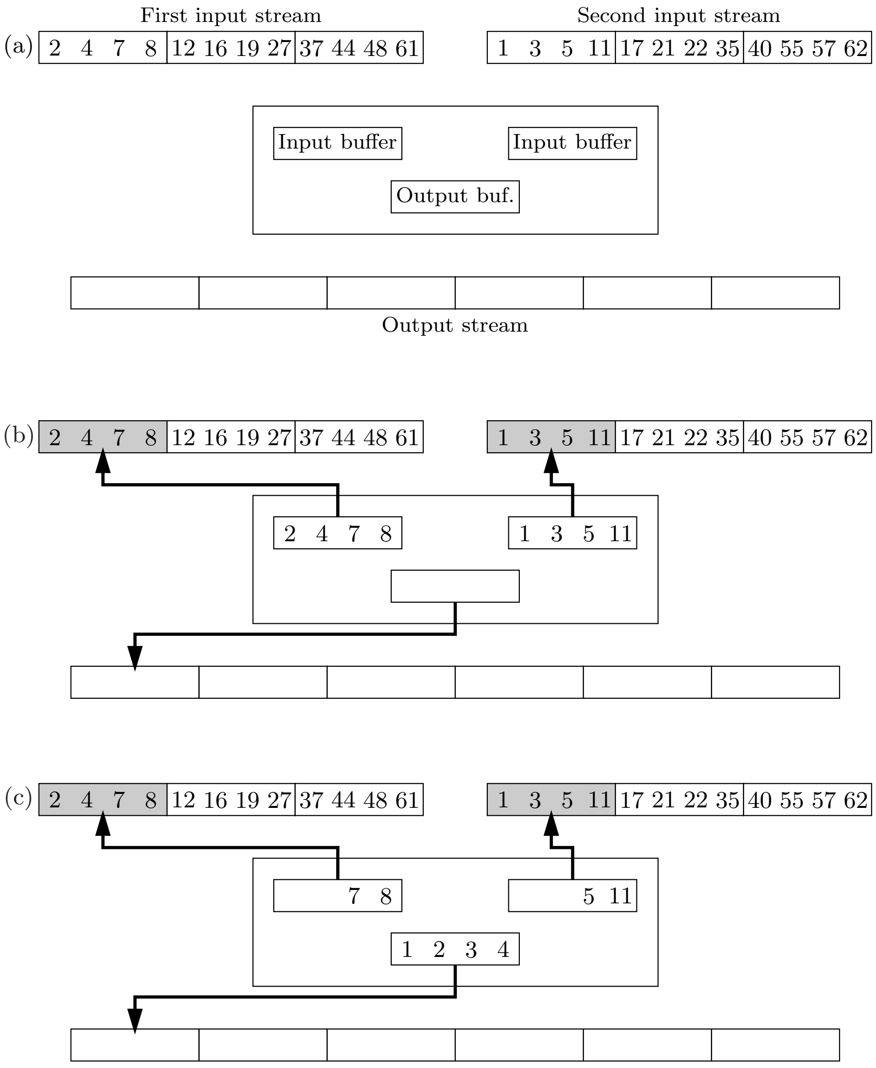
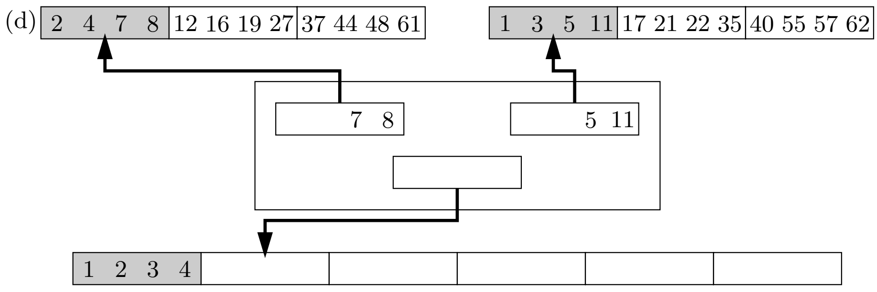


Fig. 3.1. Merging two sorted sequences. (a) The initial situation: The two lists are stored on disk. Two empty input buffers and an empty output buffer have been allocated in main memory. The output sequence does not contain any data yet. (b) The first block from each input sequence has been loaded into main memory. (c) The first B elements have been moved from the input buffers to the output buffer.



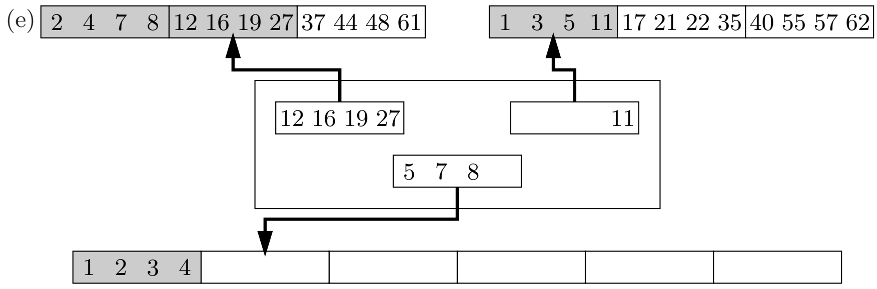


Fig. 3.1. (continued) (d) The contents of the output buffer are flushed to the output stream to make room for more data to be moved to the output buffer. (e) After moving elements 5, 7, and 8 to the output buffer, the input buffer for the first stream does not contain any more data items. Hence, the next block is read from the first input stream into the input buffer.

array A' from array A takes $\mathcal{O}(N/B)$ I/Os rather than $\Theta(N)$ I/Os, as would be required to solve this task using direct disk accesses.

In our example we apply the scanning paradigm to a problem with one input stream A and one output stream A'. It is easy to apply the above buffering technique to a problem with q input streams $S_1, \dots, S_q$ and r output streams $S'_1, \dots, S'_r$, as long as there is enough room to keep an input buffer of size B per input stream S_i and an output buffer of size B per output stream S'_j in internal memory. More precisely, $p + q$ cannot be more than M/B. Under this assumption the algorithm still takes $\mathcal{O}(N/B)$ I/Os, where $N = \sum_{i=1}^{q} |S_i| + \sum_{j=1}^{r} |S'_j|$. Note, however, that this analysis includes only the number of I/Os required to read the elements from the input streams and write the output to the output streams. It does not include the I/O-complexity of the actual computation of the output elements from the input elements. One way to guarantee that the I/O-complexity of the whole algorithm, including all computation, is $\mathcal{O}(N/B)$ is to ensure that only the $M-(q+r)B$ elements most recently read from the input streams are required for the computation of the next output element, or the required information about all elements read from the input streams can be maintained succinctly in $M - (q + r)B$ space. If this can be guaranteed, the computation of all output elements from the read input elements can be carried out in main memory and thus does not cause any I/O-operations to be performed.

An important example where the scanning paradigm is applied to more than one input stream is the merging of k sorted streams to produce a single sorted output stream (see Fig. 3.1). This procedure is applied repeatedly with a parameter of $k = 2$ in the classical internal memory MergeSort algorithm. The I/O-efficient MergeSort algorithm discussed in the next section takes advantage of the fact that up to $k = M/B$ streams can be merged in a linear number of I/Os, in order to decrease the number of recursive merge steps required to produce a single sorted output stream.

3.2.2 Sorting

Sorting is a fundamental problem that arises in almost all areas of computer science, including large scale applications such as database systems. Besides the obvious applications where the task at hand requires per definition that the output be produced in sorted order, sorting is often applied as a paradigm for eliminating random disk accesses in external memory algorithms. Consider for instance a graph algorithm that performs a traversal of the input graph to compute a labelling of its vertices. This task does not require any part of the representation of the graph to be sorted, so that the internal memory algorithm does not necessarily include a sorting step. An external memory algorithm for the same problem, on the other hand, can benefit greatly by sorting the vertex set of the graph appropriately. In particular, without sorting the vertex set, the algorithm has no control over the order in which the vertices are stored on disk. Hence, in the worst case the algorithm spends one I/O to load each vertex into internal memory when it is visited. If the order in which the vertices of the graph are visited can be determined efficiently, it is more efficient to sort the vertices in this order and then perform the traversal of the graph using a simple scan of the sorted vertex set.

The number of I/O-efficient sorting algorithms that have been proposed in the literature is too big for us to give an exhaustive survey of these algorithms at the level of detail appropriate for a tutorial as this one. Hence, we restrict our discussion to a short description of the two main paradigms applied in I/O-efficient sorting algorithms and then present the simplest I/O-optimal sorting algorithm in detail. At the end of this section we discuss a number of issues that need to be addressed in order to obtain I/O-optimal algorithms for sorting on multiple disks.

Besides algorithms that delegate the actual work required to sort the given set of data elements to I/O-efficient data structures, the existing sorting algorithms can be divided into two categories based on the basic approach taken to produce the sorted output.

Algorithms based on the *merging paradigm* proceed in two phases. In the first phase, the *run formation phase*, the input data is partitioned into more or less trivial sorted sequences, called "runs". In the second phase, the *merging phase*, these runs are merged until only one sorted run remains, where merging k runs $S_1, \dots, S_k$ means that a single sorted run S' is produced that contains all elements of runs $S_1, \dots, S_k$. The classical internal memory MergeSort algorithm is probably the simplest representative of this class of algorithms. The run formation phase is trivial in this case, as it simply declares every input element to be in a run of its own. The merging phase uses two-way merging to produce longer runs from shorter ones. That is, given the current set of runs $S_1, \dots, S_k$, these runs are grouped into pairs of runs and each pair is merged to form a longer run. Each such iteration of merging pairs of runs can be carried out in linear time. The number of runs reduces by a factor of two from one iteration to the next, so that $\mathcal{O}(\log N)$ iterations suffice to produce a single sorted run containing all data elements. Hence, the algorithm takes $\mathcal{O}(N \log N)$ time.

Algorithms based on the *distribution paradigm* compute a partition of the given data set S into subsets $S_0, \dots, S_k$ so that for all $0 \le i < j \le k$ and any two elements $x \in S_i$ and $y \in S_j$, $x \le y$. Given this partition, a sequence containing the elements of S in sorted order is produced by sorting each of the sets $S_0, \dots, S_k$ recursively and concatenating the resulting sorted sequences. In order to produce sets $S_0, \dots, S_k$, the algorithm chooses a set of *splitters* $x_1 \le \cdots \le x_k$ from S. To simplify the discussion, we also assume that there are two splitters $x_0 = -\infty$ and $x_{k+1} = +\infty$. Then set S_i, $0 \le i \le k$, is defined as the set of elements $y \in S$ so that $x_i \le y < x_{i+1}$. Given the set of splitters, sets $S_0, \dots, S_k$ are produced by comparing each element in S to the splitters. The efficiency of this procedure depends on a good choice of the splitter elements. If it can be guaranteed that each of the sets $S_0, \dots, S_k$ has size $\mathcal{O}(|S|/k)$, the procedure finishes in $\mathcal{O}(N \log_2 N)$ time because $\mathcal{O}(\log_k N)$ levels of recursion suffice to produce a partition of the input into N singleton sets that are arranged in the right order, and each level of recursion takes $\mathcal{O}(N \log k)$ time to compare each element in S to the splitters. QuickSort is a representative of this class of algorithms.

In internal memory MergeSort has the appealing property of being simpler than QuickSort because the run formation phase is trivial and the merging phase is a simple iterative process that requires only little bookkeeping. In contrast, QuickSort faces the problem of computing a good splitter, which is easy using randomization, but requires some effort if done deterministically. In external memory the situation is not much different, as long as the goal is to sort optimally on a single disk. If I/O-optimal performance is to be achieved on multiple disks, distribution-based algorithms are preferable because it is somewhat easier to make them take full advantage of the parallel nature of multi-disk systems. Since we do not go into too much detail discussing the issues involved in optimal sorting on multiple disks, we choose an I/O-efficient version of MergeSort as the sorting algorithm we discuss in detail. The algorithm is simple, resembles very much the internal memory algorithm, and achieves optimal performance on a single disk.

In order to see what needs to be done to obtain a MergeSort algorithm that performs $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os, let us analyze the I/O-complexity of the internal memory MergeSort algorithm as it is. The run formation phase does not require any I/Os. Merging two sorted runs S_1 and S_2 takes $\mathcal{O}(1 + (|S_1| + |S_2|)/B)$ I/Os using the scanning paradigm: Read the first element from each of the two streams. Let x and y be the two read elements. If $x < y$, place x into the output stream, read the next element from S_1, and repeat the whole procedure. If $x \geq y$, place y into the output stream, read the next element from S_2, and repeat the whole procedure. Since every input element is involved in $\mathcal{O}(\log N)$ merge steps, and the total number of merge steps is $\mathcal{O}(N)$, the I/O-complexity of the internal memory MergeSort algorithm is $\mathcal{O}(N + (N/B)\log_2 N)$.

The I/O-complexity of the algorithm can be reduced to $\mathcal{O}((N/B)\cdot \log_2(N/B))$ by investing $\mathcal{O}(N/B)$ I/Os during the run formation phase. In particular, the run formation phase makes sure that the merge phase starts out with N/M sorted runs of length M instead of N singleton runs. To achieve this goal, the data is partitioned into N/M chunks of size M. Then each chunk is loaded into main memory, sorted internally, and written back to disk in sorted order. Reading and writing each chunk takes $\mathcal{O}(M/B)$ I/Os, so that the total I/O-complexity of the run formation phase is $\mathcal{O}(N/M \cdot M/B) = \mathcal{O}(N/B)$. As a result of reducing the number of runs to N/M, the merge phase now takes $\mathcal{O}(N/M + (N/B)\log_2(N/M)) = \mathcal{O}((N/B)\log_2(N/B))$ I/Os.

In order to increase the base of the logarithm to M/B, it has to be ensured that the number of runs reduces by a factor of $\Omega(M/B)$ from one iteration of the merge phase to the next because then $\mathcal{O}(\log_{M/B}(N/B))$ iterations suffice to produce a single sorted run. In order to achieve this goal, the obvious thing to do is to merge $k = M/(2B)$ runs $S_1, \dots, S_k$ instead of only two runs in a single merge step. Similar to the internal memory merge step, the algorithm loads the first elements $x_1, \dots, x_k$ from runs $S_1, \dots, S_k$ into main memory, copies the smallest of them, x_i, to the output run, reads the next element from S_i and repeats the whole procedure. The I/O-complexity of this modified merge step is $\mathcal{O}(k + (\sum_{i=1}^{k} |S_i|)/B)$ because the available amount of main memory suffices to allocate a buffer of size B for each input and output run, so that the scanning paradigm can be applied.

Using this modified merge step, the number of runs reduces by a factor of $M/(2B)$ from one iteration of the merge phase to the next. Hence, the algorithm produces a single sorted run already after $\mathcal{O}(\log_{M/B}(N/B))$ iterations, and the I/O-complexity of the algorithm becomes $\mathcal{O}((N/B)\log_{M/B}(N/B))$, as desired.

An issue that does not affect the I/O-complexity of the algorithm, but is important to obtain a fast algorithm, is how the merging of k runs instead of two runs is done in internal memory. When merging two runs, choosing the next element to be moved to the output run involves a single comparison. When merging $k > 2$ runs, it becomes computationally too expensive to find the minimum of elements $x_1, \ldots, x_k$ in $\mathcal{O}(k)$ time because then the running time of the merge phase would be $\mathcal{O}(kN \log_k(N/B))$. In order to achieve optimal running time in internal memory as well, the minimum of elements $x_1, \ldots, x_k$ has to be found in $\mathcal{O}(\log k)$ time. This can be achieved by maintaining the smallest elements $x_1, \ldots, x_k$, one from each run, in a priority queue. The next element to be moved to the output run is the smallest in the priority queue and can hence be retrieved using a DeleteMin operation. Let the retrieved element be $x_i \in S_i$. Then after moving x_i to the output run, the next element is read from S_i and inserted into the priority queue, which guarantees that again the smallest unprocessed element from every run is stored in the priority queue. This process is repeated until all elements have been moved to the output run. The amount of space used by the priority queue is $\mathcal{O}(k) = \mathcal{O}(M/B)$, so that the priorty queue can be maintained in main memory. Moving one element to the output run involves the execution of one DeleteMin and one Insert operation on the priority queue, which takes $\mathcal{O}(\log k)$ time. Hence, the total running time of the MergeSort algorithm is $\mathcal{O}(N \log M + (N \log k) \log_k(N/B)) = \mathcal{O}(N \log N)$. We summarize the discussion in the following theorem.

Theorem 3.1. *[17] A set of N elements can be sorted using $\mathcal{O}((N/B) \log_{M/B}(N/B))$ I/Os and $\mathcal{O}(N \log N)$ internal memory computation time.*

Sorting optimally on multiple disks is a more challenging problem. The challenge with distribution-based algorithms is to distribute the blocks of the buckets approximately evenly across the D disks while at the same time making sure that every I/O-operation writes $\Omega(D)$ blocks to disk. The latter requirement needs to be satisfied to guarantee that the algorithm takes full advantage of the parallel disks (up to a constant factor) during write operations. The former requirement is necessary to guarantee that the recursive invocation of the sorting algorithm can utilize the full bandwidth of the parallel I/O-system when reading the data elements to be sorted. Vitter and Shriver [755] propose randomized online techniques that guarantee that with high probability each bucket is distributed evenly across the D disks. The balancing algorithm applies the classical result that if α balls are placed uniformly at random into β bins and $\alpha = \Omega(\beta \log \beta)$, then all bins contain approximately the same number of balls, with high probability. The balancing algorithm maps the blocks of each bucket uniformly at random to disks where they are to be stored. Viewing the blocks in a bucket as balls and the disks as bins, and assuming that the number of blocks is sufficiently larger than the number of disks, the blocks in each bucket are distributed evenly across the D disks, with high probability. A number of deterministic methods for performing distribution sort on multiple disks have also been proposed, including BalanceSort [586], sorting using the buffer tree [52], and algorithms obtained by simulating bulk-synchronous parallel sorting algorithms [244]. The reader may refer to these references for details.

For merge sort, it is required that each iteration in the merging phase is carried out in $\mathcal{O}(N/(DB))$ I/Os. In particular, each read operation must bring $\Omega(D)$ blocks of data into main memory, and each write operation must write $\Omega(D)$ blocks to disk. While the latter is easy to achieve, reading blocks in parallel is difficult because the runs to be merged were formed in the previous iteration without any knowledge about how they would interact with other runs in subsequent merge operations. Nodine and Vitter [587] propose an optimal deterministic merge sort for multiple disks. The algorithm first performs an approximate merge phase that guarantees that no element is too far away from its final location. In the second phase, each element is moved to its final location. Barve et al. [92, 93] claim that their sorting algorithm is the most practical one. Using their approach, each run is striped across the disks, with a random starting disk. When merging runs, the next block needed from each disk is read into main memory. If there is not sufficient room in main memory for all the blocks to be read, then the least needed blocks are discarded from main memory (without incurring any I/Os). They derive asymptotic upper bounds on the expected I/O complexity of their algorithm.

3.3 Simulation of Parallel Algorithms in External Memory

Blockwise data access is a central theme in the design of I/O-efficient algorithms. A second important issue, when more than one disk is present, is fully parallel disk I/O. A number of techniques have been proposed that address this issue by simulating parallel algorithms as external memory algorithms. Most notably, Atallah and Tsay [74] discuss how to derive I/O-efficient algorithms from parallel algorithms for mesh architectures, Chiang et al. [192] discuss how to obtain I/O-efficient algorithms from PRAM algorithms, and Sibeyn and Kaufmann [692], Dehne et al. [242, 244], and Dittrich et al. [254] discuss how to simulate coarse-grained parallel algorithms developed for the BSP, CGM, and BSP* models in external memory. In this section we discuss the simulation of PRAM algorithms in external memory, which has been proposed by Chiang et al. [192]. For a discussion of other simulation results see Chapter 15.

The *PRAM simulation* of [192] is particularly appealing as it translates the large number of PRAM-algorithms described in the literature into I/O-efficient and sometimes I/O-optimal algorithms, including algorithms for a large number of graph and geometric problems, such as connectivity, computing minimum spanning trees, planarity testing and planar embedding, computing convex hulls, Voronoi diagrams, and triangulations of point sets in the plane.

In order to describe the *simulation paradigm* of [192], let us quickly recall the definition of a PRAM. A *PRAM* consists of a number of RAM-type processing units that operate synchronously and share a global memory. Different processors can exchange information by reading and writing the same memory cell in the shared memory. The two main measures of performance of an algorithm in the PRAM-model are its running time and the amount of work it performs. The former is defined as the maximum number of computation steps performed by any of the processors. The latter is the product of the running time of the algorithm times the number of processors.

Now consider a PRAM-algorithm $\mathcal{A}$ that uses N processors and $\mathcal{O}(N)$ space and runs in $\mathcal{O}(T(N))$ time. To simulate the computation of algorithm $\mathcal{A}$, assume that the processor contexts ($\mathcal{O}(1)$ state information per processor) and the content of the shared memory are stored on disk in a suitable format. Assume furthermore that every computation step of a processor consists of a constant number of read accesses to shared memory (to retrieve the operands of the computation step), followed by $\mathcal{O}(1)$ computation, and a constant number of write accesses to shared memory (to write the results back to memory). Then one step of algorithm $\mathcal{A}$ can be simulated in $\mathcal{O}(\mathrm{sort}(N))$ I/Os as follows: First scan the list of processor contexts to translate the read accesses each processor intends to perform into read requests that are written to disk. Then sort the resulting list of read requests by the memory locations they access. Scan the sorted list of read requests and the memory representation to augment every read request with the content of the memory cell it addresses. Sort the list of read requests again, this time by the issuing processor, and finally scan the sorted list of read requests and the list of processor contexts to transfer the requested operands to each processor. The computation of all processors can now be simulated in a single scan over the processor contexts. Writing the results of the computation to the shared memory can be simulated in a manner similar to the simulation of reading the operands.

The simulation of read and write accesses to the shared memory requires $\mathcal{O}(1)$ scans of the list of processor contexts, $\mathcal{O}(1)$ scans of the representation of the shared memory, and sorting and scanning the lists of read and write requests a constant number of times. Since all these lists have size $\mathcal{O}(N)$, this takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. As argued above, the computation itself can be carried out in $\mathcal{O}(\mathrm{scan}(N))$ I/Os. Hence, a single step of algorithm $\mathcal{A}$ can be simulated in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, so that simulating the whole algorithm takes $\mathcal{O}(T(N) \cdot \mathrm{sort}(N))$ I/Os.

Theorem 3.2. *[192] A PRAM algorithm that uses N processors and $\mathcal{O}(N)$ space and runs in time $T(N)$ can be simulated in $\mathcal{O}(T(N) \cdot \mathrm{sort}(N))$ I/Os.*

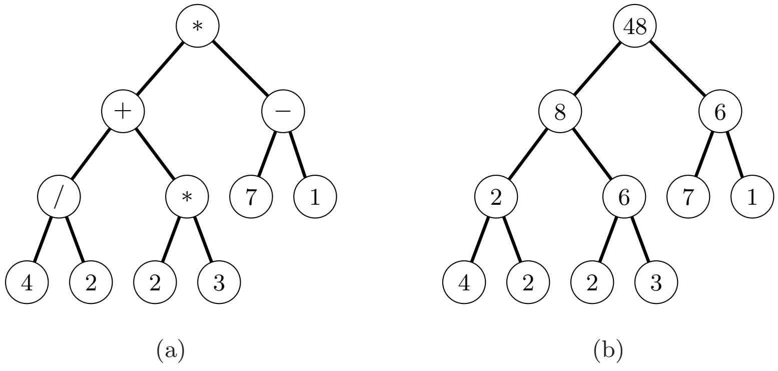


Fig. 3.2. (a) The expression tree for the expression $((4\,/\,2) + (2 * 3)) * (7 - 1)$. (b) The same tree with its vertices labelled with their values..

An interesting situation arises when the number of active processors and the amount of data to be processed decrease geometrically. That is, after a constant number of steps, the number of active processors and the amount of processed data decrease by a constant factor. Then the data and the contexts of the active processors can be compacted after each PRAM step, so that the I/O-complexity of simulating the steps of the algorithm is also geometrically decreasing. This implies that the I/O-complexity of the algorithm is dominated by the complexity of simulating the first step of the algorithm, which is $\mathcal{O}(\text{sort}(N))$.

3.4 Time-Forward Processing

Time-forward processing [52, 192] is an elegant technique for solving problems that can be expressed as a traversal of a *directed acyclic graph (DAG)* from its sources to its sinks. Problems of this type arise mostly in I/O-efficient graph algorithms, even though applications of this technique for the construction of I/O-efficient data structures are also known. Formally, the problem that can be solved using time-forward processing is that of evaluating a DAG G: Let ϕ be an assignment of labels $\phi(v)$ to the vertices of G. Then the goal is to compute another labelling ψ of the vertices of G so that for every vertex $v \in G$, label $\psi(v)$ can be computed from labels $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_k)$, where $u_1, \ldots, u_k$ are the in-neighbors of v.

As an illustration, consider the problem of expression-tree evaluation (see Fig. 3.2). For this problem, the input is a binary tree T whose leaves store real numbers and whose internal vertices are labelled with one of the four elementary binary operations $+, -, *, /$. The *value* of a vertex is defined recursively. For a leaf v, its value $val(v)$ is the real number stored at v. For an internal vertex v with label $\circ \in \{+, -, *, /\}$, left child x, and right child y, $val(v) = val(x) \circ val(y)$. The goal is to compute the value of the root of T. Cast in terms of the general DAG evaluation problem defined above, tree T is a DAG whose edges are directed from children to parents, labelling ϕ is the initial assignment of real numbers to the leaves of T and of operations to the internal vertices of T, and labelling ψ is the assignment of the values $val(v)$ to all vertices $v \in T$. For every vertex $v \in T$, its label $\psi(v) = val(v)$ is computed from the labels $\psi(x) = val(x)$ and $\psi(y) = val(y)$ of its in-neighbors (children) and its own label $\phi(v) \in \{+, -, *, /\}$.

In order to be able to evaluate a DAG G I/O-efficiently, two assumptions have to be satisfied: (1) The vertices of G have to be stored in topologically sorted order. That is, for every edge $(v, w) \in G$, vertex v precedes vertex w. (2) Label $\psi(v)$ has to be computable from labels $\phi(v)$ and $\psi(u_1), \dots, \psi(u_k)$ in $\mathcal{O}(\text{sort}(k))$ I/Os. The second condition is trivially satisfied if every vertex of G has in-degree no more than M.

Given these two assumptions, time-forward processing visits the vertices of G in topologically sorted order to compute labelling ψ. Visiting the vertices of G in this order guarantees that for every vertex $v \in G$, its in-neighbors are evaluated before v is evaluated. Thus, if these in-neighbors "send" their labels $\psi(u_1), \dots, \psi(u_k)$ to v, v has these labels and its own label $\phi(v)$ at its disposal to compute $\psi(v)$. After computing $\psi(v)$, v sends its own label $\psi(v)$ "forward in time" to its out-neighbors, which guarantees that these out-neighbors have $\psi(v)$ at their disposal when it is their turn to be evaluated.

The implementation of this technique due to Arge [52] is simple and elegant. The "sending" of information is realized using a priority queue Q (see Chapter 2 for a discussion of priority queues). When a vertex v wants to send its label $\psi(v)$ to another vertex w, it inserts $\psi(v)$ into priority queue Q and gives it priority w. When vertex w is evaluated, it removes all entries with priority w from Q. Since every in-neighbor of w sends its label to w by queuing it with priority w, this provides w with the required inputs. Moreover, every vertex removes its inputs from the priority queue before it is evaluated, and all vertices with smaller numbers are evaluated before w. Thus, at the time when w is evaluated, the entries in Q with priority w are those with lowest priority, so that they can be removed using a sequence of DELETEMIN operations.

Using the buffer tree of Arge [52] to implement priority queue Q, INSERT and DELETEMIN operations on Q can be performed in $\mathcal{O}((1/B) \cdot \log_{M/B}(|E|/B))$ I/Os amortized because priority queue Q never holds more than $|E|$ entries. The total number of priority queue operations performed by the algorithm is $\mathcal{O}(|E|)$, one INSERT and one DELETEMIN operation per edge. Hence, all updates of priority queue Q can be processed in $\mathcal{O}(\text{sort}(|E|))$ I/Os. The computation of labels $\psi(v)$ from labels $\phi(v)$ and $\psi(u_1), \dots, \psi(u_k)$, for all vertices $v \in G$, can also be carried out in $\mathcal{O}(\text{sort}(|E|))$ I/Os, using the above assumption that this computation takes $\mathcal{O}(\text{sort}(k))$ I/Os for a single vertex v. Hence, we obtain the following result.

Theorem 3.3. *[52, 192] Given a DAG $G = (V, E)$ whose vertices are stored in topologically sorted order, graph G can be evaluated in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os, provided that the computation of the label of every vertex $v \in G$ can be carried out in $\mathcal{O}(\text{sort}(\deg^-(v)))$ I/Os, where $\deg^-(v)$ is the in-degree of vertex v.*

3.5 Greedy Graph Algorithms

In this section we describe a simple technique proposed in [775] that can be used to make internal memory graph algorithms of a sufficiently simple structure I/O-efficient. For this technique to be applicable, the algorithm has to compute a labelling of the vertices of the graph, and it has to do so in a particular way. We call a vertex labelling algorithm $\mathcal{A}$ *single-pass* if it computes the desired labelling λ of the vertices of the graph by visiting every vertex exactly once and assigns label $\lambda(v)$ to v during this visit. We call $\mathcal{A}$ *local* if label $\lambda(v)$ can be computed in $\mathcal{O}(\text{sort}(k))$ I/Os from labels $\lambda(u_1), \dots, \lambda(u_k)$, where $u_1, \dots, u_k$ are the neighbors of v whose labels are computed before $\lambda(v)$. Finally, algorithm $\mathcal{A}$ is *presortable* if there is an algorithm that takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os to compute an order of the vertices of the graph so that $\mathcal{A}$ produces a correct result if it visits the vertices of the graph in this order. The technique we describe here is applicable if algorithm $\mathcal{A}$ is presortable, local, and single-pass.

So let $\mathcal{A}$ be a presortable local single-pass vertex-labelling algorithm computing some labelling λ of the vertices of a graph $G = (V, E)$. In order to make algorithm $\mathcal{A}$ I/O-efficient, the two main problems are to determine an order in which algorithm $\mathcal{A}$ should visit the vertices of G and devise a mechanism that provides every vertex v with the labels of its previously visited neighbors $u_1, \dots, u_k$. Since algorithm $\mathcal{A}$ is presortable, there exists an algorithm $\mathcal{A}'$ that takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os to compute an order of the vertices of G so that algorithm $\mathcal{A}$ produces the correct result if it visits the vertices of G in this order. Assume w.l.o.g. that this ordering of the vertices of G is expressed as a numbering. We use algorithm $\mathcal{A}'$ to number the vertices of G and then derive a DAG G' from G by directing every edge of G from the vertex with smaller number to the vertex with larger number. DAG G' has the property that for every vertex v, the in-neighbors of v in G' are exactly those neighbors of v that are labelled before v. Hence, labelling λ can be computed using time-forward processing. In particular, by the locality of $\mathcal{A}$, the label $\lambda(v)$ of every vertex can be computed in $\mathcal{O}(\text{sort}(k))$ I/Os from the labels $\lambda(u_1), \dots, \lambda(u_k)$ of its in-neighbors $u_1, \dots, u_k$ in G', which is a simplified version of the condition for the applicability of time-forward processing. This leads to the following result.

Theorem 3.4. *[775] Every graph problem $\mathcal{P}$ that can be solved by a presortable local single-pass vertex labelling algorithm can be solved in* $\mathcal{O}(\text{sort}(|V|+|E|))$ *I/Os.*

An important observation to be made is that in this application of time-forward processing, the restriction that the vertices of the DAG to be evaluated have to be given in topologically sorted order does not pose a problem because the directions of the edges are chosen only after fixing an order of the vertices that is to be the topological order.

To illustrate the power of Theorem 3.4, we apply it below to obtain deterministic $\mathcal{O}(\text{sort}(|V|+|E|))$ I/O algorithms for finding a maximal independent set of a graph G and coloring a graph of degree Δ with $\Delta+1$ colors. In [775], the approach is applied in a less obvious manner in order to compute a maximal matching of a graph $G = (V, E)$ in $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os. The problem with computing a maximal matching is that it is an edge labelling problem. However, Zeh [775] shows that it can be transformed into a vertex labelling problem of a graph with $|E|$ vertices and at most $2|E|$ edges.

3.5.1 Computing a Maximal Independent Set

In order to compute a *maximal independent set* S of a graph $G = (V, E)$ in internal memory, the following simple algorithm can be used: *Process the vertices in an arbitrary order. When a vertex $v \in V$ is visited, add it to S if none of its neighbors is in S.* Translated into a labelling problem, the goal is to compute the characteristic function $\chi_S : V \to \{0, 1\}$ of S, where $\chi_S(v) = 1$ if $v \in S$, and $\chi_S(v) = 0$ if $v \notin S$. Also note that if S is initially empty, then any neighbor w of v that is visited after v cannot be in S at the time when v is visited, so that it is sufficient for v to inspect all its neighbors that are visited before v to decide whether or not v should be added to S. The result of these modifications is a vertex-labelling algorithm that is presortable (since the order in which the vertices are visited is unimportant), local (since only previously visited neighbors of v are inspected to decide whether v should be added to S, and a single scan of labels $\chi_S(u_1), \dots, \chi_S(u_k)$ suffices to do so), and single-pass. This leads to the following result.

Theorem 3.5. *[775] Given an undirected graph $G = (V, E)$, a maximal independent set of G can be found in* $\mathcal{O}(\text{sort}(|V|+|E|))$ *I/Os and linear space.*

3.5.2 Coloring Graphs of Bounded Degree

The algorithm to compute a $(\Delta + 1)$-*coloring of a graph G* whose vertices have degree bounded by some constant Δ is similar to the algorithm for computing a maximal independent set presented in the previous section: *Process the vertices in an arbitrary order. When a vertex $v \in V$ is visited, assign a* *color $c(v) \in \{1, \dots, \Delta+1\}$ to vertex v that has not been assigned to any neighbor of v.* The algorithm is presortable and single-pass for the same reasons as the maximal independent set algorithm. The algorithm is local because the color of v can be determined as follows: Sort the colors $c(u_1), \dots, c(u_k)$ of v's in-neighbors $u_1, \dots, u_k$. Then scan this list and assign the first color not in this list to v. This takes $\mathcal{O}(\text{sort}(k))$ I/Os.

Theorem 3.6. *[775] Given an undirected graph $G = (V, E)$ whose vertices have degree at most Δ, a $(\Delta + 1)$-coloring of G can be computed in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os and linear space.*

3.6 List Ranking and the Euler Tour Technique

List ranking and the Euler tour technique are two techniques that have been applied successfully in the design of PRAM algorithms for labelling problems on lists and rooted trees and problems that can be reduced efficiently to one of these problems. Given the similarity of the issues to be addressed in parallel and external memory algorithms, it is not surprising that the same two techniques can be applied in I/O-efficient algorithms as well.

3.6.1 List Ranking

Let L be a linked list, i.e., a collection of vertices $x_1, \dots, x_N$ such that each vertex x_i, except the tail of the list, stores a pointer $\text{succ}(x_i)$ to its successor in L, no two vertices have the same successor, and every vertex can reach the tail of L by following successor pointers. Given a pointer to the head of the list (i.e., the vertex that no other vertex in the list points to), the *list ranking* problem is that of computing for every vertex x_i of list L, its distance from the head of L, i.e., the number of edges on the path from the head of L to x_i.

In internal memory this problem can easily be solved in linear time using the following algorithm: *Starting at the head of the list, follow successor pointers and number the vertices of the list from* 0 *to* $N - 1$ *in the order they are visited.* Often we use the term "list ranking" to denote the following generalization of the list ranking problem, which is solvable in linear time using a straightforward generalization of the above algorithm: Given a function $\lambda : \{x_1, \dots, x_N\} \to X$ assigning labels to the vertices of list L and a multiplication $\otimes : X \times X \to X$ defined over X, compute a label $\phi(x_i)$ for each vertex x_i of L such that $\phi\bigl(x_{\sigma(1)}\bigr) = \lambda\bigl(x_{\sigma(1)}\bigr)$ and $\phi\bigl(x_{\sigma(i)}\bigr) = \phi\bigl(x_{\sigma(i-1)}\bigr) \otimes \lambda\bigl(x_{\sigma(i)}\bigr)$, for $1 < i \le N$, where $\sigma : [1, N] \to [1, N]$ is a permutation so that $x_{\sigma(1)}$ is the head of L and $\text{succ}\bigl(x_{\sigma(i)}\bigr) = x_{\sigma(i+1)}$, for $1 \le i < N$.

Unfortunately the simple internal memory algorithm is not I/O-efficient: Since we have no control over the physical order of the vertices of L on disk, an adversary can easily arrange the vertices of L in a manner that forces the internal memory algorithm to perform one I/O per visited vertex, so that the algorithm performs $\Omega(N)$ I/Os in total. On the other hand, the lower bound for list ranking shown in [192] is only $\Omega(\text{perm}(N))$. Next we sketch a list ranking algorithm proposed in [192] that takes $\mathcal{O}(\text{sort}(N))$ I/Os and thereby closes the gap between the lower and the upper bound.

We make the simplifying assumption that multiplication over X is associative. If this is not the case, we determine the distance of every vertex from the head of L, sort the vertices of L by increasing distances, and then compute the prefix product using the internal memory algorithm. After arranging the vertices by increasing distances from the head of L, the internal memory algorithm takes $\mathcal{O}(\text{scan}(N))$ I/Os. Hence, the whole procedure still takes $\mathcal{O}(\text{sort}(N))$ I/Os, and the associativity assumption is not a restriction.

Given that multiplication over X is associative, the algorithm of [192] uses graph contraction to rank list L as follows: First an independent set I of L is found so that $|I| = \Omega(N)$. Then the elements in I are removed from L. That is, for every element $x \in I$ with predecessor y and successor z in L, the successor pointer of y is updated to $\text{succ}(y) = z$. The label of x is multiplied with the label of z, and the result is assigned to z as its new label in the compressed list. It is not hard to see that the weighted ranks of the elements in $L-I$ remain the same after adjusting the labels in this manner. Hence, their ranks can be computed by applying the list ranking algorithm recursively to the compressed list. Once the ranks of all elements in $L - I$ are known, the ranks of the elements in I are computed by multiplying their labels with the ranks of their predecessors in L.

If the algorithm excluding the recursive invocation on the compressed list takes $\mathcal{O}(\text{sort}(N))$ I/Os, the total I/O-complexity of the algorithm is given by the recurrence $\mathcal{I}(N) = \mathcal{I}(cN) + \mathcal{O}(\text{sort}(N))$, for some constant $0 < c < 1$. The solution of this recurrence is $\mathcal{O}(\text{sort}(N))$. Hence, we have to argue that every step, except the recursive invocation, can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os.

Given independent set I, it suffices to sort the vertices in I by their successors and the vertices in $L-I$ by their own IDs, and then scan the resulting two sorted lists to update the weights of the successors of all elements in I. The successor pointers of the predecessors of all elements in I can be updated in the same manner. In particular, it suffices to sort the vertices in $L - I$ by their successors and the vertices in I by their own IDs, and then scan the two sorted lists to copy the successor pointer from each vertex in I to its predecessor. Thus, the construction of the compressed list takes $\mathcal{O}(\text{sort}(N))$ I/Os, once set I is given.

In order to compute the independent set I, Chiang et al. [192] apply a 3-coloring procedure for lists, which applies time-forward processing to "monotone" sublists of L and takes $O(\text{sort}(N))$ I/Os; the largest monochromatic set is chosen to be set I. Using the maximal independent set algorithm of Section 3.5.1, a large independent set I can be obtained more directly in the same number of I/Os because a maximal independent set of a list has size at least $N/3$. Thus, we have the following result.

Theorem 3.7. *[192] A list of length N can be ranked in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

List ranking alone is of very limited use. However, combined with the Euler tour technique described in the next section, it becomes a very powerful tool for solving problems on trees that can be expressed as functions over a traversal of the tree or problems on general graphs that can be expressed in terms of a traversal of a spanning tree of the graph. An important application is the *rooting* of an undirected tree T, which is the process of directing all edges of T from parents to children after choosing one vertex of T as the root. Given a rooted tree T (i.e., one where all edges are directed from parents to children), the Euler tour technique and list ranking can be used to compute a preorder or postorder numbering of the vertices of T, or the sizes of the subtrees rooted at the vertices of T. Such labellings are used in many classical graph algorithms, so that the ability to compute them is a first step towards solving more complicated graph problems.

3.6.2 The Euler Tour Technique

An *Euler tour* of a tree $T = (V, E)$ is a traversal of T that traverses every edge exactly twice, once in each direction. Such a traversal is useful, as it produces a linear list of vertices or edges that captures the structure of the tree. Hence, it allows standard parallel or external memory algorithms to be applied to this list, in order to solve problems on tree T that can be expressed as some function to be evaluated over the Euler tour.

Formally, the tour is represented as a linked list L whose elements are the edges in the set $\{(v, w), (w, v) : \{v, w\} \in E\}$ and so that for any two consecutive edges e_1 and e_2, the target of e_1 is the source of e_2. In order to define an Euler tour, choose a circular order of the edges incident to each vertex of T. Let $\{v, w_1\}, \ldots, \{v, w_k\}$ be the edges incident to vertex v. Then let $\mathrm{succ}((w_i, v)) = (v, w_{i+1})$, for $1 \le i < k$, and $\mathrm{succ}((w_k, v)) = (v, w_1)$. The result is a circular linked list of the edges in T. Now an Euler tour of T starting at some vertex r and returning to that vertex can be obtained by choosing an edge (v, r) with $\mathrm{succ}((v, r)) = (r, w)$, setting $\mathrm{succ}((v, r)) = \mathbf{null}$, and choosing (r, w) as the first edge of the traversal.

List L can be computed from the edge set of T in $\mathcal{O}(\mathrm{sort}(N))$ I/Os: First scan set E to replace every edge $\{v, w\}$ with two directed edges (v, w) and (w, v). Then sort the resulting set of directed edges by their target vertices. This stores the incoming edges of every vertex consecutively. Hence, a scan of the sorted edge list now suffices to compute the successor of every edge in L.

Lemma 3.8. *An Euler tour L of a tree with N vertices can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

Given an unrooted (and undirected) tree T, choosing one vertex of T as the root defines a direction on the edges of T by requiring that every edge be directed from the parent to the child. The process of *rooting* tree T is that of computing these directions explicitly for all edges of T. To do this, we construct an Euler tour starting at an edge (r, v) and compute the rank of every edge in the list. For every pair of opposite edges (u, v) and (v, u), we call the edge with the lower rank a *forward edge*, and the other a *back edge*. Now it suffices to observe that for any vertex $x \neq r$ in T, edge $(\mathit{parent}(x), x)$ is traversed before edge $(x, \mathit{parent}(x))$ by any Euler tour starting at r. Hence, for every pair of adjacent vertices x and $\mathit{parent}(x)$, edge $(\mathit{parent}(x), x)$ is a forward edge, and edge $(x, \mathit{parent}(x))$ is a back edge. That is, the set of forward edges is the desired set of edges directed from parents to children. Constructing and ranking an Euler tour starting at the root r takes $\mathcal{O}(\text{sort}(N))$ I/Os, by Theorem 3.7 and Lemma 3.8. Given the ranks of all edges, the set of forward edges can be extracted by sorting all edges in L so that for any two adjacent vertices v and w, edges (v, w) and (w, v) are stored consecutively and then scanning this sorted edge list to discard the edge with higher rank from each of these edge pairs. Hence, a tree T can be rooted in $\mathcal{O}(\text{sort}(N))$ I/Os.

Instead of discarding back edges, it may be useful to keep them, but tag every edge of the Euler tour L as either a forward or back edge. Using this information, well-known labellings of the vertices of T can be computed by ranking list L after assigning appropriate weights to the edges of L. For example, consider the weighted ranks of the edges in L after assigning weight one to every forward edge and weight zero to every back edge. Then the *preorder* number of every vertex $v \neq r$ in T is one more than the weighted rank of the forward edge with target v; the preorder number of the root r is always one. The size of the subtree rooted at v is one more than the difference between the weighted ranks of the back edge with source v and the forward edge with target v. To compute a *postorder* numbering, we assign weight zero to every forward edge and weight one to every back edge. Then the postorder number of every vertex $v \neq r$ is the weighted rank of the back edge with source v. The postorder number of the root r is always N.

After labelling every edge in L as a forward or back edge, the appropriate weights for computing the above labellings can be assigned in a single scan of list L. The weighted ranks can then be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, by Theorem 3.7. Extracting preorder and postorder numbers from these ranks takes a single scan of list L again. To extract the sizes of the subtrees rooted at the vertices of T, we sort the edges in L so that opposite edges with the same endpoints are stored consecutively. Then a single scan of this sorted edge list suffices to compute the size of the subtree rooted at every vertex v. Hence, all these labels can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os for a tree with N vertices.

3.7 Graph Blocking

The final topic we consider is that of *blocking graphs*. In particular, we are interested in laying out graphs on disk so that traversals of paths in these graphs cause as few page faults as possible, using an appropriate paging algorithm that needs to be specified along with the graph layout. We make two assumptions the first of which makes the design of suitable layouts easier, while the second makes it harder. The first assumption we make is that the graph to be stored on disk is static, i.e., does not change. Hence, it is not necessary to be able to update the graph I/O-efficiently, so that redundancy can be used to obtain better layouts. That is, some or all of the vertices of the graph may be stored in more than one location on disk. In order to visit such a vertex v, it suffices to visit any of these copies. This gives the paging algorithm considerable freedom in choosing the "best" of all disk blocks containing a copy of v as the one to be brought into main memory. By choosing the right block, the paging algorithm can guarantee that the next so many steps along the path do not cause any page faults. The second assumption we make is that the paths are traversed in an online fashion. That is, the traversed path is constructed only while it is traversed and is not known in advance. This allows an adversary to choose the worst possible path based on the previous decisions made by the paging algorithm, and the paging algorithm has to be able to handle such adversarial behavior gracefully. That is, it has to minimize the number of page faults in the worst case without having any a priori knowledge about the traversed path.

Besides the obvious applications where the problem to be solved is a graph problem and the answer to a query consists of a traversal of a path in the graph, the theory of graph blocking can be applied in order to store pointer-based data structures on disk so that queries on these data structures can be answered I/O-efficiently. In particular, a pointer-based data structure can be viewed as a graph with additional information attached to its vertices. Answering a query on such a data structure often reduces to traversing a path starting at a specified vertex in the data structure. For example, laying out binary search trees on disk so that paths can be traversed I/O-efficiently, one arrives at a layout which bears considerable resemblance to a B-tree. As discussed in Chapter 2, the layout of lists discussed below needs only few modifications to be transformed into an I/O-efficient linked list data structure that allows insertions and deletions, i.e., is no longer static.

Since we allow redundant representations of the graph, the two main measures of performance for a given blocking and the used paging algorithm are the number of page faults incurred by a path traversal in the worst case and the amount of space used by the graph representation. Clearly, in order to store a graph with N vertices on disk, at least N/B blocks of storage are required. We define the *storage blow-up* of a graph blocking to be β if it uses $\beta N/B$ blocks of storage to store the graph on disk. Since space usage is a serious issue with large data sets, the goal is to design graph blockings that minimize the storage blow-up and at the same time minimize the number of page faults incurred by a path traversal. Often there is a trade-off. That is, no blocking manages to minimize both performance measures at the same time. In this section we restrict our attention to graph layouts with constant storage blow-up and bound the worst-case number of page faults achievable by these layouts using an appropriate paging algorithm. Throughout this section we denote the length of the traversed path by L. The traversal of such a path requires at least $\lceil L/B \rceil$ I/Os in any graph because at most B vertices can be brought into main memory in a single I/O-operation.

The graphs we consider include lists, trees, grids and planar graphs. The blocking for planar graphs generalizes to any class of graphs with small separators. The results presented here are described in detail in the papers of Nodine et al. [585], Hutchinson et al. [419], and Agarwal et al. [7].

Blocking Lists. The natural approach for *blocking a list* is to store the vertices of the list in an array, sorted in their order of appearance along the list. The storage blow-up of this blocking is one (i.e., there is no blow-up at all). Since every vertex is stored exactly once in the array, the paging algorithm has no choice about the block to be brought into main memory when a vertex is visited. Still, if the traversed path is simple (i.e., travels along the list in only one direction), the traversal of a path of length L incurs only L/B page faults. To see this, assume w.l.o.g. that the path traverses the list in forward direction, i.e., the vertices are visited in the same order as they are stored in the array, and consider a vertex v in the path that causes a page fault. Then v is the first vertex in the block that is brought into main memory, and the $B - 1$ vertices succeeding v in the direction of the traversal are stored in the same block. Hence, the traversal of any simple path causes one page fault every B steps along the path.

If the traversed path is not simple, there are several alternatives. Assuming that $M \geq 2B$, the same layout as for simple paths can be used; but the paging algorithm has to be changed somewhat. In particular, when a page fault occurs at a vertex v, the paging algorithm has to make sure that the block brought into main memory does not replace the block containing the vertex u visited just before v. Using this strategy, it is again guaranteed that after every page fault, at least $B - 1$ steps are required before the next page fault occurs. Indeed, the block containing vertex v contains all vertices that can be reached from v in $B - 1$ steps by continuing the traversal in the same direction, and the block containing vertex u contains all vertices that can be reached from v in B steps by continuing the traversal in the opposite direction. Hence, traversing a path of length L incurs at most L/B page faults.

In the pathological situation that $M = B$ (i.e., there is room for only one block in main memory) and given the layout described above, an adversary can construct a path whose traversal causes a page fault at every step. In particular, the adversary chooses two adjacent vertices v and w that are in

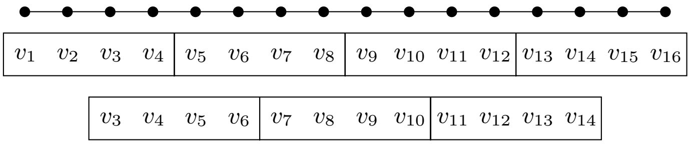

v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16}
v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14}

Fig. 3.3. A layout of a list on a disk with block size $B = 4$. The storage blow-up of the layout is two.

different blocks and then constructs the path $P = (v, w, v, w, \dots)$. Whenever vertex v is visited, the block containing v is brought into main memory, thereby overwriting the block containing w. When vertex w is visited, the whole process is reversed. The following layout with storage blow-up two thwarts the adversary's strategy: Instead of having only one array containing the vertices of the list, create a second array storing the vertices of the list in the same order, but with the block boundaries offset by $B/2$ (see Fig. 3.3). To use this layout efficiently, the paging algorithm has to change as follows: Assume that the current block is from the first array. When a page fault occurs, the vertex v to be visited is the last vertex in the block preceding the current block in the first array or the first vertex in the block succeeding the current block. Since the blocks in the second array are offset by $B/2$, this implies that v is at least $B/2 - 1$ steps away from the border of the block containing v in the second array. Hence, the paging algorithm loads this block into memory because then at least $B/2$ steps are required before the next page fault occurs. When the next page fault occurs, the algorithm switches back to a block in the first array, which again guarantees that the next $B/2 - 1$ steps cannot cause a page fault. That is, the paging algorithm alternates between the two arrays. Traversing a path of length L now incurs at most $2L/B$ page faults.

Blocking Trees. Next we discuss *blocking of trees.* Quite surprisingly, trees cannot be blocked at all if there are no additional restrictions on the tree or the type of traversal that is allowed. To see this, consider a tree whose internal vertices have degree at least M. Then for any vertex v, at most $M - 1$ of its neighbors can reside in main memory at the same time as v. Hence, there is at least one neighbor of v that is not in main memory at the time when v is in main memory. An adversary would always select this vertex as the one to be visited after v. Since at least every other vertex on any path in the tree has to be an internal vertex, the adversary can construct a path that causes a page fault every other step along the path. Note that this is true irrespective of the storage blow-up of the graph representation.

From this simple example it follows that for unrestricted traversals, a good blocking of a tree can be achieved only if the degree of the vertices of the tree is bounded by some constant d. We show that there is a blocking with storage blow-up four so that traversing a path of length L causes at most $2L/\log_d B$ page faults. To construct this layout, which is very similar to the list layout

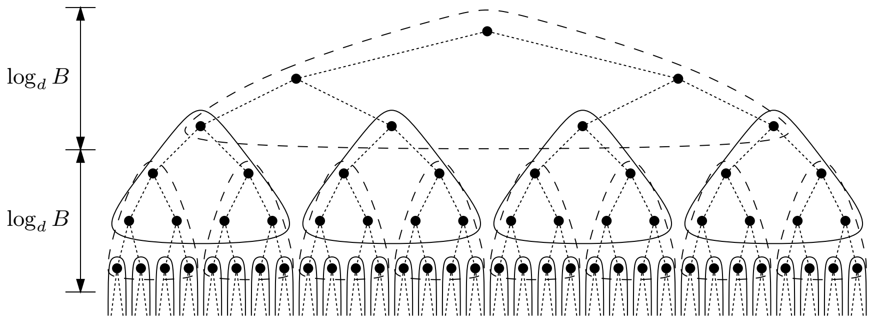


Fig. 3.4. A blocking of a binary tree with block size 7. The subtrees in the first partition are outlined with dashed lines. The subtrees in the second partition are outlined with solid lines.

shown in Fig. 3.3, we choose one vertex r of T as the root and construct two partitions of T into layers of height $\log_d B$ (see Fig. 3.4). In the first partition, the i-th layer contains all vertices at distance between $(i-1)\log_d B$ and $i\log_d B - 1$ from r. In the second partition, the i-th layer contains all vertices at distance between $(i-1/2)\log_d B$ and $(i+1/2)\log_d B = 1$ from r. Each layer in both partitions consists of subtrees of size at most B, so that each subtree can be stored in a block. Moreover, small subtrees can be packed into blocks so that no block is less than half full. Hence, both partitions together use at most $4N/B$ blocks, and the storage blow-up is at most four.

The paging algorithm now alternates between the two partitions similar to the above paging algorithm for lists. Consider the traversal of a path, and let v be a vertex that causes a page fault. Assume that the tree currently held in main memory is from the first partition. Then v is the root or a leaf of a tree in the first partition. Hence, the tree in the second partition that contains v contains all vertices that can be reached from v in $(\log_d B)/2 - 1$ steps. Thus, by loading this block into main memory, the algorithm guarantees that the next page fault occurs after at least $(\log_d B)/2 - 1$ steps, and traversing a path of length L causes at most $2L/(\log_d B)$ page faults.

If all traversed paths are restricted to travel away from the root of T, the storage blow-up can be reduced to two, and the number of page faults can be reduced to $L/\log_d B$. To see this, observe that only the first of the above partitions is needed, and for any traversed path, the vertices causing page faults are the roots of subtrees in the partition. After loading the block containing that root into main memory, $\log_d B - 1$ steps are necessary in order to reach a leaf of the subtree, and the next page fault occurs after $\log_d B$ steps. For traversals towards the root, Hutchinson et al. [419] show that using $\mathcal{O}(N/B)$ disk blocks, a page fault occurs every $\Omega(B)$ steps, so that a path of length L can be traversed in $\mathcal{O}(L/B)$ I/Os.

Blocking Two-Dimensional Grids. *Blocking of two-dimensional grids* can be done using the same ideas as for blocking lists. This is not too surprising because lists are equivalent to one-dimensional grids from a blocking point of view. In both cases, the grid is covered with subgrids of size B. In the two-dimensional case, the subgrids have dimension $\sqrt{B} \times \sqrt{B}$. We call such a covering a *tessellation.*

However, the added dimension does create a few complications. In particular, if the amount of main memory is $M = B$, a blocking that consists of three tessellations is required to guarantee that a page fault occurs only every $\omega(1)$ steps. To see that two tessellations are not sufficient, consider two tessellations offset by k and l in the x and y-dimensions. Then an adversary chooses a path containing vertices $\bigl(i\sqrt{B}+k, j\sqrt{B}\bigr)$, $\bigl(i\sqrt{B}+k+1, \sqrt{B}j\bigr)$, $\bigl(i\sqrt{B}+k, j\sqrt{B}+1\bigr)$, and $\bigl(i\sqrt{B}+k+1, j\sqrt{B}+1\bigr)$, for two integers i and j. Neither of the two tessellations contains a subgrid that contains more than two of these vertices. Hence, given that the traversal is at one of the four vertices, and only one of the two subgrids containing this vertex is in main memory, an adversary can always choose the neighbor of the current vertex that does not belong to the subgrid in main memory. Thus, every step causes a page fault.

By increasing the storage blow-up to three, it can be guaranteed that a page fault occurs at most every $\sqrt{B}/6$ steps. In particular, the blocking consists of three tessellations so that the second tessellation has offset $\sqrt{B}/3$ in both directions w.r.t. the first tessellation, and the third tessellation has offset $2\sqrt{B}/3$ w.r.t. the first tessellation (see Fig. 3.5). Then it is not hard to see that for every vertex in the grid, there exists at least one subgrid in one of the three tessellations, so that the vertex is at least $\sqrt{B}/6$ steps away from the boundary of the subgrid. Hence, whenever a page fault occurs at some vertex v, the paging algorithm brings the appropriate grid for vertex v into main memory.

If $M \geq 2B$, the storage blow-up can be reduced to two, and the number of steps between two page faults can be increased to $\sqrt{B}/4$. In particular, the blocking consists of two tessellations so that the second tessellation has offset $\sqrt{B}/2$ in both directions w.r.t. the first tessellation. Let v be a vertex that causes a page fault, and let u be the vertex visited before v in the traversal. Then at the time when the paging algorithm has to bring vertex v into main memory, a block containing u from one of the two tessellations is already in main memory. Now it is easy to verify that the block containing u that is already in main memory together with one of the two blocks containing v in the two tessellations covers at least the $\sqrt{B}/4$-neighborhood of v, i.e., the subgrid containing all vertices that can be reached from v in at most $\sqrt{B}/4$ steps. Hence, by bringing the appropriate block containing v into main memory, the paging algorithm can guarantee that the next page fault occurs after only $\sqrt{B}/4$ steps.

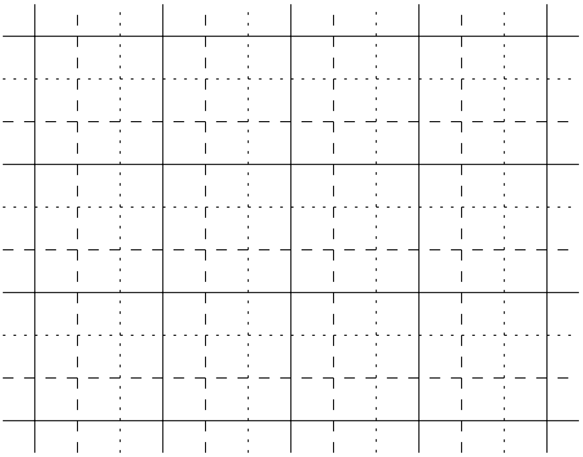


Fig. 3.5. A blocking for $M = B$.

Finally, if $M \geq 3B$, the storage blow-up can be brought down to one while keeping the number of page faults incurred by the traversal of a path of length L at $4L/\sqrt{B}$. To achieve this, the tessellation shown in Fig. 3.6 is used. To prove that the traversal of a path of length L incurs at most $4L/\sqrt{B}$ page faults, we show that at most two page faults can occur within $\sqrt{B}/2$ steps. So assume the opposite. Then let v be a vertex that causes a page fault, and let u be the vertex visited immediately before v. Let u be in the solid bold subgrid in Fig. 3.6 and assume that it is in the top left quarter of the subgrid. Then all vertices that can be reached from u in $\sqrt{B}/2$ steps are contained in the solid thin subgrids. In particular, v is contained in one of these subgrids. If v is in subgrid A, it takes at least $\sqrt{B}/2$ steps after visiting v to reach a vertex in subgrid C. If v is in subgrid C, it takes at least $\sqrt{B}/2$ steps after visiting v to reach a vertex in subgrid A. Hence, in both cases only a vertex in subgrid B can cause another page fault within $\sqrt{B}/2$ steps after visiting vertex u. If v is in subgrid B, consider the next vertex w after v that causes a page fault and is at most $\sqrt{B}/2$ steps away from u. Vertex w is either in subgrid A, or in subgrid C. W.l.o.g. let w be in subgrid A. Then it takes at least $\sqrt{B}/2$ steps to reach a vertex in subgrid C, so that again only two page faults can occur within $\sqrt{B}/2$ steps after visiting u. This shows that the traversal of a path of length L incurs at most $4L/\sqrt{B}$ page faults.

Blocking Planar Graphs. The final result we discuss here concerns the *blocking of planar graphs* of bounded degree. Quite surprisingly, planar graphs allow blockings with the same performance as for trees, up to constant factors. That is, with constant storage blow-up it can be guaranteed that traversing a path of length L incurs at most $4L/\log_d B$ page faults, where d is the maximal degree of the vertices in the graph. To achieve this, Agarwal et al. [7] make use of separator results due to Frederickson [315]. In particular,

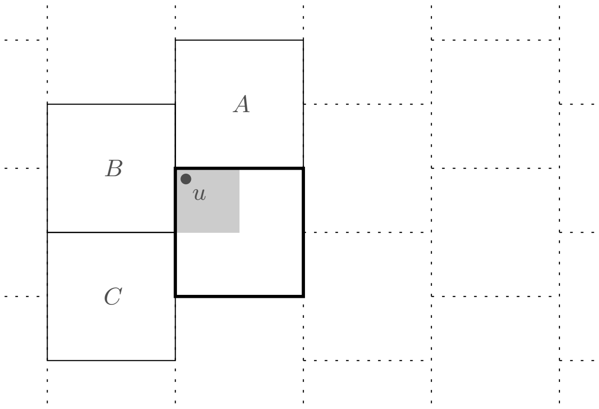


Fig. 3.6. A blocking with storage blow-up one.

Frederickson shows that for every planar graph G, there exists a set S of $\mathcal{O}\big(N/\sqrt{B}\big)$ vertices so that no connected component of $G - S$ has size more than B. Based on this result, the following graph representation can be used to achieve the above result. First ensure that every connected component of $G - S$ is stored in a single block and pack small connected components into blocks so that every block is at least half full. This representation of $G - S$ uses at most $2N/B$ disk blocks. The second part of the blocking consists of the $(\log_d B)/2$-neighborhoods of the vertices in S. That is, for every vertex $v \in S$, the vertices reachable from v in at most $(\log_d B)/2$ steps are stored in a single block. These vertices fit into a single block because at most $d^{(\log_d B)/2} = \sqrt{B}$ vertices can be reached in that many steps from v. Packing these neighborhoods into blocks so that every block is at least half full, this second part of the blocking uses $\mathcal{O}\big(\sqrt{B}|S|/B\big) = \mathcal{O}(N/B)$ blocks. Hence, the storage blow-up is $\mathcal{O}(1)$.

Now consider the exploration of an arbitrary path in G. Let v be a vertex that causes a page fault. If $v \in S$, the paging algorithm brings the block containing the $(\log_d B)/2$-neighborhood of v into main memory. This guarantees that at least $(\log_d B)/2$ steps along the path are required before the next page fault occurs. If $v \notin S$, $v \in G - S$. Then the paging algorithm brings the block containing the connected component of $G - S$ that contains v into main memory. As long as the path stays inside this connected component, no further page faults occur. When the next page fault occurs, it has to happen at a vertex $w \in S$. Hence, the paging algorithm brings the block containing the neighborhood of w into main memory, and at least $(\log_d B)/2$ steps are required before the next page fault occurs. Thus, at most two page faults occur every $(\log_d B)/2$ steps, and traversing a path of length L incurs at most $4L/(\log_d B)$ page faults. This is summarized in the following theorem.

Theorem 3.9 (Agarwal et al. [7]). *A planar graph with N vertices of degree at most d can be stored in $\mathcal{O}(N/B)$ blocks so that any path of length L can be traversed in $\mathcal{O}(L/\log_d B)$ I/Os.*

3.8 Remarks

In this chapter we have seen some of the fundamental techniques used in the design of efficient external memory algorithms. There are many other techniques that are not discussed in this chapter, but are scattered across various other chapters in this volume. I/O-efficient data structures are discussed in Chapter 2. Techniques used in computational geometry, including distribution sweeping and batch filtering, are discussed in Chapter 6. Many specialized graph algorithms are discussed in Chapters 4 and 5. These include algorithms for connectivity problems, breadth-first search, depth-first search, shortest paths, partitioning planar graphs, and computing planar embeddings of planar graphs. The simulation of bulk synchronous (coarse grained) parallel algorithms is discussed in Chapter 15.

4. Elementary Graph Algorithms in External Memory*

Irit Katriel and Ulrich Meyer**

4.1 Introduction

Solving real-world optimization problems frequently boils down to processing *graphs*. The graphs themselves are used to represent and structure relationships of the problem's components. In this chapter we review external-memory (**EM**) graph algorithms for a few representative problems:

Shortest path problems are among the most fundamental and also the most commonly encountered graph problems, both in themselves and as subproblems in more complex settings [21]. Besides obvious applications like preparing travel time and distance charts [337], shortest path computations are frequently needed in telecommunications and transportation industries [677], where messages or vehicles must be sent between two geographical locations as quickly or as cheaply as possible. Other examples are complex traffic flow simulations and planning tools [337], which rely on solving a large number of individual shortest path problems. One of the most commonly encountered subtypes is the *Single-Source Shortest-Path* (SSSP) version: let $G = (V, E)$ be a graph with $|V|$ nodes and $|E|$ edges, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real *weight* to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the weight $\text{dist}(v)$ of a minimum-weight ("shortest") path from s to v; the weight of a path is the sum of the weights of its edges.

Breadth-First Search (BFS) [554] can be seen as the unweighted version of SSSP; it decomposes a graph into levels where level i comprises all nodes that can be reached from the source via i edges. The BFS numbers also impose an order on the nodes within the levels. BFS has been widely used since the late 1950's; for example, it is an ingredient of the classical separator algorithm for planar graphs [507].

Another basic graph-traversal approach is *Depth-First Search* (DFS) [407]; instead of exploring the graph in levels, DFS tries to visit as many graph vertices as possible in a long, deep path. When no edge to an unvisited node

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the DFG grant SA 933/1-1.

** Partially supported by the Center of Excellence programme of the EU under contract number ICAI-CT-2000-70025. Parts of this work were done while the author was visiting the Computer and Automation Research Institute of the Hungarian Academy of Sciences, Center of Excellence, MTA SZTAKI, Budapest.

can be found from the current node then DFS backtracks to the most recently visited node with unvisited neighbor(s) and continues there. Similar to BFS, DFS has proved to be a useful tool, especially in artificial intelligence [177]. Another well-known application of DFS is in the linear-time algorithm for finding strongly connected components [713].

Graph connectivity problems include *Connected Components* (CC), *Biconnected Components* (BCC) and *Minimum Spanning Forest* (MST/MSF). In CC we are given a graph $G = (V, E)$ and we are to find and enumerate maximal subsets of the nodes of the graph in which there is a path between every two nodes. In BCC, two nodes are in the same subset iff there are *two* edge-disjoint paths connecting them. In MST/MSF the objective is to find a spanning tree of G (spanning forest if G is not connected) with a minimum total edge weight. Both problems are central in network design; the obvious applications are checking whether a communications network is connected or designing a minimum cost network. Other applications for CC include clustering, e.g., in computational biology [386] and MST can be used to approximate the *traveling salesman* problem within a factor of 1.5 [201].

We use the standard model of external memory computation [755]: There is a main memory of size M and an external memory consisting of D disks. Data is moved in blocks of size B consecutive words. An I/O-operation can move up to D blocks, one from each disk. Further details about models for memory hierarchies can be found in Chapter 1. We will usually describe the algorithms under the assumption $D = 1$. In the final results, however, we will provide the I/O-bounds for general $D \geq 1$ as well. Furthermore, we shall frequently use the following notational shortcuts: $\mathrm{scan}(x) := \mathcal{O}(x/(D \cdot B))$, $\mathrm{sort}(x) := \mathcal{O}(x/(D \cdot B) \cdot \log_{M/B}(x/B))$, and $\mathrm{perm}(x) := \mathcal{O}(\min\{x/D, \mathrm{sort}(x)\})$.

Organization of the Chapter. We discuss external-memory algorithms for all the problems listed above. In Sections 4.2 – 4.7 we cover graph traversal problems (BFS, DFS, SSSP) and Sections 4.8 – 4.13 provide algorithms for graph connectivity problems (CC, BCC, MSF).

4.2 Graph-Traversal Problems: BFS, DFS, SSSP

In the following sections we will consider the classical graph-traversal problems Breadth-First Search (BFS), Depth-First Search (DFS), and Single-Source Shortest-Paths (SSSP). All these problems are well-understood in *internal memory* (**IM**): BFS and DFS can be solved in $\mathcal{O}(|V| + |E|)$ time [21], SSSP with nonnegative edge weights requires $\mathcal{O}(|V| \cdot \log |V| + |E|)$ time [252, 317]. On more powerful machine models, SSSP can be solved even faster [374, 724].

Most **IM** algorithms for BFS, DFS, and SSSP visit the vertices of the input graph G in a one-by-one fashion; appropriate candidate nodes for the next vertex to be visited are kept in some data-structure Q (a queue for BFS, a stack for DFS, and a priority-queue for SSSP). After a vertex v is extracted from Q, the *adjacency list* of v, i.e., the set of neighbors of v in G, is examined in order to update Q: unvisited neighboring nodes are inserted into Q; the priorities of nodes already in Q may be updated.

The Key Problems. The short description above already contains the main difficulties for I/O-efficient graph-traversal algorithms:

(a) *Unstructured indexed access to adjacency lists.*
(b) Remembering visited nodes.
(c) (The lack of) *Decrease_Key* operations in external priority-queues.

Whether (a) is problematic or not depends on the sizes of the adjacency lists; if a list contains k edges then it takes $\Theta(1 + k/B)$ I/Os to retrieve all its edges. That is fine if $k = \Omega(B)$, but wasteful if $k = \mathcal{O}(1)$. In spite of intensive research, so far there is no general solution for (a) on sparse graphs: unless the input is known to have special properties (for example *planarity*), virtually all **EM** graph-traversal algorithms require $\Theta(|V|)$ I/Os to access adjacency lists. Hence, we will mainly focus on methods to avoid spending one I/O for each edge on general graphs[1]. However, there is recent progress for BFS on arbitrary *undirected* graphs [542]; e.g., if $|E| = \mathcal{O}(|V|)$, the new algorithm requires just $\mathcal{O}(|V|/\sqrt{B} + \text{sort}(|V|))$ I/Os. While this is a major step forward for BFS on undirected graphs, it is currently unclear whether similar results can be achieved for undirected DFS/SSSP or BFS/DFS/SSSP on general directed graphs.

Problem (b) can be partially overcome by solving the graph problems *in phases* [192]: a dictionary DI of maximum capacity $|\text{DI}| < M$ is kept in internal memory; DI serves to remember visited nodes. Whenever the capacity of DI is exhausted, the algorithms make a pass through the external graph representation: all edges pointing to visited nodes are discarded, and the remaining edges are compacted into new adjacency lists. Then DI is emptied, and a new phase starts by visiting the next element of Q. This *phase-approach* explored in [192] is most efficient if the quotient $|V|/|\text{DI}|$ is small[2]; $\mathcal{O}(\lceil |V|/|\text{DI}| \rceil \cdot \text{scan}(|V| + |E|))$ I/Os are needed in total to perform all graph compactions. Additionally, $\mathcal{O}(|V| + |E|)$ operations are performed on Q.

As for SSSP, problem (c) is less severe if (b) is resolved by the phase-approach: instead of actually performing Decrease_Key operations, several priorities may be kept for each node in the external priority-queue; after a node v is dequeued for the first time (with the smallest key) any further appearance of v in Q will be ignored. In order to make this work, superfluous

[1] In contrast, the chapter by Toma and Zeh in this volume (Chapter 5) reviews improved algorithms for special graph classes such as planar graphs.

[2] The chapters by Stefan Edelkamp (Chapter 11) and Rasmus Pagh (Chapter 2) in this book provide more details about space-efficient data-structures.

elements still kept in the **EM** data structure of Q are marked obsolete right before DI is emptied at the end of a phase; the marking can be done by scanning Q.

Plugging-in the I/O-bounds for external queues, stacks, and priority-queues as presented in Chapter 2 we obtain the following results:

Problem	Performance with the phase-approach [192]
BFS, DFS	$\mathcal{O}\left(\lvert V\rvert + \left\lceil \frac{\lvert V\rvert}{M} \right\rceil \cdot \mathrm{scan}(\lvert V\rvert + \lvert E\rvert)\right)$ I/Os
SSSP	$\mathcal{O}\left(\lvert V\rvert + \left\lceil \frac{\lvert V\rvert}{M} \right\rceil \cdot \mathrm{scan}(\lvert V\rvert + \lvert E\rvert) + \mathrm{sort}(\lvert E\rvert)\right)$ I/Os

Another possibility is to solve (b) and (c) by applying extra bookkeeping and extra data structures like the I/O-efficient *tournament tree* of Kumar and Schwabe [485]. In that case the graph traversal can be done in *one* phase, even if $n \gg M$.

It turns out that the best known **EM** algorithms for *undirected* graphs are simpler and/or more efficient than their respective counterparts for directed graphs; due to the new algorithm of [542], the difference for BFS on sparse graphs is currently as big as $\Omega(\sqrt{B} \cdot \log |V|)$.

Problems (b) and (c) usually disappear in the *semi-external memory* (**SEM**) setting where it is assumed that $M = c \cdot |V| < |E|$ for some appropriately chosen positive constant c: e.g., the **SEM** model may allow to keep a boolean array for (b) in internal memory; similarly, a node priority queue with Decrease_Key operation for (c) could reside completely in **IM**.

Still, due to (a), the currently best **EM**/**SEM** algorithms for BFS, DFS and SSSP require $\Omega(|V| + |E|/B)$ I/Os on general *directed* graphs. Taking into consideration that naive applications of the best **IM** algorithms in external memory cause $\mathcal{O}(|V| \cdot \log |V| + |E|)$ I/Os we see how little has been achieved so far concerning general sparse directed graphs. On the other hand, it is perfectly unclear, whether one can do significantly better at all: the best known lower-bound is only $\Omega(\mathrm{perm}(|V|) + |E|/B)$ I/Os (by the trivial reductions of list ranking to BFS, DFS, and SSSP).

In the following we will present some *one-pass approaches* in more detail. In Section 4.3 we concentrate on algorithms for undirected BFS. Section 4.4 introduces I/O-efficient Tournament Trees; their application to undirected **EM** SSSP is discussed in Section 4.5. Finally, Section 4.6 provides traversal algorithms for directed graphs.

4.3 Undirected Breadth-First Search

Our exposition of one-pass BFS algorithms for undirected graphs is structured as follows. After some preliminary observations we review the basic BFS algorithm of Munagala and Ranade [567] in Section 4.3.1. Then, in Section 4.3.2, we present the recent improvement by Mehlhorn and Meyer [542]. This algorithm can be seen as a refined implementation of the Munagala/Ranade approach. The new algorithm clearly outperforms a previous BFS approach [550], which was the first to achieve $o(|V|)$ I/Os on undirected sparse graphs with bounded node degrees. A tricky combination of both strategies might help to solve *directed* BFS with sublinear I/O; see Section 4.7 for more details.

We restrict our attention to computing the BFS *level* of each node v, i.e., the minimum number of edges needed to reach v from the source. For undirected graphs, the respective BFS tree or the BFS numbers (order of the nodes in a level) can be obtained efficiently: in [164] it is shown that each of the following transformations can be done using $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os: **BFS Numbers → BFS Tree → BFS Levels → BFS Numbers**.

The conversion BFS Numbers → BFS Tree is done as follows: for each node $v \in V$, the parent of v in the BFS tree is the node v' with BFS number $\mathit{bfsnum}(v') = \min_{(v,w)\in E} \mathit{bfsnum}(w)$. The adjacency lists can be augmented with the BFS numbers by sorting. Another scan suffices to extract the BFS tree edges.

As for the conversion BFS Tree → BFS Levels, an Euler tour [215] around the undirected BFS tree can be constructed and processed using scanning and list ranking; Euler tour edges directed towards the leaves are assigned a weight $+1$ whereas edges pointing towards the root get weight -1. A subsequent prefix-sum computation [192] on the weights of the Euler tour yields the appropriate levels.

The last transformation BFS Levels → BFS Numbers proceeds level-by-level: having computed correct numbers for level i, the order (BFS numbers) of the nodes in level $i+1$ is given as follows: each level-$(i+1)$ node v must be a child (in the BFS tree) of its adjacent level-i node with least BFS number. After sorting the nodes of level i and the edges between levels i and $i+1$, a scan provides the adjacency lists of level-$(i+1)$ nodes with the required information.

4.3.1 The Algorithm of Munagala and Ranade

We turn to the basic BFS algorithm of Munagala and Ranade [567], MR_BFS for short. It is also used as a subroutine in more recent BFS approaches [542, 550]. Furthermore, MR_BFS is applied in the deterministic CC algorithm of [567] (which we discuss in Section 4.9).

Let $L(t)$ denote the set of nodes in BFS level t, and let $|L(t)|$ be the number of nodes in $L(t)$. MR_BFS builds $L(t)$ as follows: let $A(t) := N(L(t-1))$

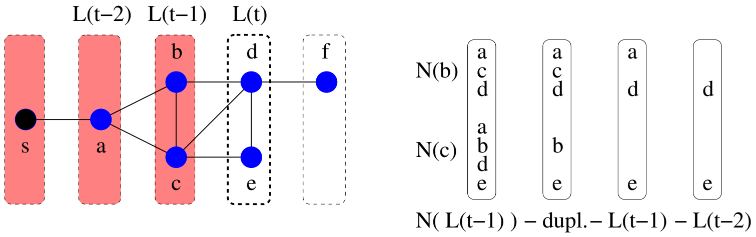


Fig. 4.1. A phase in the BFS algorithm of Munagala and Ranade [567]. Level $L(t)$ is composed out of the disjoint neighbor vertices of level $L(t-1)$ excluding those vertices already existing in either $L(t-2)$ or $L(t-1)$.

be the multi-set of neighbor vertices of nodes in $L(t-1)$; $N(L(t-1))$ is created by $|L(t-1)|$ accesses to the adjacency lists, one for each node in $L(t-1)$. Since the graph is stored in adjacency-list representation, this takes $\mathcal{O}\left(|L(t-1)| + |N(L(t-1))|/B\right)$ I/Os. Then the algorithm removes duplicates from the multi-set A. This can be done by sorting $A(t)$ according to the node indices, followed by a scan and compaction phase; hence, the duplicate elimination takes $\mathcal{O}(\text{sort}(|A(t)|)$ I/Os. The resulting set $A'(t)$ is still sorted.

Now the algorithm computes $L(t) := A'(t) \setminus \{L(t-1) \cup L(t-2)\}$. Fig. 4.1 provides an example. Filtering out the nodes already contained in the sorted lists $L(t-1)$ or $L(t-2)$ is possible by parallel scanning. Therefore, this step can be done using

$$\mathcal{O}\Big(\text{sort}\big(\ |N(L(t-1))|\ \big) + \text{scan}\big(\ |L(t-1)| + |L(t-2)|\ \big)\Big) \text{ I/Os.}$$

Since $\sum_t |N(L(t))| = \mathcal{O}(|E|)$ and $\sum_t |L(t)| = \mathcal{O}(|V|)$, the whole execution of MR_BFS requires $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os.

The correctness of this BFS algorithm crucially depends on the fact that the input graph is undirected. Assume that the levels $L(0), \ldots, L(t-1)$ have already been computed correctly. We consider a neighbor v of a node $u \in L(t-1)$: the distance from s to v is at least $t-2$ because otherwise the distance of u would be less than $t-1$. Thus $v \in L(t-2) \cup L(t-1) \cup L(t)$ and hence it is correct to assign precisely the nodes in $A'(t) \setminus \{L(t-1) \cup L(t-2)\}$ to $L(t)$.

Theorem 4.1 ([567]). *BFS on arbitrary undirected graphs can be solved using $\mathcal{O}(|V| + \text{sort}(|V| + |E|))$ I/Os.*

4.3.2 An Improved BFS Algorithm

The FAST_BFS algorithm of Mehlhorn and Meyer [542] refines the approach of Munagala and Ranade [567]. It trades-off unstructured I/Os with increasing the number of iterations in which an edge may be involved. FAST_BFS operates in two phases: in a first phase it preprocesses the graph and in a second phase it performs BFS using the information gathered in the first phase. We first sketch a variant with a randomized preprocessing. Then we outline a deterministic version.

The Randomized Partitioning Phase. The preprocessing step partitions the graph into disjoint connected subgraphs $\mathcal{S}_i$, $0 \leq i \leq K$, with small expected diameter. It also partitions the adjacency lists accordingly, i.e., it constructs an external file $\mathcal{F} = \mathcal{F}_0\mathcal{F}_1 \dots \mathcal{F}_i \dots \mathcal{F}_{K-1}$ where $\mathcal{F}_i$ contains the adjacency lists of all nodes in $\mathcal{S}_i$. The partition is built by choosing *master nodes* independently and uniformly at random with probability $\mu = \min\{1, \sqrt{(|V| + |E|)/(B \cdot |V|)}\}$ and running a local BFS from all master nodes "in parallel" (for technical reasons, the source node s is made the master node of $\mathcal{S}_0$): in each round, each master node s_i tries to capture all unvisited neighbors of its current sub-graph $\mathcal{S}_i$; this is done by first sorting the nodes of the active fringes of all $\mathcal{S}_i$ (the nodes that have been captured in the previous round) and then scanning the dynamically shrinking adjacency-lists representation of the yet unexplored graph. If several master nodes want to include a certain node v into their partitions then an arbitrary master node among them succeeds. The selection can be done by sorting and scanning the created set of neighbor nodes.

The expected number of master nodes is $K := \mathcal{O}(1 + \mu \cdot n)$ and the expected shortest-path distance (number of edges) between any two nodes of a subgraph is at most $2/\mu$. Hence, the expected total amount of data being scanned from the adjacency-lists representation during the "parallel partition growing" is bounded by

$$X := \mathcal{O}(\sum_{v \in V} 1/\mu \cdot (1 + \text{degree}(v))) = \mathcal{O}((|V| + |E|)/\mu).$$

The total number of fringe nodes and neighbor nodes sorted and scanned during the partitioning is at most $Y := \mathcal{O}(|V| + |E|)$. Therefore, the partitioning requires

$$\mathcal{O}(\text{scan}(X) + \text{sort}(Y)) = \mathcal{O}(\text{scan}(|V| + |E|)/\mu + \text{sort}(|V| + |E|))$$

expected I/Os.

After the partitioning phase each node knows the (index of the) subgraph to which it belongs. With a constant number of sort and scan operations FAST_BFS can reorganize the adjacency lists into the format $\mathcal{F}_0\mathcal{F}_1 \dots \mathcal{F}_i \dots \mathcal{F}_{|\mathcal{S}|-1}$, where $\mathcal{F}_i$ contains the adjacency lists of the nodes in partition $\mathcal{S}_i$; an entry $(v, w, \mathcal{S}(w), f_{\mathcal{S}(w)})$ from the adjacency list of $v \in \mathcal{F}_i$ stands for the edge (v, w) and provides the additional information that w belongs to subgraph $\mathcal{S}(w)$ whose subfile $\mathcal{F}_{\mathcal{S}(w)}$ starts at position $f_{\mathcal{S}(w)}$ within $\mathcal{F}$. The edge entries of each $\mathcal{F}_i$ are lexicographically sorted. In total, $\mathcal{F}$ occupies $\mathcal{O}((|V| + |E|)/B)$ blocks of external storage.

The BFS Phase. In the second phase the algorithm performs BFS as described by Munagala and Ranade (Section 4.3.1) with one crucial difference: FAST_BFS maintains an external file $\mathcal{H}$ (= hot adjacency lists); it comprises unused parts of subfiles $\mathcal{F}_i$ that contain a node in the current level $L(t-1)$. FAST_BFS initializes $\mathcal{H}$ with $\mathcal{F}_0$. Thus, initially, $\mathcal{H}$ contains the adjacency list of the root node s of level $L(0)$. The nodes of each created BFS level will also carry identifiers for the subfiles $\mathcal{F}_i$ of their respective subgraphs $\mathcal{S}_i$.

When creating level $L(t)$ based on $L(t-1)$ and $L(t-2)$, FAST_BFS does not access single adjacency lists like MR_BFS does. Instead, it performs a parallel scan of the sorted lists $L(t-1)$ and $\mathcal{H}$ and extracts $N(L(t-1))$; In order to maintain the invariant that $\mathcal{H}$ contains the adjacency lists of all vertices on the current level, the subfiles $\mathcal{F}_i$ of nodes whose adjacency lists are not yet included in $\mathcal{H}$ will be merged with $\mathcal{H}$. This can be done by first sorting the respective subfiles and then merging the sorted set with $\mathcal{H}$ using one scan. Each subfile $\mathcal{F}_i$ is added to $\mathcal{H}$ at most once. After an adjacency list was copied to $\mathcal{H}$, it will be used only for $\mathcal{O}(1/\mu)$ expected steps; afterwards it can be discarded from $\mathcal{H}$. Thus, the expected total data volume for scanning $\mathcal{H}$ is $\mathcal{O}(1/\mu \cdot (|V|+|E|))$, and the expected total number of I/Os to handle $\mathcal{H}$ and $\mathcal{F}_i$ is $\mathcal{O}\left(\mu \cdot |V| + \text{sort}(|V|+|E|) + 1/\mu \cdot \text{scan}(|V|+|E|)\right)$. The final result follows with $\mu = \min\{1, \sqrt{\text{scan}(|V|+|E|)/|V|}\}$.

Theorem 4.2 ([542]). *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \text{scan}(|V|+|E|)} + \text{sort}(|V|+|E|)\right)$ *expected I/Os.*

The Deterministic Variant. In order to obtain the result of Theorem 4.2 in the worst case, too, it is sufficient to modify the preprocessing phase of Section 4.3.2 as follows: instead of growing subgraphs around randomly selected master nodes, the deterministic variant extracts the subfiles $\mathcal{F}_i$ from an Euler tour [215] around a spanning tree for the connected component C_s that contains the source node s. Observe that C_s can be obtained with the deterministic connected-components algorithm of [567] using
$\mathcal{O}((1+\log\log(B \cdot |V|/|E|)) \cdot \text{sort}(|V|+|E|)) =$
$\mathcal{O}(\sqrt{|V| \cdot \text{scan}(|V|+|E|)} + \text{sort}(|V|+|E|))$ I/Os. The same number of I/Os suffices to compute a (minimum) spanning tree T_s for C_s [60].

After T_s has been built, the preprocessing constructs an Euler tour around T_s using a constant number of sort- and scan-steps [192]. Then the tour is broken at the root node s; the elements of the resulting list can be stored in consecutive order using the deterministic list ranking algorithm of [192]. This takes $\mathcal{O}(\text{sort}(|V|))$ I/Os. Subsequently, the Euler tour can be cut into pieces of size $2/\mu$ in a single scan. These Euler tour pieces account for subgraphs $\mathcal{S}_i$ with the property that the distance between any two nodes of $\mathcal{S}_i$ in G is at most $2/\mu - 1$. See Fig. 4.2 for an example. Observe that a node v of degree d may be part of $\Theta(d)$ different subgraphs $\mathcal{S}_i$. However, with a constant number of sorting steps it is possible to remove multiple node appearances and make sure that each node of C_s is part of exactly one subgraph $\mathcal{S}_i$ (actually there

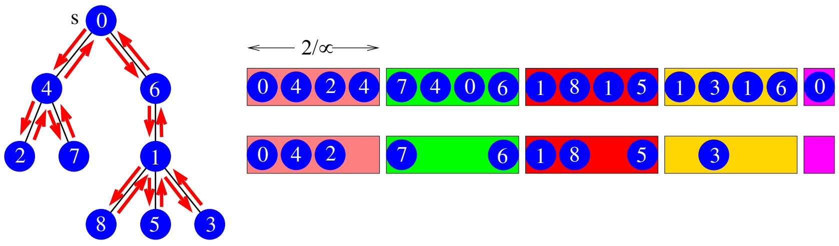


Fig. 4.2. Using an Euler tour around a spanning tree of the input graph in order to obtain a partition for the deterministic BFS algorithm.

are special algorithms for duplicate elimination, e.g. [1, 534]). Eventually, the reduced subgraphs $\mathcal{S}_i$ are used to create the reordered adjacency-list files $\mathcal{F}_i$; this is done as in the randomized preprocessing and takes another $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os. Note that the reduced subgraphs $\mathcal{S}_i$ may not be connected any more; however, this does not matter as our approach only requires that any two nodes in a subgraph are relatively close in the original input graph.

The BFS-phase of the algorithm remains unchanged; the modified preprocessing, however, guarantees that each adjacency-list will be part of the external set $\mathcal{H}$ for at most $2/\mu$ BFS levels: if a subfile $\mathcal{F}_i$ is merged with $\mathcal{H}$ for BFS level $L(t)$, then the BFS level of any node v in $\mathcal{S}_i$ is at most $L(t) + 2/\mu - 1$. Therefore, the adjacency list of v in $\mathcal{F}_i$ will be kept in $\mathcal{H}$ for at most $2/\mu$ BFS levels.

Theorem 4.3 ([542]). *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \text{scan}(|V| + |E|)} + \text{sort}(|V| + |E|)\right)$ *I/Os in the worst case.*

4.4 I/O-Efficient Tournament Trees

In this section we review a data structure due to Kumar and Schwabe [485] which proved helpful in the design of better **EM** graph algorithms: the I/O-efficient tournament tree, *I/O-TT* for short. A tournament tree is a complete binary tree, where some rightmost leaves may be missing. In a figurative sense, a standard tournament tree models the outcome of a k-phase knock-out game between $|V| \leq 2^k$ players, where player i is associated with the i-th leaf of the tree; winners move up in the tree.

The I/O-TT as described in [485] is more powerful: it works as a priority queue with the Decrease_Key operation. However, both the size of the data structure and the I/O-bounds for the priority queue operations depend on the size of the universe from which the entries are drawn. Used in connection with graph algorithms, the static I/O-TT can host at most $|V|$ elements with

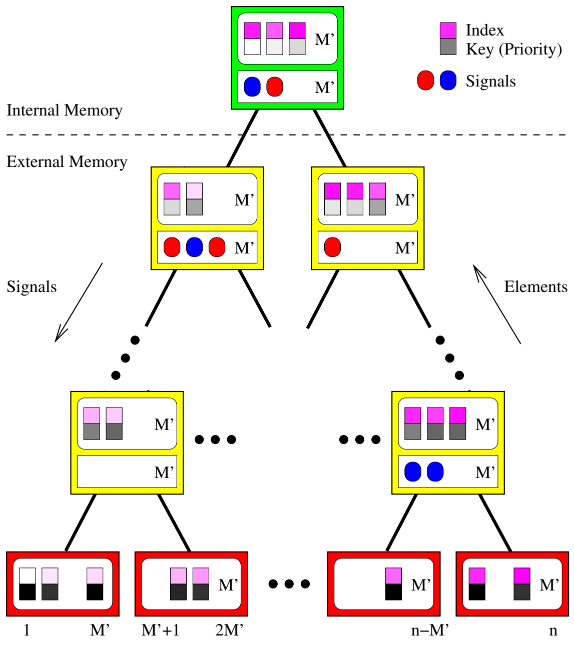


Fig. 4.3. Principle of an I/O-efficient tournament tree. Signals are traveling from the root to the leaves; elements move in opposite direction.

pairwise disjoint indices in $\{1, \ldots, |V|\}$. Besides its index x, each element also has a key k (priority). An element $\langle x_1, k_1 \rangle$ is called smaller than $\langle x_2, k_2 \rangle$ if $k_1 < k_2$.
The I/O-TT supports the following operations:

(i) *deletemin:* extract the element $\langle x, k \rangle$ with smallest key k and replace it by the new entry $\langle x, \infty \rangle$.
(ii) *delete(x):* replace $\langle x, \mathit{oldkey} \rangle$ by $\langle x, \infty \rangle$.
(iii) *update(x,newkey):* replace $\langle x, \mathit{oldkey} \rangle$ by $\langle x, \mathit{newkey} \rangle$ if $\mathit{newkey} < \mathit{oldkey}$.

Note that (ii) and (iii) do not require the old key to be known. This feature will help to implement the graph-traversal algorithms of Section 4.5 without paying one I/O for each edge (for example an SSSP algorithm does not have to find out explicitly whether an edge relaxation leads to an improved tentative distance).

Similar to other I/O-efficient priority queue data structures (see Chapter 2 of Rasmus Pagh for an overview) I/O-TTs rely on the concept of lazy batched processing. Let $M' = c \cdot M$ for some positive constant $c < 1$; the static I/O-TT for $|V|$ entries only has $\lceil |V|/M' \rceil$ leaves (instead of $|V|$ leaves in the standard tournament tree). Hence, there are $\mathcal{O}(\log_2(|V|/M))$ levels. Elements with indices in the range $\{(i-1) \cdot M' + 1, \ldots, i \cdot M'\}$ are mapped to the i-th leaf. The index range of internal nodes of the I/O-TT is given by the union of the index ranges of their children. Internal nodes of the I/O-TT keep a list of at least $M'/2$ and at most M' elements each (sorted according to their priorities). If the list of a tree node v contains z elements, then they are the smallest z out of all those elements in the tree being mapped to the leaves that are descendants of v. Furthermore, each internal node is equipped with a *signal buffer* of size M'. Initially, the I/O-TT stores the elements $\langle 1, +\infty \rangle, \langle 2, +\infty \rangle, \ldots, \langle |V|, +\infty \rangle$, out of which the lists of internal nodes keep at least $M'/2$ elements each. Fig. 4.3 illustrates the principle of an I/O-TT.

4.4.1 Implementation of the Operations

The operations (i)–(iii) generate *signals* which serve to propagate information down the tree; signals are inserted into the root node, which is kept in internal memory. When a signal arrives in a node it may create, delete or modify an element kept in this node; the signal itself may be discarded, altered or remain unchanged. Non-discarded signals are stored until the capacity of the node's buffer is exceeded; then they are sent down the tree towards the unique leaf node its associated element is mapped to.

Operation (i) removes an element $\langle x, k \rangle$ with smallest key k from the root node (in case there are no elements in the root it is recursively refilled from its children). A signal is sent on the path towards the leaf associated with x in order to reinsert $\langle x, +\infty \rangle$. The reinsertion takes place at the first tree node on the path with free capacity whose descendents are either empty or exclusively host elements with key infinity.

Operation (ii) is done in a similar way as (i); a delete signal is sent towards the leaf node that index x is mapped to; the signal will eventually meet the element $\langle x, \mathit{oldkey} \rangle$ and cause its deletion. Subsequently, the delete signal is converted into a signal to reinsert $\langle x, +\infty \rangle$ and proceeds as in case (i).

Finally, the signal for operation (iii) traverses its predefined tree path until either some node v_{newkey} with appropriate key range is found or the element $\langle x, \mathit{oldkey} \rangle$ is met in some node v_{oldkey}. In the latter case, if $\mathit{oldkey} > \mathit{newkey}$ then $\langle x, \mathit{newkey} \rangle$ will replace $\langle x, \mathit{oldkey} \rangle$ in the list of v_{oldkey}; if $\mathit{oldkey} \leq \mathit{newkey}$ nothing changes. Otherwise, i.e., if $\langle x, \mathit{newkey} \rangle$ belongs to a tree node v_{newkey} closer to the root than v_{oldkey}, then $\langle x, \mathit{newkey} \rangle$ will be added to the list of v_{newkey} (in case this exceeds the capacity of v_{newkey} then the largest list element is recursively flushed to the respective child node of v_{newkey} using a special flush signal). The update signal for *update(x,newkey)* is altered into a delete signal for $\langle x, \mathit{oldkey} \rangle$, which is sent down the tree in order to eliminate the obsolete entry for x with the old key.

It can be observed that each operation from (i)–(iii) causes at most two signals to travel all the way down to a leaf node. Overflowing buffers with $X > M'$ signals can be emptied using $\mathcal{O}(X/B)$ I/Os. Elements moving up the tree can be charged to signals traveling down the tree. Arguing more formally along these lines, the following amortized bound can be shown:

Theorem 4.4 ([485]). *On an I/O-efficient tournament tree with $|V|$ elements, any sequence of z delete/deletemin/update operations requires at most $\mathcal{O}(z/B \cdot \log_2(|V|/B))$ I/Os.*

4.5 Undirected SSSP with Tournament Trees

In the following we sketch how the I/O-efficient tournament tree of Section 4.4 can be used in order to obtain improved **EM** algorithms for the single source shortest path problem. The basic idea is to replace the data structure Q for the candidate nodes of **IM** traversal-algorithms (Section 4.2) by the **EM** tournament tree. The resulting SSSP algorithm works for undirected graphs with strictly positive edge weights.

The SSSP algorithm of [485] constructs an I/O-TT for the $|V|$ vertices of the graph and sets all keys to infinity. Then the key of the source node is updated to zero. Subsequently, the algorithm operates in $|V|$ iterations similarly to Dijkstra's approach [252]: iteration i first performs a *deletemin* operation in order to extract an element $\langle v_i, k_i \rangle$; the final distance of the extracted node v_i is given by $\text{dist}(v_i) = k_i$. Then the algorithm issues $\textit{update}(w_j, \text{dist}(v_i) + c(v_i, w_j))$ operations on the I/O-TT for each adjacent edge (v_i, w_j), $v_i \neq w_j$, having weight $c(v_i, w_j)$; in case of improvements the new tentative distances will automatically materialize in the I/O-TT.

However, there is a problem with this simple approach; consider an edge (u, v) where $\text{dist}(u) < \text{dist}(v)$. By the time v is extracted from the I/O-TT, u is already settled; in particular, after removing u, the I/O-TT replaces the extracted entry $\langle u, \text{dist}(u) \rangle$ by $\langle u, +\infty \rangle$. Thus, performing $\textit{update}(u, \text{dist}(v) + c(v, u) < \infty)$ for the edge (v, u) after the extraction of v would reinsert the settled node u into the set Q of candidate nodes. In the following we sketch how this problem can be circumvented:

A second **EM** priority-queue[3], denoted by *SPQ*, supporting a sequence of z *deletemin* and *insert* operations with (amortized) $\mathcal{O}(z/B \cdot \log_2(z/B))$ I/Os is used in order to remember settled nodes "at the right time". Initially, SPQ is empty. At the beginning of iteration i, the modified algorithm additionally checks the smallest element $\langle u'_i, k'_i \rangle$ from SPQ and compares its key k'_i with the key k_i of the smallest element $\langle u_i, k_i \rangle$ in I/O-TT. Subsequently, only the element with smaller key is extracted (in case of a tie, the element in the I/O-TT is processed first). If $k_i < k'_i$ then the algorithm proceeds as described above; however, for each $\textit{update}(v, \text{dist}(u) + c(u, v))$ on the I/O-TT it additionally inserts $\langle u, \text{dist}(u) + c(u, v) \rangle$ into the SPQ. On the other hand, if $k'_i < k_i$ then a $\textit{delete}(u'_i)$ operation is performed on I/O-TT as well and a new phase starts.

[3] Several priority queue data structures are appropriate; see Chapter 2 for an overview.

Operation	I/O-TT	SPQ
...	$\langle u, \mathrm{dist}(u)\rangle$, $\langle v, *\rangle$	
$u = $ *TT_deletemin()*	$\langle v, *\rangle$	
TT_update($v, \mathrm{dist}(u) + c(u, v)$*)*	$\langle v, *'\rangle$	
SPQ_insert($u, \mathrm{dist}(u) + c(u, v)$*)*	$\langle v, *'\rangle$	$\langle u, \mathrm{dist}(u) + c(u, v)\rangle$
...	$\langle v, \mathrm{dist}(v)\rangle$	$\langle u, \mathrm{dist}(u) + c(u, v)\rangle$
$v = $ *TT_deletemin()*		$\langle u, \mathrm{dist}(u) + c(u, v)\rangle$
TT_update($u, \mathrm{dist}(v) + c(u, v)$*)*	$\langle u, \mathrm{dist}(v) + c(u, v)\rangle$	$\langle u, \mathrm{dist}(u) + c(u, v)\rangle$
...	$\langle u, \mathrm{dist}(v) + c(u, v)\rangle$	$\langle u, \mathrm{dist}(u) + c(u, v)\rangle$
$u = $ *SPQ_deletemin()*	$\langle u, \mathrm{dist}(v) + c(u, v)\rangle$	
TT_delete(u*)*		

Fig. 4.4. Identifying spurious entries in the I/O-TT with the help of a second priority queue SPQ.

In Fig. 4.4 we demonstrate the effect for the previously stated problem concerning an edge (u, v) with $\mathrm{dist}(u) < \mathrm{dist}(v)$: after node u is extracted from the I/O-TT for the first time, $\langle u, \mathrm{dist}(u) + c(u, v)\rangle$ is inserted into SPQ. Since $\mathrm{dist}(u) < \mathrm{dist}(v) \leq \mathrm{dist}(u) + c(u, v)$, node v will be extracted from I/O-TT while u is still in SPQ. The extraction of v triggers a spurious reinsertion of u into I/O-TT having key $\mathrm{dist}(v) + c(v, u) = \mathrm{dist}(v) + c(u, v) > \mathrm{dist}(u) + c(u, v)$. Thus, u is extracted as the smallest element in SPQ before the re-inserted node u becomes the smallest element in I/O-TT; as a consequence, the resulting *delete*(u) operation for I/O-TT eliminates the spurious node u in I/O-TT just in time. Extra rules apply for nodes with identical shortest path distances.

As already indicated in Section 4.2, one-pass algorithms like the one just presented still require $\Theta(|V| + (|V| + |E|)/B)$ I/Os for accessing the adjacency lists. However, the remaining operations are more I/O-efficient: $\mathcal{O}(|E|)$ operations on the I/O-TT and SPQ add another $\mathcal{O}(|E|/B \cdot \log_2(|E|/B))$ I/Os. Altogether this amounts to $\mathcal{O}(|V| + |E|/B \cdot \log_2(|E|/B))$ I/Os.

Theorem 4.5. *SSSP on undirected graphs can be solved using* $\mathcal{O}(|V| + |E|/B \cdot \log_2(|E|/B))$ *I/Os.*

The unpublished full version of [485] also provides a one-pass **EM** algorithm for DFS on undirected graphs. It requires $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ I/Os. A different algorithm for directed graphs achieving the same bound will be sketched in the next section.

4.6 Graph-Traversal in Directed Graphs

The best known one-pass traversal-algorithms for general directed graphs are often less efficient and less appealing than their undirected counterparts from the previous sections. The key difference is that it becomes much more complicated to keep track of previously visited nodes of the graph; the nice trick of checking a constant number of previous levels for visited nodes as discussed for undirected BFS does not work for directed graphs. Therefore we store edges that point to previously seen nodes in a so-called *buffered repository tree* (BRT) [164]: A BRT maintains $|E|$ elements with keys in $\{1, \ldots, |V|\}$ and supports the operations $insert(edge, key)$ and $extract\_all(key)$; the latter operation reports and deletes all edges in the BRT that are associated with the specified key.

A BRT can be built as a height-balanced static binary tree with $|V|$ leaves and buffers of size B for each internal tree node; leaf i is associated with graph node v_i and stores up to degree(v_i) edges. Insertions into the BRT happen at the root; in case of buffer overflow an inserted element (e, i) is flushed down towards the i-th leaf. Thus, an insert operation requires amortized $\mathcal{O}(1/B \cdot \log_2 |V|)$ I/Os. If $extract\_all(i)$ reports x edges then it needs to read $\mathcal{O}(\log_2 |V|)$ buffers on the path from the root to the i-th leaf; another $\mathcal{O}(x/B)$ disk blocks may have to be read at the leaf itself. This accounts for $\mathcal{O}(x/B + \log_2 |V|)$ I/Os.

For DFS, an external stack S is used to store the vertices on the path from the root node of the DFS tree to the currently visited vertex. A step of the DFS algorithm checks the previously unexplored outgoing edges of the topmost vertex u from S. If the target node v of such an edge (u, v) has not been visited before then u is the father of v in the DFS tree. In that case, v is pushed on the stack and the search continues for v. Otherwise, i.e., if v has already been visited before, the next unexplored outgoing edge of u will be checked. Once all outgoing edges of the topmost node u on the stack have been checked, node u is popped and the algorithm continues with the new topmost node on the stack.

Using the BRT the DFS procedure above can be implemented I/O efficiently as follows: when a node v is encountered for the first time, then for each incoming edge $e_i = (u_i, v)$ the algorithm performs $insert(e_i, u_i)$. If at some later point u_i is visited then $extract\_all(u_i)$ provides a list of all edges out of u_i that should not be traversed again (since they lead to nodes already seen before). If the (ordered) adjacency list of u_i is kept in some **EM** priority-queue $P(u_i)$ then all superfluous edges can be deleted from $P(u_i)$ in an I/O-efficient way. Subsequently, the next edge to follow is given by extracting the minimum element from $P(u_i)$.

The algorithm takes $\mathcal{O}(|V| + |E|/B)$ I/Os to access adjacency lists. There are $\mathcal{O}(|E|)$ operations on the n priority queues $P(\cdot)$ (implemented as external buffer trees). As the DFS algorithm performs an inorder traversal of a DFS tree, it needs to change between different $P(\cdot)$ at most $\mathcal{O}(|V|)$ times. Therefore, $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os suffice to handle the operations on all $P(\cdot)$. Additionally, there are $\mathcal{O}(|E|)$ *insert* and $\mathcal{O}(|V|)$ *extract_all* operations on the BRT; the I/Os required for them add up to $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ I/Os.

The algorithm for BFS works similarly, except that the stack is replaced by an external queue.

Theorem 4.6 ([164, 485]). *BFS and DFS on directed graphs can be solved using* $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ *I/Os.*

4.7 Conclusions and Open Problems for Graph Traversal

In the previous sections we presented the currently best **EM** traversal algorithms for general graphs assuming a single disk. With small modifications, the results of Table 4.1 can be obtained for $D \geq 1$ disks.

Table 4.1. Graph traversal with D parallel disks.

Problem	I/O-Bound
Undir. BFS	$\mathcal{O}\left(\sqrt{\|V\| \cdot \text{scan}(\|V\| + \|E\|)} + \text{sort}(\|V\| + \|E\|)\right)$
Dir. BFS, DFS	$\mathcal{O}\left(\min\left\{\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{scan}(\|V\| + \|E\|),\ \left(\|V\| + \frac{\|E\|}{D \cdot B}\right) \cdot \log_2 \|V\|\right\}\right)$
Undir. SSSP	$\mathcal{O}\left(\min\left\{\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{sort}(\|V\| + \|E\|),\ \|V\| + \frac{\|E\|}{D \cdot B} \cdot \log_2 \|V\|\right\}\right)$
Dir. SSSP	$\mathcal{O}\left(\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{sort}(\|V\| + \|E\|)\right)$

One of the central goals in **EM** graph-traversal is the reduction of unstructured I/O for accessing adjacency-lists. The FAST_BFS algorithm of Section 4.3.2 provides a first solution for the case of undirected BFS. However, it also raises new questions: For example, is $\Omega(|V|/\sqrt{D \cdot B})$ I/Os a lower bound for sparse graphs? Can similar results be obtained for DFS or SSSP with arbitrary nonnegative edge weights? In the following we shortly discuss difficulties concerning possible extensions towards *directed* BFS.

The improved I/O-bound of FAST_BFS stems from partitioning the undirected input graph G into disjoint node sets $\mathcal{S}_i$ such that the distance in G between any two nodes of $\mathcal{S}_i$ is relatively small. For directed graphs, a partitioning of that kind may not always exist. It could be beneficial to have a larger number of overlapping node sets where the algorithm accesses just a small fraction of them. Similar ideas underlie a previous BFS algorithm for undirected graphs with small node degrees [550]. However, there are deep problems concerning space blow-up and efficiently identifying the "right" partitioning for arbitrary directed graphs. Besides all that, an $o(|V|)$-I/O algorithm for directed BFS must also feature novel strategies to remember previously visited nodes. Maybe, for the time being, this additional complication should be left aside by restricting attention to the semi-external case; first results for semi-external BFS on directed Eulerian graphs are given in [542].

4.8 Graph Connectivity: Undirected CC, BCC, and MSF

The *Connected Components*(*CC*) problem is to create, for a graph $G = (V, E)$, a list of the nodes of the graph, sorted by component, with a special record marking the end of each component. The Connected Components Labeling (*CCL*) problem is to create a vector L of size $|V|$ such that for every $i, j \in \{1, \cdots, |V|\}$, $L[i] = L[j]$ iff nodes i and j are in the same component of G. A solution to CCL can be converted into a CC output in $\mathcal{O}(\text{sort}(|V|))$ I/Os, by sorting the nodes according to their component label and adding the separators.

Minimum Spanning Forest (*MSF*) is the problem of finding a subgraph F of the input graph G such that every connected component in G is connected in F and the total weight of the edges of F is minimal.

Biconnected Components (*BCC*) is the problem of finding subsets of the nodes such that u and v are in the same subset iff there are two node-disjoint paths between u and v.

CC and MSF are related, as an MSF yields the connected components of the graph. Another common point is that typical algorithms for both problems involve reading the adjacency lists of nodes. Reading a node's adjacency list L takes $\mathcal{O}(\text{scan}(|L|))$ I/Os, so going over the nodes in an arbitrary order and reading each node's adjacency list once requires a total of $\mathcal{O}(\text{scan}(|E|) + |V|)$ I/Os. If $|V| \leq |E|/B$, the scan$(|E|)$ term dominates.

For both CC and MST, we will see algorithms that have this $|V|$ term in their complexity. Before using such an algorithm, a *node reduction* step will be applied to reduce the number of nodes to at most $|E|/B$. Then, the $|V|$ term is dominated by the other terms in the complexity. A node reduction step should have the property that the transformations it performs on the graph preserve the solutions to the problem in question. It should also be possible to reintegrate the nodes or edges that were removed into the solution of the problem for the reduced graph. For both CC and MST, the node reduction step will apply a small number of phases of an iterative algorithm for the problem. We could repeat such phases until completion, but that would require $\Theta(\log\log|V|)$ phases, each of which is quite expensive. The combination of a small number, $\mathcal{O}(\log\log(|V|B/|E|))$, of phases with an efficient algorithm for dense graphs, gives a better complexity. Note that when $|V| \leq |E|$, $\mathcal{O}(\text{sort}(|E|)\log\log(|V|B/|E|)) = \mathcal{O}(\text{sort}(|E|)\log\log B)$.

For BCC, we show an algorithm that transforms the graph and then applies CC. BCC is clearly not easier than CC; given an input $G = (E, V)$ to CC, we can construct a graph G' by adding a new node and connecting it with each of the nodes of G. Each biconnected component of G' is the union of a connected component of G with the new node.

4.9 Connected Components

The undirected BFS algorithm of Section 4.3.1 can trivially be modified to solve the CCL problem: whenever $L(t-1)$ is empty, the algorithm has spanned a complete component and selects some unvisited node for $L(t)$. It can therefore number the components, and label each node in $L(t)$ with the current component number before $L(t)$ is discarded. This adds another $\mathcal{O}(|V|)$ I/Os, so the complexity is still $\mathcal{O}(|V| + \text{sort}(|V| + |E|))$. For a dense graph, where $|V| \leq |E|/B$, we have $\mathcal{O}(|V| + \text{sort}(|V| + |E|)) = \mathcal{O}(\text{sort}(|V| + |E|))$. For a general graph, Munagala and Ranade [567] suggest to precede the BFS run with the following node reduction step:

The idea is to find, in each phase, sets of nodes such that the nodes in each set belong to the same connected component. Among each set a leader is selected, all edges between nodes of the set are contracted and the set is reduced to its leader. The process is then repeated on the reduced graph.

Algorithm 1. Repeat until $|V| \leq |E|/B$:

1. For each node, select a neighbor. Assuming that each node has a unique integer node-id, let this be the neighbor with the smallest id. This partitions the nodes into *pseudotrees* (directed graphs where each node has outdegree 1).
2. Select a leader from each pseudotree.
3. Replace each edge (u, v) in E by an edge $(R(u), R(v))$, where $R(v)$ is the leader of the pseudotree v belongs to.
4. Remove isolated nodes, parallel edges and self loops.

For Step 1 we sort two copies of the edges, one by source node and one by target node, and scan both lists simultaneously to find the lowest numbered neighbor of each node.

For step 2 we note that a pseudotree is a tree with one additional edge. Since each node selected its smallest neighbor, the smallest node in each pseudotree must be on the pseudotree's single cycle. In addition, this node can be identified by the fact that the edge selected for it goes to a node with a higher ID. Hence, we can scan the list of edges of the pseudoforest, remove each edge for which the source is smaller than the target, and obtain a forest while implicitly selecting the node with the smallest id of each pseudotree as the leader. On a forest, Step 3 can be implemented by pointer doubling as in the algorithm for list ranking in Chapter 3 with $\mathcal{O}(\text{sort}(|E|))$ I/Os. Step 4

requires an additional constant number of sorts and scans of the edges. The total I/Os for one iteration is then $\mathcal{O}(\text{sort}(|E|))$. Since each iteration at least halves the number of nodes, $\log_2(|V|B/|E|)$ iterations are enough, for a total of $\mathcal{O}(\text{sort}(|E|)\log(|V|B/|E|))$ I/Os.
After $\log_2(|V|B/|E|)$ iterations, we have a contracted graph in which each node represents a set of nodes from the original graph. Applying BFS on the contracted graph gives a component label to each supernode. We then need to go over the nodes of the original graph and assign to each of them the label of the supernode it was contracted to. This can be done by sorting the list of nodes by the id of the supernode that the node was contracted to and the list of component labels by supernode id, and then scanning both lists simultaneously.
The complexity can be further improved by contracting more edges per phase at the same cost. More precisely, in phase i up to $\sqrt{S_i}$ edges adjacent to each node will be contracted, where $S_i = 2^{(3/2)^i}\left(= S_{i-1}^{3/2}\right)$ (less edges will be contracted only if some nodes have become singletons, in which case they become inactive). This means that the number of active nodes at the beginning of phase i, $|V_i|$, is at most $|V|/S_{i-1} \cdot S_{i-2} \leq |V|/(S_i)^{2/3}(S_i)^{4/9} \leq |V|/S_i$, and $\log\log(|V|B/|E|)$ phases are sufficient to reduce the number of nodes as desired.

To stay within the same complexity per phase, phase i is executed on a reduced graph G_i, which contains only the relevant edges: those that will be contracted in the current phase. Then $|E_i| \leq |V_i|\sqrt{S_i}$. We will later see how this helps, but first we describe the algorithm:

Algorithm 2. Phase i:

1. For each active node, select up to $d = \sqrt{S_i}$ adjacent edges (less if the node's degree is smaller). Generate a graph $G_i = (V_i, E_i)$ over the active nodes with the selected edges.
2. Apply $\log d$ phases of Algorithm 1 to G_i.
3. Replace each edge (u, v) in E by an edge $(R(u), R(v))$ and remove redundant edges and nodes as in Algorithm 1.

Complexity Analysis. Steps 1 an 3 take $\mathcal{O}(\text{sort}(|E|))$ I/Os as in Algorithm 1. In Step 2, each phase of Algorithm 1 takes $\mathcal{O}(\text{sort}(|E_i|))$ I/Os. With $|E_i| \leq |V_i|\sqrt{S_i}$ (due to Step 1) and $|V_i| \leq (|V|/S_i)$ (as shown above), this is $\mathcal{O}\big(\text{sort}(|V|/\sqrt{S_i})\big)$ and all $\log\sqrt{S_i}$ phases need a total of $\mathcal{O}(\text{sort}(|V|))$ I/Os. Hence, one phase of Algorithm 2 needs $\mathcal{O}(\text{sort}(|E|))$ I/Os as before, giving a total complexity of $\mathcal{O}(\text{sort}(|E|)\log\log(|V|B/|E|))$ I/Os for the node reduction. The BFS-based CC algorithm can then be executed in $\mathcal{O}(\text{sort}(|E|))$ I/Os.

For $D > 1$, we perform node reduction phases until $|V| \leq |E|/BD$, giving:

Theorem 4.7 ([567]). *CC can be solved using*

$$\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \text{ I/Os.}$$

4.10 Minimum Spanning Forest

An $\mathcal{O}(|V| + \text{sort}(|E|))$ MSF algorithm. The Jarník-Prim [428, 615] MSF algorithm builds the MSF incrementally, one tree at a time. It starts with an arbitrary node and in each iteration, finds the node that is connected to the current tree by the lightest edge, and adds it to the tree. When there do not exist any more edges connecting the current tree with unvisited nodes, it means that a connected component of the graph has been spanned. The algorithm then selects an arbitrary unvisited node and repeats the process to find the next tree of the forest. For this purpose, the nodes which are not in the MSF are kept in a priority queue, where the priority of a node is the weight of the lightest edge that connects it to the current MSF, and ∞ if such an edge does not exist. Whenever a node v is added to the MSF, the algorithm looks at v's neighbors. For each neighbor u which is in the queue, the algorithm compares the weight of the edge (u, v) with the priority of u. If the priority is higher than the weight of the new edge that connects u to the tree, u's priority is updated. In **EM**, this algorithm requires at least one I/O per edge to check the priority of a node, hence the number of I/Os $\Theta(|E|)$.

However, Arge et al. [60] pointed out that if *edges* are kept in the queue instead of nodes, the need for updating priorities is eliminated. During the execution of the algorithm, the queue contains (at least) all edges that connect nodes in the current MSF with nodes outside of it. It can also contain edges between nodes that are in the MSF. The algorithm proceeds as follows: repeatedly perform *extract minimum* to extract the minimum weight edge (u, v) from the queue. If v is already in the MSF, the edge is discarded. Otherwise, v is included in the MSF and all edges incident to it, except for (u, v), are inserted into the queue. Since each node in the tree inserts all its adjacent edges, if v is in the MSF, the queue contains two copies of the edge (u, v). Assuming that all edge weights are distinct, after the first copy was extracted from the queue, the second copy is the new minimum. Therefore, the algorithm can use one more *extract-minimum* operation to know whether the edge should be discarded or not.

The algorithm reads each node's adjacency list once, with $\mathcal{O}(|V| + |E|/B)$ I/Os. It also performs $\mathcal{O}(|E|)$ *insert* and *extract minimum* operations on the queue. By using an external memory priority queue that supports these operations in $\mathcal{O}\Big(1/B \log_{M/B}(N/B)\Big)$ I/Os amortized, we get that the total complexity is $\mathcal{O}(|V| + \text{sort}(|E|))$.

Node Reduction for MSF. As in the CC case, we wish to reduce the number of nodes to at most $|E|/B$. The idea is to contract edges that are in the MSF, merging the adjacent nodes into supernodes. The result is a reduced graph G' such that the MSF of the original graph is the union of the contracted edges and the edges of the MSF of G', which can be found recursively.

A *Boruvka* [143] step selects for each node the minimum-weight edge incident to it. It contracts all the selected edges, replacing each connected component they define by a supernode that represents the component, removing all isolated nodes, self-edges, and all but the lowest weight edge among each set of multiple edges.

Each Boruvka step reduces the number of nodes by at least a factor of two, while contracting only edges that belong to the MSF. After i steps, each supernode represents a set of at least 2^i original nodes, hence the number of supernodes is at most $|V|/2^i$. In order to reduce the number of nodes to $|E|/B$, $\lceil \log(|V|B/|E|) \rceil$ phases are necessary. Since one phase requires $\mathcal{O}(\text{sort}(|E|))$ I/Os, this algorithm has complexity of $\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log(|V|B/|E|))\})$.

As with the CC node reduction algorithm, this can be improved by combining phases into superphases, where each superphase still needs $\mathcal{O}(\text{sort}(|E|))$ I/Os, and reduces more nodes than the basic step. Each superphase is the same, except that the edges selected for E_i are not the smallest numbered $\sqrt{S_i}$ edges adjacent to each node, but the *lightest* $\sqrt{S_i}$ edges. Then a superphase which is equivalent to $\log \sqrt{s}_i$ *Boruvka* steps is executed with $\mathcal{O}(\text{sort}(|E|))$ I/Os. The total number of I/Os for node reduction is $\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|B/|E|)\})$. The output is the union of the edges that were contracted in the node reduction phase and the MSF of the reduced graph.

Theorem 4.8 ([567]). *MSF can be solved using*

$$\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \textit{ I/Os.}$$

4.11 Randomized *CC* and *MSF*

We now turn to the randomized MSF algorithm that was proposed by Abello, Buchsbaum and Westbrook [1], which is based on an internal memory algorithm that was found by Karger, Klein and Tarjan [445]. As noted above, an MSF algorithm is also a CC algorithm.

Let $G = (V, E)$ be the input graph with edge weights $c(e)$. Let $E' \subseteq E$ be an arbitrary subset of the edges of G and let F' be the MSF of $G' = (V, E')$. Denote by $W_{F'}(u, v)$ the weight of the heaviest edge on the path connecting u and v in F'. If such a path does not exist, $W_{F'}(u, v) = \infty$.

Lemma 4.9. *For any edge $e = (u, v) \in E$, if $w(e) \geq W_{F'}(u, v)$, then there exists an MSF that does not include e.*

The lemma follows immediately from the *cycle property*: the heaviest edge of each cycle is not in the MSF. □

The following MSF algorithm is based on Lemma 4.9 [445]:

1. Apply two Boruvka phases to reduce the number of nodes by at least a factor of 4. Call the contracted graph G'.

2. Choose a subgraph H of G' by selecting each edge independently with probability p.
3. Apply the algorithm recursively to find the MSF F of H.
4. Delete from G' each edge $e = (u, v)$ for which $w(e) > W_{F'}(u, v)$. Call the resulting graph G''.
5. Apply the algorithm recursively to find the MSF F' of G''.
6. return the union of the edges contracted in step 1 and the edges of F'.

Step 1 requires $\mathcal{O}(\text{sort}(|E|))$ I/Os and Step 2 an additional $\mathcal{O}(\text{scan}(|E|))$ I/Os.
Step 4 can be done with $\mathcal{O}(\text{sort}(|E|))$ I/Os as well [192]. The idea is based on the MST verification algorithm of King [456], which in turn uses a simplification of Komlós's algorithm for finding the heaviest weight edge on a path in a tree [464]. Since Komlós's algorithm only works on full branching trees[4], the MSF F' is first converted into a forest of full branching trees F'' such that $W_{F'}(u, v) = W_{F''}(u, v)$ and F'' has $\mathcal{O}(|V|)$ nodes. The conversion is done as follows: each node of F' is a leaf in F''. A sequence of Boruvka steps is applied to F' and each step defines one level of each tree in F'': the nodes at distance i from the leaves are the supernodes that exist after i Boruvka steps and the parent of each non-root node is the supernode into which is was contracted. Since contraction involves at least two nodes, the degree of each non-leaf node is at least 2. All Boruvka steps require a total of $\mathcal{O}(\text{sort}(|V|))$ I/Os.

The next step is to calculate, for each edge (u, v) in G', the least common ancestor (LCA) of u and v in F''. Chiang et al. show that this can be done with $\mathcal{O}(\text{sort}(|E|))$ I/Os [192]. An additional $\mathcal{O}(\text{sort}(|E|))$ I/Os are necessary to construct a list of tuples, one for each edge of G', of the weight of the edge, its endpoints and the LCA of the endpoints. These tuples are then filtered through F'' with $\mathcal{O}((|E|/|V|)\text{sort}(|V|)) = \mathcal{O}(\text{sort}(|E|))$ I/Os as follows: each tuple is sent as a query to the root of the tree and traverses down from it towards the leaves that represent the endpoints of the edge. When the query reaches the LCA, it is split into two weight-queries, one continuing towards each of the endpoints. If a weight query traverses a tree edge which is heavier than the query edge, the edge is not discarded. Otherwise, it is. To make this I/O efficient, queries are passed along tree paths using batch filtering [345]. This means that instead of traversing the tree from root to leaves for each query, a batch of $|V|$ queries is handled at each node before moving on to the next.

The non-recursive stages then require $\mathcal{O}(\text{sort}(|E|))$ I/Os. Karger et al., have shown that the expected number of edges in G'' is $|V|/p$ [445]. The algorithm includes two recursive calls. One with at most $|V|/4$ nodes and expected $p|E|$ edges, and the other with at most $|V|/4$ nodes and expected $|V|/p$ edges. The total expected I/O complexity is then:

[4] A full branching tree is a tree in which all leaves are at the same level and each non-leaf node has at least two descendants

$$t(|E|,|V|) \leq \mathcal{O}(\mathrm{sort}(|E|)) + t\left(p|E|, \frac{|V|}{4}\right) + t\left(\frac{|V|}{p}, \frac{|V|}{4}\right) = \mathcal{O}(\mathrm{sort}(|E|))$$

Theorem 4.10 ([1]). *MSF and CC can be solved by a randomized algorithm that uses $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os in the expected case.*

4.12 Biconnected Components

Tarjan and Vishkin [714] propose a parallel algorithm that reduces *BCC* to an instance of *CC*. The idea is to transform the graph G into a graph G' such that the connected components of G' correspond to the biconnected components of G. Each node of G' represents an edge of G. If two edges e_1 and e_2 are on the same simple cycle in G, the edge (e_1, e_2) exists in G'. The problem with this construction is that the graph G' is very large; it has $|E|$ nodes and up to $\Omega\left(|E|^2\right)$ edges. However, they show how to generate a smaller graph G'' with the desired properties: instead of including all edges of G', first find (any) rooted spanning tree T of G and generate G'' as the the subgraph of G' induced by the edges of T. Formally, we say that two nodes of T are *unrelated* if neither is a predecessor of the other. G'' then includes the edges:

1. $\{(u,v),(x,w)\}$ such that $(u,v),(x,w) \in T$, $(v,w) \in G - T$ and v, w are unrelated in T.
2. $\{(u,v),(v,w)\}$ such that $(u,v),(v,w) \in T$ and there exists an edge in G between a descendant of w and a non-descendant of v.

It is not difficult to see that the connected components of G'' correspond to biconnected components of G, and that the number of edges in G'' is $O(|E|)$.

Chiang et al. [192] adapt this algorithm to the I/O model. Finding an arbitrary spanning tree is obviously not more difficult than finding an MST, which can be done with $\mathcal{O}(\mathrm{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\})$ I/Os (Theorem 4.8). To construct G'', we can use Euler tour and list ranking (Chapter 3) to number the nodes by preorder and find the number of descendants of each node, followed by a constant number of sorts and scans of the edges to check the conditions described above. Hence, the construction of G'' after we have found the MSF needs $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os. Finding the connected components of G'' has the same complexity as MSF (Theorem 4.7). Deriving the biconnected components of G from the connected components of G'' can then be done with a constant number of sorts and scans of the edges of G and G''.

Theorem 4.11 ([192]). *BCC can be solved using*

$$\mathcal{O}(\mathrm{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \text{ I/Os.}$$

Using the result of Theorem 4.10 we get:

Theorem 4.12. *BCC can be solved by a randomized algorithm using $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os.*

4.13 Conclusion for Graph Connectivity

Munagala and Ranade [567] prove a lower bound of $\Omega(|E|/|V| \cdot \text{sort}(|V|))$ I/Os for CC, BCC and MSF. Note that $|E|/|V| \cdot \text{sort}(|V|) = \Theta(\text{sort}(|E|))$.

We have surveyed randomized algorithms that achieve this bound, but the best known deterministic algorithms have a slightly higher I/O complexity. Therefore, while both deterministic and randomized algorithms are efficient, there still exists a gap between the upper bound and the lower bound in the deterministic case.

Acknowledgements We would like to thank the participants of the GI-Dagstuhl Forschungsseminar "Algorithms for Memory Hierarchies" for a number of fruitful discussions and helpful comments on this chapter.

5. I/O-Efficient Algorithms for Sparse Graphs

Laura Toma and Norbert Zeh*

5.1 Introduction

Massive graphs arise naturally in many applications. Recent web crawls, for example, produce graphs with on the order of 200 million nodes and 2 billion edges. Recent research in web modelling uses depth-first search, breadth-first search, and the computation of shortest paths and connected components as primitive routines for investigating the structure of the web [158]. Massive graphs are also often manipulated in Geographic Information Systems (GIS), where many problems can be formulated as fundamental graph problems. When working with such massive data sets, only a fraction of the data can be held in the main memory of a state-of-the-art computer. Thus, the transfer of data between main memory and secondary, disk-based memory, and not the internal memory computation, is often the bottleneck. A number of models have been developed for the purpose of analyzing this bottleneck and designing algorithms that minimize the traffic between main memory and disk. The algorithms discussed in this chapter are designed and analyzed in the *parallel disk model* (PDM) of Vitter and Shriver [755]. For a definition and discussion of this model, the reader may refer to Chapter 1.

Despite the efforts of many researchers [1, 7, 52, 53, 60, 164, 192, 302, 419, 485, 521, 522, 550, 567, 737], the design of I/O-efficient algorithms for basic graph problems is still a research area with many challenging open problems. For most graph problems, $\Omega(\text{perm}(|V|))$ or $\Omega(\text{sort}(|V|))$ are lower bounds on the number of I/Os required to solve them [53, 192], while the best known algorithms for these problems on general graphs perform considerably more I/Os. For example, the best known algorithms for DFS and SSSP perform $\Omega(|V|)$ I/Os in the worst case; for BFS an algorithm performing $o(|V|)$ I/Os has been proposed only recently (see Table 5.1). While these algorithms are I/O-efficient for graphs with at least $B \cdot |V|$ edges, they are inefficient for sparse graphs.

In this chapter we focus on algorithms that solve a number of fundamental graph problems I/O-efficiently on sparse graphs. The algorithms we discuss, besides exploiting the combinatorial and geometric properties of special classes of sparse graphs, demonstrate the power of two general techniques applied in I/O-efficient graph algorithms: graph contraction and time-forward processing. The problems we consider are computing the connected and biconnected components (CC and BCC), a minimum spanning tree (MST), or

* Part of this work was done when the second author was a Ph.D. student at the School of Computer Science of Carleton University, Ottawa, Canada.

Table 5.1. The best known upper bounds for fundamental graph problems on undirected graphs. The algorithms are deterministic and use linear space.

Problem	General undirected graphs	
CC, MST	$\mathcal{O}\left(\text{sort}(\lvert E\rvert)\log\log\frac{\lvert V\rvert B}{\lvert E\rvert}\right)$	[567, 60]
SSSP	$\mathcal{O}\left(\lvert V\rvert+\frac{\lvert E\rvert}{B}\cdot\log\frac{\lvert V\rvert}{B}\right)$	[485]
DFS	$\mathcal{O}\left(\lvert V\rvert+\frac{\lvert V\rvert}{M}\cdot\text{scan}(E)\right)$	[192]
	$\mathcal{O}\left((\lvert V\rvert+\text{scan}(\lvert E\rvert))\cdot\log_2\lvert V\rvert\right)$	[485]
BFS	$\mathcal{O}\left(\sqrt{\frac{\lvert V\rvert\lvert E\rvert}{B}}+\text{sort}(\lvert V\rvert+\lvert E\rvert)\log\log\frac{\lvert V\rvert B}{\lvert E\rvert}\right)$	[542]

an ear decomposition of the given graph, breadth-first search (BFS), depth-first search (DFS), and single source shortest paths (SSSP). The first four problems can be categorized as connectivity problems, while the latter three can be considered graph searching problems.

In our discussion we emphasize that graph contraction almost immediately leads to I/O-efficient solutions for connectivity problems on sparse graphs because edge contractions preserve the connectivity of the graph. For graph searching problems, the algorithms we discuss exploit structural properties of the graph classes we consider such as having small separators, outerplanar or planar embeddings, or tree-decompositions of constant width. The graph classes we consider are outerplanar graphs, planar graphs, grid graphs and graphs of bounded treewidth. A crucial condition for exploiting the structural properties of these graph classes in algorithms is the ability to compute such information explicitly. We discuss I/O-efficient algorithms that compute outerplanar and planar embeddings of outerplanar and planar graphs, tree-decompositions of graphs of bounded treewidth, and small separators of graphs in any of these classes. Table 5.2 gives an overview of the algorithms for sparse graphs discussed in this chapter.

In Section 5.2 we define the graph classes we consider. In Section 5.3 we describe the two fundamental algorithmic techniques used to solve graph problems I/O-efficiently on sparse graphs. In each of Sections 5.4 through 5.8 we discuss a different graph problem. Sections 5.5 through 5.8 are further divided into subsections describing the different solutions of the considered problem for different classes of sparse graphs. While we present these solutions separately, we emphasize the common ideas behind these solutions. We conclude in Section 5.9 with a summary and a discussion of some open problems.

Table 5.2. The problems that can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space on sparse graphs and the sections where they are discussed. A left-arrow indicates that the problem can be solved using the more general algorithm to the left.

Problem	**Graph class**				
	sparse	*planar*	*grid*	*outerplanar*	*bounded treewidth*
CC, BCC, MST	5.4	←	←	←	←
Ear decomposition	5.4	←	←	←	←
BFS + SSSP	open	5.5.1	5.5.1	5.5.2	5.5.2
DFS	open	5.6.1	open(5.6.2)	5.6.3	open
Graph partition	N/A	5.7.1	5.7.2	5.7.3	5.7.3
Embedding	N/A	5.8.1	N/A	5.8.3	N/A
Tree-decomposition	N/A	N/A	N/A	5.8.3	5.8.2

5.2 Definitions and Graph Classes

The notation and terminology used in this chapter is quite standard. The reader may refer to [283, 382] for definitions of basic graph-theoretic concepts. For clarity, we review a few basic definitions and define the graph classes considered in this chapter.

Given a graph $G = (V, E)$ and an edge $(v, w) \in E$, the *contraction* of edge (v, w) is the operation of replacing vertices v and w with a new vertex x and every edge (u, y), where $u \in \{v, w\}$ and $y \notin \{v, w\}$, with an edge (x, y). This may introduce duplicate edges into the edge set of G. These edges are removed. We call graph G *sparse* if $|E'| = \mathcal{O}(|V'|)$ for any graph $H = (V', E')$ that can be obtained from G through a series of edge contractions.[1]

A *planar embedding* $\hat{G}$ of a graph $G = (V, E)$ is a drawing of G in the plane so that every vertex is represented as a unique point, every edge is represented as a contiguous curve connecting its two endpoints, and no two edges intersect, except possibly at their endpoints. A graph G is *planar* if it has a planar embedding. Given an embedded planar graph, the *faces* of G are the connected components of $\mathbb{R}^2 \setminus \hat{G}$. The *boundary* of a face f is the set of vertices and edges contained in the closure of f.

A graph $G = (V, E)$ is *outerplanar* if it has a planar embedding one of whose faces has all vertices of G on its boundary. We call this face the *outer face* of G.

A *grid graph* is a graph whose vertices are a subset of the vertices of a $\sqrt{N} \times \sqrt{N}$ regular grid. Every vertex is denoted by its coordinates (i, j) and

[1] The authors of [192] call these graphs "sparse under edge contraction", thereby emphasizing the fact that the condition $|E| = \mathcal{O}(|V|)$ is not sufficient for a graph to belong to this class.

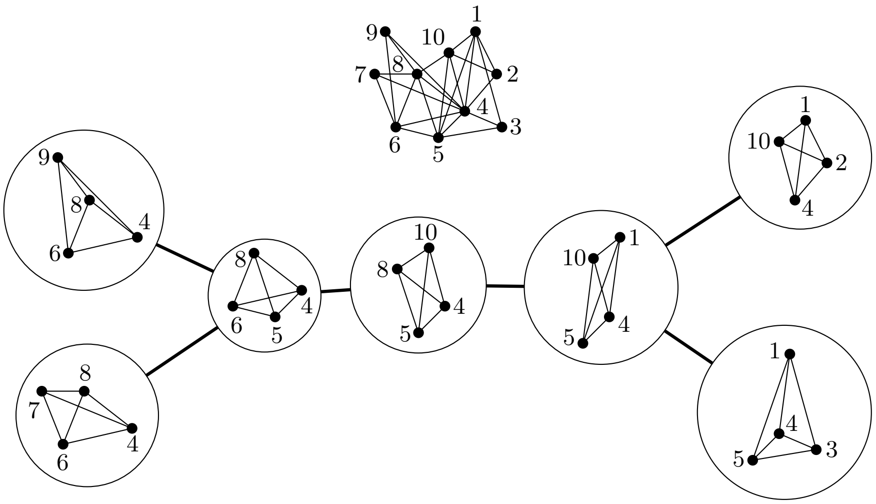


Fig. 5.1. A graph of treewidth three and a tree-decomposition of width three for the graph.

can be connected only to those vertices whose coordinates differ by at most one from its own coordinates. Note that a grid graph is not necessarily planar because diagonal edges may intersect.

A *tree-decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{D} = (T, \mathcal{X})$, where $T = (I, F)$ is a tree and $\mathcal{X} = (X_i)_{i \in I}$ is a collection of vertex sets satisfying the following properties (see Fig. 5.1):

(i) $\bigcup_{i \in I} X_i = V$,
(ii) For every edge $(v, w) \in E$, there exists a node $i \in I$ so that $v, w \in X_i$, and
(iii) For two nodes $i, k \in T$ and any node j on the path from i to k in T, $X_i \cap X_k \subseteq X_j$.

The *width* of tree-decomposition $\mathcal{D}$ is defined as $\max\{|X_i| : i \in I\} - 1$. The *treewidth* of graph G is the minimum width of all its tree-decompositions. Intuitively, the treewidth of a graph measures how close the graph is to being a tree or forest. A class $\mathcal{C}$ of graphs is said to have *bounded treewidth* if there exists a constant k so that all graphs in $\mathcal{C}$ have treewidth at most k. Outerplanar graphs, for example, have treewidth two. For algorithmic purposes, a particular type of tree-decomposition is especially useful: A *nice* tree-decomposition of a graph G is a tree-decomposition $\mathcal{D} = (T, \mathcal{X})$ with the following additional properties:

(iv) Tree T is a rooted binary tree, and
(v) Every internal node of T is of one of the following types: A *forget node* $i \in T$ has one child j, and $X_i = X_j \setminus \{x\}$, for some vertex $x \in X_j$. An *introduce node* $i \in T$ has one child j, and $X_i = X_j \cup \{x\}$, for some vertex

$x \notin X_j$. A *join node* $i \in T$ has two children j and k, and $X_i = X_j = X_k$. The leaves of T are also referred to as *start nodes*.

Bodlaender and Kloks [136] show that every graph of treewidth k has a nice tree-decomposition of width k and size $\mathcal{O}(N)$, where the size of a tree-decomposition is the number of nodes in T.

The relationships between the different graph classes are visualized in Fig. 5.2. In general, the more classes a graph belongs to, the more restrictive is its structure, so that it is not surprising that outerplanar graphs allow very simple solutions to the problems discussed in this chapter.

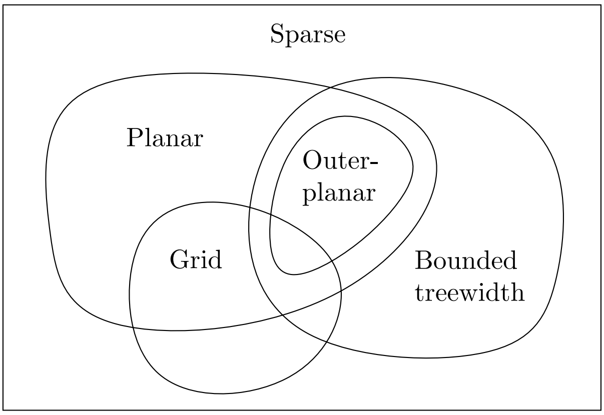


Fig. 5.2. The relationships between the different graph classes considered in this survey.

5.3 Techniques

Before discussing the particular algorithms in this survey, we sketch the two fundamental algorithmic techniques used in these algorithms: graph contraction and time-forward processing.

5.3.1 Graph Contraction

At a very abstract level, *graph contraction* is simple and elegant: Identify a number of edge-disjoint subgraphs of G so that representing each such subgraph by a graph of smaller size reduces the size of G by a constant factor and preserves the properties of interest. This approach is often taken in parallel graph algorithms, where the contraction procedure is applied recursively $q = \mathcal{O}(\log N)$ times, in order to produce a sequence $G = G_0, G_1, \ldots, G_q$ of graphs with $|G_q| = \mathcal{O}(1)$. Then the problem is solved in $\mathcal{O}(1)$ time on G_q. A solution for graph G is constructed by undoing the contraction steps and at each step deriving a solution for G_i from the given solution for graph G_{i+1}. When designing I/O-efficient algorithms, the contraction can usually stop after $q = \mathcal{O}(\log B)$ contraction levels. At that point, the resulting graph is guaranteed to have $\mathcal{O}(|V|/B)$ vertices, so that the algorithm can afford to spend $\mathcal{O}(1)$ I/Os per vertex to solve the problem on the contracted graph. The edges can usually be handled using I/O-efficient data structures.

5.3.2 Time-Forward Processing

Time-forward processing is a very elegant technique for evaluating directed acyclic graphs. This technique has been proposed in [192] and improved in [52]. Formally, the following problem can be solved using this technique:

Let G be a directed acyclic graph (DAG) whose vertices are numbered so that every edge in G leads from a vertex with lower number to a vertex with higher number. That is, this numbering is a topological numbering of G. Let every vertex v of G store a label $\phi(v)$, and let f be a function to be applied in order to compute for every vertex v, a new label $\psi(v) = f(\phi(v), \lambda(u_1), \dots, \lambda(u_k))$, where $u_1, \dots, u_k$ are the in-neighbors of v in G, and $\lambda(u_i)$ is some piece of information "sent" from u_i to v after computing $\psi(u_i)$. The goal is to "evaluate" G, i.e., to compute $\psi(v)$, for all vertices $v \in G$.

While time-forward processing does not solve the problem of computing $\psi(v)$ I/O-efficiently in the case where the input data $\phi(v)$ and $\lambda(u_1), \dots, \lambda(u_k)$ do not fit into internal memory, it provides an elegant way to supply vertex v with this information at the time when $\psi(v)$ is computed. The idea is to process the vertices in G by increasing numbers. This guarantees that all in-neighbors of vertex v are evaluated before v. Thus, if these in-neighbors "send" their outputs $\lambda(u_1), \dots, \lambda(u_k)$ to v, v has these inputs and its own label $\phi(v)$ at its disposal to compute $\psi(v)$. After computing $\psi(v)$, v sends its output $\lambda(v)$ "forward in time" to its out-neighbors, which guarantees that these out-neighbors have $\lambda(v)$ at their disposal when it is their turn to be evaluated.

The implementation of this technique due to Arge [52] uses a priority queue Q to realize the "sending" of information along the edges of G. When a vertex v wants to send its output $\lambda(v)$ to another vertex w, it inserts $\lambda(v)$ into priority queue Q and gives it priority w. When vertex w is being evaluated, it removes all entries with priority w from Q. As every in-neighbor of w sends its output to w by queuing it with priority w, this provides w with the required inputs. Moreover, every vertex removes its inputs from the priority queue when it is evaluated, and all vertices with smaller numbers are evaluated before w. Thus, the entries in Q with priority w are in fact those with lowest priority in Q at the time when w is evaluated. Therefore they can be removed using a sequence of DELETEMIN operations. Since this procedure involves $\mathcal{O}(|V| + |E|)$ priority queue operations, graph G can be evaluated in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os using an I/O-efficient priority queue (see Chapter 2).

The power of time-forward processing lies in the fact that many problems on undirected graphs can be expressed as evaluation problems of DAGs derived from these graphs.

5.4 Connectivity Problems

5.4.1 Connected Components

The I/O-efficient connectivity algorithm of [192] uses ideas from the PRAM algorithm of Chin et al. [197] for this problem. First the graph contraction technique from Section 5.3.1 is applied in order to compute a sequence $G = G_0, \ldots, G_q$ of graphs whose vertex sets have geometrically decreasing sizes and so that the vertex set of graph G_q fits into main memory. The latter implies that the connected components of G_q can be computed using a simple semi-external connectivity algorithm as outlined below. Given the connected components of G_q, the connected components of G are computed by undoing the contraction steps used to construct graphs $G_1, \ldots, G_q$ one by one and in each step computing the connected components of G_i from those of G_{i+1}. The details of the algorithm are as follows:

In order to compute graph G_{i+1} from graph G_i during the contraction phase, every vertex in $G_i = (V_i, E_i)$ selects its incident edge leading to its neighbor with smallest number. The selected edges form a forest F_i each of whose trees contains at least two vertices. Every tree in F_i is then contracted into a single vertex, which produces a new graph $G_{i+1} = (V_{i+1}, E_{i+1})$ with at most half as many vertices as G_i. In particular, for $0 \le i \le q$, $|V_i| \le |V|/2^i$. Choosing $q = \log(|V|/M)$, this implies that $|V_q| \le M$, i.e., the vertex set of G_q fits into main memory. Hence, the connected components of G_q can be computed using the following simple algorithm:

Load the vertices of G_q into main memory and label each of them as being in a separate connected component. Now scan the edge set of G_q and merge connected components whenever the endpoints of an edge are found to be in different connected components. The computation of this algorithm is carried out in main memory, so that computing the connected components of G_q takes $\mathcal{O}(\text{scan}(|V_q| + |E_q|))$ I/Os.

To construct the connected components of graph G_i from those of graph G_{i+1} when undoing the contraction steps, all that is required is to replace each vertex v of G_i with the tree in F_i it represents and assign v's component label to all vertices in this tree.

In [192] it is shown that the construction of G_{i+1} from G_i as well as computing the connected components of G_i from those of G_{i+1} takes $\mathcal{O}(\text{sort}(|E_i|))$ I/Os. Hence, the whole connectivity algorithm takes $\sum_{i=0}^{\log(|V|/M)} \mathcal{O}(\text{sort}(|E_i|))$ I/Os. Since the graphs we consider are sparse,

$|E_i| = \mathcal{O}(|V_i|) = \mathcal{O}(|V|/2^i)$, so that $\sum_{i=0}^{\log(|V|/M)} \mathcal{O}(\mathrm{sort}(|E_i|)) = \mathcal{O}(\mathrm{sort}(|V|))$. That is, the contraction-based connectivity algorithm computes the connected components of a sparse graph in $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os.

5.4.2 Minimum Spanning Tree

The algorithm outlined in Section 5.4.1 can be modified so that it computes an MST (a minimum spanning forest if the graph is disconnected). Instead of selecting for every vertex, the edge connecting it to its neighbor with smallest number, the MST-algorithm chooses the edge of minimum weight incident to each vertex. The weight of an edge in G_i is the minimum weight of its corresponding edges in G. When an edge in G_i is selected, its corresponding minimum weight edge in G is added to the MST. For details and a correctness proof see [192, 197]. Clearly, these modifications do not increase the I/O-complexity of the connectivity algorithm. Hence, the algorithm takes $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os on sparse graphs.

5.4.3 Biconnected Components

Tarjan and Vishkin [714] present an elegant parallel algorithm to compute the biconnected components of a graph G by computing the connected components of an auxiliary graph H. Given a spanning tree T of G, every non-tree edge (v, w) of G (i.e., $(v, w) \in E(G)\backslash E(T)$) defines a *fundamental cycle*, which consists of the path from v to w in T and edge (v, w) itself. The auxiliary graph H contains one vertex per edge of G. Two vertices in H are adjacent if the corresponding edges appear consecutively on a fundamental cycle in G. Using this definition of H, it is easy to verify that two edges of G are in the same biconnected component of G if the two corresponding vertices in H are in the same connected component of H. In [192], Chiang et al. show that the construction of H from G can be carried out in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os. Since H has $\mathcal{O}(|E|)$ vertices and edges, the connected components of H can be computed in $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os using the connectivity algorithm from Section 5.4.1. Hence, the biconnected components of G can be computed in $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os if G is sparse.

5.4.4 Ear Decomposition

An *ear decomposition* $\mathcal{E} = (P_0, P_1, \ldots, P_k)$ of a graph G is an ordered partition of G into edge-disjoint simple paths P_i with endpoints s_i and t_i. Ear P_0 is an edge. For $1 \leq i \leq k$, ear P_i shares its two endpoints s_i and t_i, but none of its internal vertices, with the union $P_0 \cup \cdots \cup P_{i-1}$ of all previous ears. An ear P_i is *open* if $s_i \neq t_i$. Ear decomposition $\mathcal{E}$ is *open* if all its ears are open.

For a graph to have an ear decomposition, it has to be two-edge connected. That is, there cannot be an edge whose removal disconnects the graph. For the graph to have an open ear decomposition, it has to be 2-vertex connected (biconnected). That is, there cannot be a vertex whose removal disconnects the graph. So let G be a 2-edge connected graph, and let $\mathcal{E}$ be an ear decomposition of G. Removing an arbitrary edge from each ear in $\mathcal{E}$ results in a spanning tree of G. Conversely, an ear decomposition of G can be obtained from any spanning tree T of G as follows: Consider the fundamental cycles defined by the non-tree edges of G and sort them by their distances in T from the root r of T, where the distance of a cycle from r is the minimum distance of its vertices from r. Remove from each fundamental cycle all those edges that are already contained in a previous cycle. It can be shown that the resulting sorted set of cycles and paths is an ear decomposition of G.

In order to obtain a parallel ear decomposition algorithm, Maon et al. [530] propose the following implementation of the above idea: Their algorithm assigns labels to the edges of G so that two edges are in the same ear if and only if they have the same label. For a non-tree edge $e = (v, w)$, let LCA(e) be the lowest common ancestor of v and w in T, and let depth(e) be the distance of LCA(e) from the root of T. Then label(e) = (depth(e), e). For a tree-edge e, label(e) is the minimum label of all non-tree edges so that edge e is on the fundamental cycles defined by these edges. Now every non-tree edge e defines a cycle or simple path P_e with edge set $\{e' \in G : \text{label}(e') = \text{label}(e)\}$. Maon et al. show that the collection of these cycles and paths, sorted by their labels, is an ear decomposition of G. This ear decomposition is not necessarily open, even if G is biconnected; but the computation can be modified to produce an open ear decomposition in this case. See [530] for details.

Given spanning tree T, the computation of edge labels as described above involves only standard tree computations such as answering LCA queries or computing the depth of a vertex in T. In [192] it is shown that these computations can be carried out in $\mathcal{O}(\text{sort}(|V|))$ I/Os. Since we consider sparse graphs, tree T can be computed in $\mathcal{O}(\text{sort}(|V|))$ I/Os using for example the minimum spanning tree algorithm of Section 5.4.2. Hence, for sparse graphs an (open) ear decomposition can be computed in $\mathcal{O}(\text{sort}(|V|))$ I/Os.

5.5 Breadth-First Search and Single Source Shortest Paths

After covering connectivity problems, we now turn to the first two graph searching problems: breadth-first search (BFS) and the single source shortest path (SSSP) problem. Since BFS is the same as the SSSP problem if all edges in the graph have unit weight, and both problems have an $\Omega(\text{perm}(|V|))$ I/O lower bound, we restrict our attention to SSSP-algorithms. Even though the details of the SSSP-algorithms for different graph classes differ, their efficiency is based on the fact that the considered graph classes have small separators. In particular, a separator decomposition of a graph in each such class can be

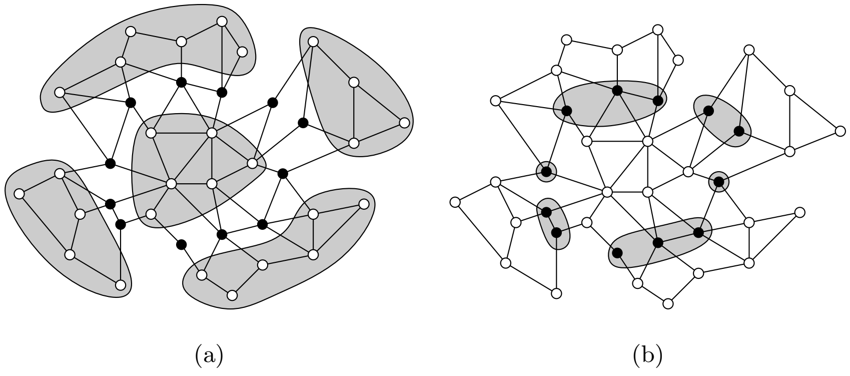


Fig. 5.3. (a) A partition of a planar graph into the shaded subgraphs using the black separator vertices. (b) The boundary sets of the partition.

obtained I/O-efficiently (see Section 5.7), and the shortest path algorithms apply dynamic programming to such a decomposition in order to solve the SSSP problem.

5.5.1 Planar Graphs and Grid Graphs

Every planar graph with N vertices or grid graph embedded into a $\sqrt{N} \times \sqrt{N}$ grid contains a set S of $\mathcal{O}(N/B)$ separator vertices whose removal partitions the graph into $\mathcal{O}\big(N/B^2\big)$ subgraphs of size at most B^2 and boundary size at most B, where the *boundary* ∂G_i of a subgraph G_i is the set of separator vertices adjacent to vertices in G_i [315]. The set of separator vertices can be partitioned into maximal subsets so that the vertices in each subset are adjacent to the same set of subgraphs G_i. These sets are called the *boundary sets* of the partition (see Fig. 5.3). If the graph has bounded degree, which is true for grid graphs and can be ensured for planar graphs using a simple transformation, there exists a partition that, in addition to the above properties, has only $\mathcal{O}\big(N/B^2\big)$ boundary sets [315].

Given such a partition of G into subgraphs $G_1, \dots, G_q$, the single source shortest path algorithm first computes a graph G_R with vertex set S so that the distances between two separator vertices in G and G_R are the same. Assuming that $s \in S$, the distances from s to all separator vertices can hence be computed by solving the single source shortest path problem on G_R, which can be done I/O-efficiently due to the reduced size of the vertex set of G_R. Given the distances from s to all separator vertices, the distances from s to all vertices in a graph G_i can be computed as $\text{dist}_G(s, v) = \min\{\text{dist}_G(s, u) + \text{dist}_{R_i}(u, v) : u \in \partial G_i\}$, where R_i is the subgraph of G induced by the vertices in $V(G_i) \cup \partial G_i$.

The construction of graph G_R from graph G is done as follows: For each graph R_i as defined above, compute the distances in R_i between all pairs of vertices in ∂G_i. Then construct a complete graph R'_i with vertex set ∂G_i and assign weight $\text{dist}_{R_i}(v, w)$ to every edge $(v, w) \in R'_i$. Graph G_R is the union of graphs $R'_1, \dots, R'_q$.

Assuming that $M = \Omega(B^2)$, there is enough room in main memory to store one graph R_i and its compressed version R'_i. Hence, graph G_R can be computed from graph G by loading graphs $R_1, \dots, R_q$ into main memory, one at a time, computing for each graph R_i the compressed version R'_i without incurring any I/Os and writing R'_i to disk. As this procedure requires a single scan of the list of graphs $R_1, \dots, R_q$, and these graphs have a total size of $\mathcal{O}(N)$, graph G_R can be constructed in $\mathcal{O}(\text{scan}(N))$ I/Os. Similarly, once the distances from s to all separator vertices are known, the computation of the distances from s to all non-separator vertices can be carried out in another scan of the list of graphs $R_1, \dots, R_q$ because the computation for the vertices in R_i is local to R_i.

From the above discussion it follows that the SSSP problem can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os on G if it can be solved in that many I/Os on G_R. Since G_R has only $\mathcal{O}(N/B)$ vertices and $\mathcal{O}\big((N/B^2) \cdot B^2\big) = \mathcal{O}(N)$ edges, the SSSP problem on G_R can be solved in $\mathcal{O}((N/B)\log_2(N/B))$ I/Os using the shortest path algorithm described in Chapter 4. In order to reduce the I/O-complexity of this step to $\mathcal{O}(\text{sort}(N))$, Arge et al. [60, 68] propose a modified version of Dijkstra's algorithm, which avoids the use of a DecreaseKey operation. This is necessary because the best known external priority queue that supports this operation [485] takes $\mathcal{O}((N/B)\log_2(N/B))$ I/Os to process a sequence of N priority queue operations, while there are priority queues that do not support this operation, but can process a sequence of N Insert, Delete, and DeleteMin operations in $\mathcal{O}(\text{sort}(N))$ I/Os [52, 157].

In addition to a priority queue Q storing the unvisited vertices of G_R, the algorithm of Arge et al. maintains a list L of the vertices of G_R, each labelled with its tentative distance from s. That is, for every vertex stored in Q, its label in L is the same as its priority in Q. For a vertex not in Q, list L stores its final distance from s. Initially, all distances, except that of s, are ∞. Vertex s has distance 0. Now the algorithm repeatedly performs DeleteMin operations on Q to obtain the next vertex to process. For every retrieved vertex v, the algorithm loads the adjacency list of v into main memory and updates the distances from s to v's neighbors as necessary. (The adjacency list of v fits into main memory because every vertex in S has degree $\mathcal{O}(B)$ in G_R. To see this, observe that each vertex in S is on the boundary of $\mathcal{O}(1)$ subgraphs G_i because graph G has bounded degree, and each subgraph has at most B boundary vertices.) In order to update these distances, the algorithm retrieves the entries corresponding to v's neighbors from L and compares the current tentative distance of each neighbor w of v to the length of the path from s to w through v. If the path through v is shorter, the distance from s to w is updated in L and Q. Since the old tentative distance from s to w is known, the update on Q can be performed by deleting the old copy of w

and inserting a new copy with the updated distance as priority. That is, the required DecreaseKey operation is replaced by a Delete and an Insert operation.

Since graph G_R has $\mathcal{O}(N/B)$ vertices and $\mathcal{O}(N)$ edges, retrieving all adjacency lists takes $\mathcal{O}(N/B + \text{scan}(N)) = \mathcal{O}(\text{scan}(N))$ I/Os. For the same reason, the algorithm performs only $\mathcal{O}(N)$ priority operations on Q, which takes $\mathcal{O}(\text{sort}(N))$ I/Os. It remains to analyze the number of I/Os spent on accessing list L. If the vertices in L are not arranged carefully, the algorithm may spend one I/O per access to a vertex in L, $\mathcal{O}(N)$ I/Os in total. In order to reduce this I/O-bound to $\mathcal{O}(N/B)$, Arge et al. use the fact that there are only $\mathcal{O}(N/B^2)$ boundary sets, each of size $\mathcal{O}(B)$. If the vertices in each boundary set are stored consecutively in L, the bound on the size of each boundary set implies that the vertices in the set can be accessed in $\mathcal{O}(1)$ I/Os. Moreover, every boundary set is accessed only $\mathcal{O}(B)$ times, once per vertex on the boundaries of the subgraphs defining this boundary set. Since there are $\mathcal{O}(N/B^2)$ boundary sets, the total number of I/Os spent on loading boundary sets from L is hence $\mathcal{O}\big(B \cdot N/B^2\big) = \mathcal{O}(N/B)$.

The algorithm described above computes only the distances from s to all vertices in G. However, it is easy to augment the algorithm so that it computes shortest paths in $\mathcal{O}(\text{sort}(N))$ I/Os using an additional post-processing step.

5.5.2 Graphs of Bounded Treewidth and Outerplanar Graphs

The SSSP algorithm for planar graphs and grid graphs computes shortest paths in three steps: First it encodes the distances between separator vertices in a compressed graph. Then it computes the distances from the source to all separator vertices in this compressed graph. And finally it computes the distances from the source to all non-separator vertices using the distance information computed for the separator vertices on the boundary of the subgraph G_i containing each such vertex. The shortest path algorithm for outerplanar graphs and graphs of bounded treewidth [522, 775] applies this approach iteratively, using the fact that a tree-decomposition of the graph provides a hierarchical decomposition of the graph using separators of constant size.

Assume that the given tree-decomposition $\mathcal{D} = (T, \mathcal{X})$ of G is nice in the sense defined in Section 5.2 and that $s \in X_v$, for all $v \in T$.[2] Then every subtree of T rooted at some node $v \in T$ represents a subgraph $G(v)$ of G, which shares only the vertices in X_v with the rest of G.

The first phase of the algorithm processes T from the leaves towards the root and computes for every node $v \in T$ and every pair of vertices $x, y \in X_v$, the distance from x to y in $G(v)$. Since $G(r) = G$, for the root r of T, this produces the distances in G between all vertices in X_r. In particular, the

[2] Explicitly adding s to all sets X_v to ensure the latter assumption increases the width of the decomposition by at most one.

distances from s to all other vertices in X_r are known at the end of the first phase. The second phase processes tree T from the root towards the leaves to compute for every node $v \in T$, the distances from s to all vertices in X_v.

During the first phase, the computation at a node v uses only the weights of the edges between vertices in X_v and distance information computed for the vertices stored at v's children. During the second phase, the computation at node v uses the distance information computed for the vertices in X_v during the first phase of the algorithm and the distances from s to all vertices in $X_{p(v)}$, where $p(v)$ denotes v's parent in T. Since the computation at every node involves only a constant amount of information, it can be carried out in main memory. All that is required is passing distance information from children to parents in the first phase of the algorithm and from parents to children in the second phase. This can be done in $\mathcal{O}(\text{sort}(N))$ I/Os using time-forward processing because tree T has size $\mathcal{O}(N)$, and $\mathcal{O}(1)$ information is sent along every edge.

To provide at least some insight into the computation carried out at the nodes of T, we discuss the first phase of the algorithm. For a leaf v, $G(v)$ is the graph induced by the vertices in X_v. In particular, $|G(v)| = \mathcal{O}(1)$, and the distances in $G(v)$ between all vertices in X_v can be computed in main memory. For a forget node v with child w, $G(v) = G(w)$ and $X_v \subset X_w$, so that the distance information for the vertices in X_v has already been computed at node w and can easily be copied to node v. For an introduce node v with child w, $X_v = X_w \cup \{x\}$. A shortest path in $G(v)$ between two vertices in X_v consists of shortest paths in $G(w)$ between vertices in X_w and edges between x and vertices in X_w. Hence, the distances between vertices in X_v are the same in $G(v)$ and in a complete graph $G'(v)$ with vertex set X_v whose edges have the following weights: If $y, z \in X_w$, then edge (y, z) has weight $\text{dist}_{G(w)}(y, z)$. Otherwise assume w.l.o.g. that $y = x$. Then the weight of edge (x, z) is the same in $G'(v)$ as in G. The distances in $G(v)$ between all vertices in X_v can now be computed by solving all pairs shortest paths on $G'(v)$. This can be done in main memory because $|G'(v)| = \mathcal{O}(1)$. For a join node u with children v and w, a similar graph $G'(u)$ of constant size is computed, which captures the lengths of the shortest paths between all vertices in $X_u = X_v = X_w$ that stay either completely in $G(v)$ or completely in $G(w)$. The distances in $G(u)$ between all vertices in X_u are again computed in main memory by solving all pairs shortest paths on $G'(u)$.

The second phase of the algorithm proceeds in a similar fashion, using the fact that a shortest path from s to a vertex x in X_v either stays completely inside $G(v)$, in which case the shortest path information between s and x computed in the first phase is correct, or it consists of a shortest path from s to a vertex y in $X_{p(v)}$ followed by a shortest path from y to x in $G(p(v))$.

Since outerplanar graphs have treewidth 2, the algorithm sketched above can be used to solve SSSP on outerplanar graphs in $\mathcal{O}(\text{sort}(N))$ I/Os. Alternatively, one can derive a separator decomposition of an outerplanar graph

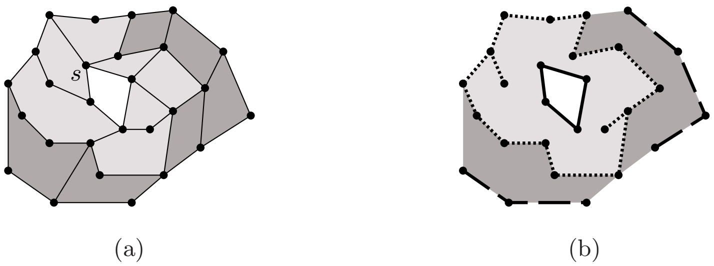


Fig. 5.4. (a) A planar graph G with its faces colored according to their levels. Level-0 faces are white. Level-1-faces are light grey. Level-2 faces are dark grey. (b) The corresponding partition of the graph into outerplanar subgraphs H_0 (solid), H_1 (dotted), and H_2 (dashed).

directly from an outerplanar embedding of the graph. This separator decomposition has a somewhat simpler structure, which allows the algorithm to be simplified and the constants to be improved. However, the overall structure of the algorithm remains the same. The interested reader may refer to [775] for details.

5.6 Depth-First Search

The algorithms of the previous section use small separators to compute shortest paths I/O-efficiently. In this section we discuss algorithms that construct DFS-trees of planar graphs, grid graphs, and outerplanar graphs. These algorithms exploit the geometric structure of these graphs to carry out their task in an I/O-efficient manner. Since graphs of bounded treewidth do not exhibit such a geometric structure in general, the techniques used in these algorithms fail on graphs of bounded treewidth, so that the problem of computing a DFS-tree of a graph of bounded treewidth I/O-efficiently is open.

5.6.1 Planar Graphs

For the sake of simplicity, assume that the given planar graph G is biconnected. If this is not the case, a DFS-tree of G can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os by identifying the biconnected components of G using the biconnectivity algorithm from Section 5.4.3 and merging appropriate DFS-trees computed separately for each of these biconnected components.

In order to perform DFS in an embedded biconnected planar graph G, the algorithm of [62], which follows ideas from [372], uses the following approach: First the faces of G are partitioned into layers around a central face that has the source of the DFS on its boundary (see Fig. 5.4a). The partition of the faces of G into layers induces a partition of G into outerplanar graphs of a particularly simple structure, so that DFS-trees of these graphs can be

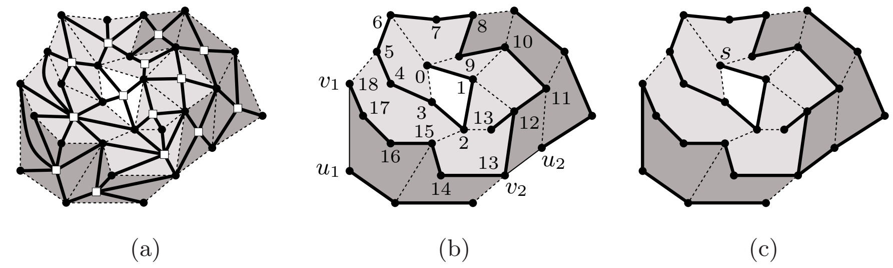


Fig. 5.5. (a) The face-on-vertex graph G_F shown in bold. (b) Spanning tree T_1 and layer graph H_2 are shown in bold. Attachment edges (u_i, v_i) are thin solid edges. The vertices in T_1 are labelled with their DFS-depths. (c) The final DFS tree of G.

computed I/O-efficiently (see Fig. 5.4b). A DFS-tree of G is then obtained by merging appropriate DFS-trees of these layer graphs.

Formally, the layers are defined as follows: The first layer consists of a single face r that has the source s of the DFS on its boundary. Given the first i layers, the $(i+1)$-st layer contains all faces that share a vertex with a face in the i-th layer and are not contained in layers 0 through i. Such a partition of the faces of G into layers can be obtained using BFS in the *face-on-vertex graph* G_F of G. This graph contains all vertices of G as well as one vertex f^* per face f of G. There is an edge (v, f^*) in G_F if and only if vertex v is on the boundary of face f in G. Given the BFS-distances of all vertices in G_F from r^*, a face f is in the i-th layer if the BFS-distance of vertex f^* from r^* is $2i$ (see Fig. 5.5a).

The computed layer partition of the faces of G defines the following partition of G into outerplanar subgraphs $H_0, H_1, \ldots, H_k$ (see Fig. 5.4b). Let G_i be the subgraph of G induced by all faces at levels 0 through i, and let $E_i = E(G_i) \setminus E(G_{i-1})$. Then edge set E_i can be partitioned into two sets E_i' and E_i''. An edge is in E_i' if none of its endpoints is on the boundary of a level-$(i-1)$ face. Otherwise it is in E_i''. Now graph H_i is defined as the subgraph of G induced by the edges in E_i'. The edges in E_i'' are the *attachment edges* of H_i, since they form the connection between H_i and G_{i-1}.

In order to compute a DFS-tree of G, the algorithm now computes DFS-trees $T_0, \ldots, T_k$ of graphs $H_0 = G_0, \ldots, G_k = G$, where for all $1 \leq i \leq k$, tree T_i is obtained by augmenting tree T_{i-1} appropriately. To facilitate this incremental construction, the algorithm maintains the invariant that the vertices on each boundary cycle of G_i appear on a root-to-leaf path in T_i. We call this the *boundary cycle invariant.* Using this invariant, a DFS-tree T_{i+1} of G_{i+1} can be obtained as follows: For every connected component H' of H_{i+1}, find the attachment edge in E_{i+1}'' that connects a vertex v in H' to a vertex u of maximal depth in T_i. Compute a DFS-tree of H' rooted at v and join it to T_i using edge (u, v). Since H' is enclosed by exactly one boundary cycle of G_i, the choice of edge (u, v) and the boundary cycle invariant guarantee that for any other attachment edge (x, y) of H', its endpoint

$x \in G_i$ is an ancestor of u in T_i. The endpoint $y \in H'$ is a descendant of w, so that (x, y) is a back edge. Hence, T_{i+1} is a DFS-tree of G_{i+1}.

We have to discuss how to compute a DFS-tree of H' in a manner that maintains the boundary cycle invariant. This is where the simple structure of H_{i+1} comes into play. In particular, it can be shown that graph H_{i+1} is a "forest of cycles". That is, every non-trivial biconnected component of H_{i+1} is a simple cycle. Now consider the biconnected components of H'. These components form a tree of biconnected components, which can be rooted at a component containing vertex v. For every biconnected component $\mathcal{B}$, except the root component, its *parent cutpoint* is the cutpoint shared by $\mathcal{B}$ and its parent component in the tree of biconnected components. The parent cutpoint of the root component is defined to be v, even though v is not necessarily a cutpoint of H'. A DFS-tree of H' is now obtained by removing from every non-trivial biconnected component of H' one of the two edges incident to its parent cutpoint. Since the boundary cycles of G_{i+1} are cycles in H_{i+1}, and each such cycle is a non-trivial biconnected component of H_{i+1}, this maintains the boundary cycle invariant for T_{i+1}.

The computation of a DFS-tree for H' involves only computing the biconnected components of H' and the removal of appropriate edges from the non-trivial biconnected components. The former can be done in $\mathcal{O}(\text{sort}(|H'|))$ I/Os using the biconnectivity algorithm from Section 5.4.3. The latter can be done in a constant number of sorting and scanning steps. Hence, computing DFS-trees for all connected components of H_{i+1} takes $\mathcal{O}(\text{sort}(|H_{i+1}|))$ I/Os. Finding attachment edges (u, v) for all connected components of H_{i+1} requires sorting and scanning the vertex set of H_i and the set E''_{i+1} of attachment edges, which takes $\mathcal{O}(\text{sort}(|H_i| + |H_{i+1}|))$ I/Os. Summing these complexities over all layers of G, the whole DFS-algorithm takes $\mathcal{O}(\text{sort}(|V|))$ I/Os because graphs $H_0, \dots, H_k$ are disjoint.

5.6.2 Grid Graphs

Finding an I/O-efficient algorithm for DFS in grid graphs is still an open problem, even though the standard internal memory DFS-algorithm performs only $\mathcal{O}\big(N/\sqrt{B}\big)$ I/Os if carried out carefully. In particular, Meyer [550] made the following observation: Consider a grid of size $\sqrt{N}$ by $\sqrt{N}$ divided into subgrids of size $\sqrt{B}$ by $\sqrt{B}$. Then a DFS-tree of G can be computed using the standard internal memory algorithm: Start from the block (subgrid) containing the source vertex, perform DFS until the algorithm visits a vertex that is not in the block, load the block containing this vertex into main memory, and continue in this fashion until the DFS-tree is complete. A block may be loaded several times during a run of the algorithm, each time to compute a different part of the DFS tree that lies within this block. However, the DFS tree enters a block through a boundary vertex, and leaves it through a boundary vertex. Every vertex is visited $O(1)$ times by the DFS-algorithm. Since a block has $\mathcal{O}(\sqrt{B})$ boundary vertices, this implies that every block is loaded $\mathcal{O}(\sqrt{B})$ times, so that the algorithm takes $\mathcal{O}(N/B \cdot \sqrt{B}) = \mathcal{O}(N/\sqrt{B})$ I/Os.

5.6.3 Outerplanar Graphs

The DFS-algorithm for outerplanar graphs is hardly worth being called an algorithm. It merely exploits the fact that a DFS-tree of the graph is already encoded in an outerplanar embedding of the graph. In particular, if the graph is biconnected, the path obtained by removing an edge from the boundary cycle of the outer face of G is a DFS-tree of G. This path can be extracted in $\mathcal{O}(\text{scan}(N))$ I/Os from an appropriate representation of the outerplanar embedding.

If the graph is not biconnected, the basic idea of the algorithm is to walk along the outer boundary of G and keep track of the number of times a vertex has been visited so far. If a vertex is visited for the first time, its predecessor along the outer boundary is made its parent in the DFS-tree. Otherwise nothing is done for the vertex. This strategy can easily be realized in $\mathcal{O}(\text{scan}(N))$ I/Os using a stack, again assuming an appropriate representation of the outerplanar embedding. The interested reader may refer to [775] for details.

5.7 Graph Partitions

In this section we review algorithms for partitioning sparse graphs into smaller subgraphs. In particular, given a graph G belonging to a class $\mathcal{C}$ of sparse graphs and an integer $h > 0$, the goal is to compute a small set S of vertices whose removal partitions G into $k = \mathcal{O}(N/h)$ subgraphs $G_1, \ldots, G_k$ of size at most h. We refer to the pair $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ as an *h-partition* of G. The vertices in S are referred to as *separator vertices*. Set S as a whole is referred to as the *separator* that induces partition $\mathcal{P}$. Finally, let $\sigma(\mathcal{C}, N, h) = \max_G \min_S \{|S| : \text{separator } S \text{ induces an } h\text{-partition of } G\}$, where the maximum is taken over all N-vertex graphs in class $\mathcal{C}$. We call partition $\mathcal{P}$ *optimal* if $|S| = \mathcal{O}(\sigma(\mathcal{C}, |G|, h))$. Among the algorithms presented so far, only the shortest path algorithm for planar graphs requires an optimal h-partition $\big(\text{for } h = B^2\big)$, while the algorithm for outerplanar graphs and graphs of bounded treewidth only relies on the fact that a recursive partition of the graph using separators of constant size is encoded in the tree-decomposition of the graph. Nevertheless graph partitions have been applied to design efficient sequential and parallel algorithms for a wide range of problems and may prove useful for designing I/O-efficient algorithms for these problems as well.

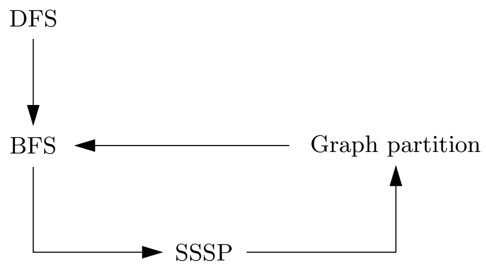


Fig. 5.6. $\mathcal{O}(\mathrm{sort}(N))$ I/O reductions between fundamental problems on planar graphs. An arrow indicates that the pointing problem can be solved in $\mathcal{O}(\mathrm{sort}(N))$ I/Os if the problem the arrow points to can be solved in that many I/Os.

5.7.1 Planar Graphs

A number of researchers have put considerable effort into designing algorithms that compute optimal partitions of planar graphs I/O-efficiently. The first step towards a solution has been made in [419], where it is shown that an optimal $\frac{2}{3}N$-partition of a planar graph G can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using an I/O-efficient version of Lipton and Tarjan's algorithm [507]. Unfortunately the I/O-bound holds only if a BFS-tree and a planar embedding of G are given. Arge et al. [60] present an I/O-efficient variant of an algorithm due to Goodrich [344]. Given a planar graph G and an integer $h > 0$, the algorithm takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os to compute a separator of size $\mathcal{O}\big(\mathrm{sort}(N) + N/\sqrt{h}\big)$ that induces an h-partition of G. Again, a BFS-tree and a planar embedding of G are required. In addition, the amount of available memory is required to be $M > B^{2+\alpha}$, for some $\alpha > 0$. Together with the shortest path algorithm from Section 5.5.1, the DFS-algorithm from Section 5.6.1, and the observation that BFS can be solved using an SSSP-algorithm, the latter result leads to circular dependencies between different fundamental problems on embedded planar graphs as shown in Fig. 5.6. In particular, if any of the three problems in the cycle—i.e., computing optimal h-partitions, BFS, or SSSP—can be solved in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, all problems in Fig. 5.6 can be solved in this number of I/Os on embedded planar graphs.

The break-through has been achieved in [523], where it is shown that an optimal h-partition of a planar graph G can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os without using a solution to any of the other problems, provided that the amount of available main memory is $\Omega\big(h \log^2 B\big)$. Since the shortest path algorithm from Section 5.5.1 requires an optimal B^2-partition, this implies that BFS, DFS, and SSSP can be solved in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, provided that $M = \Omega\big(B^2 \log^2 B\big)$. In [775], it is shown that using these results and an I/O-efficient version of a recent result of [26], optimal partitions of planar graphs with costs and weights on their vertices and optimal edge separators of weighted planar graphs can be obtained in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. We sketch here the result of [523] on computing unweighted partitions because it is the key to the efficiency of the other algorithms, and refer the reader to [775] for the details of the more general separator algorithm.

The algorithm of [523] obtains an optimal h-partition of G by careful application of the graph contraction technique, combined with a linear-time internal memory algorithm for this problem. In particular, it first constructs a hierarchy of planar graphs $G = H_0, \ldots, H_r$ whose sizes are geometrically decreasing and so that $|H_r| = \mathcal{O}(N/B)$. The latter implies that applying the internal memory algorithm to H_r in order to compute an optimal partition of H_r takes $\mathcal{O}(N/B) = \mathcal{O}(\text{scan}(N))$ I/Os. Given the partition of H_r, the algorithm now iterates over graphs $H_{r-1}, \ldots, H_0$, in each iteration deriving a partition of H_i from the partition of H_{i+1} computed in the previous iteration. The construction of a separator S_i for H_i starts with the set S'_i of vertices in G_i that were contracted into the vertices in S_{i+1} during the construction of H_{i+1} from H_i. Set S'_i induces a preliminary partition of H_i, which is then refined by adding new separator vertices to S'_i. The resulting set is S_i.

The efficiency of the procedure and the quality of its output depend heavily on the properties of the computed graph hierarchy. In [523] it is shown that a graph hierarchy $G = H_0, \ldots, H_r$ with the following properties can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os:

(i) For all $0 \leq i \leq r$, graph H_i is planar,
(ii) For all $1 \leq i \leq r$, every vertex in H_i represents a constant number of vertices in H_{i-1} and at most 2^i vertices in G, and
(iii) For all $0 \leq i \leq r$, $|H_i| = \mathcal{O}\big(N/2^i\big)$.

Choosing $r = \log B$, Property (iii) implies that $|H_r| = \mathcal{O}(N/B)$, as required by the algorithm. Properties (ii) and (iii) can be combined with an appropriate choice of the size of the subgraphs in the partitions of graphs $H_r, \ldots, H_1$ to guarantee that the final partition of G is optimal. In particular, the algorithm makes sure that for $1 \leq i \leq r$, separator S_i induces an $\big(h \log^2 B\big)$-partition of H_i, and only the refinement step computing $S = S_0$ from S'_0 has the goal of producing an h-partition of $G = H_0$. Aleksandrov and Djidjev [25] show that for any graph of size N and any $h' > 0$, their algorithm computes a separator of size $\mathcal{O}\big(N/\sqrt{h'}\big)$ that induces an h'-partition of the graph. Hence, since we use the algorithm of [25] to compute S_r and to derive separator S_i from S'_i, for $0 \leq i < r$, $|S_r| = \mathcal{O}\big(|H_r|/\big(\sqrt{h}\log B\big)\big)$, and for $i > 0$, the construction of S_i from S'_i adds $\mathcal{O}\big(|H_i|/\big(\sqrt{h}\log B\big)\big)$ separator vertices to S'_i. By Properties (ii) and (iii), this implies that $|S'_0| = \mathcal{O}\big(N/\sqrt{h}\big)$. In order to obtain an h-partition of G, the algorithm of [25] adds another $\mathcal{O}\big(N/\sqrt{h}\big)$ separator vertices to S'_0, so that S induces an *optimal* h-partition of G.

The efficiency of the algorithm also follows from Properties (ii) and (iii). We have already argued that for $r = \log B$, $|H_r| = \mathcal{O}(N/B)$, so that the linear-time separator algorithm takes $\mathcal{O}(\text{scan}(N))$ I/Os to compute the initial $\big(h \log^2 B\big)$-partition of H_r. Property (ii) implies that separator S'_i induces a

$\left(ch \log^2 B\right)$-partition of H_i, for some constant $c \geq 1$. Under the assumption that $M \geq ch \log^2 B$, this implies that every connected component of $H_i - S'_i$ fits into main memory. Hence, the algorithm of [523] computes the connected components of $H_i - S'_i$, loads each of them into main memory and applies the internal memory algorithm of [25] to partition it into subgraphs of size at most $h \log^2 B$ (or h, if $i = 0$).

Since S_r can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os and the only external memory computation required to derive S_i from S'_i is computing the connected components of $H_i - S'_i$, the whole algorithm takes $\mathcal{O}(\text{scan}(N)) + \sum_{i=0}^{r-1} \mathcal{O}(\text{sort}(|H_i|)) = \sum_{i=0}^{r-1} \mathcal{O}\left(\text{sort}\left(N/2^i\right)\right) = \mathcal{O}(\text{sort}(N))$ I/Os.

In order to use the computed partition in the SSSP-algorithm from Section 5.5.1, it has to satisfy a few more stringent properties than optimality in the above sense. In particular, it has to be guaranteed that each of the $\mathcal{O}(N/h)$ subgraphs in the partition is adjacent to at most $\sqrt{h}$ separator vertices and that there are only $\mathcal{O}(N/h)$ boundary sets as defined in Section 5.5.1. In [775], it is shown that these properties can be ensured using a post-processing that takes $\mathcal{O}(\text{sort}(N))$ I/Os and increases the size of the computed separator by at most a constant factor. The construction is based on ideas from [315].

5.7.2 Grid Graphs

For a grid graph G, the geometric information associated with its vertices makes it very easy to compute an h-partition of G. In particular, every vertex stores its coordinates (i, j) in the grid. Then the separator S is chosen to contain all vertices in rows and columns $\sqrt{h}, 2\sqrt{h}, 3\sqrt{h}, \ldots$. Separator S has size $\mathcal{O}\left(N/\sqrt{h}\right)$ and partitions G into subgraphs of size at most h. That is, the computed partition is optimal. Since every vertex in a grid graph can be connected only to its eight neighboring grid vertices, each subgraph in the computed partition is adjacent to at most $4\sqrt{h}$ separator vertices. The number of boundary sets in the partition is $\mathcal{O}(N/h)$. Hence, this partition can be used in the shortest path algorithm from Section 5.5.1.

5.7.3 Graphs of Bounded Treewidth and Outerplanar Graphs

As in the case of the single-source shortest path problem, the computation of h-partitions for graphs of bounded treewidth and outerplanar graphs is similar. Again, for outerplanar graphs, a simplified algorithm producing a slightly better partition is presented in [775]; but the basic approach is the same as in the general algorithm for graphs of bounded treewidth, which we describe next.

As the shortest path algorithm, the algorithm starts by computing a nice tree-decomposition $\mathcal{D} = (T, \mathcal{X})$ of G using either the tree-decomposition algorithm from Section 5.8.2 or the outerplanar embedding algorithm from Section 5.8.3. The goal of the algorithm is to use the tree-decomposition to compute an optimal h-partition of G, i.e., a partition of G into subgraphs of size at most h using a separator of size $\mathcal{O}(kN/h)$, where k is the treewidth of G. To do this, tree T is processed from the leaves towards the root, starting with an empty separator S. For every node $i \in T$, a weight $\omega(i)$ is computed, which equals the size of the connected component of $G(i) - S$ containing the vertices in X_i. At a leaf, $\omega(i)$ is computed as follows: If $|G(i)| > h/2$, all vertices in X_i are added to S, and $\omega(i) = 0$. Otherwise $\omega(i) = |G(i)|$. At a forget node, $\omega(i) = \omega(j)$. At an introduce node, let $\omega'(i) = \omega(j) + 1$. If $\omega'(i) > h/2$, the vertices in X_i are added to S, and $\omega(i) = 0$. Otherwise $\omega(i) = \omega'(i)$. Finally, at a join node, let $\omega'(i) = \omega(j) + \omega(k)$. Then $\omega(i)$ is computed using the same rules as for an introduce node. It is easily verified that the computed vertex set S induces an h-partition of G. To see that S has size $\mathcal{O}(kN/h)$, observe that every group of $k + 1$ vertices added to S can be charged to a set of at least $h/2$ vertices in G and that these groups of charged vertices are disjoint. Hence, at most $(k + 1)N/(h/2) = \mathcal{O}(kN/h)$ vertices are added to S.

5.8 Gathering Structural Information

Having small separators is a structural property that all the graph classes we consider have in common. In this section we review I/O-efficient algorithms to gather more specific information about each class. In particular, we sketch algorithms for computing outerplanar and planar embeddings of outerplanar and planar graphs and tree-decompositions of graphs of bounded treewidth. These algorithms are essential, at least from a theoretical point of view, as all of the algorithms presented in previous sections, except the separator algorithm for planar graphs, require an embedding or tree-decomposition to be given as part of the input.

5.8.1 Planarity Testing and Planar Embedding

In order to test whether a given graph is planar, the algorithm of [523] exploits the fact that the separator algorithm from Section 5.7.1 does not require a planar embedding to be given as part of the input. In fact, the algorithm can be applied even without knowing whether G is planar. The strategy of the planar embedding algorithm is to use the separator algorithm and *try* to compute an optimal B^2-partition of G whose subgraphs $G_1, \dots, G_q$ have boundary size at most B. If the separator algorithm fails to produce the desired partition in $\mathcal{O}(\text{sort}(N))$ I/Os, the planar embedding algorithm terminates and reports that G is not planar. Otherwise the algorithm first tests whether each of the graphs $G_1, \dots, G_q$ is planar. If one of these graphs is non-planar, graph G cannot be planar. If graphs $G_1, \dots, G_q$ are planar, each graph G_i is replaced with a constraint graph C_i of size $\mathcal{O}(B)$. These constraint graphs have the property that graph G is planar if and only if the approximate graph A obtained by replacing each subgraph G_i with its constraint graph C_i is planar. If A is planar, a planar embedding of G is obtained from a planar embedding of A by locally replacing the embedding of each constraint graph C_i with a consistent planar embedding of G_i.

This approach leads to an I/O-efficient algorithm using the following observations: (1) Graphs $G_1, \dots, G_q$ have size at most B^2, and graphs $C_1, \dots, C_q$ have size $\mathcal{O}(B)$ each. Thus, the test of each graph G_i for planarity and the construction of the constraint graph C_i from G_i can be carried out in main memory, provided that $M \geq B^2$, which has to be true already in order to apply the separator algorithm. (2) Graph A has size $\mathcal{O}(N/B)$ because it is constructed from $\mathcal{O}(N/B^2)$ constraint graphs of size $\mathcal{O}(B)$, so that a linear time planarity testing and planar embedding algorithm (e.g., [144]) takes $\mathcal{O}(\text{scan}(N))$ I/Os to test whether A is planar and if so, produce a planar embedding of A. The construction of consistent planar embeddings of graphs $G_1, \dots, G_q$ from the embeddings of graphs $C_1, \dots, C_q$ can again be carried out in main memory.

This seemingly simple approach involves a few technicalities that are discussed in detail in [775]. At the core of the algorithm is the construction of the constraint graph C_i of a graph G_i. This construction is based on a careful analysis of the structure of graph G_i and beyond the scope of this survey. We refer the reader to [775] for details. However, we sketch the main ideas.

The construction is based on the fact that triconnected planar graphs are rigid in the sense that they have only two different planar embeddings, which can be obtained from each other by "flipping" the whole graph. The construction of constraint graph C_i partitions graph G_i into its connected components, each connected component into its biconnected components, and each biconnected component into its triconnected components. The connected components can be handled separately, as they do not interact with each other. The constraint graph of a connected component is constructed bottom-up from constraint graphs of its biconnected components, which in turn are constructed from constraint graphs of their triconnected components.

The constraint graph of a triconnected component is triconnected, and its embedding contains all faces of the triconnected component so that other parts of G may be embedded in these faces. The rest of the triconnected component is compressed as far as possible while preserving triconnectivity and planarity.

The constraint graph of a biconnected component is constructed from the constraint graphs of its triconnected components by analyzing the amount of interaction of these triconnected components with the rest of G. Depending on these interactions, the constraint graph of each triconnected component is either (1) preserved in the constraint graph of the biconnected component, (2) grouped with the constraint graphs of a number of other triconnected components, or (3) does not appear in the constraint graph of the biconnected component at all because it has no influence on the embedding of any part of G that is not in this biconnected component. In the second case, the group of constraint graphs is replaced with a new constraint graph of constant size. The constraint graph of a connected component is constructed in a similar manner from the constraint graphs of its biconnected components.

5.8.2 Computing a Tree-Decomposition

The algorithm of [522] for computing a tree-decomposition of width k for a graph G of treewidth k follows the framework of the linear-time algorithm for this problem by Bodlaender and Kloks [135, 136]. The algorithm can also be used to test whether a given graph G has treewidth at most k, as long as k is constant. The details of the algorithm are rather involved, so that we sketch only the main ideas here. In [135] it is shown that for every $k > 0$, there exist two constants $c_1, c_2 > 0$ so that for every graph G of treewidth k one of the following is true: (1) Every maximal matchingf G contains at least $c_1 N$ edges. (2) G contains a set X of at least $c_2 N$ vertices so that a tree-decomposition of width k for G can be obtained by attaching one node of size at most $k + 1$ per vertex in X to a tree-decomposition of width k for the graph $G - X$. In the latter case, the algorithm of [522] recursively computes a tree-decomposition of $G - X$ and then attaches the additional nodes in $\mathcal{O}(\text{sort}(N))$ I/Os. In the former case, it computes a tree-decomposition $\mathcal{D}'$ of width at most k for the graph G' obtained by contracting the edges in a maximal matching. The maximal matching can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using an algorithm of [775]. By replacing every vertex in G' that corresponds to an edge in the matching by the two endpoints of this edge, one immediately obtains a tree-decomposition of width at most $2k + 1$ for G. In order to obtain a tree-decomposition of width at most k, an I/O-efficient version of the algorithm of [136] is used, which computes the desired tree-decomposition in $\mathcal{O}(\text{sort}(N))$ I/Os. This algorithm starts by transforming the given tree-decomposition $\mathcal{D}'$ of width at most $2k + 1$ into a nice tree-decomposition of width at most $2k + 1$ and size $\mathcal{O}(N)$. Then dynamic programming is applied to this tree-decomposition, from the leaves towards the root, in order to compute an implicit representation of a tree-decomposition of width k for graph G. In a second pass of time-forward processing from the leaves towards the root, the tree-decomposition is extracted from this implicit representation. The details of this algorithm are complex and beyond the scope of this survey.

Given that each recursive step of the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os, the I/O-complexity of the whole algorithm is $\mathcal{O}(\text{sort}(N))$, since graphs G' and $G - X$ passed to recursive invocations of the algorithm contain only a constant fraction of the vertices and edges of G.

5.8.3 Outerplanarity Testing and Outerplanar Embedding

In order to compute an outerplanar embedding of an outerplanar graph, the algorithm of [775][3] exploits the following two observations: (i) An outerplanar embedding of an outerplanar graph can be obtained from outerplanar embeddings of the biconnected components of the graph, by making sure that no biconnected component is embedded in an interior face of another biconnected component. (ii) The boundary of the outer face of an outerplanar embedding of a biconnected outerplanar graph G is the only cycle in G that contains all vertices of G.

Observation (i) can be used to reduce the problem of computing an outerplanar embedding of G to that of computing outerplanar embeddings of its biconnected components. These components can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using the biconnectivity algorithm from Section 5.4.3, so that any outerplanar graph can be embedded in $\mathcal{O}(\text{sort}(N))$ I/Os if a biconnected outerplanar graph can be embedded in this number of I/Os. To achieve the latter, Observation (ii) is exploited.

The algorithm for embedding a biconnected outerplanar graph computes the boundary cycle C of G, numbers the vertices along C, and uses this numbering of the vertices to derive the final embedding of G. Assume for now that cycle C is given. Then the desired numbering of the vertices of G can be obtained by removing an arbitrary edge from cycle C and applying the Euler tour technique and list ranking to the resulting path in order to compute the distances of all vertices in this path from one of the endpoints of the removed edge. Given the numbering of the vertices along C, the edges incident to every vertex in G can be ordered clockwise around this vertex using the observation that these edges appear in the same order clockwise around v as their endpoints clockwise along C. Hence, an outerplanar embedding of G can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os if cycle C can be identified in $\mathcal{O}(\text{sort}(N))$ I/Os.

To compute cycle C, the algorithm exploits the fact that this cycle is unique. In particular, an algorithm that computes *any* cycle containing all vertices in G must produce cycle C. A cycle containing all vertices of G can be computed as follows from an open ear decomposition of G. Let $P_0, \dots, P_q$ be the ears in the ear decomposition. Then remove every ear P_i, $i > 0$, that consists of a single edge. The resulting graph is a biconnected subgraph of G that contains all vertices of G. For every remaining ear P_i, $i > 0$, remove edge (a, b) from G, where a and b are the endpoints of P_i. This procedure can easily be carried out using a constant number of sorts and scans, so that it takes $\mathcal{O}(\text{sort}(N))$ I/Os. The following argument shows that the resulting graph is the desired cycle C.

Let $P'_0, \dots, P'_r$ be the set of ears remaining after removing all ears P_i, $i > 0$, consisting of a single edge. Then the above construction is equivalent

[3] The algorithm for this problem originally presented in [521] is more complicated, so that we present the simplified version from [775] here.

to the following construction of graphs $G_1, \dots, G_r$. Graph G_1 is the union of ears P'_0 and P'_1. Since ear P'_0 consists of a single edge, and the endpoints of ear P'_1 are in P'_0, graph G_1 is a cycle. In order to construct graph G_i from cycle G_{i-1}, remove the edge connecting the endpoints of ear P'_i from G_{i-1} and add P'_i to the resulting graph. If the endpoints of ear P'_i are adjacent in G_{i-1}, graph G_i is again a cycle. But if the endpoints of P_i are not adjacent in G_{i-1}, it follows from the biconnectivity of G_{i-1} and the fact that P_i contains at least one internal vertex that graph G_i contains a subgraph that is homeomorphic to $K_{2,3}$, so that G_i and hence G cannot be outerplanar. Applying this argument inductively, we obtain that $G_r = C$.

The algorithm sketched above can easily be augmented to test whether a given graph is outerplanar. For details, we refer the reader to [775].

5.9 Conclusions and Open Problems

The algorithms for BFS, DFS, and SSSP on special classes of sparse graphs are a major step towards solving these problems on sparse graphs in general. In particular, the results on planar graphs have answered the long standing question whether these graphs allow $\mathcal{O}(\text{sort}(N))$ I/O solutions for these problems. However, all these algorithms are complex because they are based on computing separators. Thus, the presented results pose a new challenge, namely that of finding simpler, practical algorithms for these problems.

Since the currently best known separator algorithm for planar graphs requires that $M = \Omega\big(B^2 \log^2 B\big)$, the algorithms for BFS, DFS, and SSSP on planar graphs inherit this constraint. It seems that this memory requirement of the separator algorithm (and hence of the other algorithms as well) can be removed or at least reduced if the semi-external single source shortest path problem can be solved in $\mathcal{O}(\text{sort}(|E|))$ I/Os on arbitrary graphs. ("Semi-external" means that the vertices of the graph fit into main memory, but the edges do not.)

For graphs of bounded treewidth, the main open problem is finding an I/O-efficient DFS-algorithm. Practicality is not an issue here, as the chances to obtain practical algorithms for these graphs are minimal, as soon as the algorithms rely on a tree-decomposition.

For grid graphs, the presented shortest path algorithm uses a partition of the graph into a number of cells that depends on the size of the grid. This may be non-optimal if the graph is an extremely sparse subgraph of the grid. An interesting question here is whether it is possible to exploit the geometric information provided by the grid to obtain a partition of the same quality as the one obtained by the separator algorithm for planar graphs, but with much less effort, i.e., in a way that leads to a practical algorithm.

6. External Memory Computational Geometry Revisited

Christian Breimann and Jan Vahrenhold*

6.1 Introduction

Computational Geometry is an area in Computer Science basically concerned with the design and analysis of algorithms and data structures for problems involving geometric objects. This field started in the 1970's and has evolved into a discipline reaching out to areas such as Complexity Theory, Discrete and Combinatorial Geometry, or Algorithm Engineering. Geometric problems occur in a variety of applications, e.g., Computer Graphics, Databases, Geosciences, or Medical Imaging, and there are several textbooks presenting (internal memory) geometric algorithms [239, 275, 342, 541, 566, 596, 614, 647]. The systematic investigation of geometric algorithms specifically designed for massive data sets started in the early 1990's, most noticeably after Goodrich *et al.* presented their pioneering paper "External Memory Computational Geometry" [345].

In our survey, we intend to give an overview of results that have been obtained during the last decade and try to relate these results to internal memory algorithms as well. We will review algorithms and data structures for geometric problems involving massive data sets. Our focus will be both on theoretical results and on practical applications. Due to this double focus, this chapter contains not only an overview of fundamental geometric problems and corresponding specialized algorithms and data structures developed in the areas of Computational Geometry and Spatial Databases, but we also discuss how general-purpose index structures already implemented in commercial database systems can be used for solving geometric problems.

As a prominent application area involving massive data sets, *spatial database systems* have attracted increasing interest both in research communities and among professional users. In addition to the growing number of applications, the increasing availability of spatial data in form of digital maps and images is one of the main reasons for this trend, and tightly coupled to this, applications demand sophisticated computations and complex analyses of such data. In this area, it is not uncommon to use sub-optimal algorithms (in terms of their asymptotic complexity) if they lead to better performance in practice.

* Part of this work was done while on leave at the University for Health Informatics and Technology Tyrol, 6020 Innsbruck, Austria.

The geometric problems described in the remainder of this survey are grouped according to the kind of objects for which the problem is defined. In Section 6.3, we describe problems involving sets of points (Problems 6.1–6.11), in Section 6.4, we present a discussion of problems involving sets of segments (Problems 6.12–6.20), and in Section 6.5, we conclude by surveying problems involving sets of polygons (Problem 6.21 and Problem 6.22). Table 6.1 contains an overview of all problems covered in this chapter.

Table 6.1. Geometric problems surveyed in this chapter.

No.	Problem name	No.	Problem Name
1	Convex Hull	12	Segment Stabbing
2	Halfspace Intersection	13	Segment Sorting
3	Closest Pair	14	Endpoint Dominance
4	K-Bichromatic Closest Pairs	15	Trapezoidal Decomposition
5	Nearest Neighbor	16	Polygon Triangulation
6	All Nearest Neighbors	17	Vertical Ray-Shooting
7	Reverse Nearest Neighbors	18	Planar Point Location
8	K-Nearest Neighbors	19	Bichromatic Segment Intersection
9	Halfspace Range Searching	20	Segment Intersection
10	Orthogonal Range Searching	21	Rectangle Intersection
11	Voronoi Diagram	22	Polygon Intersection

Whenever appropriate, we will sub-classify problems according to the extent to which the sets may be updated:

Static Setting: All data items are fixed prior to running an algorithm or building a data structure, and no changes to the data items may occur afterwards.

Dynamic Setting: The set of items that forms the problem instance can be updated by insertions as well as deletions.

Semidynamic Setting: The set of items that forms the problem instance can be updated by either insertions or deletions, but not both.

The dynamic and semidynamic setting can also be considered in a *batched* variant, that is, all updates have to be known in advance. For problems that involve answering queries, we additionally distinguish between two kinds of queries:

Single-Shot Queries: Each query has to be answered independent of other queries and before the next query may be posed.

Batched Queries: The user specifies a collection of queries, and the only requirement is that all queries are answered by the end of the algorithm.

Before surveying geometric problems and corresponding solutions, we will briefly review the model of computation and introduce three general techniques for solving large-scale geometric problems.

6.2 General Methods for Solving Geometric Problems

External memory algorithms are investigated analytically in the *parallel disk model* introduced by Aggarwal and Vitter [17] and later refined by Vitter and Shriver [755]. The parallel disk model, which is based on blocked transfers, uses the following parameters:

N	=	Number of objects in the problem instance
M	=	Number of objects that fit simultaneously into main memory
B	=	Number of objects that fit into one disk block
D	=	Number of independent disks
P	=	Number of parallel processors

In this survey, however, we will restrict ourselves to algorithms for single-disk/single-processor settings, that is, we assume $D = 1$ and $P = 1$. For algorithms involving multiple queries, we consider two additional parameters:

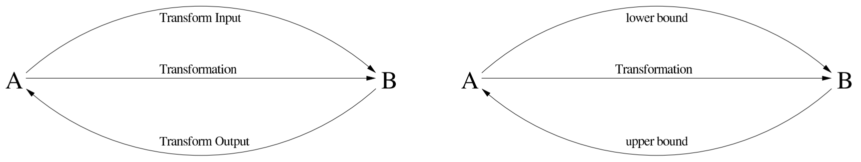
Q	=	Number of queries
Z	=	Number of objects in the answer set

This model allows computations only on elements that are in main memory, and whenever additional elements are needed in main memory, they have to be read from disk. The measures of performance for an external memory algorithm are the number of I/Os performed during its execution and the amount of disk space occupied (in terms of disk blocks).

In the remainder of this section, we present some general methods which are often used to solve geometric problems. First of all, we briefly discuss the implications of solving a problem by reducing it to a problem for which an efficient algorithm is known and present the concept of duality which sometimes can be used for this purpose (Section 6.2.1). In Section 6.2.2, we describe the general *distribution sweeping* paradigm, an external version of the well-known *plane sweeping*. Section 6.2.3 covers the R-tree, a spatial index structure frequently used in spatial database systems, and some of its variants.

6.2.1 Reduction of Problems

A common technique for proving lower bounds for (geometric) problems is to reduce the problem to some fundamental problem for which a lower bound is known. Among these fundamental problems is the *Element Uniqueness* problem which is, given a collection of N objects, to determine whether any two are identical. The lower bound for this problem is $\Omega((N/B)\log_{M/B}(N/B))$ [59], and—looking at the reduction from the opposite direction—a matching upper bound for the *Element Uniqueness* problem for points can be obtained by solving what is called the *Closest Pair* problem (see Problem 6.3). For a given collection of points, this problem consists of computing a pair with minimal distance. This distance is non-zero if and only if the collection does not contain duplicates, that is if and only if the answer to the *Element Uniqueness* problem is negative.



(a) Concept of transformation. (b) Transformation of bounds.

Fig. 6.1. Reduction of problems.

A more general view is given by Figure 6.1. It shows that reducing (transforming) a problem A to a problem B means transforming the input of A first, then solving problem B, and transforming its solution back afterwards (see Figure 6.1 (a)). Such a transformation is said to be a $\tau(N)$-transformation if and only if transforming both the input and the solution can be done in $\mathcal{O}(\tau(N))$ time. If an algorithm for solving the problem B has an asymptotic complexity of $\mathcal{O}(f_B(N))$, the problem A can be solved in $\mathcal{O}(f_B(N)+\tau(N))$ time. In addition, if the intrinsic complexity of the problem A is $\Omega(f_A(N))$ and if $\tau(N)\in o(f_A(N))$, then B also has a lower bound of $\Omega(f_A(N))$ (see Figure 6.1 (b) and, e.g., the textbook by Preparata and Shamos [614, Chap. 1.4]).

Reduction via Duality In Section 6.3, which is entitled "Problems Involving Sets of Points", we will discuss the following problem (Problem 6.2):

> "Given a set $\mathcal{S}$ of N halfspaces in $\mathbb{R}^d$, compute the common intersection of these halfspaces."

At first, it seems surprising that this problem should be discussed in a section devoted to problems involving set of points. Using the concept of *geometric duality*, however, points and halfspaces can be identified in a consistent way: A *duality transform* maps points in $\mathbb{R}^d$ into the set $\mathcal{G}^d$ of non-vertical hyperplanes in $\mathbb{R}^d$ and vice versa. The classical duality transform between points and hyperplanes is defined as follows:

$$\mathcal{D}:\begin{cases}\mathcal{G}^d \to \mathbb{R}^d : x_d = a_d + \sum_{i=1}^{d-1} a_i x_i \mapsto (a_1,\dots,a_d)\\ \mathbb{R}^d \to \mathcal{G}^d \;: (b_1,\dots,b_d) \qquad\qquad\quad \mapsto x_d = b_d - \sum_{i=1}^{d-1} b_i x_i\end{cases}$$

Another well-known transform $\mathcal{D}$ that is used, e.g., in the context of the *Convex Hull* problem (Problem 6.1), maps a point p on the unit parabola to the unique hyperplane that is tangent to the parabola in p. For the sake of simplicity, this duality transform is stated for $d = 2$.

$$\mathcal{D} : \begin{cases} \mathcal{G}^2 \to \mathbb{R}^2 : y = 2ax - b \mapsto (a, b) \\ \mathbb{R}^2 \to \mathcal{G}^2 \ : (a, b) \qquad\quad \mapsto y = 2ax - b \end{cases}$$

An important property of these transforms is that they are their own inverses and that they preserve the "above-below" relation: a point p lies above (below) the hyperplane ℓ with respect to the d-th dimension if and only if the line $\mathcal{D}(p)$ lies above (below) the point $\mathcal{D}(\ell)$ with respect to the d-th dimension. This property is exploited in several algorithms for, e.g., the *Range Searching* problem, the *Convex Hull* problem, the *K-Nearest Neighbors* problem, or the *Voronoi Diagram* problem. These algorithms first reduce the original problem to a problem stated for the duals of the original objects, solve the problem in the dual setting, and finally employ the same duality transform to obtain the solution to the original problem. We refer the interested reader to textbooks on Computational Geometry (e.g. [275, 541, 596]) for a more detailed treatment of these duality transforms.

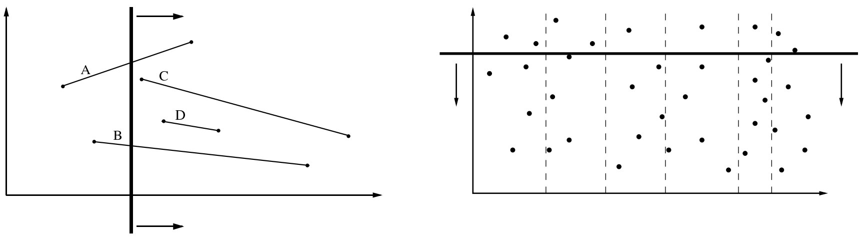
6.2.2 Distribution Sweeping

A large number of internal memory algorithms is based upon *plane sweeping*, a general technique for turning a static $(d+1)$-dimensional problem into a (finite) collection of instances of a dynamic d-dimensional problem. Although the general approach is independent of the dimension d, it is most efficient when the (original) problem is two-dimensional, and therefore we will restrict the following description to this setting.

The characterizing feature of the *plane sweeping* technique is an (imaginary) line (or, in the general setting, a hyperplane) that is swept over the entire data set. For sake of simplicity, this *sweep-line* is usually assumed to be perpendicular to the x-axis of the coordinate system and to move from left to right. Any object intersected by the sweep-line at $x = t$ is called *active at time t*, and only active objects are involved in geometric computations at that time. In the situation depicted in Figure 6.2(a), the sweep-line is drawn in bold, and the active objects, i.e., the objects intersected by the sweep-line, are the line segments A and B.

To guarantee the correctness of a plane-sweep algorithm, one has to take care of restating the original problem in such a way that operations involving only active objects are sufficient to determine the proper solution to the problem, e.g., Graf [350] summarized several formulations of plane-sweep algorithms.

All objects active at a given time are usually stored in a dictionary called *sweep-line structure*. The status of the sweep-line structure is updated as soon



(a) Internal plane sweeping. (b) External distribution sweeping.

Fig. 6.2. The *plane sweeping* and *distribution sweeping* techniques.

as the sweep-line moves to a point where the topology of the active objects changes discontinuously: for example, an object must be inserted into the sweep-line structure as soon as the sweep-line hits its leftmost point, and it must be removed after the sweep-line has passed its rightmost point. The sweep-line structure can be maintained in logarithmic time per update if the objects can be ordered linearly, e.g., by the y-value of their intersection with the sweep line.

For a finite set of objects, there are only finitely many points where the topology of the active objects changes discontinuously, e.g., when objects are inserted into or deleted from the sweep-line structure; these points are called *events* and are stored in increasing order of their x-coordinates, e.g, in a priority queue. Depending on the problem to be solved, there may exist additional event types apart from *insert* and *delete* events. The data structure for storing the events is called *event queue*, and maintaining it as a priority queue under insertions and deletions can be accomplished in logarithmic time per update. That is, if the active objects can be ordered linearly, each event can be processed in logarithmic time (excluding the time needed for operations involving active objects). As a consequence, the *plane sweeping* technique often leads to optimal algorithms, e.g., the *Closest Pair* problem can be solved in optimal time $\mathcal{O}(N \log_2 N)$ [398].

The straightforward approach for externalizing the *plane sweeping* technique would be to replace the (internal) sweep-line structure by a corresponding external data structure, e.g., a B-tree [96]. A plane-sweep algorithm with an internal memory time complexity of $\mathcal{O}(N \log_2 N)$ then spends $\mathcal{O}(N \log_B N)$ I/Os. For problems with an external memory lower bound of $\Omega((N/B) \log_{M/B}(N/B))$, however, the latter bound is at least a factor of B away from optimal.[1] The key to an efficient external sweeping technique is to combine sweeping with ideas similar to *divide-and-conquer*, that is, to subdivide the plane prior to sweeping. To aid imagination, consider the plane subdivided into $\Theta(M/B)$ parallel (vertical) strips, each containing the same

[1] Often the (realistic) assumption $M/B > B$ is made. In such a situation, an additional (non-trivial) factor of $\log_B M > 2$ is lost.

number of data objects (see Figure 6.2(b)).[2] Each of these strips is then processed using a sweep over the data and eventually by recursion. However, in contrast to the description of the internal case, the sweep-line is perpendicular to the y-axis, and sweeping is done from top to bottom. The motivation behind this modified description is to facilitate the intuition behind the novel ingredient of distribution sweeping, namely the subdivision into vertical strips.

The subdivision proceeds using a technique originally proposed for distribution sort [17], hence, the resulting external *plane sweeping* technique has been christened *distribution sweeping* [345]. While in the situation of distribution sort all partitioning elements have to be selected using an external variant of the *median find* algorithm [133, 307], distribution sweeping can resort to having an optimal external sorting algorithm at hand. The set of all x-coordinates is sorted in ascending order, and for each (recursive) subdivision of a strip, the $\Theta(M/B)$ partitioning elements can be selected from the sorted sequence spending an overall number of $\mathcal{O}(N/B)$ I/Os per level of recursion.

Using this linear partitioning algorithm as a subroutine, the distribution sweeping technique can be stated as follows: Prior to entering the recursive procedure, all objects are sorted with respect to the sweeping direction, and the set of x-coordinates is sorted such that the partitioning elements can be found efficiently. During each recursive call, the current data set is partitioned into M/B strips. Objects that interact with objects from other strips are found and processed during a sweep over the strips, while interactions between objects assigned to the same strip are found recursively. The recursion terminates when the number of objects assigned to a strip falls below M and the subproblem can be solved in main memory. If the sweep for finding inter-strip interactions can be performed using only a linear number of I/Os, i.e., $\Theta(N/B)$ I/Os, the overall I/O complexity for distribution sweeping is $\mathcal{O}((N/B)\log_{M/B}(N/B))$.

6.2.3 The R-tree Spatial Index Structure

Many algorithms proposed in the context of spatial databases assume that the data set is indexed by a hierarchy of bounding boxes. This assumption is justified by the popularity of the R-tree spatial index structure (and its variants) in academic and commercial database systems.

The *R-tree*, originally proposed by Guttman [368], is a height-balanced multiway tree similar to a B-tree. An R-tree stores d-dimensional data objects approximated by their axis-parallel minimum bounding boxes. For ease of

[2] Non-point objects cannot always be assigned to a unique strip because they may interact with several strips. In such a situation, objects are assigned to a maximal contiguous interval of strips they interact with. See the discussion of, e.g., Problem 6.20 for more details on how to deal with such a situation.

presentation, we restrict the following discussion to the situation $d = 2$ and assume that each data object itself is an axis-parallel rectangle.

The leaf nodes in an R-tree contain $\Theta(B)$ data rectangles each, where B is the maximum fanout of the tree. Internal nodes contain $\Theta(B)$ entries of the form (Ptr,R), where Ptr is a pointer to a child node and R the minimum bounding rectangle covering all rectangles which are stored in the subtree rooted in that child. Each entry in a leaf stores a data object or, in the general setting, the bounding rectangle of a data object and a pointer to the data object itself. Since the bounding rectangles stored within internal nodes are used to guide the insertion, deletion, and querying processes (see below), they are referred to as *routing rectangles*, whereas the bounding rectangles stored in the leaves are called *data rectangles*. An R-tree for N rectangles consists of $\mathcal{O}(N/B)$ nodes and has height $\mathcal{O}(\log_B N)$. Figure 6.3 shows an example of an R-tree for a set of two-dimensional rectangles.

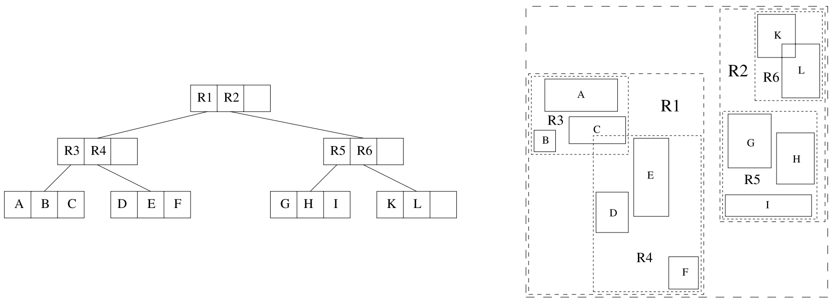


Fig. 6.3. R-tree for data rectangles A, B, C, ..., I, K, L. The tree in this example has maximum fanout $B = 3$.

To insert a new rectangle r into an already existing R-tree with root v, we select the subtree rooted at v whose bounding rectangle needs least enlargement to include the new rectangle. The insertion process continues recursively until a leaf is reached, adjusting routing rectangles as necessary. Since recursion takes place along a single root-to-leaf path, an insertion can be performed touching only $\mathcal{O}(\log_B N)$ nodes. If a leaf overflows due to an insertion, a rebalancing process similar to B-tree rebalancing is triggered, and therefore R-trees also grow and shrink only at the root. The insertion path depends not only on the heuristic chosen for breaking ties in case of non-unique subtrees for recursion, but also on the objects already present in the R-tree. Hence, there is no unique R-tree for a given set of rectangles, and different orders of insertion for the same set of rectangles usually result in different R-trees.

During the insertion process, a new rectangle r might overlap the routing rectangles of several subtrees of the node v currently visited. However, the rectangle r is routed to exactly one such subtree. Since the routing rectangle of this subtree might be extended to include r, the routing rectangles stored within v can overlap. This overlap directly affects the performance of R-tree query operations: When querying an R-tree to find all rectangles overlapping a given query rectangle r, we have to branch at each internal node into all subtrees whose minimum bounding rectangle overlaps r. (Queries for all rectangles containing a given query point p can be stated in the same way by regarding p as an infinitesimally small rectangle.) In the worst case, the search process has to branch at each internal node into all subtrees which results in $\mathcal{O}(N/B)$ nodes being touched—even though the number of reported overlapping data rectangles might be much smaller. Intuitively, it is thus desirable that the routing rectangles stored within a node overlap as little as possible.

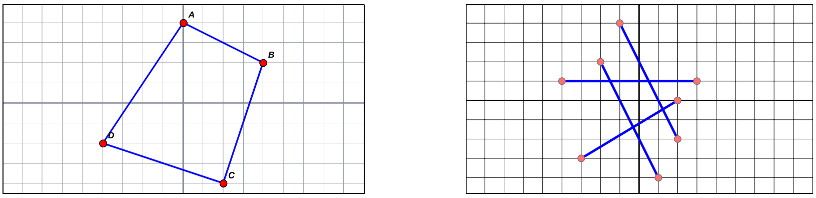
Another heuristic is to minimize the area covered by each routing rectangle. As a consequence routing rectangles cover less *dead space*, i. e., space covered by a routing rectangle which is not covered by any child, such that unsuccessful searches may terminate earlier. Similar heuristics are used in several variants of the R-tree including the R^+-tree [683], the *Hilbert R-tree* [442], and the R^*-tree [102], which is widely recognized to be the most practical R-tree variant. This is especially due to the fact that the heuristics used in the R^*-tree re-insert a certain number of elements if routing rectangles have to be split. This usually results in a re-structured tree with less overlapping of routing rectangles permitting fast answers for queries. We refer the interested reader also to the *Generalized Search Tree* [50, 389] and to more detailed overviews [323, 754].

As mentioned above, overlapping routing rectangles decrease the query performance of R-trees, and with increasing dimension, this overlap grows rapidly. Therefore, other data structures, e.g., the X-tree [114], which uses so-called *supernodes* permitting a sequential scan of their children, have been developed. But as the percentage of the data space covered by routing rectangles grows quickly with increasing dimensionality, for $d > 10$, nearly every node is accessed when querying the data structure as long as nodes are split in a balanced way. For many data distributions, a sequential scan can have better query performance in terms of overall running time than the random I/Os caused by querying data structures which are based on data-partitioning [758]. With the Pyramid-Technique [113], points and ranges in d-dimensional data space are transformed to 1-dimensional values which can be stored and queried using any 1-dimensional data structure, e.g., a B^+-tree. The authors claim that the Pyramid-Technique using a B^+-tree outperforms not only the data structures presented above but also the sequential scan.

We have presented some *hierarchical spatial index structures* which are used to efficiently store and query multi-dimensional data objects. From now on, whenever we refer to a hierarchical spatial index structure, any of these structures may be used unless explicitly stated otherwise.

6.3 Problems Involving Sets of Points

The first problem we discuss in this section is not only one of the most fundamental problems studied in Computational Geometry but also one of the rare problems where finding an optimal external algorithm for the two-dimensional case is completely straightforward.



(a) Planar convex hull. (b) Intersection of halfspaces in dual space.

Fig. 6.4. Computing the convex hull of a finite point set.

Problem 6.1 (Convex Hull). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$, find the smallest (convex) polytope enclosing $\mathcal{S}$ (see Figure 6.4(a)).

Among the earliest internal memory algorithms for computing the convex hull in two dimensions was a sort-and-scan algorithm due to Graham [352]. This algorithm, called *Graham's Scan*, is based upon the invariant that when traversing the boundary of a convex polygon in counterclockwise direction, any three consecutive points form a left turn. The algorithm first selects a point p that is known to be interior to the convex hull, e.g., the center of gravity of the triangle formed by three non-collinear points in $\mathcal{S}$. All points in $\mathcal{S}$ are then sorted by increasing polar angle with respect to p. The convex hull is constructed by pushing the points onto a stack in sorted order, maintaining the above invariant. As soon as the next point to be pushed and the topmost two points on the stack do not form a left turn, points are repeatedly removed from the stack until only one point is left or the invariant is fulfilled. After all points have been processed, the stack contains the points lying on the convex hull in clockwise direction. As each point can be pushed onto (removed from) the stack only once, $\Theta(N)$ stack operations are performed, and the (optimal) internal memory complexity, dominated by the sorting step, is $\mathcal{O}(N \log_2 N)$.

This algorithm is one of the rare cases where externalization is completely straightforward [345]. Sorting can be done using $\mathcal{O}((N/B) \log_{M/B}(N/B))$ I/Os [17], and an external stack can be implemented such that $\Theta(N)$ stack operations require $\mathcal{O}(N/B)$ I/Os (see Chapter 2). The external algorithm we obtain this way has an optimal complexity of $\mathcal{O}((N/B) \log_{M/B}(N/B))$.

In general, $\mathcal{O}(N)$ points of $\mathcal{S}$ can lie on the convex hull, but there are situations where the number Z of points on the convex hull is (asymptotically) much smaller. An *output-sensitive* algorithm for computing the convex hull in two dimensions has been obtained by Goodrich *et al.* [345]. Building upon the concept of *marriage-before-conquest* [458], the authors combine external versions of finding the median of an unsorted set [17] and of computing the convex hull of a partially sorted point set [343] to obtain an optimal output-sensitive external algorithm with complexity $\mathcal{O}((N/B)\log_{M/B}(Z/B))$.

Independent from this particular problem, Hoel and Samet [402] claimed that accessing disjoint decompositions of data space tends to be faster than other decompositions for a wide range of hierarchical spatial index structures. Along these lines, Böhm and Kriegel [138] presented two algorithms for solving the *Convex Hull* problem using spatial index structures. One algorithm, computing the minimum and maximum values for each dimension and traversing the index depth-first, is shown to be optimal in the number of disk accesses as it reads only the pages containing points not enclosed by the convex hull once. The second algorithm performs worse in terms of I/O but needs less CPU time. It is unclear, however, how to extend these algorithms to higher dimensions.

An approach to the d-dimensional *Convex Hull* problem is based on the observation that the convex hull of $\mathcal{S} \subset \mathbb{R}^d$ can be inferred from the intersection of halfspaces in the dual space $(\mathbb{R}^d)^*$ [781] (see also Figures 6.4(a) and (b)). For each point $p \in \mathcal{S}$, the corresponding dual halfspace is given by $p^* := \{x \in (\mathbb{R}^d)^* \mid \sum_{i=1}^{d} x_i p_i \leq 1\}$. At least for $d \in \{2,3\}$, the intersection of halfspaces can be computed I/O-efficiently (see the following Problem 6.2), and this results in corresponding I/O-efficient algorithms for the *Convex Hull* problem in these dimensions.

Problem 6.2 (Halfspace Intersection). Given a set $\mathcal{S}$ of N halfspaces in $\mathbb{R}^d$, compute the common intersection of these halfspaces.

In the context of the *Halfspace Intersection* problem, efficient external algorithms are known only for the situation $d \leq 3$. The intersection in three dimensions can be computed by either using an externalization of Reif and Sen's parallel algorithm [630] (as proposed by Goodrich *et al.* [345]) or by an algorithm that can be derived in the framework of randomized incremental construction with gradations [228] (see Section 6.4). Both algorithms require $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os (for the first approach, this bound holds with high probability, while it is the expected complexity for the second approach).

The problem we discuss next has already been mentioned in the context of solving problems by reduction (Section 6.2.1):

Problem 6.3 (Closest Pair). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ and a distance metric $\mathbf{d}$, find a pair $(p,q) \in \mathcal{S} \times \mathcal{S}$, $p \neq q$, for which $\mathbf{d}(p,q) = \min\{\mathbf{d}(r,s) \mid r,s \in \mathcal{S}, r \neq s\}$ (see Figure 6.5(a)).

There is a variety of optimal algorithms in the internal memory setting that solve the problem either directly or by exploiting reductions to other

(a) Closest pair. (b) Nearest neighbor for query point p.

Fig. 6.5. Closest-point problems.

problems (see also the survey by Smid [700]). In the external memory setting, the (static) problem of finding the closest pair in a fixed set $\mathcal{S}$ of N points can be solved by exploiting the reduction to the *All Nearest Neighbors* problem (Problem 6.6), where for each point $p \in \mathcal{S}$, we wish to determine its nearest neighbor in $\mathcal{S} \setminus \{p\}$ (see Problem 6.5). Having computed this list of N pairs of points, we can easily select two points forming a closest pair by scanning the list while keeping track of the closest pair seen so far. As we will discuss below, the complexity of solving the *All Nearest Neighbors* is $\mathcal{O}((N/B) \log_{M/B}(N/B))$, which gives us an optimal algorithm for solving the static *Closest Pair* problem.

Handling the dynamic case is considerably more involved, as an insertion or a deletion could change a large number of "nearest neighbors", and consequently, the reduction to the *All Nearest Neighbors* problem would require touching at least the same number of objects.

Callahan, Goodrich, and Ramaiyer [168] introduced an external variant of *topology trees* [316], and building upon this data structure, they managed to develop an external version of the dynamic closest pair algorithm by Bespamyatnikh [121]. The data structure presented by Callahan *et al.* can be used to dynamically maintain the closest pair spending $\mathcal{O}(\log_B N)$ I/Os per update.

The *Closest Pair* problem can also be considered in a bichromatic setting, where each point is labeled with either of two colors, and where we wish to report a pair with minimal distance among all pairs of points having different colors [10, 351]. This problem can be generalized to the case of reporting the K bichromatic closest pairs.

Problem 6.4 (K-Bichromatic Closest Pairs). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ with $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ and $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$, find K closest pairs $(p, q) \in \mathcal{S}_1 \times \mathcal{S}_2$.

Some efficient internal memory algorithms for solving this problem have been proposed [10, 451], but it seems that none of them can be externalized efficiently. In the context of spatial databases, the *K-Bichromatic Closest Pairs* problem can be seen as a special instance of a so-called *θ-join* which is defined as follows: Given two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of objects and a predicate $\theta : \mathcal{S}_1 \times \mathcal{S}_2 \to \mathbb{B}$, compute all pairs $(s_1, s_2) \in \mathcal{S}_1 \times \mathcal{S}_2$, for which $\theta(s_1, s_2) = \texttt{true}$.

In his approach to the *K-Bichromatic Closest Pairs* problem,Henrich [393] considered the special case $|\mathcal{S}_2| = 1$, and assuming that $\mathcal{S}_1$ is indexed hierarchically, he proposed to perform a priority-driven traversal of the spatial index structure storing $\mathcal{S}_1$. Hjaltason and Samet [401] later generalized this approach and referred to Problem 6.4 as a special instance of a θ-join, namely the *incremental distance join* (again assuming that each relation is indexed hierarchically). Their algorithm schedules a priority-driven synchronous traversal of both trees, repeatedly looking at two nodes, one from each tree. The processing is guided by the distance between the (routing) rectangles corresponding to the nodes, and to each pair this distance is assigned as the pair's priority. Initially, the priority queue contains all pairs that can be formed by grouping the root of one tree and the children of the root of the other tree, and the first element in the queue always forms the closest pair of objects stored in the queue. For each removed pair of nodes, the pairs formed by the children (if any) are inserted into the queue. Whenever a pair of data objects appears at the front of the queue, its associated distance is minimal among all unconsidered distances, hence, all K bichromatic closest pairs can be reported ordered by increasing distance. This algorithm benefits from the observation that in practical applications $K \ll |\mathcal{S}_1 \times \mathcal{S}_2|$, but nevertheless, the priority queue might contain a large number of pairs. Hjaltason and Samet described several approaches for how to organize the priority queue such that only a small portion of it actually resides in main memory. This means that only the promising candidate pairs are kept in main memory whereas all pairs having a large distance are off-loaded to external memory. The authors argue that except for unlikely worst-case configurations, their approaches perform without accessing the off-loaded data and that worst-case configurations can be handled gracefully as well. Worst-case optimal external priority queues are also discussed in Chapter 2 and Chapter 3.

Corral *et al.* [219] presented a collection of algorithms that improve the effective running time of the above algorithms for solving the *Bichromatic Closest Pair* problem. These improvements include a separate treatment for the case $K = 1$ and choosing a heap-based priority queue.

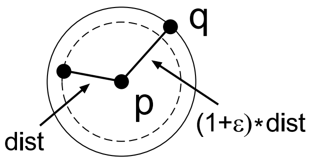
These algorithms for solving Problem 6.4 can be modified to solve (the monochromatic) Problem 6.3. This modification is not generally possible [219] for an arbitrary algorithm solving Problem 6.4. A description of modifications for the latter algorithm has been given by Corral *et al.* [218]. The authors claim that these modifications do not seriously affect the performance of their algorithm.

As mentioned before, spatial index structures try to cluster objects based upon their spatial location, and consequently, several approaches have been made to exploit this structural property when dealing with *proximity problems*. A fundamental proximity problem is to organize a set of points such that for each query point the point closest to it can be reported quickly.

Problem 6.5 (Nearest Neighbor). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$, a distance metric $\mathbf{d}$, and a query point p in $\mathbb{R}^d$, report a point $q \in \mathcal{S}$, for which $\mathbf{d}(p, q) = \min\{\mathbf{d}(p, r) \mid r \in \mathcal{S}\}$ (see Figure 6.5(b)).

Problem 6.5 and its relatives occur in a variety of conventional geographical applications, e.g., when searching for the closest geometric feature of some kind relative to some given spatial location. This problem is also referred to as the *Post Office* problem [460][3]. Since the metric **d** defining the "closeness" of two objects is also a parameter in the problem setting, this problem can be found in new application areas like multimedia database systems. In this setting, multimedia objects, e.g., text, image, or video objects, are described by high-dimensional *feature vectors* which in turn are considered as points in the *feature space*. Proximity among these feature vectors implies similarity between the objects represented, and in combination with carefully chosen metrics, spatial index structures can be used for efficiently performing similarity search [137, 473, 668, 681].

The *Nearest Neighbor* problem can also be restated in the context of *Voronoi diagrams* (see Problem 6.11), and using techniques by Goodrich *et al.* [345], one can obtain a static data structure that answers nearest neighbor queries in $\mathcal{O}(\log_B N)$ I/Os. We will comment on this approach when discussing algorithms for computing the Voronoi diagram.



(a) Approximate nearest neighbor.

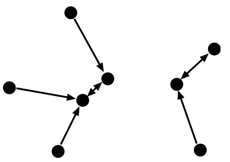

(b) All nearest neighbors.

Fig. 6.6. Nearest-neighbor problems.

A variant of the *Nearest Neighbor* problem is to compute an *approximate nearest neighbor* for a given query point. Here, an additional parameter ε is used to allow for certain slack in the reported "minimum" distance. For $\varepsilon > 0$, a $(1+\varepsilon)$-approximate nearest neighbor of a query point p is a point q that is no further than $(1+\varepsilon)$ times the distance *dist* to the actual nearest neighbor of p (see Figure 6.6(a)).

Using external topology trees, Callahan *et al.* [168] derived an external version of the data structure by Arya *et al.* [72] that can be used to maintain $\mathcal{S}$ under insertions and deletions with $\mathcal{O}(\log_B N)$ I/Os per update such that an approximate nearest neighbor query can be answered spending $\mathcal{O}(\log_B N)$ I/Os.[4] Even in the internal memory setting, it is an open problem to find an efficient dynamic data structure with $\mathcal{O}(N \log^{\mathcal{O}(1)} N)$ space that can be used for the exact *Nearest Neighbor* problem and has $\mathcal{O}(\log^{\mathcal{O}(1)} N)$ update

[3] This reference is ascribed to Knuth as he discusses a data structure called *post-office tree* which can be used for answering a query of the kind "What is the nearest city to point x?", given the value of x [460, page 563].

[4] The constants hidden in the "Big-Oh"-notation depend on d and ε.

and query time [700], and not surprisingly, the external memory variant of this problem is unsolved as well.

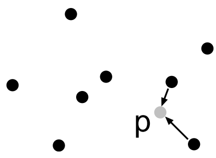
Berchtold *et al.* [112] proposed to use hierarchical spatial index structures to store the data points. They also introduced a different cost model and compared the predicted and actual cost of solving the *Nearest Neighbor* problem for real-world data using an X-tree [114] and a Hilbert-R-tree [287]. Brin [148] introduced the *GNAT* index structure which resembles a hierarchical Voronoi diagram (see Problem 6.11). He also gave empirical evidence that this structure outperforms most other index structures for high-dimensional data spaces.

The practical relevance of the nearest neighbor, however, becomes less significant as the number of dimensions increases. For both real-world and synthetic data sets in high-dimensional space ($d > 10$), Weber, Schek, and Blott [759] as well as Beyer *et al.* [123] showed that under several distance metrics the distance to the nearest neighbor is larger than the distance between the nearest neighbor and the farthest neighbor of the query point. Their observation raises an additional *quality issue*: The exact nearest neighbor of a query point might not be relevant at all. As an approach to cope with this complication, Hinneburg, Aggarwal, and Keim [397] modified the *Nearest Neighbor* problem by introducing the notion of *important dimensions.* They introduced a quality criterion to determine which dimensions are relevant to the specific proximity problem in question and examined the data distribution resulting from projections of the data set to these dimensions. Obviously, their approach yields improvements over standard techniques only if the number of "important" dimensions is significantly smaller than the dimension of the data space.

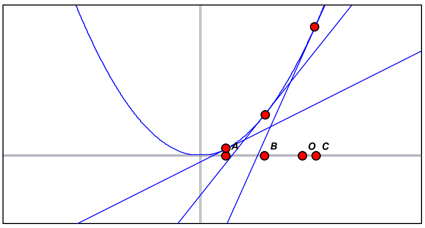
Problem 6.6 (All Nearest Neighbors). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ and a distance metric $\mathbf{d}$, report for each point $p \in \mathcal{S}$ a point $q \in \mathcal{S}$, for which $\mathbf{d}(p,q) = \min\{\mathbf{d}(p,r) \mid r \in \mathcal{S}, p \neq r\}$ (see Figure 6.6(b)).

The *All Nearest Neighbors* problem, which can also be seen as a special batched variant of the (single-shot) *Nearest Neighbor* problem, can be posed, e.g., in order to find clusters within a point set. Goodrich *et al.* [345] proposed an algorithm with $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/O-complexity based on the distribution sweeping paradigm: Their approach is to externalize a parallel algorithm by Atallah and Tsay [74] replacing work on each processor by work within a single memory load. Recall that on each level of distribution sweeping, only interactions *between* strips are handled, and that interactions within a strip are handled recursively. In the situation of finding nearest neighbors, the algorithm performs a top-down sweep keeping track of each point whose nearest neighbor above does not lie within the same strip. The crucial observation by Atallah and Tsay is that there are at most four such points in each strip, and by choosing the branching factor of distribution sweeping as $M/(5B)$, the (at most) four blocks per strip containing these points as well as the $M/(5B)$ blocks needed to produce the input for the recursive steps can be kept in main memory. Nearest neighbors within the same strip are found recursively, and the result is combined with the result of a second bottom-up sweep to produce the final answer.

In several applications, it it desirable to compute not only the exact nearest neighbors but to additionally compute for each point the K points closest to it. An algorithm for this so-called *All K-Nearest Neighbors* problem has been presented by Govindarajan *et al.* [346]. Their approach (which works for an arbitrary number d of dimensions) builds upon an external data structure to efficiently maintain a *well-separated pair decomposition* [169]. A well-separated pair decomposition a set $\mathcal{S}$ of points is a hierarchical clustering of $\mathcal{S}$ such that any two clusters on the same level of the hierarchy are farther apart than any to points within the same cluster, and several internal memory algorithms have been developed building upon properties of such a decomposition. The external data structure of Govindarajan *et al.* occupies $\mathcal{O}(KN/B)$ disk blocks and can be used to compute all K-nearest neighbors in $\mathcal{O}((KN/B)\log_{M/B}(KN/B))$ I/Os. Their method can also be used to compute the K closest pairs in d dimensions in $\mathcal{O}(((N+K)/B)\log_{M/B}((N+K)/B))$ I/Os using $\mathcal{O}((N+K)/B)$ disk blocks.



(a) Reverse nearest neighbors for point p.



(b) K-nearest neighbors via lifting.

Fig. 6.7. Non-standard nearest-neighbor problems.

Problem 6.7 (Reverse Nearest Neighbors). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$, a distance metric $\mathbf{d}$, and a query point p in $\mathbb{R}^d$, report all points $q \in \mathcal{S}$, for which $\mathbf{d}(q,p) = \min\{\mathbf{d}(q,r) \mid r \in (\mathcal{S} \cup \{p\}) \setminus \{q\}\}$ (see Figure 6.7(a)).

The *Reverse Nearest Neighbors* problem has been introduced in the spatial database setting by Korn and Muthukrishnan [472] who also presented static and dynamic solutions for the bichromatic an monochromatic problem. For simplicity, we only discuss the solution to the static monochromatic problem here, as for their approach only minor modifications are needed to solve the other three problems. In a preprocessing step, the *All Nearest Neighbors* problem is solved for $\mathcal{S}$. Each point q and its nearest neighbor r define a ball centered at q with radius $\mathbf{d}(q,r)$. All N such balls are stored in a spatial index structure that can be used to report, given a query point p, all balls containing p. It is easy to verify that the points corresponding to the balls that contain p are exactly the points having p as their nearest neighbor in $\mathcal{S} \cup \{p\}$. In the internal memory setting, at least the static version of the *Reverse Nearest Neighbor* problem can be solved efficiently [524]. The main problem when trying to efficiently solve the problem in a dynamic setting is that updating $\mathcal{S}$ essentially involves finding nearest neighbors in a dynamically changing point set, and—as discussed in the context of Problem 6.5—no efficient solution with at most polylogarithmic space overhead is known.

Problem 6.8 (K-Nearest Neighbors). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$, an integer K with $1 \leq K \leq N$, and a query point p in $\mathbb{R}^d$, report K points $q_i \in \mathcal{S}$ closest to p.

Agarwal *et al.* [6] solved the two-dimensional *K-Nearest Neighbors* problem in the dual setting: using a duality transform, they proposed to map each two-dimensional point (a_1, a_2) to the hyperplane $z = a_1^2 + a_2^2 - 2a_1x - 2a_2y$ which is tangent to the unit parabola at the (lifted) point $(a_1, a_2, a_1^2 + a_2^2)$. In this setting, the problem of finding the K nearest neighbors for a point $p = (x_p, y_p)$ can be restated as finding the K highest hyperplanes above the point $(x_p, y_p, 0)$ (For the sake of simplicity, the corresponding one-dimensional problem is sketched in Figure 6.7(b). Consider, e.g., point O: The two highest hyperplanes lying above O are defined by lifting points B and C which are also the two nearest neighbors of O). Using an external version of Chan's algorithm for computing $(\leq k)$-levels of an arrangement [175], Agarwal *et al.* [6] developed a data structure for range searching among halfplanes that, after spending $\mathcal{O}((N/B)\log_2 N \log_B N)$ expected I/Os for preprocessing, occupies an expected number of $\mathcal{O}((N/B)\log_2(N/B))$ disk blocks. This data structure can be used to report the K highest halfplanes above a query point, and by duality, the K nearest neighbors in the original setting, spending $\mathcal{O}(\log_B N + K/B)$ expected I/Os per query.

In addition to the quite involved data structure mentioned above, spatial index structures have been considered to solve the *K-Nearest Neighbor* problem [190, 473, 640, 681]. Much attention has been paid to pruning parts of the candidate set [681] and to removing inefficient heuristics [190]. As mentioned above, the performance of most index structures degrades for high dimensions, and even while the Pyramid-Technique [113] can be used for uniformly distributed data in high dimensions, its performance degrades for non-uniformly distributed data. To overcome this deficiency, Yu *et al.* [774] presented a new approach called *iDistance* which is adaptable with respect to data distribution. They propose to partition the data space according to its characteristics and, for each partition, to index the *distance* between contained data points and a reference point using a B$^+$-tree. Their algorithm can be used to incrementally refine approximate answers such that early during the algorithm, approximate results can be output if desired. In contrast, the *VA-file* of Weber *et al.* [758, 759] uses approximated data to produce a set of candidate pairs during nearest neighbor search. It partitions the data space into cells and stores unique bit strings for these cells in an (optionally compressed) array. During a sequential scan of this array, candidates are determined by using the stored approximations, before these candidates are further examined to obtain the final result.

Establishing a trade-off between used disk space and obtained query time, Goldstein and Ranakrishnan [338] presented an approach to reduce query time by examining some characteristics of the data and storing redundant information. Following their approach the user can explicitly relate query performance and disk space, i.e., more redundant information can be stored to improve query performance and vice versa. With a small percentage of only approximately correct answers in the final result, this approach leads to sub-linear query processing for high dimensions.

The description of algorithms for the *K-Nearest Neighbors* problem concludes our discussion of proximity problems, that is of selecting certain points according to their proximity to one or more query points. The next two problems also consist of selecting a subset of the original data, namely the set contained in a given query range. These problems, however, have been discussed in detail by recent surveys [11, 56, 754], so we only sketch the main results in this area.

(a) Halfspace range searching. (b) Orthogonal range searching.

Fig. 6.8. Range searching problems.

Problem 6.9 (Halfspace Range Searching). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ and a vector $\mathbf{a} \in \mathbb{R}^d$, report all Z points $x \in \mathcal{S}$, for which $x_d \leq a_d + \sum_{i=1}^{d-1} a_i x_i$.

The main source for solutions to the halfspace range searching problem in the external memory setting is the paper by Agarwal *et al.* [6]. The authors presented a variety of data structures that can be used for halfspace range searching classifying their solutions in linear and non-linear space data structures. All proposed algorithms rely on the following duality transform and the fact that it preserves the "above-below" relation.

$$\mathcal{D} : \begin{cases} \mathcal{G}^d \to \mathbb{R}^d : x_d = a_d + \sum_{i=1}^{d-1} a_i x_i \mapsto (a_1, \ldots, a_d) \\ \mathbb{R}^d \to \mathcal{G}^d : (b_1, \ldots, b_d) \mapsto x_d = b_d - \sum_{i=1}^{d-1} b_i x_i \end{cases}$$

In the linear space setting, the general problem for $d > 3$ can be solved using an external version of a *partition tree* [535] spending for any fixed $\varepsilon > 0$ $\mathcal{O}((N/B)^{1-1/d+\varepsilon} + Z/B)$ I/Os per query. The expected preprocessing complexity is $\mathcal{O}(N \log_2 N)$ I/Os. For *simplex range searching* queries, that is for reporting all points in $\mathcal{S}$ lying inside a given query simplex with μ faces of all dimensions, $\mathcal{O}((\mu N/B)^{1-1/d+\varepsilon} + Z/B)$ I/Os are sufficient. For *halfspace range searching* and $d = 2$, the query cost can be reduced to $\mathcal{O}(\log_B N + Z/B)$ I/Os (using $\mathcal{O}(N \log_2 N \log_B N)$ expected I/Os to preprocess an external version of a data structure by Chazelle, Guibas, and Lee [184]). Using partial rebuilding, points can also be inserted into/removed from $\mathcal{S}$ spending amortized $\mathcal{O}(\log_2(N/B) \log_B N)$ I/Os per update.

If one is willing to spend slightly super-linear space, the query cost in the three-dimensional setting can be reduced to $\mathcal{O}(\log_B N + Z/B)$ I/Os at the expense of an expected overall space requirement of $\mathcal{O}((N/B) \log_2(N/B))$ disk blocks. This data structure externalizes a result of Chan [175] and can be constructed spending an expected number of $\mathcal{O}((N/B) \log_2(N/B) \log_B N)$ I/Os. Alternatively, Agarwal *et al.* [6] propose to use external versions of *shallow partition trees* [536] that use $\mathcal{O}((N/B) \log_B N)$ space and can answer a query spending $\mathcal{O}((N/B)^{\varepsilon} + Z/B)$ I/Os. This approach can also be generalized to an arbitrary number d of dimensions: a halfspace range searching query can be answered spending $\mathcal{O}((N/B)^{1-1/\lfloor d/2 \rfloor+\varepsilon} + Z/B)$ I/Os. The exact complexity of halfspace range searching is unknown—even in the well-investigated internal memory setting, there exist several machine model/query type combinations where no matching upper and lower bounds are known [11].

Problem 6.10 (Orthogonal Range Searching). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ and d (possibly unbounded) intervals $[l_i, r_i]$, report all Z points $x \in \mathcal{S}$ for which $x \in [l_1, r_2] \times \ldots \times [l_d, r_d]$.

The more restricted *Orthogonal Range Searching* problem can obviously be solved by storing the data points (considered as infinitesimally small rectangles) in a spatial index structure and by performing a range query (*window query*). The actual query time, however, depends on the heuristic for clustering nodes, and in the worst case, the index structure has to be traversed completely—even if $Z \in \mathcal{O}(1)$. Despite this disadvantage, most of these index structures occupy only linear space and support updates I/O-efficiently. Occupying only linear space has been recognized as a conceptual advantage that may cancel the disadvantage of a theoretically high query cost, and the notion of indexability has been introduced to investigate possible trade-offs between storage redundancy and access overhead in the context of range searching [388].

An external data structure that uses linear space and efficiently supports both updates and queries has been proposed by Grossi and Italiano [360]. The authors externalized their internal memory *cross-tree*, which can be seen as a cross-product of d one-dimensional index structures, and obtained a data structure that can be updated in $\mathcal{O}(\log_B N)$ I/Os per update and orthogonal range queries in $\mathcal{O}((N/B)^{1-1/d} + Z/B)$ I/Os per query. The external cross-tree can be built in $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os. In a different model that excludes threaded data structures like the cross-tree, Kanth and Singh [444] obtained similar bounds (but with amortized update complexity) by layering B-trees and k-D-trees. Their paper additionally includes a proof of a matching lower bound.

The *Orthogonal Range Searching* problem has also been considered in the batched setting: Arge *et al.* [65] and Goodrich *et al.* [345] showed how to solve the two-dimensional problem spending $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/Os using linear space. Arge *et al.* [65] extended this result to higher dimensions and obtained a complexity of $\mathcal{O}((N/B)\log_{M/B}^{d-1}(N/B) + Z/B)$ I/Os. The one-dimensional batched dynamic problem, i.e., all Q updates are known in advance, can be solved in $\mathcal{O}(((N+Q)/B)\log_{M/B}(N+Q)/B + Z/B)$ I/Os [65], but no corresponding bound is known in higher dimensions.

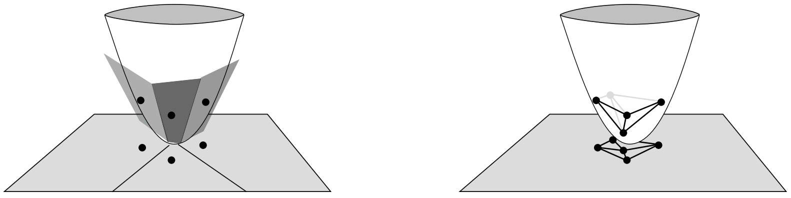
Problems that are slightly less general than the *Orthogonal Range Searching* problem are the (two-dimensional) *Three-Sided Orthogonal Range Searching* and *Two-Sided Orthogonal Range Searching* problem, where the query range is unbounded at one or two sides. Both problems have been considered by several authors [129, 421, 443, 624, 709, 750], most recently by Arge, Samoladas, and Vitter [67] in the context of *indexability* [388]—see also more specific surveys [11, 56, 754].

Another recent development in the area of range searching are algorithms for range searching among moving objects. In this setting, each object is assigned a (static) "flight plan" that determines how the position of an object changes as a (continuous) function of time. Using external versions of *partition trees* [535], Agarwal, Arge, and Erickson [5] and Kollios and Tsotras [463] developed efficient data structures that can be used to answer orthogonal range queries in one and two dimensions spending $\mathcal{O}((N/B)^{1/2+\varepsilon} + Z/B)$ I/Os. These solutions are *time-oblivious* in the sense that the complexity of a range query does not depend on how far the point of time of the query is in the future. *Time-responsive* solutions that answer queries in the near future (or past) faster than queries further away in time have been proposed by Agarwal *et al.* [5] and by Agarwal, Arge, and Vahrenhold [8].

We conclude this section by discussing the *Voronoi diagram* and its graph-theoretic dual, the *Delaunay triangulation.* Both structures have a variety of proximity-related applications, e.g., in Geographic Information Systems, and we refer the interested reader to more specific treatments of how to work with these structures [76, 275, 336].

Problem 6.11 (Voronoi Diagram). Given a set $\mathcal{S}$ of N points in $\mathbb{R}^d$ and a distance metric $\mathbf{d}$, compute for each point $p \in \mathcal{S}$ its Voronoi region $V(p, \mathcal{S}) := \{x \in \mathbb{R}^d \mid \mathbf{d}(x,p) \leq \mathbf{d}(x,q), q \in \mathcal{S} \setminus \{p\}\}$.

Given the above definition, the *Voronoi diagram* consists of the union of all N Voronoi regions which are disjoint except for a possibly shared boundary.



(a) Voronoi diagram via lifting. (b) Delaunay triangulation via lifting.

Fig. 6.9. Computing the Voronoi diagram and the Delaunay triangulation.

An optimal algorithm for computing the Voronoi diagram can be obtained by a transformation already used for solving the *K-Nearest Neighbors* problem (Problem 6.8). The key idea is that a Voronoi region for a point $p \in \mathcal{S}$ contains exactly those points in $\mathbb{R}^d$ that have p as their nearest neighbor with respect to $\mathcal{S}$. To compute the Voronoi diagram in d dimensions, each point is lifted to the $(d+1)$-dimensional unit parabola, and the intersection of the halfspaces dual to these points is computed (see Figure 6.9(a)). As already mentioned in the discussion of the *K-Nearest Neighbors* problem (Problem 6.8), the highest plane above a d-dimensional point is dual to the lifted version of its nearest neighbor [275], and consequently, the projection of the intersection of halfspaces back to d-dimensional space results in the Voronoi diagram. As the intersection of halfspaces can be computed efficiently in two and three dimensions (see Problem 6.2), the Voronoi diagram in one and two dimensions can be constructed using the above transformation. It should be noted that a similar transformation, namely computing the convex hull (Problem 6.1) of the lifted points (see Figure 6.9(b)) can be used to compute the graph-theoretic dual of the Voronoi diagram, the *Delaunay triangulation*, in two and three dimensions.

The Voronoi diagram can be used to solve the (static) *Nearest Neighbor* problem (Problem 6.5). This is due to the observation that each query point q that does not lie on a shared boundary of Voronoi regions falls into exactly one Voronoi region, say the region belonging to some point $p \in \mathcal{S}$. By definition, this region contains all points in the plane that are closer to p than to any other point of $\mathcal{S}$, that is, all points for which p is the nearest neighbor with respect to $\mathcal{S}$. In order to find the region containing the query point q, one has to solve the *Point Location* problem. An algorithm for solving this problem—formally defined as Problem 6.18 in Section 6.4—can be used to answer a *Nearest Neighbor* query for a static set $\mathcal{S}$ in $\mathcal{O}(\log_B N)$ I/Os.

In the internal memory setting, a variety of two-dimensional problems can be solved by using either the Voronoi diagram or the Delaunay triangulation. Almost all these solutions require one of these structures to be traversed, and as both structures are planar graphs, we refer to Chapter 5 for details on the external memory complexity of such traversals.

6.4 Problems Involving Sets of Line Segments

We begin this section by stating a geometric problem that is inherently one-dimensional even though it is formulated in a two-dimensional setting. This problem serves also as a vehicle for introducing the *interval tree* data structure. The external memory version of this data structure is a building block for several efficient algorithms and its description can also be used to demonstrate design techniques for externalizing data structures.

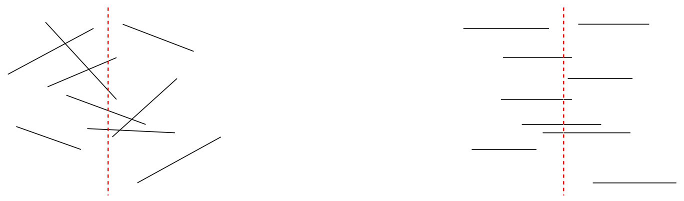

(a) Stabbing a set of segments. (b) Stabbing a set of intervals.

Fig. 6.10. Reducing the *Segment Stabbing* problem to a one-dimensional setting.

Problem 6.12 (Segment Stabbing). Given a set $\mathcal{S}$ of N segments in the plane and a vertical line $\ell = x$, compute all Z segments in $\mathcal{S}$ intersected by ℓ (see Figure 6.10(a)).

The key observation leading towards an optimal algorithm is that the segments stabbed by ℓ are exactly those segments in $\mathcal{S}$ whose projections onto the x-axis contain the point $(x, 0)$ (see Figure 6.10(b)). In the internal memory setting, this reduced problem can be solved optimally, that is spending $\mathcal{O}(\log_2 N + Z)$ time and linear space, by using the so-called *interval tree* [274]. An interval tree is a perfectly balanced binary search tree over the set of x-coordinates of all endpoints in $\mathcal{S}$ (hereafter referred to as "x-coordinates in $\mathcal{S}$"), and data elements are stored in internal nodes as well as in leaf nodes. Each node corresponds to the median of (interval of) all x-coordinates in $\mathcal{S}$ stored in the subtree rooted at that node, e.g., the root corresponds to the median of all x-coordinates in $\mathcal{S}$. The x-coordinate stored at an internal node v naturally partitions the set stored in the corresponding subtree into two slabs, and a segment in $\mathcal{S}$ is stored in a secondary data structure associated with v, if and only if it crosses the boundary between these slabs and does not cross any slab boundary induced by v's parent. The interval tree storing $\mathcal{S}$ can be updated (that is insertions and deletions can be performed) in $\mathcal{O}(\log_2 N)$ time per update.[5]

Arge and Vitter [71] obtained an optimal external memory solution for the *Segment Stabbing* problem by developing an external version of the interval

[5] The insertion bound is amortized if the set of x-coordinates in $\mathcal{S}$ is augmented due to this insertion.

tree. Their data structure occupies linear space and can be used to answer stabbing queries spending $\mathcal{O}(\log_B N + Z/B)$ I/Os per query. As in the internal setting, the data structure can be made dynamic, and the resulting dynamic data structure supports both insertions and deletions with $\mathcal{O}(\log_B N)$ worst-case I/O-complexity.

The externalization technique used by Arge and Vitter is of independent interest, hence, we will present it in a little more detail. In order to obtain a query complexity of $\mathcal{O}(\log_B N + Z/B)$ I/Os, the fan-out of the base tree has to be in $\mathcal{O}(B^c)$ for some constant $c > 0$, and for reasons that will become clear immediately, this constant is chosen as $c = 1/2$. As mentioned above, the boundaries between the children of a node v are stored at v and partition the interval associated with v into consecutive slabs, and a segment s intersecting the boundary of such a slab (but of no slab corresponding to a child of v's parent) is stored at v. The slabs intersected by s form a contiguous subinterval $[s_l, s_r]$ of $[s_1, s_{\sqrt{B}}]$. In the situation of Figure 6.11(a), for example, the segment s intersects the slabs s_1, s_2, s_3, and s_4, hence, $l = 1$ and $r = 4$. The indices l and r induce a partition of s into three (possibly empty) subsegments: a left subsegment $s \cap s_l$, a middle subsegment $s \cap [s_{l+1}, s_{r-1}]$, and a right subsegment $s \cap s_r$.

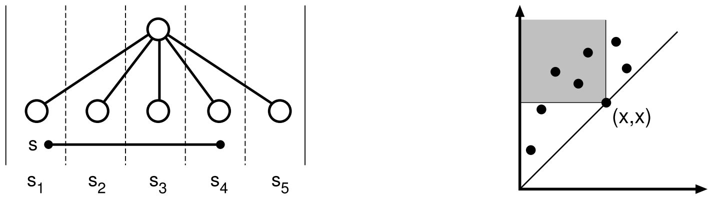
Each of the $\sqrt{B}$ slabs associated with a node v has a left and right structure that stores left and right subsegments falling into the slab. In the situation of the interval tree, these structures are lists ordered by the x-coordinates of the endpoints that do not lie on the slab boundary. Handling of middle subsegments is complicated by the fact that a subsegment might span more that one slab, and storing the segment at each such slab would increase both space requirement and update time. To resolve this problem, Arge and Vitter introduced the notion of *multislabs*: a multislab is a contiguous subinterval of $[s_1, s_{\sqrt{B}}]$, and it is easy to realize that there are $\Theta(\sqrt{B}\sqrt{B}) = \Theta(B)$ such multislabs. Each middle subsegment is stored in a secondary data structure corresponding to the (unique) maximal multislab it spans, and as there are only $\Theta(B)$ multislabs, the node v can accommodate pointers to all these structures in $\mathcal{O}(1)$ disk blocks.[6]

As in the internal memory setting, a stabbing query with $\ell = x$ is answered by performing a search for x and querying all secondary structures of the nodes visited along the path. As the tree is of height $\mathcal{O}(\log_B N)$, and as each left and right structure that contributes $Z' \geq 0$ elements to the answer set can be queried in $\mathcal{O}(1 + Z'/B)$ I/Os, the overall query complexity is $\mathcal{O}(\log_B N + Z/B)$ I/Os.[7]

[6] To ensure that the overall space requirement is $\mathcal{O}(N/B)$ disk blocks, multislab lists containing too few segments are grouped together into a special underflow structure [71].

[7] Note that each multislab structure queried contributes *all* its elements to the answer set, hence, the complexity of querying $\mathcal{O}(\sqrt{B}\log_B N)$ multislab structures is $\mathcal{O}(Z/B)$.

The main problem with making the interval tree dynamic is that the insertion of a new interval might augment the set of x-coordinates in $\mathcal{S}$. As a consequence, the base tree structure of the interval tree has to be reorganized, and this in turn might require several segments moved between secondary structures of different nodes. Using weight-balanced B-trees (see Chapter 2) and a variant of the global rebuilding technique [599], Arge and Vitter obtained a linear-space dynamic version of the interval tree that answers stabbing queries in $\mathcal{O}(\log_B N + Z/B)$ I/Os and can be updated in $\mathcal{O}(\log_B N)$ I/Os worst-case.



(a) A node in an external interval tree. (b) A diagonal corner query.

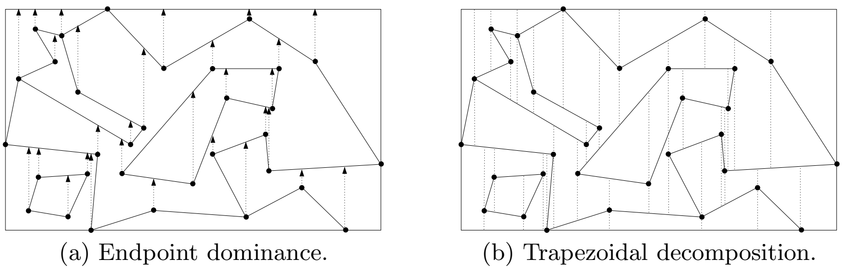
Fig. 6.11. Different approaches to the *Segment Stabbing* problem.

A completely different approach to solving the *Segment Stabbing* problem is to regard this problem as a special case of two-sided range searching in two dimensions, namely as a so-called *diagonal corner query*. By regarding the (one-dimensional) interval $[x_l, x_r]$ as the two-dimensional point (x_l, x_r) lying above the main diagonal, a stabbing query for the vertical line $\ell = x$ corresponds to a two-sided range query with apex at (x, x) (see Figure 6.11(b)). As diagonal corner queries can be answered by any data structure proposed for two-dimensional (two-sided, three-sided, or general orthogonal) range searching, all solutions discussed for the *Orthogonal Range Searching* problem (Problem 6.10) can be applied to the *Segment Stabbing* problem.

We now state a problem that occurs as a preprocessing step in a variety of other problems.

Problem 6.13 (Segment Sorting). Given a set $\mathcal{S}$ of N non-intersecting segments in the plane, compute the partial order given by the "above-below" relation and extend this order to a total order on $\mathcal{S}$.

Computing a total order on a set of non-intersecting segments in the plane has important applications, e.g., for the *Vertical Ray-Shooting* problem [69, 613] (see Problem 6.17) or the *Bichromatic Segment Intersection* problem [70]. The solution to the *Segment Sorting* problem makes use of what is called an *extended external segment tree*. This data structure has been proposed for solving the *Endpoint Dominance* problem which we discuss next.



(a) Endpoint dominance. (b) Trapezoidal decomposition.

Fig. 6.12. Problems involving multiple query points.

Problem 6.14 (Endpoint Dominance). Given a set $\mathcal{S}$ of N non-intersecting segments in the plane, find for each endpoint of a segment in $\mathcal{S}$ the segment in $\mathcal{S}$ (if any) directly above this endpoint (see Figure 6.12(a)).

Even though it seems that the *Endpoint Dominance* problem could be solved by repeatedly querying an external interval tree,[8] the main motivation behind developing a different approach is that the *Endpoint Dominance* problem is a batched static problem. For batched static problems, there is no need to employ a data structure whose I/O-complexity per single operation is worst-case optimal. Instead, a better overall I/O-complexity can be obtained by building on certain aspects of lazy data processing as in the *buffer tree* data structure (see Chapter 2).

As the interval tree data structure, the *segment tree* is a data structure for storing a set of one-dimensional intervals [108, 614]. The main idea again is to organize the x-coordinates in $\mathcal{S}$ as a binary search tree, but this time the x-coordinates are stored exclusively in the leaves of the tree. For $x_{[1]}, \ldots, x_{[2N]}$ denoting the sorted sequence of x-coordinates in $\mathcal{S}$ and $1 \leq i \leq 2N-1$, the i-th leaf (in left-to-right order) corresponds to the interval $[x_{[i]}, x_{[i+1]}[$ while the $2N$-th leaf corresponds to the point $x_{[2N]}$. An internal node then corresponds to the union of all intervals stored in the subtree below it. A segment is stored at each node v, where it (or rather its projection onto the x-axis) contains the interval corresponding to v, and this implies that each segment can be stored in up to two nodes per level. This in turn implies that an external segment tree occupies $\mathcal{O}((N/B)\log_{M/B}(N/B))$ blocks, and consequently each algorithm that relies on a set of segments being sorted requires the same amount of (temporary) disk space for at least the duration of the preprocessing step.

An external segment tree as proposed by Arge, Vengroff, and Vitter [70] can be seen as a hierarchical representation of the slabs visited during an algorithm based upon distribution sweeping. Corresponding to this intuition, the tree can be constructed efficiently top-down, distributing middle subsegments to secondary multislab structures. This requires that one block for each

[8] In fact, a solution can be obtained using an *augmented* version of this data structure (see Problem 6.17).

multislab can be held in main memory, and since the number of multislabs is quadratic in the number of slabs, the number of slabs, that is, the fan-out of the base tree (and thus of the corresponding distribution sweeping process), is chosen as $\Theta(\sqrt{M/B})$.[9]

To facilitate finding the segment immediately above another segment's endpoint, the segments in the multislab structures have to be sorted according to the "above-below" relation. Given that the solution to the *Endpoint Dominance* problem will be applied to solve the *Segment Sorting* problem (Problem 6.13), this seems a prohibited operation. Exploiting the fact, however, that the middle subsegments have their endpoints on a set of $\Theta(\sqrt{M/B})$ slab boundaries, Arge *et al.* [70] demonstrated how these segments can be sorted in a linear number of I/Os using only a standard (one-dimensional) sorting algorithm. Extending the external segment tree by keeping left and right subsegments in sorted order as they are distributed to slabs on the next level and using a simple counting argument, it can be shown that such an extended external segment tree can be constructed top-down spending $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os.[10]

The endpoint dominance queries are then filtered through the tree remembering for each query point the lowest dominating segment seen so far. Filtering is done bottom-up reflecting the fact that the segment tree has been built top-down. Arge *et al.* [70] built on the concept of fractional cascading [182] and proposed to use segments sampled from the multislab lists of a node v to each child (instead of the other way round) as bridges that help finding the dominating segment in v once the dominating segment in the nodes below v (if any) has been found. The number of sampled segments is chosen such that the overall space requirement of the tree does not (asymptotically) increase and that, simultaneously for all multislabs of a node v, all segments between two sampled segments can be held in main memory. Then, Q queries can be filtered through the extended external segment tree spending $\mathcal{O}(((N+Q)/B)\log_{M/B}(N/B))$ I/Os, and after the filtering process, all dominating segments are found.

A second approach is based upon the close relationship to the *Trapezoidal Decomposition* problem (Problem 6.15), namely that the solution for the *Endpoint Dominance* problem can be derived from the trapezoidal decomposition spending $\mathcal{O}(N/B)$ I/Os. As we will sketch, an algorithm derived in the framework of Crauser *et al.* [228] computes the *Trapezoidal Decomposition* of N non-intersecting segments spending an expected number of

[9] Using a base-tree with $\sqrt{M/B}$ fan-out does not asymptotically change the complexity as $\mathcal{O}((N/B)\log_{\sqrt{M/B}}(N/B)) = \mathcal{O}((N/B)\log_{M/B}(N/B))$. More precisely, the smaller fan-out results in a tree with twice as much levels.

[10] At present, it is unknown whether an extended external segment tree can be built efficiently in a multi-disk environment, that is, whether the complexity of building this structure is $\mathcal{O}((N/DB)\log_{M/B}(N/B))$ I/Os for $D \notin \mathcal{O}(1)$ [70].

$\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os, hence the *Endpoint Dominance* problem can be solved spending asymptotically the same number of I/Os.

Arge *et al.* [70] demonstrate how the *Segment Sorting* problem (Problem 6.13) can be solved by reduction to the *Endpoint Dominance* problem (Problem 6.14). Just as for computing the trapezoidal decomposition, two instances of the *Endpoint Dominance* problem are solved, this time augmented with horizontal segments at $y = +\infty$ and $y = -\infty$. Based upon the solution of these two instances, a directed graph $\mathcal{G}$ is created as follows: each segment corresponds to a node, and if a segment u is dominated from above (from below) by a segment v, the edge (u, v) (the edge (v, u)) is added to the graph. The two additional segments ensure that each of the original segments is dominated from above and from below, hence, the resulting graph is a planar (s, t)-graph. Computing the desired total order on $\mathcal{S}$ then corresponds to topologically sorting $\mathcal{G}$. As $\mathcal{G}$ is a planar (s, t)-graph of complexity $\Theta(N)$, this can be accomplished spending no more than $\mathcal{O}((N/B)\log_{M/B}(N/B))$ I/Os [192].

Problem 6.15 (Trapezoidal Decomposition). Given a set $\mathcal{S}$ of N non-intersecting segments in the plane, compute the planar partition induced by extending a vertical ray in $(+y)$- and $(-y)$-direction from each endpoint p of each segment until it hits the segment of $\mathcal{S}$ (if any) directly above resp. below p (see Figure 6.12(b)).

While the *Trapezoidal Decomposition* problem is closely related to the *Endpoint Dominance* problem (Problem 6.14) and to the *Polygon Triangulation* problem (Problem 6.16), it is also of independent interest. In internal memory, computing the trapezoid decomposition as a preprocessing step helps solving the *Planar Point Location* problem [457, 679] (Problem 6.18) and performing map-overlay [304] (see Problem 6.22).

In the external memory setting, two algorithms are known for solving the *Trapezoidal Decomposition* problem. The first approach, proposed by Arge *et al.* [70], exploits the simple fact that combining the results of two instances of the *Endpoint Dominance* problem (one with negated y-coordinates of all objects) yields the desired decomposition. All vertical extensions can be computed explicitly by linearly scanning the output of both *Endpoint Dominance* instances. The resulting extensions are then sorted by the name of the original segment they lie on (ties are broken by x-coordinates), and during one scan of the sorted output, all trapezoids can be reported in explicit form.

The second approach can be obtained within the framework of *randomized incremental construction* with *gradations* as proposed by Crauser *et al.* [228]. Even if the segments in $\mathcal{S}$ are not intersection-free but induce Z intersections, the *Trapezoidal Decomposition* problem can be solved spending an expected optimal number of $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/Os. The basic idea behind this framework is to externalize the paradigm of randomized incremental construction (considering elements from the problem instance one after the other, but in random order). Externalization is facilitated using gradations (see, e.g., [566]), a concept originating in the design of parallel algorithms. A gradation is a geometrically increasing random sequence of subsets $\emptyset = \mathcal{S}_0 \subseteq \cdots \subseteq \mathcal{S}_\ell = \mathcal{S}$. The randomized incremental construction with gradations refines the (intermediate) solution for a $\mathcal{S}_i$ by simultaneously adding all objects in $\mathcal{S}_{i+1} \setminus \mathcal{S}_i$ (that is, in parallel respectively blockwise). This framework is both general and powerful enough to yield algorithms with expected optimal complexity for a variety of geometric problems. As discussing the sophisticated details and the analysis of the resulting algorithms would be beyond the scope of this survey, we will only mention these results whenever appropriate and instead refer the interested reader to the original article [228].

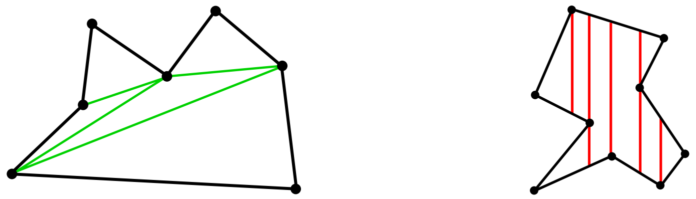

(a) Triangulation of a unimontone polygon. (b) Trapezoidal decomposition.

Fig. 6.13. Polygon triangulation and its relation to trapezoid decomposition.

Problem 6.16 (Polygon Triangulation). Given a simple polygon P in the plane with N edges, partition the interior of P into $N-2$ faces bounded by three segments each by adding $N-3$ non-intersecting line segments connecting two vertices of P (see Figure 6.13(a)).

Fornier and Montuno [310] proved that in the internal memory setting the *Polygon Triangulation* problem is (linear-time) equivalent to the *Trapezoidal Decomposition* problem (Problem 6.15) applied to the interior of the polygon (see Figure 6.13(b)). Subsequently, all internal memory algorithms built upon this fact, culminating in an optimal linear-time algorithm by Chazelle [180]. The main idea of computing a triangulation from a trapezoidal decomposition is to subdivide the original polygon into a collection of *unimonotone* polygons. A simple polygon with vertices $v_1, \ldots, v_N$ is called unimonotone if there are vertices v_i and v_{i+1} such that the projections of $v_{i+1}, \ldots, v_{i+N}$ onto the line supporting the edge (v_i, v_{i+1}) (all indices are to be read modulo N) form a sorted sequence. A unimonotone polygon can then be triangulated by repeatedly cutting off convex corners during a stack-driven traversal of the polygon's boundary (see Figure 6.13(a)).

While the traversal of a polygon's boundary can be done spending no more than a linear number of I/Os, explicitly constructing the unimonotone polygons is more involved. The key observation is that all necessary information for subdividing a polygon into unimonotone polygons can be inferred locally,

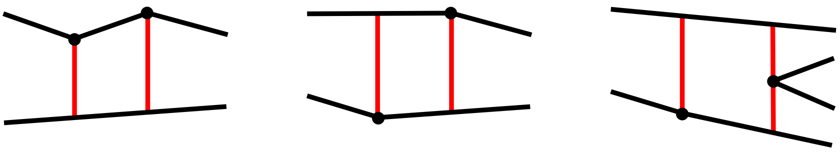
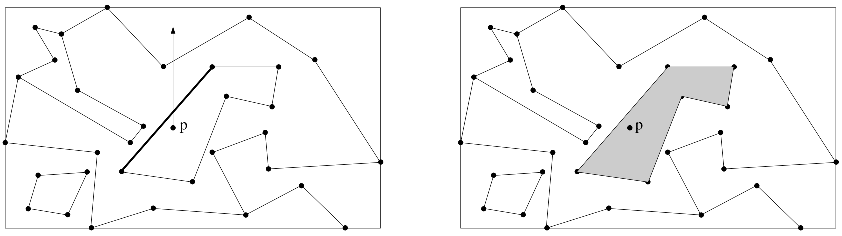


Fig. 6.14. Three classes of trapezoids.

i.e., by looking at isolated trapezoids. Each trapezoid is either a triangle or it is determined by vertical lines originating from two polygon vertices (see Figure 6.14). Fournier and Montuno [310] showed that by adding a diagonal between every such pair of vertices that do not already form a polygon edge, the polygon is partitioned into unimonotone polygons.

Arge *et al.* [70] built upon this observation and proposed the following algorithm for computing a triangulation of the given polygon. First, the trapezoidal decomposition is computed and all resulting trapezoids are scanned to see whether they induce diagonals as described above. For each vertex determining a qualifying trapezoid, a pointer to the matching vertex is stored. In the second phase, the sequence of vertices on the boundary is transformed into a linked list representing the vertices of the unimonotone subpolygons as they appear in clockwise order on the respective boundaries.

Applying a list ranking algorithm (see Chapter 3) to this linked list yields the sequence of vertices for each unimonotone subpolygon in sorted order. The I/O-complexity of list ranking is $\mathcal{O}((N/B)\log_{M/B}(N/B))$ [52, 192]. As mentioned above, each subpolygon can then be triangulated spending a linear number of I/Os. Summing up, we obtain an $\mathcal{O}((N/B)\log_{M/B}(N/B))$ algorithm for triangulating a simple polygon. As the internal memory complexity of this problem is $\Theta(N)$, a natural question is whether there exists an external algorithm with matching $\mathcal{O}(N/B)$ I/O-complexity. At present, however, it is unknown whether either the *Trapezoidal Decomposition* problem or the *Polygon Triangulation* problem can be solved spending $o((N/B)\log_{M/B}(N/B))$ I/Os.



(a) Vertical ray shooting from point p. (b) Point location query for point p.

Fig. 6.15. Problems involving a single query point.

Problem 6.17 (Vertical Ray-Shooting). Given a set $\mathcal{S}$ of N non-intersecting segments in the plane and a query point p in the plane, find the segment in $\mathcal{S}$ (if any) first hit by a ray emanating from p in $(+y)$-direction (see Figure 6.15(a)).

The first approach to external memory vertical ray-shooting has been proposed by Goodrich *et al.* [345] for the special case of $\mathcal{S}$ forming a monotone[11] subdivision. Combining an on-line filtering technique with an external version of a fractional-cascaded data structure [182, 183], they obtained a linear space external data structure that can be used to answer a vertical ray-shooting query in $\mathcal{O}(\log_B N)$ I/Os. Applying a batch filtering technique, a batch of Q vertical ray-shooting queries can be answered in $\mathcal{O}(((N+Q)/B)\log_{M/B}(N/B))$ I/Os.

Arge *et al.* [70] extended this result to a set $\mathcal{S}$ forming a general planar subdivision. Their solution is based upon the observation that each of the Q query points can be regarded as a (infinitesimally short) segment and that solving the *Endpoint Dominance* problem (see Problem 6.14) for the union of $\mathcal{S}$ and these Q segments yields the dominating segment for each query point. Their solution, using an extended external segment tree, requires $\mathcal{O}(((N+Q)/B)\log_{M/B}(N/B))$ I/Os and $\mathcal{O}((N/B)\log_{M/B}(N/B))$ space. Along similar lines, namely by reduction to the *Trapezoidal Decomposition* problem (Problem 6.15), an algorithm can be derived in the framework of Crauser *et al.* [228]. The resulting algorithm then answers a batch of Q vertical ray-shooting queries in expected $\mathcal{O}(((N+Q)/B)\log_{M/B}(N/B))$ I/Os using linear space.

The *Vertical Ray-Shooting* problem can be stated in a dynamic version, in which the set $\mathcal{S}$ additionally needs to be maintained under insertions and deletions of segments. For algorithms building upon the assumption that $\mathcal{S}$ forms a monotone subdivision Π, this implies that Π remains monotone after *each* update.

The most successful internal memory approaches [95, 188] to the dynamic version of the *Vertical Ray-Shooting* problem are based upon interval trees. In contrast to the *Segment Stabbing* problem (Problem 6.12), however, one cannot afford to report all segments above the query point p (there might be $\Theta(N)$ of them) just to find the one immediately above p. As a consequence, the secondary data structures associated with the nodes of the base interval tree have to reflect both the horizontal order of the endpoints within the slab (if applicable) and the vertical ordering of the (left, right, and middle) segments. These requirements increase the complexity of dynamically maintaining $\mathcal{S}$ under insertions and deletions.

For the left and right structures associated with each slab of a node in the interval tree, Agarwal *et al.* [4] built upon ideas due to Cheng and Jar-

[11] A polygon is called *monotone* in direction θ if any line in direction $\pi/2+\theta$ intersects the polygon in a connected interval. A planar subdivision Π is *monotone* if all faces of Π are monotone in the same direction.

nadan [188] and described a dynamic data structure for storing ν left and right subsegments with $\mathcal{O}(\log_B \nu)$ update time. Maintenance of the middle segments is complicated by the fact that not all segments are comparable according to the above-below relation (Problem 6.13), and that insertion of a new segment might globally affect the total order induced by this (local) partial order. Using level-balanced B-trees (see Chapter 2) and exploiting special properties of monotone subdivisions, Agarwal *et al.* [4] obtained a dynamic data structure for storing ν middle subsegments with $\mathcal{O}(\log_B^2 \nu)$ update time. The global data structure uses linear space and can be used to answer a vertical ray-shooting query in a monotone subdivision spending $\mathcal{O}(\log_B^2 N)$ I/Os. The amortized update complexity is $\mathcal{O}(\log_B^2 N)$.

This result was improved by Arge and Vahrenhold [69] who applied the logarithmic method (see Chapter 2) and an external variant of dynamic fractional cascading [182, 183] to obtain the same update and query complexity for general subdivisions.[12] The analysis is based upon the (realistic) assumption $B^2 < M$. Under the more restrictive assumption $2B < M$, the amortized insertion bound becomes $\mathcal{O}(\log_B N \cdot \log_{M/B}(N/B))$ I/Os while all other bounds remain the same.

A batched semidynamic version, that is, only deletions or only insertions are allowed, and all updates have to be known in advance, has been proposed by Arge *et al.* [65]. Using an external decomposition approach to the problem, $\mathcal{O}(Q)$ point location queries and $\mathcal{O}(N)$ updates can be performed in $\mathcal{O}(((N+Q)/B)\log^2_{M/B}((N+Q)/B))$ I/Os using $\mathcal{O}((N+Q)/B)$ space.

Problem 6.18 (Planar Point Location). Given a planar partition Π with N edges and a query point P in the plane, find the face of Π containing p (see Figure 6.15(b)).

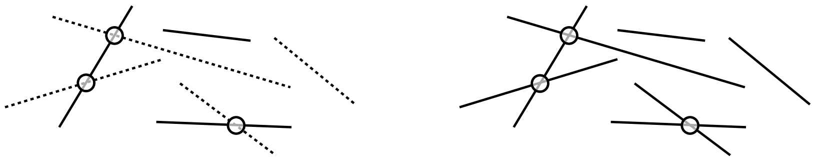
Usually, each edge in a planar partition stores the names of the two faces of Π it separates. Then, algorithms for solving the *Vertical Ray-Shooting* problem (Problem 6.17) can be used to answer point location queries with constant additional work.

Most algorithms for vertical ray-shooting exploit hierarchical decompositions which can be generalized to a so-called *trapezoidal search graph* [680]. Using balanced hierarchical decompositions, searching then can be done efficiently in both the internal and external memory setting. As the query points and thus the search paths to be followed are not known in advance, external memory searching in such a graph will most likely result in unpredictable access patterns and random I/O operations. The same is true for using general-purpose tree-based spatial index structures.

It is well known that disk technologies and operating systems support sequential I/O operations more efficiently than random I/O operations [519, 739]. Additionally, for practical applications, it is often desirable to trade asymptotically optimal performance for simpler structures if there is

[12] The deletion bound can be improved to $\mathcal{O}(\log_B N)$ I/Os amortized.

hope for comparable or even faster performance in practice. Vahrenhold and Hinrichs [740] extended the bucketing technique of Edahiro, Kokubo, and Asano [261] to the external memory setting incorporating both single-shot and batched queries into a single algorithm. The resulting algorithm relies on nothing more than sorting and scanning, and as the worst case I/O complexity of $\mathcal{O}(N/B)$ I/Os for a single-shot query is obtained for only pathological situations, the algorithm is both easy to implement and fast in practice [740].



(a) Bichromatic segment intersection. (b) General segment intersection.

Fig. 6.16. Segment intersection problems.

Problem 6.19 (Bichromatic Segment Intersection). Given a set $\mathcal{S}_1$ of non-intersecting "blue" segments in the plane and a set $\mathcal{S}_2$ of non-intersecting "red" segments in the plane with $|\mathcal{S}_1 \cup \mathcal{S}_2| \in \Theta(N)$, compute all Z "red-blue" pairs of intersecting segments in $\mathcal{S}_1 \times \mathcal{S}_2$ (see Figure 6.16(a)).

To facilitate exposition of the algorithm for the *Bichromatic Segment Intersection* problem, we first describe an algorithm for solving the special case of *Orthogonal Segment Intersection*, where we want to report all intersections between a set $\mathcal{S}_1$ of horizontal segments and a set $\mathcal{S}_2$ of vertical segments. To solve this problem, Goodrich *et al.* [345] described an optimal algorithm with $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/O-complexity that is based upon the distribution sweeping paradigm. For each slab, a so-called active list A_i is maintained. If, during the top-down sweep, the upper (lower) endpoint of a vertical segment is encountered, the segment is added to (removed from) the active list of the slab it falls into. If a left endpoint of a segment s is encountered, the intersection with each segment in the active lists of the slabs spanned by s are reported. Intersections within slabs intersected but not spanned by s are reported while sweeping at lower levels of recursion. Using lazy deletions from the active lists and an amortization argument, the method can be shown to require a linear number of I/Os per level of recursion. As recursion stops as soon as the subproblem can be solved in main memory, the overall I/O-complexity is $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ [345].

This approach has been refined by Arge *et al.* [70] to obtain an optimal $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ algorithm for the *Bichromatic Segment Intersection* problem. In a preprocessing step, the red segments and the endpoints of the blue segments (regarded as infinitesimal short segments) are merged into one set and sorted according to the "above-below" relation. The same process is repeated for the set constructed from the blue segments and the endpoints of the red segments. We now describe the work done on each level of recursion during the distribution sweeping.

In the terminology of the description of the external interval tree (see Problem 6.12), the algorithm first detects intersections between red middle subsegments and blue left and right subsegments. The key to an efficient solution is to explicitly construct the endpoints of the blue left and right subsegments that lie on the slab boundaries and to merge them into the sorted list of red middle subsegments and the (proper) endpoints of the blue left and right subsegments. During a top-down sweep over the plane (in segment order), blue left and right subsegments are then inserted into active lists of their respective slab as soon as their topmost endpoint is encountered, and for each red middle subsegment s encountered, the active lists of the slabs spanned by s are scanned to produce red-blue pairs of intersecting segments. As soon as a red middle subsegment does not intersect a blue left or right subsegment, this blue segment cannot be intersected by any other red segment, hence, it can be removed from the slab's active list. An amortization argument shows that all intersections can be reported in a linear number of I/Os. An analogous scan is performed to report intersections between blue middle subsegments and red left and right subsegments.

In a second phase, intersections between middle subsegments of different colors are reported. For each multislab, a multislab list is created, and each red middle subsegment is then distributed to the list of the maximal multislab that it spans. An immediate consequence of the red segments being sorted is that each multislab list is sorted by construction. Using a synchronized traversal of the sorted list of blue middle subsegments and multislab lists and repeating the process for the situation of the blue middle subsegments being distributed, all red-blue pairs of intersecting middle subsegments can be reported spending a linear number of I/Os. Intersections between non-middle subsegments of different colors are found by recursion within the slabs. As in the orthogonal setting, a linear number of I/Os is spent on each level of recursion, hence, the overall I/O-complexity is $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$.

Since computing the trapezoidal decomposition of a set of segments yields the Z intersections points without additional work, an algorithm with expected optimal $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/O-complexity can be derived in the framework of Crauser *et al.* [228].

Problem 6.20 (Segment Intersection). Given a set $\mathcal{S}$ of N segments in the plane, compute all Z pairs of intersecting segments in $\mathcal{S} \times \mathcal{S}$ (see Figure 6.16(b)).

Even though the general *Segment Intersection* problem appears considerably more complicated than its bichromatic variant, its intrinsic complexity is the same [88, 181]. An external algorithm with (suboptimal) I/O-complexity of $\mathcal{O}(((N+Z)/B)\log_{M/B}(N/B))$ has been proposed by Arge *et al.* [70]. The main idea is to integrate all phases of the deterministic solution described for the *Bichromatic Segment Intersection* problem (see Problem 6.19) into one single phase. The distribution sweeping paradigm is not directly applicable because there is no total order on a set of intersecting segments. Arge *et al.* [70] proposed to construct an extended external segment tree on the segments and (during the construction of this data structure) to break the segments stored in the same multislab lists into non-intersecting fragments. The resulting segment tree can then be used to detect intersections between segments stored in different multislab lists. For details and the analysis of this second phase, we refer the reader to the full version of the paper [70].

Since computing the trapezoidal decomposition of a set of segments yields the Z intersections points without additional work, an algorithm with expected optimal $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/O-complexity can be derived in the framework of Crauser *et al.* [228]. It remains a open problem, though, to find a deterministic optimal solution for the *Segment Intersection* problem.

Jagadish [426] developed a completely different approach to finding all line segments that intersect a given line segment. Applying this algorithm to all segments and removing duplicates, it can also be used to solve Problem 6.20. This algorithm, which has experimentally shown to perform well for real-world data sets [426], partitions the d-dimensional data space into d partitions (one for each axis) and stores a small amount of data for each line segment in the partition with whose axis this line segment defines the smallest angle. The data stored is determined by using a modified version of Hough transform [408]. For simplicity, the planar case is considered here first, before we show how to generalize it to higher dimensions. In the plane, each line segment determines a line given by either $y = m \cdot x + b$ or $x = m \cdot y + b$, and at least one of these lines has a slope in $[-1, 1]$. This equation is taken to map m and b to a point in (2-dimensional) transform space by a duality transform. An intersection test for a given line segment works as follows. The two endpoints are transformed into lines first. Assuming for simplicity, that these lines intersect (the approach also works for parallel lines), we know that these two lines divide the transform space into four regions. Transforming a third point of the line segment, the two regions *between* the transformed lines can be determined easily. The points contained in these regions (or, rather, the segment supported by their dual lines) are candidates for intersecting line segments. Whether they really intersect can be tested by comparing the projections on the partition axis of both segments which have been stored along with each point in transform space. For d-dimensional data space, only little changes occur. After determining the partition axis, the projections of each line segment on the $d-1$ planes involving this axis are treated as above resulting in $d-1$ lines and a point in $(2(d-1))$-dimensional transform space. In addition, the interval of the projection on the partition axis is stored. Note that this technique needs $2dN$ space to store the line segments. Unfortunately, no asymptotic bounds for query time are given, but experiments show that this approach is more efficient than using spatial index structures or transforming the two d-dimensional endpoints into one point in $2d$-dimensional data space, which are both very common approaches. Some other problems including finding all line segments passing through or lying in the vicinity of a specified point can be solved by this technique [426].

The *Segment Intersection* problem has a natural extension: Given a set of polygonal objects, report all intersecting pairs of objects. While at first it seems that this extension is quite straightforward, we will demonstrate in the next section that only special cases can be solved efficiently.

6.5 Problems Involving Set of Polygonal Objects

In spatial databases that store sets of polygonal objects, combining two planar partitions (*maps*) m_1 and m_2 by *map overlay* or *spatial overlay join* is an important operation. The *spatial overlay join* of m_1 and m_2 produces a set of pairs of polygonal objects (o_1, o_2) where $o_1 \in m_1, o_2 \in m_2$, and o_1 and o_2 intersect. In contrast, the *map overlay* produces a set of polygonal objects consisting of the following objects:

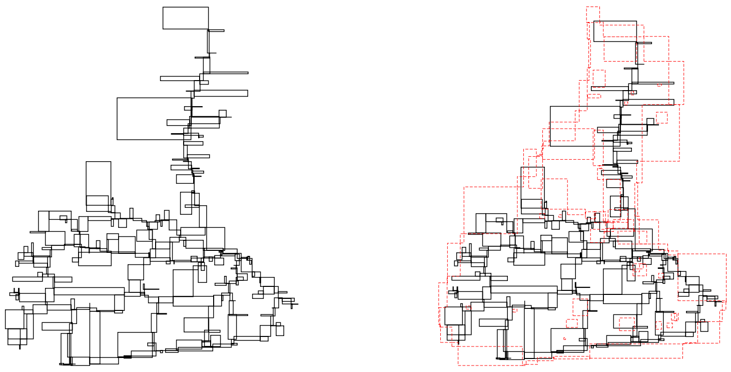
– All objects of m_1 intersecting no object of m_2
– All objects of m_2 intersecting no object of m_1
– All polygonal objects produced by two intersecting objects of m_1 and m_2

Usually spatial join operations are performed in two steps [594]:

– In the *filter step*, a conservative approximation of each spatial object is used to eliminate objects that cannot be part of the result.
– In the *refinement step*, each pair of objects passing the filter step is examined according to the spatial join condition.

In the context of spatial join, the most common approximation for a spatial object is by means of its minimum bounding box (see Section 6.2.3 and Figure 6.17(a)). Performing the filter step then can be restated as finding all pairs of intersecting rectangles between two sets of rectangles (see Figure 6.17(b)). The minimum bounding box is chosen from several possible approximations as it realizes a good trade-off between approximation quality and storage requirements. In addition, most spatial index structures are based on minimum bounding boxes. Nevertheless, some approaches use additional approximations to further reduce the number of pairs passing the filter step [150].

Problem 6.21 (Rectangle Intersection). Given a set $\mathcal{S}$ of N axis-aligned rectangles in the plane, compute all Z pairs of intersecting rectangles in $\mathcal{S} \times \mathcal{S}$. In our definition, this includes pairs of rectangles where one rectangle is fully contained in the other rectangle.


(a) Bounding boxes of road features (Block Island, RI).

(b) Bounding boxes of roads and hydrography features (Block Island).

Fig. 6.17. Using rectangular bounding boxes for a spatial join operation.

An efficient approach based upon the distribution sweeping paradigm has been proposed by Goodrich *et al.* [345] and later restated in the context of bichromatic decomposable problems by Arge *et al.* [64, 65]. The main observation is that, for any pair of intersecting rectangles, there exists a horizontal line that passes through both rectangles, and thus their projections onto this line consist of overlapping intervals. In Figure 6.18, three pairs of intersecting rectangles and their projections onto the sweep-line are shown.

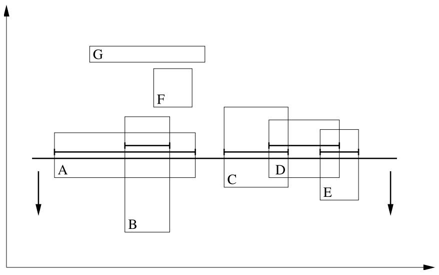


Fig. 6.18. Intersecting rectangles correspond to overlapping intervals.

The proposed algorithm for solving the *Rectangle Intersection* problem searches for intersections between active rectangles by exploiting this rectangle–interval correspondence: As the sweep-line advances over the data, the algorithm maintains the projections of all active rectangles onto the sweep-line and checks which of these intervals intersect, thus reducing the static two-dimensional rectangle intersection problem to the dynamic one-dimensional interval intersection problem. During the top-down sweep, $\Theta(M/B)$ multislab lists are maintained, and the (projected) intervals are first used to query the multislab lists for overlap with other intervals before the middle subsegments are inserted into the multislab lists themselves (left and right subsegments are treated recursively).[13] A middle subsegment is removed from the multislab lists when the sweep-line passes the lower boundary of the original rectangle. Making sure that these deletions are performed in a blocked manner, one can show that the overall I/O-complexity is $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$. In the case of rectangles, the reduction to finding intersection of edges yields an efficient algorithm as the number of intersecting pairs of objects is asymptotically the same as the number of intersecting pairs of edges—in the more general case of polygons, this is not the case (see Problem 6.22).

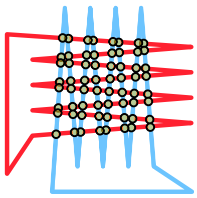
In the database community, this problem is considered almost exclusively in the *bichromatic* case of the filter step of spatial join operations, and several heuristics implementing the filter step and performing well for real-world data sets have been proposed during the last decade. Most of the proposed algorithms [101, 150, 151, 362, 401, 414, 603, 704] need index structures for both data sets, while others only require one data set to be indexed [63, 365, 511, 527], but also spatial hash joins [512] and other non index-based algorithms [64, 477, 606] have been presented. Moreover, other conservative approximation techniques besides minimum bounding boxes—mainly in the planar case—like convex hull, minimum bounding m-corner (especially $m \in \{4, 5\}$), smallest enclosing circle, or smallest enclosing ellipse [149, 150] as well as four-colors raster signature [782] have been considered. Using additional progressive approximations like maximum enclosed circle or rectangle leads to fast identification of object pairs which can be reported without testing the exact geometry [149, 150]. Rotem [639] proposed to transform the idea of join indices [741] to n-dimensional data space using the grid file [583].

The central idea behind all approaches summarized above is to repeatedly reduce the working set by pruning or partitioning until it fits into main memory where an internal memory algorithm can be used. Most index-based algorithms exploit the hierarchical representation implicitly given by the index structures to prune parts of the data sets that cannot contribute to the output of the join operator. In contrast, algorithms for non-indexed spatial join try to reduce the working set by either imposing an (artificial) order and then performing some kind of merging according to this order or by hashing the data to smaller partitions that can be treated separately. The overall performance of algorithms for the filter step, however, often depends on subtle design choices and characteristics of the data set [63], and therefore discussing these approaches in sufficient detail would be beyond the scope of this survey.

The *refinement step* of the spatial join cannot rely on approximations of the polygonal objects but has to perform computations on the exact repre-

[13] In the bichromatic setting, two sets of multislab lists are used, one for each color.

sentations of the objects that passed the filter step. In this step, the problem is to determine all pairs of polygonal objects that fulfill the join predicate.


(a) Two simple polygons may have $\Theta(N^2)$ intersecting pairs of edges.

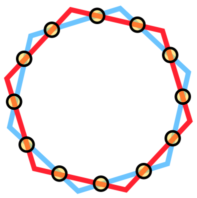

(b) Two convex polygons may have $\Theta(N)$ intersecting pairs of edges.

Fig. 6.19. Output-sensitivity for intersecting polygons.

Problem 6.22 (Polygon Intersection). Given a set $\mathcal{S}$ of polygonal objects in the plane consisting of N edges in total, compute all Z pairs of intersecting polygons in $\mathcal{S} \times \mathcal{S}$. By definition this includes pairs of polygons where one polygon is fully contained in the other polygon.

The main problem in developing efficient algorithms for the *Polygon Intersection* problem is the notion of *output sensitivity*. The problem, as stated above, requires the output to depend on the number of intersecting pairs of polygons and not on the number of intersecting pairs of polygon edges. The problem could be easily solved by employing algorithms for the *Segment Intersection* problem (Problem 6.20), however, the number of intersecting pairs of edges can be asymptotically much larger than the number of intersecting pairs of polygons. For simple polygons, each pair of intersecting polygons can even give rise to a quadratic number of intersecting pairs of edges (see Figure 6.19(a)). Even for convex polygons, any one pair of intersecting polygons can give rise to a linear number of intersecting pairs of edges (see Figure 6.19(b)), but exploiting the convexity of the polygons, efficient output-sensitive algorithms have been developed in the internal memory setting [9, 364].

If the *Polygon Intersection* problem is considered in the context of the bichromatic map overlay join, the output is no longer the set of intersecting pairs of polygons, but it additionally includes the planar partition induced by the overlay. This in turn removes the limitation of not to compute Z pairs of intersecting segments. In the internal memory setting, an optimal algorithm for computing the map overlay join of two simply connected planar partitions in $\mathcal{O}(N + Z)$ time and space has been proposed by Finke and Hinrichs [304]. This algorithm heavily relies on a trapezoidal decomposition of the partitions and on the ability to efficiently traverse a connected planar subdivision, so unless both problems can be solved optimally in the external memory setting, there is little hope for an optimal external memory variant.

Some effort has also been made to combine spatial index structures and internal memory algorithms ([174, 584]) for finding line segment intersections [100, 480], but these results rely on practical considerations about the input data. Another approach which is also claimed to be efficient for real world data sets [149], generates variants of R-trees, namely TR*-trees [671] for both data sets and uses them to compute the result afterwards.

6.6 Conclusions

In this survey, we have discussed algorithms and data structures that can be used for solving large-scale geometric problems. While a lot of research has been done both in the context of spatial databases and in algorithmics, one of the most challenging problems is to combine the best of these two worlds, that is algorithmic design techniques and insights gained from experiments for real-world instances. The field of external memory experimental algorithmics is still wide open.

Several important issues in large-scale Geographic Information Systems have not been addressed in the context of external memory algorithms, including how to externalize algorithms on triangulated irregular networks or how to (I/O-efficiently) perform map-overlay on large digital maps. We conclude this chapter by stating two prominent open problems for which optimal algorithms are known only in the internal memory setting:

– Is it possible to triangulate a simple polygon given its vertices in counterclockwise order along its boundary spending only a linear number of I/Os?
– Is it possible to compute all Z pairs of intersecting line segments in a set of N line segments in the plane using a *deterministic* algorithm that spends only $\mathcal{O}((N/B)\log_{M/B}(N/B) + Z/B)$ I/Os?

7. Full-Text Indexes in External Memory

Juha Kärkkäinen* and S. Srinivasa Rao

7.1 Introduction

A *full-text index* is a data structure storing a text (a string or a set of strings) and supporting *string matching queries*: Given a pattern string P, find all occurrences of P in the text. The best-known full-text index is the suffix tree [761], but numerous others have been developed. Due to their fast construction and the wealth of combinatorial information they reveal, full-text indexes (and suffix trees in particular) also have many uses beyond basic string matching. For example, the number of distinct substrings of a string or the longest common substrings of two strings can be computed in linear time [231]. Gusfield [366] describes several applications in computational biology, and many others are listed in [359].

Most of the work on full-text indexes has been done on the RAM model, i.e., assuming that the text and the index fit into the internal memory. However, the size of digital libraries, biosequence databases and other textual information collections often exceed the size of the main memory on most computers. For example, the GenBank [107] database contains more than 20 GB of DNA sequences in its August 2002 release. Furthermore, the size of a full-text index is usually 4–20 times larger than the size of the text itself [487]. Finally, if an index is needed only occasionally over a long period of time, one has to keep it either in internal memory reducing the memory available to other tasks or on disk requiring a costly loading into memory every time it is needed.

In their standard form, full-text indexes have poor memory locality. This has led to several recent results on adapting full-text indexes to external memory. In this chapter, we review the recent work focusing on two issues, full-text indexes supporting I/O-efficient string matching queries (and updates), and external memory algorithms for constructing full-text indexes (and for sorting strings, a closely related task).

We do not treat other string techniques in detail here. Most string matching algorithms that do not use an index work by scanning the text more or less sequentially (see, e.g., [231, 366]), and are relatively trivial to adapt to an externally stored text. Worth mentioning, however, are algorithms that may generate very large automata in pattern preprocessing, such as [486, 533, 573, 735, 770], but we are not aware of external memory versions of these algorithms.

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

In information retrieval [85, 314], a common alternative to full-text indexes is the *inverted file* [460], which takes advantage of the natural division of linguistic texts into a limited number of distinct words. An inverted file stores each distinct word together with a list of pointers to the occurrences of the word in the text. The main advantage of inverted files is their space requirement (about half of the size of the text [85]), but they cannot be used with unstructured texts such as biosequences. Also, the space requirement of the data structures described here can be significantly reduced when the text is seen as a sequence of atomic words (see Section 7.3.2).

Finally, we mention another related string technique, compression. Two recent developments are compressed indexes [299, 300, 361, 448, 648, 649] and sequential string matching in compressed text without decompression [40, 289, 447, 575, 576]. Besides trying to fit the text or index into main memory, these techniques can be useful for reducing the time for moving data from disk to memory.

7.2 Preliminaries

We begin with a formal description of the problems and the model of computation.

The Problems Let us define some terminology and notation. An *alphabet* Σ is a finite ordered set of *characters*. A *string* S is an array of characters, $S[1,n] = S[1]S[2]\dots S[n]$. For $1 \leq i \leq j \leq n$, $S[i,j] = S[i]\dots S[j]$ is a *substring* of S, $S[1,j]$ is a *prefix* of S, and $S[i,n]$ is a *suffix* of S. The set of all strings over alphabet Σ is denoted by Σ^*.

The main problem considered here is the following.

Problem 7.1 (Indexed String Matching). Let the *text* $\mathcal{T}$ be a set of K strings in Σ^* with a total length N. A *string matching query* on the text is: Given a *pattern* $P \in \Sigma^*$, find all occurrences of P as a substring of the strings in $\mathcal{T}$. The static problem is to store the text in a data structure, called a *full-text index*, that supports string matching queries. The dynamic version of the problem additionally requires support for insertion and deletion of strings into/from $\mathcal{T}$.

All the full-text indexes described here have a linear space complexity. Therefore, the focus will be on the time complexity of queries and updates (Section 7.4), and of construction (Section 7.5).

Additionally, the *string sorting problem* will be considered in Section 7.5.5.

Problem 7.2 (String Sorting). Given a set $\mathcal{S}$ of K strings in Σ^* with a total length N, sort them into the lexicographic order.

The Model Our computational model is the standard external memory model introduced in [17, 755] and described in Chapter 1 of this volume. In particular, we use the following main parameters:

N = number of characters in the text or in the strings to be sorted
M = number of characters that fit into the internal memory
B = number of characters that fit into a disk block

and the following shorthand notations:

$$\begin{aligned} \mathrm{scan}(N) &= \Theta\left(N/B\right) \\ \mathrm{sort}(N) &= \Theta\left((N/B)\log_{M/B}(N/B)\right) \\ \mathrm{search}(N) &= \Theta\left(\log_B N\right) \end{aligned}$$

The following parameters are additionally used:

K = number of strings in the text or in the set to be sorted
Z = size of the answer to a query (the number of occurrences)
$|\Sigma|$ = size of the alphabet
$|P|$ = number of characters in a pattern P
$|S|$ = number of characters in an inserted/deleted string S

For simplicity, we mostly ignore the space complexity, the CPU complexity, and the parallel (multiple disks) I/O complexity of the algorithms. However, significant deviations from optimality are noted.

With respect to *string representation*, we mostly assume the *integer alphabet model*, where characters are integers in the range $\{1, \ldots, N\}$. Each character occupies a single machine word, and all usual integer operations on characters can be performed in constant time. For internal memory computation, we sometimes assume the *constant alphabet model*, which differs from the integer alphabet model in that dictionary operations on sets of characters can be performed in constant time and linear space.[1] Additionally, the *packed string model* is discussed in 7.5.6.

7.3 Basic Techniques

In this section, we introduce some basic techniques. We start with the (for our purposes) most important internal memory data structures and algorithms. Then, we describe two external memory techniques that are used more than once later.

[1] With techniques such as hashing, this is *nearly* true even for the integer alphabet model. However, integer dictionaries are a complex issue and outside the scope of this article.

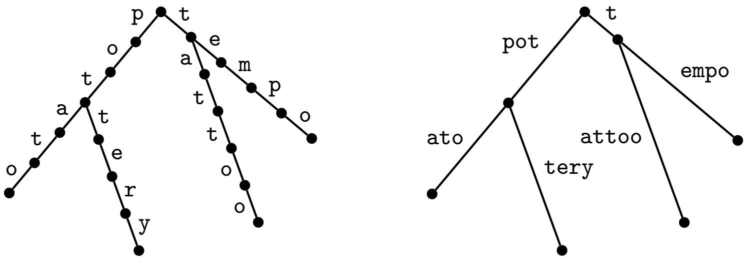


Fig. 7.1. Trie and compact trie for the set {`potato`, `pottery`, `tattoo`, `tempo`}

7.3.1 Internal Memory Techniques

Most full-text indexes are variations of three data structures, suffix arrays [340, 528], suffix trees [761] and DAWGs (Directed Acyclic Word Graphs) [134, 230]. In this section, we describe suffix arrays and suffix trees, which form the basis for the external memory data structures described here. We are not aware of any adaptation of DAWG for external memory.

Let us start with an observation that underlies almost all full-text indexes. If an occurrence of a pattern P starts at position i in a string $S \in \mathcal{T}$, then P is a prefix of the suffix $S[i, |S|]$. Therefore, we can find all occurrences of P by performing a prefix search query on the set of all suffixes of the text: A *prefix search query* asks for all the strings in the set that contain the query string P as a prefix. Consequently, a data structure that stores the set of all suffixes of the text and supports prefix searching is a full-text index.

The simplest data structure supporting efficient prefix searching is the lexicographically sorted array, where the strings with a given prefix always form a contiguous interval. The *suffix array* of a text $\mathcal{T}$, denoted by $SA_{\mathcal{T}}$, is the sorted array of pointers to the suffixes of $\mathcal{T}$ (see Fig. 7.2). By a binary search, a string matching (prefix search) query can be answered with $\mathcal{O}(\log_2 N)$ string comparisons, which needs $\mathcal{O}(|P| \log_2 N)$ time in the worst case. Manber and Myers [528] describe how the binary search can be done in $\mathcal{O}(|P| + \log_2 N)$ time if additional (linear amount of) information is stored about longest common prefixes. Manber and Myers also show how the suffix array can be constructed in time $\mathcal{O}(N \log_2 N)$. Suffix arrays do not support efficient updates.

The trie is another simple data structure for storing a set of strings [460]. A *trie* (see Fig. 7.1) is a rooted tree with edges labeled by characters. A node in a trie represents the concatenation of the edge labels on the path from the root to the node. A trie for a set of strings is the minimal trie whose nodes represent all the strings in the set. If the set is *prefix free*, i.e., no string is a proper prefix of another string, all the nodes representing the strings are leaves. A *compact trie* is derived from a trie by replacing each maximal branchless path with a single edge labeled by the concatenation of the replaced edge labels (see Fig. 7.1).

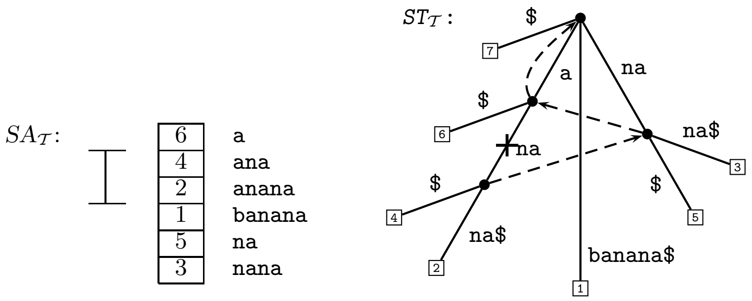


Fig. 7.2. Suffix array $SA_{\mathcal{T}}$ and suffix tree $ST_{\mathcal{T}}$ for the text $\mathcal{T} = \{$banana$\}$. For the suffix tree, a sentinel character $ has been added to the end. Suffix links are shown with dashed arrows. Also shown are the answers to a string matching query $P =$ an: in $SA_{\mathcal{T}}$ the marked interval, in $ST_{\mathcal{T}}$ the subtree rooted at $+$. Note that the strings shown in the figure are not stored explicitly in the data structures but are represented by pointers to the text.

The *suffix tree* of a text $\mathcal{T}$, denoted by $ST_{\mathcal{T}}$, is the compact trie of the set of suffixes of $\mathcal{T}$ (see Fig. 7.2). With suffix trees, it is customary to add a sentinel character $ to the end of each string in $\mathcal{T}$ to make the set of suffixes prefix free. String matching (prefix searching) in a suffix tree is done by walking down the tree along the path labeled by the pattern (see Fig. 7.2). The leaves in the subtree rooted at where the walk ends represent the set of suffixes whose prefix is the pattern. The time complexity is $\mathcal{O}(|P|)$ for walking down the path (under the constant alphabet model) and $\mathcal{O}(Z)$ for searching the subtree, where Z is the size of the answer.

The suffix tree has $\mathcal{O}(N)$ nodes, requires $\mathcal{O}(N)$ space, and can be constructed in $\mathcal{O}(N)$ time. Most linear-time construction algorithms, e.g. [540, 736, 761], assume the constant alphabet model, but Farach's algorithm [288] also works in the integer alphabet model. All the fast construction algorithms rely on a feature of suffix trees called suffix links. A *suffix link* is a pointer from a node representing the string $a\alpha$, where a is a single character, to a node representing α (see Fig. 7.2). Suffix links are not used in searching but they are necessary for an insertion or a deletion of a string S in time $\mathcal{O}(|S|)$ [297] (under the constant alphabet model).

7.3.2 External Memory Techniques

In this section, we describe two useful string algorithm techniques, Patricia tries and lexicographic naming. While useful for internal memory algorithms too, they are particularly important for external memory algorithms.

Suffix arrays and trees do not store the actual strings they represent. Instead, they store pointers to the text and access the text whenever necessary. This means that the text is accessed frequently, and often in a nearly random manner. Consequently, the performance of algorithms is poor when the text is stored on disk. Both of the techniques presented here address this issue.

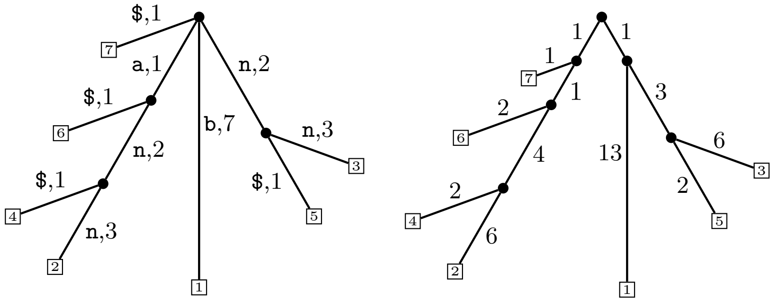


Fig. 7.3. Pat tree $PT_{\mathcal{T}}$ for $\mathcal{T} = \{\texttt{banana\$}\}$ using native encoding and binary encoding. The binary encoding of characters is `$=00`, `a=01`, `b=10`, `n=11`.

The first technique is the *Patricia trie* [557], which is a close relative of the compact trie. The difference is that, in a Patricia trie, the edge labels contain only the first character (branching character) and the length (skip value) of the corresponding compact trie label. The Patricia trie for the set of suffixes of a text $\mathcal{T}$, denoted by $PT_{\mathcal{T}}$, is called the *Pat tree* [340]. An example is given in Fig. 7.3.

The central idea of Patricia tries and Pat trees is to delay access to the text as long as possible. This is illustrated by the string matching procedure. String matching in a Pat tree proceeds as in a suffix tree except only the first character of each edge is compared to the corresponding character in the pattern P. The length/skip value tells how many characters are skipped. If the search succeeds (reaches the end of the pattern), all the strings in the resulting subtree have the same prefix of length $|P|$. Therefore, either all of them or none of them have the prefix P. A single string comparison between the pattern and some string in the subtree is required to find out which is the case. Thus, the string matching time is $\mathcal{O}(|P| + Z)$ as with the suffix tree, but there is now only a single contiguous access to the text.

Any string can be seen as a binary string through a *binary encoding* of the characters. A prefix search on a set of such binary strings is equivalent to a prefix search on the original strings. Patricia tries and Pat trees are commonly defined to use the binary encoding instead of the native encoding, because it simplifies the structure in two ways. First, every internal node has degree two. Second, there is no need to store even the first bit of the edge label because the left/right distinction already encodes for that. An example is shown in Fig. 7.3.

The second technique is lexicographic naming introduced by Karp, Miller and Rosenberg [450]. A *lexicographic naming* of a (multi)set $\mathcal{S}$ of strings is an assignment of an integer (the name) to each string such that any order comparison of two names gives the same result as the lexicographic order comparison of the corresponding strings. Using lexicographic names, arbitrarily long strings can be compared in constant time without a reference to

4	`ban`
2	`ana`
6	`nan`
2	`ana`
5	`na$`
1	`a$$`

4	`banana`
3	`anana`
6	`nana`
2	`ana`
5	`na`
1	`a`

Fig. 7.4. Lexicographic naming of the substrings of length three in `banana$$`, and of the suffixes of `banana`

the actual strings. The latter property makes lexicographic naming a suitable technique for external memory algorithms.

A simple way to construct a lexicographic naming for a set $\mathcal{S}$ is to sort $\mathcal{S}$ and use the rank of a string as its name, where the rank is the number of lexicographically smaller strings in the set (plus one). Fig. 7.4 displays two examples that are related to the use of lexicographic naming in Section 7.5.

Lexicographic naming has an application with linguistic texts, where words can be considered as 'atomic' elements. As mentioned in the introduction, inverted files are often preferred to full-text indexes in this case because of their smaller space requirement. However, the space requirement of full-text indexes (at least suffix arrays) can be reduced to the same level by storing only suffixes starting at the beginning of a word [340] (making them no more full-text indexes). A problem with this approach is that most fast construction algorithms rely on the inclusion of all suffixes. A solution is to apply lexicographic naming to the set of distinct words and transform the text into strings of names. Full-text indexes on such transformed texts are called *word-based indexes* [46, 227].

7.4 I/O-Efficient Queries

In this section, we look at some I/O-efficient index structures. In particular, we look at the structures described by Baeza-Yates et al. [83], Clark and Munro [206] and Ferragina and Grossi [296]. We then briefly sketch some recent results.

7.4.1 Hierarchies of Indexes

Baeza-Yates et al. [83] present an efficient implementation of an index for text databases when the database is stored in external memory. The implementation is built on top of a suffix array. The best known internal memory algorithm, of Manber and Myers [528], for string matching using a suffix array is not I/O-efficient when the text and the index reside in external memory. Baeza-Yates et al. propose additional index structures and searching algorithms for suffix arrays that reduce the number of disk accesses. In

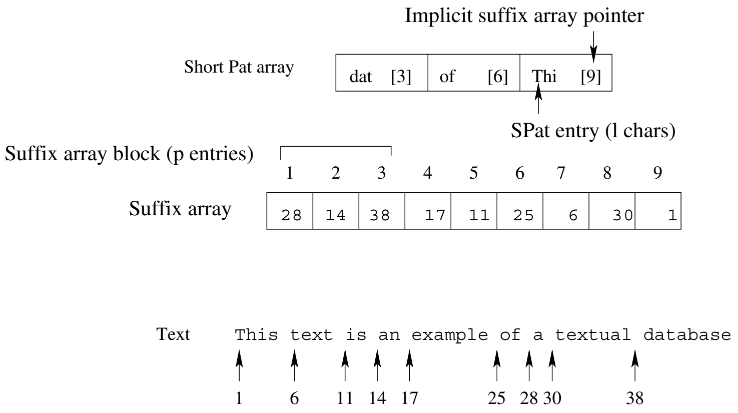


Fig. 7.5. Short Pat array

particular, they introduce two index structures for two and three level memory hierarchy (that use main memory and one/two levels of external storage), and present experimental and analytical results for these. These additional index structures are much smaller in terms of space compared to the text and the suffix array. Though, theoretically these structures only improve the performance by a constant factor, one can adjust the parameters of the structure to get good practical performance. Here, we briefly describe the structure for a two-level hierarchy. One can use a similar approach for building efficient indexes for a steeper hierarchy.

Two-Level Hierarchy. The main idea is to divide the suffix array into blocks of size p, where p is a parameter, and move one element of each block into main memory, together with the first few characters of the corresponding suffix. This structure can be considered a reduced representation of the suffix array and the text file, and is called Short Pat array or SPat array[2]. The SPat array is a set of suffix array entries where each entry also carries a fixed number, say ℓ, of characters from the text, where ℓ is a parameter (see Fig. 7.5). Due to the additional information about the text in the SPat array, a binary search can be performed directly without accessing the disk. As a result, most of the searching work is done in main memory, thus reducing the number of disk accesses. Searching for a pattern using this structure is done in two phases:

First, a binary search is performed on the SPat array, with no disk accesses, to find the suffix array block containing the pattern occurrence. Additional disk accesses are necessary only if the pattern is longer than ℓ and there are multiple entries in the SPat array that match the prefix of length ℓ of the pattern. Then, $\mathcal{O}(\log_2 r)$ disk accesses are needed, where r is the number matching entries.

[2] A suffix array is sometimes also referred to as a Pat array.

Second, the suffix array block encountered in the first phase is moved from disk to main memory. A binary search is performed between main memory (suffix array block containing the answer) and disk (text file) to find the first and last entries that match the pattern. If the pattern occurs more than p times in the text, these occurrences may be bounded by at most two SPat array entries. In this case the left and right blocks are used in the last phase of the binary search following the same procedure.

The main advantages of this structure are its space efficiency (little more than a suffix array) and ease of implementation. See [83] for the analytical and experimental results. This structure does not support updates efficiently.

7.4.2 Compact Pat Trees

Clark and Munro [206] have presented a Pat tree data structure for full-text indexing that can be adapted to external memory to reduce the number of disk accesses while searching, and also handles updates efficiently. It requires little more storage than $\log_2 N$ bits per suffix, required to store the suffix array. It uses a compact tree encoding to represent the tree portion of the Pat tree and to obtain an efficient data structure for searching static text in primary storage. This structure, called a *Compact Pat Tree* (CPT), is then used to obtain a data structure for searching on external memory. The main idea here is to partition the Pat tree into pieces that fit into a disk block, so that no disk accesses are required while searching within a partition.

Compact Representation. To represent the Pat tree in a compact form, first the strings are converted to binary using a binary encoding of the characters, as explained in Section 7.3.2, which gets rid of the space needed to store the edge labels. The underlying binary tree (without the skip values and the pointers to the suffixes at the leaves) is then stored using an encoding similar to the well known compact tree encoding of Jacobson [425]. The compact tree representation of Clark and Munro takes less than three bits per node to represent a given binary tree and supports the required tree navigational operations (parent, left child, right child and subtree size) in constant time.

For storing the skip values at the internal nodes in a Pat tree, they use the observation that large skip values are unlikely (occur very rarely). This low likelihood of large skip values leads to a simple method of compactly encoding the skip values. A small fixed number of bits are reserved to hold the skip value for each internal node. Problems caused by overflows are handled by inserting a new node and a leaf into the tree and distributing the skip bits from the original node across the skip fields of the new node and the original node. A special key value that can be easily recognized (say all 0s) is stored with these dummy leaf nodes. Multiple overflow nodes and leaves can be inserted for extremely large skip values. Note that skip values are not needed at the leaves, as we store the pointers to the suffixes at the leaves.

Under the assumption that the given text (in binary) is generated by a uniform symmetric random process, and that the bit strings in the suffixes are independent, Clark and Munro show that the expected size of the CPT can be made less than $3.5 + \log_2 N + \log_2 \log_2 N + \mathcal{O}(\log_2 \log_2 \log_2 N / \log_2 N)$ bits per node. This is achieved by setting the skip field size to $\log_2 \log_2 \log_2 N$. They also use some space saving techniques to reduce the storage requirement even further, by compromising on the query performance.

External Memory Representation. To control the accesses to the external memory during searching, Clark and Munro use the method of decomposing the tree into disk block sized pieces, each called a partition. Each partition of the tree is stored using the CPT structure described above. The only change required to the CPT structure for storing the partitions is that the offset pointers in a block may now point to either a suffix in the text or to a subtree (partition). Thus an extra bit is required to distinguish these two cases. They use a greedy bottom-up partitioning algorithm and show that such a partitioning minimizes the maximum number of disk blocks accessed when traversing from the root to any leaf. While the partitioning rules described by Clark and Munro minimize the maximum number of external memory accesses, these rules can produce many small pages and poor fill ratios. They also suggest several methods to overcome this problem. They show that the maximum number of pages traversed on any root to leaf path is at most $1 + \lceil H/\sqrt{B} \rceil + \lceil 2 \log_B N \rceil$, where H is the height of the Pat tree. Thus searching using the CPT structure takes $\mathcal{O}(\text{scan}(|P| + Z) + \text{search}(N))$ I/Os, assuming that the height H is $\mathcal{O}(\sqrt{B} \log_B N)$. Although H could be $\Theta(N)$ in the worst case, it is logarithmic for a random text under some reasonable conditions on the distribution [711].

Updates. The general approach to updating the static CPT representation is to search each suffix of the modified document and then make appropriate changes to the structure based on the path searched. While updating the tree, it may become necessary to re-partition the tree in order to retain the optimality. The solution described by Clark and Munro to insert or delete a suffix requires time proportional to the depth of the tree, and operates on the compact form of the tree. A string is inserted to or deleted from the text by inserting/deleting all its suffixes separately. See [206] for details and some experimental results.

7.4.3 String B-trees

Ferragina and Grossi [296] have introduced the *string B-tree* which is a combination of B-trees (see Chapter 2) and Patricia tries. String B-trees link external memory data structures to string matching data structures, and overcome the theoretical limitations of inverted files (modifiability and atomic keys), suffix arrays (modifiability and contiguous space) and Pat trees (unbalanced tree topology). It has the same worst case performance as B-trees but handles unbounded length strings and performs powerful search operations such as the ones supported by Pat trees. String B-trees have also been applied to

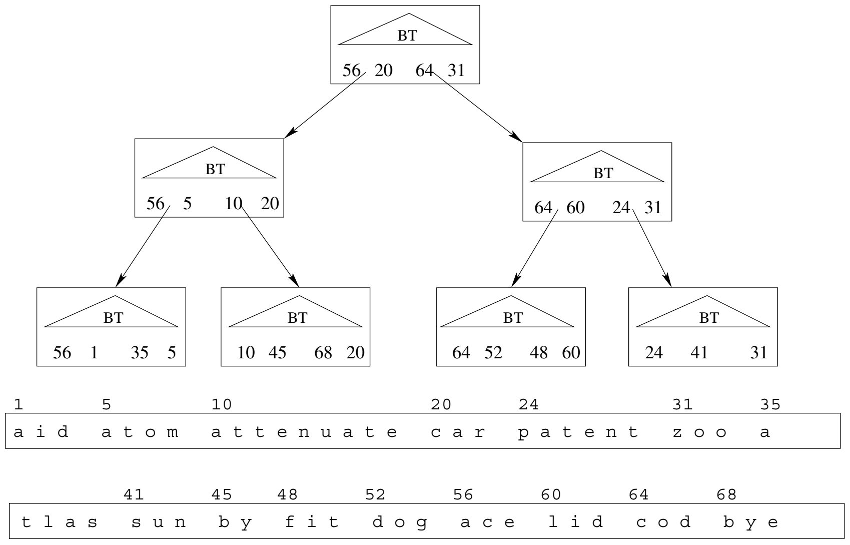


Fig. 7.6. String B-tree

(external and internal) dynamic dictionary matching [298] and some other internal memory problems [296].

String B-trees are designed to solve the dynamic version of the indexed string matching problem (Problem 1). For simplicity, we mainly describe the structure for solving the prefix search problem. As mentioned in Section 7.3.1, a string matching query can be supported by storing the suffixes of all the text strings, and supporting prefix search on the set of all suffixes.

String B-tree Data Structure. Given a set $\mathcal{S} = \{s_1, \ldots, s_N\}$ of N strings (the suffixes), a string B-tree for $\mathcal{S}$ is a B-tree in which all the keys are stored at the leaves and the internal nodes contain copies of some of these keys. The keys are the logical pointers to the strings (stored in external memory) and the order between the keys is the lexicographic order among the strings pointed to by them. Each node v of the string B-tree is stored in a disk block and contains an ordered string set $\mathcal{S}_v \subseteq \mathcal{S}$, such that $b \leq |\mathcal{S}_v| \leq 2b$, where $b = \Theta(B)$ is a parameter which depends on the disk block size B. If we denote the leftmost (rightmost) string in $\mathcal{S}_v$ by $L(v)$ $(R(v))$, then the strings in $\mathcal{S}$ are distributed among the string B-tree nodes as follows (see Fig. 7.6 for an example):

– Partition $\mathcal{S}$ into groups of b strings except for the last group, which may contain from b to $2b$ strings. Each group is mapped into a leaf v (with string set $\mathcal{S}_v$) in such a way that the left-to-right scanning of the string B-tree leaves gives the strings in $\mathcal{S}$ in lexicographic order. The longest common prefix length $lcp(S_j, S_{j+1})$ is associated with each pair (S_j, S_{j+1}) of $\mathcal{S}_v$'s strings.

– Each internal node v of the string B-tree has $d(v)$ children $u_1, \ldots, u_{d(v)}$, with $b/2 \leq d(v) \leq b$ (except for the root, which has from 2 to b children). The set $\mathcal{S}_v$ is formed by copying the leftmost and rightmost strings contained in each of its children, from left to right. More formally, $\mathcal{S}_v$ is the ordered string set $\{L(u_1), R(u_1), L(u_2), R(u_2), \ldots, L(u_{d(v)}), R(u_{d(v)})\}$.

Since the branching factor of the string B-tree is $\Theta(B)$, its height is $\Theta(\log_B N)$.

Each node v of the string B-tree stores the set $\mathcal{S}_v$ (associated with the node v) as a Patricia trie (also called a *blind trie*). To maximize the number b of strings stored in each node for a given value of B, these blind tries are stored in a succinct form (the tree encoding of Clark and Munro, for example). When a node v is transferred to the main memory, the explicit representation of its blind trie is obtained by uncompressing the succinct form, in order to perform computation on it.

Search Algorithm. To search for a given pattern P, we start from the root of the string B-tree and follow a path to a leaf, searching for the position of P at each node. At each internal node, we search for its child node u whose interval $[L(u), R(u)]$ contains P. The search at node v is done by first following the path governed by the pattern to reach a leaf l in the blind trie. If the search stops at an internal node because the pattern has exhausted, choose l to be any descendant leaf of that node. This leaf does not necessarily identify the position of P in $\mathcal{S}_v$, but it provides enough information to find this position, namely, it points to one of the strings in $\mathcal{S}_v$ that shares the longest common prefix with P. Now, we compare the string pointed to by l with P to determine the length p of their longest common prefix. Then we know that P matches the search path leading to l up to depth p, and the mismatch character $P[p+1]$ identifies the branches of the blind trie between which P lies, allowing us to find the position of P in S_v. The search is then continued in the child of v that contains this position.

Updates. To insert a string S into the set $\mathcal{S}$, we first find the leaf v and the position j inside the leaf where S has to be inserted by searching for the string S. We then insert S into the set $\mathcal{S}_v$ at position j. If $L(v)$ or $R(v)$ change in v, then we extend the change to v's ancestor. If v gets full (i.e., contains more than $2b$ strings), we split the node v by creating a new leaf u and making it an adjacent leaf of v. We then split the set S_v into two roughly equal parts of at least b strings each and store them as the new string sets for v and u. We copy the strings $L(v)$, $R(v)$, $L(u)$ and $R(u)$ in their parent node, and delete the old strings $L(v)$ and $R(v)$. If the parent also gets full, then we split it. In the worst case the splitting can extend up to the root and the resulting string B-tree's height can increase by one. Deletions of the strings are handled in a similar way, merging a node with its adjacent node whenever it gets half-full. The I/O complexity of insertion or deletion of a string S is $\mathcal{O}(\mathrm{scan}(|S|) + \mathrm{search}(N))$.

For the dynamic indexed string matching problem, to insert a string S into the text, we have to insert all its suffixes. A straightforward way of doing this requires $\mathcal{O}(\text{scan}(|S|^2) + |S|\text{search}(N + |S|))$ I/Os. By storing additional information with the nodes (similar to the suffix links described in Section 7.3.1), the quadratic dependence on the length of S can be eliminated. The same holds for deletions.

Results. Using string B-tree, one can get the following bounds for the dynamic indexed string matching problem:

Theorem 7.3. *The string B-tree of a text $\mathcal{T}$ of total length N supports*

– *string matching with a pattern P in $\mathcal{O}(\text{scan}(|P| + Z) + \text{search}(N))$ I/Os,*
– *inserting or deleting a string S into/from $\mathcal{T}$ in $\mathcal{O}(|S|\text{search}(N + |S|))$ I/Os,*

and occupies $\Theta(N/B)$ disk blocks.

The space occupied by the string B-tree is asymptotically optimal, as the space required to store the given set of strings is also $\Theta(N/B)$ disk blocks. Also the string B-tree operations take asymptotically optimal CPU time, that is, $\mathcal{O}(Bd)$ time if d disk blocks are read or written, and they only need to keep a constant number of disk blocks in the main memory at any time.

See [295] for some experimental results on string B-trees.

7.4.4 Other Data Structures

Recently, Ciriani et al. [205] have given a randomized data structure that supports lexicographic predecessor queries (which can be used for implementing prefix searching) and achieves optimal time and space bounds in the amortized sense. More specifically, given a set of N strings $S_1, \ldots, S_N$ and a sequence of m patterns $P_1, \ldots, P_m$, their solution takes $\mathcal{O}(\sum_{i=1}^{m} \text{scan}(|P_i|) + \sum_{i=1}^{N}(n_i \log_B(m/n_i))$ expected amortized I/Os, where n_i is the number of times S_i is the answer to a query. Inserting or deleting a string S takes $\mathcal{O}(\text{scan}(|S|) + \text{search}(N))$ expected amortized I/Os. The search time matches the performance of string B-trees for uniform distribution of the answers, but improves on it for biased distributions. This result is the analog of the Static Optimality Theorem of Sleator and Tarjan [699] and is achieved by designing a self-adjusting data structure based on the well-known skip lists [616].

7.5 External Construction

There are several efficient algorithms for constructing full-text indexes in internal memory [288, 528, 540, 736]. However, these algorithms access memory in a nearly random manner and are poorly suited for external construction. String B-trees provide the possibility of construction by insertion, but the construction time of $\mathcal{O}(N\text{search}(N))$ I/Os can be improved with specialized construction algorithms.

In this section, we describe several I/O-efficient algorithms for external memory construction of full-text indexes. We start by showing that the different full-text indexes can be transformed into each other efficiently. Therefore, any construction algorithm for one type of index works for others, too. Then, we describe two practical algorithms for constructing suffix arrays, and a theoretically optimal algorithm for constructing Pat trees. We also a look at the related problem of sorting strings.

For the sake of clarity, we assume that the text consists of a single string of length N, but all the algorithms can be easily modified to construct the full-text index of a set of strings of total length N with the same complexity. Unless otherwise mentioned, disk space requirement, CPU time, and speedup with parallel disks are optimal.

7.5.1 Construction from Another Index

In this section, we show that the different forms of full-text indexes we have seen are equivalent in the sense that any of them can be constructed from another in $\mathcal{O}(\text{sort}(N))$ I/Os. To be precise, this is not true for the plain suffix array, which needs to be augmented with the longest common prefix array: $LCP[i]$ is the *longest common prefix* of the suffixes starting at $SA[i-1]$ and $SA[i]$. The suffix array construction algorithms described below can be modified to construct the LCP array, too, with the same complexity. The transformation algorithms are taken from [290].

We begin with the construction of a suffix array SA (and the LCP array) from a suffix tree ST (or a Pat tree PT which has the same structure differing only in edge labels). We assume that the children of a node are ordered lexicographically. First, construct the Euler tour of the tree in $\mathcal{O}(\text{sort}(N))$ I/Os (see Chapter 3). The order of the leaves of the tree in the Euler tour is the lexicographic order of the suffixes they represent. Thus, the suffix array can be formed by a simple scan of the Euler tour. Furthermore, let w be the highest node that is between two adjacent leaves u and v in the Euler tour. Then, w is the lowest common ancestor of u and v, and the depth of w is the length of the longest common prefix of the suffixes that u and v represent. Thus LCP can also be computed by a scan of the Euler tour.

The opposite transformation, constructing ST (or PT) given SA and LCP, proceeds by inserting the suffixes into the tree in lexicographic order, i.e., inserting the leaves from left to right. Thus, a new leaf u always becomes the rightmost child of a node, say v, on the rightmost path in the tree (see Fig. 7.7). Furthermore, the longest common prefix tells the depth of v (the insertion depth of u). The nodes on the rightmost path are kept in a stack with the leaf on top. For each new leaf u, nodes are popped from the stack until the insertion depth is reached. If there was no node at the insertion depth, a new node v is created there by splitting the edge. After inserting u as the child of v, v and u are pushed on the stack. All the stack operations can be performed with $\mathcal{O}(\text{scan}(N))$ I/Os using an external stack (see Chapter 2). The

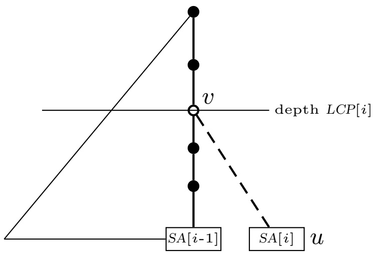


Fig. 7.7. Inserting a new leaf u representing the suffix $SA[i]$ into the suffix tree

construction numbers the nodes in the order they are created and represents the tree structure by storing with each node its parent's number. Other tree representations can then be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.

The string B-tree described in Section 7.4.3 can also be constructed from the suffix array and the *LCP* array in $\mathcal{O}(\text{sort}(N))$ I/Os with a procedure similar to the suffix tree construction. The opposite transformation is also similar.

7.5.2 Merging Algorithm

The merging algorithm was introduced by Gonnet, Baeza-Yates and Snider [340] and improved by Crauser and Ferragina [227]. The basic idea is to build the suffix array in memory-sized pieces and merge them incrementally. More precisely, the algorithm divides the text $\mathcal{T}$ into $h = \Theta(N/M)$ pieces of size $\ell = \Theta(M)$, i.e., $\mathcal{T} = \mathcal{T}_h\mathcal{T}_{h-1}\dots\mathcal{T}_2\mathcal{T}_1$ (note the order). The suffix array is built incrementally in h stages. In stage k, the algorithm constructs $SA_{\mathcal{T}_k}$ internally, and merges it with $SA_{\mathcal{T}_{k-1}\dots\mathcal{T}_1}$ externally.

The algorithm is best described as constructing the *inverse* SA^{-1} of the suffix array SA (see Figs. 7.2 and 7.8). While $SA[i]$ is the starting position of the ith suffix in the lexicographic order, $SA^{-1}[j]$ is the lexicographic rank of the suffix starting at j, i.e., $SA^{-1}[SA[i]] = i$. Obviously, SA can be computed from SA^{-1} by permutation in $\mathcal{O}(\text{sort}(N))$ I/Os. Note that $SA^{-1}[j]$ is a lexicographic name of the suffix j in the set of suffixes.

A stage k of the algorithm consists of three steps:

1. build $SA_{\mathcal{T}_k}$
2. update $SA^{-1}_{\mathcal{T}_{k-1}\dots\mathcal{T}_1}$ into the $k-1$ last pieces of $SA^{-1}_{\mathcal{T}_k\dots\mathcal{T}_1}$
3. transform $SA_{\mathcal{T}_k}$ into the first piece of $SA^{-1}_{\mathcal{T}_k\dots\mathcal{T}_1}$

Let us call the suffixes starting in $\mathcal{T}_k$ the new suffixes and the suffixes starting in $\mathcal{T}_{k-1}\dots\mathcal{T}_1$ the old suffixes. During the stage, a suffix starting at i is represented by the pair $\langle\mathcal{T}[i\dots i+\ell-1], SA^{-1}[i+\ell]\rangle$.[3] Since $SA^{-1}[i+\ell]$ is a lexicographic name, this information is enough to determine the order of

[3] The text is logically appended with ℓ copies of the character \$ to make the pair well-defined for all suffixes.

suffixes. The first step loads into internal memory $\mathcal{T}_k$, $\mathcal{T}_{k-1}$, and the first ℓ entries of $SA^{-1}_{\mathcal{T}_{k-1}\dots\mathcal{T}_1}$, i.e., the part corresponding to $\mathcal{T}_{k-1}$. Using this information, the representative pairs are formed for all new suffixes and the suffix array $SA_{\mathcal{T}_k}$ of the new suffixes is built, all in internal memory.

The second step is performed by scanning $\mathcal{T}_{k-1}\dots\mathcal{T}_1$ and $SA^{-1}_{\mathcal{T}_{k-1}\dots\mathcal{T}_1}$ simultaneously. When processing a suffix starting at i, $SA^{-1}[i]$, $SA^{-1}[i+\ell]$, and $\mathcal{T}[i, i+\ell-1]$ are in internal memory. The latter two are needed for the representative pair of the suffix and the first is modified. For each i, the algorithm determines using $SA_{\mathcal{T}_k}$ how many of the new suffixes are lexicographically smaller than the suffix starting at i, and $SA^{-1}[i]$ is increased by that amount. During the scan, the algorithm also keeps an array C of counters in memory. The value $C[j]$ is incremented during the scan when an old suffix is found to be between the new suffixes starting at $SA_{\mathcal{T}_k}[j-1]$ and $SA_{\mathcal{T}_k}[j]$. After the scan, the ranks of the new suffixes are easy to compute from the counter array C and $SA_{\mathcal{T}_k}$ allowing the execution of the third step.

The algorithm performs $\mathcal{O}(N/M)$ stages, each requiring a scan through an array of size $\mathcal{O}(N)$. Thus, the I/O complexity is $\mathcal{O}((N/M)\operatorname{scan}(N))$. The CPU complexity deserves a closer analysis, since, according to the experiments in [227], it can be the performance bottleneck. In each stage, the algorithm needs to construct the suffix array of the new suffixes and perform $\mathcal{O}(N)$ queries. Using the techniques by Manber and Myers [528], the construction requires $\mathcal{O}(M\log_2 M)$ time and the queries $\mathcal{O}(NM)$ time. In practice, the query time is $\mathcal{O}(N\log_2 M)$ with a constant depending on the type of the text. Thus, the total CPU time is $\mathcal{O}(N^2)$ in the worst case and $\mathcal{O}((N^2/M)\log_2 M)$ in practice.

Despite the quadratic dependence on the length of the text, the algorithm is fast in practice up to moderate sized texts, i.e., for texts with small ratio N/M [227]. For larger texts, the doubling algorithm described next is preferable.

7.5.3 Doubling Algorithm

The *doubling algorithm* is based on the lexicographic naming and doubling technique of Karp, Miller and Rosenberg [450]. It was introduced for external sorting of strings by Arge et al. [61] (see Section 7.5.5) and modified for suffix array construction by Crauser and Ferragina [227]. We present an improved version of the algorithm.

Let r_k be the lexicographic naming of the set of substrings of length 2^k in the text appended with $2^k - 1$ copies of the character \$ (see Fig. 7.8). In other words, $r_k(i)$ is one plus the number of substrings of length 2^k that are strictly smaller than the substring starting at i. The doubling algorithm constructs r_k for $k = 1, 2, \dots, \lceil\log_2 N\rceil$. As Fig. 7.8 illustrates, $r_{\lceil\log_2 N\rceil}$ is the same as the inverse suffix array SA^{-1}.

Let us see what happens in a stage k that constructs r_k. In the beginning, each position i is represented by the triple $\langle r_{k-1}(i), r_{k-1}(i+2^{k-1}), i\rangle$, and

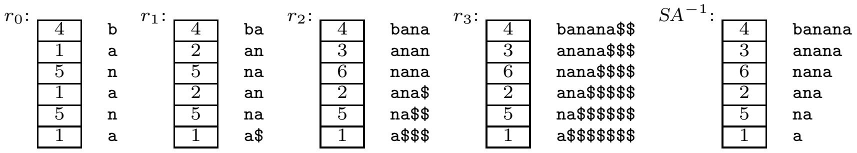


Fig. 7.8. The doubling algorithm for the text banana

the triples are stored in the order of the last component. The following steps are then performed:

1. sort the triples by the first two components (which is equivalent to sorting substrings of length 2^k)
2. scan to compute r_k and update the triples to $\langle r_k(i), r_{k-1}(i+2^{k-1}), i\rangle$
3. sort the triples by the last component
4. scan to update the triples to $\langle r_k(i), r_k(i+2^k), i\rangle$

The algorithm does $\mathcal{O}(\text{sort}(N))$ I/Os in each stage, and thus requires a total of $\mathcal{O}(\text{sort}(N)\log_2 N)$ I/Os for constructing the suffix array.

The algorithm can be improved using the observation that, if a name $r_k(i)$ is unique, then $r_h(i) = r_k(i)$ for all $h > k$. We call a triple with a unique first component *finished*. Crauser and Ferragina [227] show how step 2 can be performed without using finished triples allowing the exclusion of finished triples from the sorting steps. This reduces the I/O complexity of stage k to $\mathcal{O}(\text{sort}(N_{k-1}) + \text{scan}(N))$, where N_{k-1} is the number of unfinished triples after stage $k-1$. We show how step 4 can also be done without finished triples, improving the I/O complexity further to $\mathcal{O}(\text{sort}(N_{k-1}))$.

With only the unfinished triples available in step 2, the new rank of a triple can no more be computed as its rank in the sorted list. Instead, the new rank of a triple $\langle x, y, i\rangle$ is $x+c$, where c is the number of triples in the list with the first component x and the second component smaller than y. This works correctly because $x = r_{k-1}(i)$ already counts the smaller substrings that differ in the first 2^{k-1} characters, and all the triples that have the same first component are unfinished and thus on the list.

The newly finished triples are identified and marked in step 2 but not removed until in step 4, which we describe next. When the scan in step 4 processes $\langle x, y, i\rangle$, the triples $\langle x', y', i'\rangle$, $i' = i + 2^{k-1}$, and $\langle x'', y'', i''\rangle$, $i'' = i + 2^k$ are also brought into memory if they were unfinished after stage $k-1$. The following three cases are possible:

1. If $\langle x'', y'', i''\rangle$ exists (is unfinished), the new triple is $\langle x, x'', i\rangle$.
2. If $\langle x', y', i'\rangle$ exists but $\langle x'', y'', i''\rangle$ does not, $\langle x'', y'', i''\rangle$ was already finished before stage k, and thus y' is its final rank. Then, $\langle x, y', i\rangle$ is the new triple.
3. If $\langle x', y', i'\rangle$ does not exist, it was already finished before stage k. Then, the triple $\langle x, y, i\rangle$ must now be finished and is removed.

The finished triples are collected in a separate file and used for constructing the suffix array in the end.

Let us analyze the algorithm. Let N_k be the number of non-unique text substrings of length 2^k (with each occurrence counted separately), and let s be the largest integer such that $N_s > 0$. The algorithm needs $\mathcal{O}(\text{sort}(N_{k-1}))$ I/Os in stage k for $k = 1, \dots, s+1$. Including the initial stage, this gives the I/O complexity $\mathcal{O}(\text{sort}(N) + \sum_{k=0}^{s} \text{sort}(N_k))$. In the worst case, such as the text $\mathcal{T} = \texttt{aaa...aa}$, the I/O complexity is still $\mathcal{O}(\text{sort}(N) \log_2(N))$. In practice, the number of unfinished suffixes starts to decrease significantly much before stage $\log_2(N)$.

7.5.4 I/O-Optimal Construction

An algorithm for constructing the Pat tree using optimal $\mathcal{O}(\text{sort}(N))$ I/Os has been described by Farach-Colton et al. [290]. As explained in Section 7.5.1, the bound extends to the other full-text indexes. The outline of the algorithm is as follows:

1. Given the string $\mathcal{T}$ construct a string $\mathcal{T}'$ of half the length by replacing pairs of characters with lexicographic names.
2. Recursively compute the Pat tree of $\mathcal{T}'$ and derive the arrays $SA_{\mathcal{T}'}$ and $LCP_{\mathcal{T}'}$ from it.
3. Let SA_o and LCP_o be the string and LCP arrays for the suffixes of $\mathcal{T}$ that start at odd positions. Compute SA_o and LCP_o from $SA_{\mathcal{T}'}$ and $LCP_{\mathcal{T}'}$.
4. Let SA_e and LCP_e be the string and LCP arrays for the suffixes of $\mathcal{T}$ that start at even positions. Compute SA_e and LCP_e from SA_o and LCP_o.
5. Construct the Patricia tries PT_o and PT_e of odd and even suffixes from the suffix and LCP arrays.
6. Merge PT_o and PT_e into $PT_{\mathcal{T}}$.

Below, we sketch how all the above steps except the recursive call can be done in $\mathcal{O}(\text{sort}(N))$ I/Os. Since the recursive call involves a string of length $N/2$, the total number of I/Os is $\mathcal{O}(\text{sort}(N))$.

The naming in the first step is done by sorting the pairs of characters and using the rank as the name. The transformations in the second and fifth step were described in Section 7.5.1. The odd suffix array SA_o is computed by $SA_o[i] = 2 \cdot SA_{\mathcal{T}'}[i] - 1$. The value $LCP_o[i]$ is first set to $2 \cdot LCP_{\mathcal{T}'}[i]$, and then increased by one if $\mathcal{T}[SA_o[i] + LCP_o[i]] = \mathcal{T}[SA_o[i-1] + LCP_o[i]]$. This last step can be done by batched lookups using $\mathcal{O}(\text{sort}(N))$ I/Os.

Each even suffix is a single character followed by an odd suffix. Let SA_o^{-1} be the inverse of SA_o, i.e., a lexicographical naming of the odd suffixes. Then, SA_e can be constructed by sorting pairs of the form $\langle \mathcal{T}[2i], SA_o^{-1}[2i+1] \rangle$. The LCP of two adjacent even suffixes is zero if the first character does not match, and one plus the LCP of the corresponding odd suffixes otherwise. However, the corresponding odd suffixes may not be adjacent in SA_o. Therefore, to compute LCP_e we need to perform LCP queries between $\mathcal{O}(N)$ arbitrary pairs of odd suffixes. By a well-known property of LCP arrays, the length of the longest common prefix of the suffixes starting at $SA_o[i]$ and $SA_o[j]$ is $\min_{i<k\leq j} LCP_o[k]$. Therefore, we need to answer a batch of $\mathcal{O}(N)$ range minimum queries, which takes $\mathcal{O}(\text{sort}(N))$ I/Os [192].

The remaining and the most complex part of the algorithm is the merging of the trees. We will only outline it here. Our description differs from the one in [290] in some aspects. However, the high level procedure — identify trunk nodes (called odd/even nodes in [290]), overmerge, then unmerge — is the same.

We want to create the Pat tree $PT_\mathcal{T}$ of the whole text $\mathcal{T}$ by merging the Patricia tries PT_o and PT_e. $PT_\mathcal{T}$ inherits a part of the tree from PT_o, a part from PT_e, and a part from both. The part coming from both is a connected component that contains the root. We will call it the *trunk*. Recall that a Patricia trie is a compact representation of a trie. Thus it represents all the nodes of the trie, some of them *explicitly*, most of them *implicitly*. The trunk of $PT_\mathcal{T}$ contains three types of (explicit) nodes:

1. nodes that are explicit in both PT_o and PT_e,
2. nodes that are explicit in one and implicit in the other, and
3. nodes that are implicit in both.

Before starting the merge, the trunk nodes in PT_o and PT_e are marked. This is a complicated procedure and we refer to [290] for details. The procedure may mark some non-trunk nodes, too, but only ones that are descendants of the *leaves* of the trunk. In particular, the nearest explicit descendant of each implicit trunk leaf should be marked.

The merging process performs a coupled-DFS of PT_o and PT_e using the Euler tours of the trees. During the process the type 1 trunk nodes are easy to merge but the other two types cause problems. Let us look at how the merging proceeds. Suppose we have just merged nodes u_o of PT_o and u_e of PT_e into a type 1 node u. Next in the Euler tours are the subtrees rooted at the children of u_o and u_e. The subtrees are ordered by the first character of the edges leading to them, which makes it easy to find the pairs of subtrees that should be merged. If the edges leading to the subtrees have the same full label, the roots of the subtrees are merged and the merging process continues recursively in the subtree. However, the full labels of the edges are not available in Patricia tries, only the first character and the length. The first character is already known to match. With respect to the edge lengths there are two cases:

1. If the lengths are the same, the subtrees are merged. If the full labels are different, the correct procedure would be to merge the edges only partially, creating a type 3 node where the edges are separated. The algorithm, however, does merge the edges fully; this is called *overmerging*. Only afterwards these incorrectly merged edges are identified and unmerged. What happens in the incorrect recursive merging of the subtrees

does not matter much, because the unmerging will throw away the incorrect subtree and replace it with the original subtrees from PT_o and PT_e. For further details on unmerging, we refer to [290].

2. If the lengths are different, the longer edge is split by inserting a new node v' on it. The new node is then merged with end node v'' of the other edge to form a trunk node v of type 2. This could already be overmerging, which would be corrected later as above. In any case, the recursive merging of the subtrees continues, but there is a problem: the initial character of the edge leading to the only child w' of v' (i.e., the lower part of the split edge) is not known. Retrieving the character from the text could require an I/O, which would be too expensive. Instead, the algorithm uses the trunk markings. Suppose the correct procedure would be to merge the edge (v', w') with an edge (v'', w'') at least partially. Then w'' must be a marked node, and furthermore, w'' must be the only marked child of v'', enabling the correct merge to be performed. If (v', w') should not be merged with any child edge of v'', i.e., if v'' does not have a child edge with the first character matching the unknown first character of (v', w'), then any merge the algorithm does is later identified in the unmerging step and correctly unmerged. Then the algorithm still needs to determine the initial character of the edge, which can be done in one batch for all such edges.

Theorem 7.4 (Farach-Colton et al. [290]). *The suffix array, suffix tree, Pat tree, and string B-tree of a text of total length N can be constructed in optimal $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

7.5.5 Sorting Strings

In this section, we consider the problem of sorting strings, which is closely related to the construction of full-text indexes. The results are all by Arge et al. [61].

We begin with a practical algorithm using the doubling technique.[4] The algorithm starts by storing the strings in a single file, padded to make the string lengths powers of two and each string aligned so that its starting position in the file is a multiple of its (padded) length. The algorithm proceeds in $\mathcal{O}(\log_2 N)$ stages similar to the doubling algorithm of Section 7.5.3. In stage k, it names substrings of size 2^k by sorting pairs of names for substrings of half the length. However, in contrast to the algorithm of Section 7.5.3, the substrings in each stage are *non-overlapping.* As a consequence, the number of substrings is halved in each stage, and the whole algorithm runs in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

The rank of a string of (padded) length 2^k is computed in stage k, where it is one of the named substrings. The number of lexicographically smaller

[4] The algorithm was only mentioned, not described in [61]; thus, the details are ours.

strings of the same or greater length is determined by counting the number of lexicographically smaller substrings that are prefixes of a string. The number of lexicographically smaller, shorter strings is determined by maintaining a count of those strings at each substring. The counts are updated at the end of each stage and passed to the next stage. All the counting needs only $\mathcal{O}(\text{scan}(N))$ I/Os over the whole algorithm.

Arge et al. also show the following theoretical upper bound.

Theorem 7.5 (Arge et al. [61]). *The I/O complexity of sorting K strings of total length N with K_1 strings of length less than B of total length N_1, and K_2 strings of length at least B of total length N_2, is*

$$\mathcal{O}\left(\min\left\{K_1 \log_M K_1, \frac{N_1}{B} \log_{M/B} \frac{N_1}{B}\right\} + K_2 \log_M K_2 + \text{scan}(N)\right) .$$

As suggested by the equation, the result is achieved by processing short and long strings separately. The algorithm for long strings works by inserting the strings into a *buffered string B-tree*, which is similar to the string B-tree (Section 7.4.3), but all the insertions are done in one batch using buffering techniques from the buffer tree (see Chapter 2).

Arge et al. also show lower bounds for sorting strings in somewhat artificially weakened models. One of their lower bounds nearly matches the upper bound of the above theorem. (The algorithms behind Theorem 7.5 work in this model.) The full characterization of the I/O complexity of sorting strings, however, remains an open problem.

7.5.6 Packed Strings

So far we have used the integer alphabet model, which assumes that each character occupies a full machine word. In practice, alphabets are often small and multiple characters can be packed into one machine word. For example, DNA sequences could be stored using just two bits per character. Some of the algorithms, in particular the doubling algorithms, can take advantage of this. To analyze the effect, we have to modify the model of computation.

The *packed string model* assumes that characters are integers in the range $\{1, \ldots, |\Sigma|\}$, where $|\Sigma| \leq N$. Strings are stored in packed form with each machine word containing $\Theta(\log_{|\Sigma|} N)$ characters. The main parameters of the model are:

N = number of characters in the input strings
$n = \Theta(N/\log_{|\Sigma|} N)$ = size of input in units of machine words
M = size of internal memory in units of machine words
B = size of disk blocks in units of machine words

Note that, while the size of the text is $\Theta(n)$, the size of a full-text index is $\Theta(N)$ machine words.

Under the packed string model, the results presented in this chapter remain mostly unaffected, since the algorithms still have to deal with $\Theta(N)$ word-sized entities, such as pointers and ranks. There are some changes, though. The worst case CPU complexity of the merging algorithm is reduced by a factor $\Theta(\log_{|\Sigma|} N)$ due to the reduction in the complexity of comparing strings.

The doubling algorithm for both index construction and sorting can be modified to name substrings of length $\Theta(\log_{|\Sigma|} N)$ in the initial stage. The I/O complexity of the index construction algorithm then becomes $\mathcal{O}(\mathrm{sort}(N) + \sum_{k=0}^{s} \mathrm{sort}(n_k))$, where n_k is the number of non-unique text substrings of length $2^k \log_{|\Sigma|} N$. On a random text with independent, uniform distribution of characters, this is $\mathcal{O}(\mathrm{sort}(N))$ with high probability [711]. The I/O complexity of the sorting algorithm becomes $\mathcal{O}(\mathrm{sort}(n + K))$.

7.6 Concluding Remarks

We have considered only the simple string matching queries. Performing more complex forms of queries, in particular *approximate string matching* [574], in external memory is an important open problem. A common approach is to resort to sequential searching either on the whole text (e.g, the most widely used genomic sequence search engine BLAST [37]) or on the word list of an inverted file [51, 84, 529]. Recently, Chávez and Navarro [179] turned approximate string matching into nearest neighbor searching in metric space, and suggested using existing external memory data structures for the latter problem (see Chapter 6).

8. Algorithms for Hardware Caches and TLB

Naila Rahman*

8.1 Introduction

Over the last 20 years or so CPU clock rates have grown explosively, and CPUs with clock rates exceeding 2 GHz are now available in the mass market. Unfortunately, the speed of main memory has not increased as rapidly: today's main memory typically has a latency of about 60 ns. This implies that the cost of accessing main memory can be 120 times greater than the cost of performing an operation on data which are in the CPU's registers. Since the driving force behind CPU technology is speed and that behind memory technology is storage capacity, this trend is likely to continue. Researchers have long been aware of the importance of reducing the number of accesses to main memory in order to avoid having the CPU wait for data.

8.1.1 Internal Memory Hierarchy

Several studies, for example [499], have shown that many computer programs have good *locality* of reference. They may have good *temporal locality*, where a memory location once accessed is soon accessed again, and have good *spatial locality*, where an access to a memory location is followed by an access to a nearby memory location. The hardware solution to the problem of a slow main memory is to exploit the locality inherent in many programs by having a *memory hierarchy* which inserts multiple levels of *cache* between the CPU registers (or just CPU) and main memory. A cache is a fast memory which holds the contents of some main memory locations. If the CPU requests the contents of a main memory location, and the contents of that location is held in some level of cache, the CPU's request is answered by the cache itself (a *cache hit*); otherwise it is answered by consulting the main memory (a *cache miss*). A cache hit has small or no cost (*penalty*), 1-3 CPU cycles is fairly typical, but a cache miss requires a main memory access, and is therefore very expensive. To amortise the cost of a main memory access in case of a cache miss, an entire *block* of consecutive main memory locations which contains the location accessed is brought into cache on a miss. Programs which have good spatial locality benefit from the fact that data are transferred from main memory to cache in blocks, while programs which have good temporal locality benefit from the fact that caches hold several blocks of data. Such programs make fewer cache misses and consequently run faster. Caches which

* Supported by EPSRC grant GR/L92150

are increasing closer to the CPU are faster and smaller than caches further from the CPU. The cache closest to the CPU is referred to as the L1 cache and the cache at level i is referred to as the Li cache. Systems nowadays have at-least two levels of cache. For high performance, the L1 cache is almost always on the same chip as the CPU.

There is an important related optimisation which can contribute as much or more to performance, namely minimising misses in the *translation-lookaside buffer (TLB).* Some papers from the early and mid 90's (see e.g. [556, 12]) note the importance of minimising TLB misses when implementing algorithms, but there has been no systematic study of this optimisation, even though TLB misses are often at least as expensive as cache misses.

The TLB is used to support *virtual memory* in multi-processing operating systems [392]. Virtual memory means that the memory addresses accessed by a process refer to its own unique *logical address* space. This logical address space contains as many locations as can be addressed on the underlying architecture, which far exceeds the number of physical main memory locations in a typical system. Furthermore, there may be several active processes in a system, each with its own logical address space. To allow this, most operating systems partition main memory and the logical address space of each process into contiguous fixed-size *pages*, and store only some pages from the logical address space of each active process in main memory at a time. Owing to its myriad benefits, virtual memory is considered to be "essential to current computer systems" [392]. The disadvantage of virtual memory is that every time a process accesses a memory location, the reference to the corresponding logical page must be *translated* to a physical page reference. This is done by looking up the *page table*, a data structure in main memory. Using it in this way would lead to unacceptable slowdown. Note that if the logical page is not present in main memory at all, it is brought in from disk. The time for this is generally not counted in the CPU times, as some other task is allowed to execute on the CPU while the I/O is taking place.

The TLB is used to speed up address translation. It is a fast associative memory which holds the translations of recently-accessed logical pages. If a memory access results in a TLB hit, there is no delay, but a TLB miss can be significantly more expensive than a cache miss; hence locality at the page level is also very desirable. In most computer systems, a memory access can result in a TLB miss alone, a cache miss alone, neither, or both. Algorithms which make few cache misses can nevertheless have poor performance if they make many TLB misses. Fig. 8.1 shows large virtual memories for two processes, a smaller physical memory and a TLB which holds translations for a subset of the pages in physical memory.

The sizes of cache and TLB are limited by several factors including cost and speed [378]. Cache capacities are typically 16KB to 4MB, which is considerably smaller than the size of main memory. The size of TLBs are also

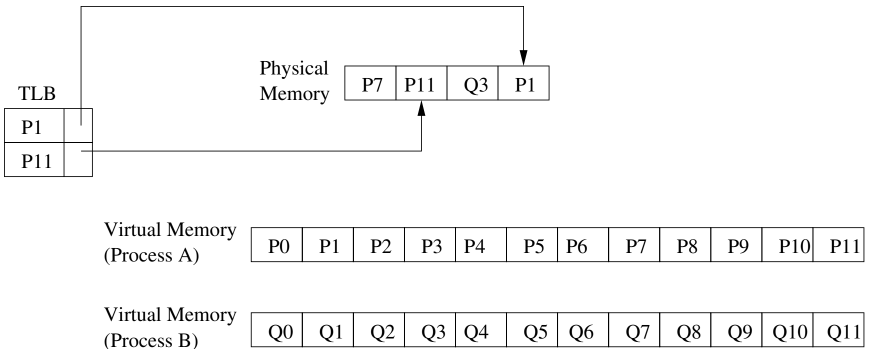


Fig. 8.1. Virtual memory, physical memory and TLB. Virtual memory for processes A and B, each with 12 pages. Physical memory with 4 pages. TLB holds virtual (logical) page to physical page translations for 2 pages.

very limited, with 64 to 128 entries being typical. Hence *simultaneous* locality at the cache and page level is important for internal-memory computation.

We refer to the caches, TLB and main memory of a computer system as the *internal memory hierarchy.*

8.1.2 Overview

In Section 8.2 we describe the basic characteristics of caches and TLB. In Section 8.3 we present two models that have been used for the internal memory hierarchy. In Section 8.4 we briefly survey techniques and algorithms that have been designed in order to improve TLB and/or cache utilisation. In Section 8.5 we briefly consider the impact of caches on power consumption, an increasingly important issue in system design, especially for mobile computing. In Section 8.6 we consider the usefulness and limitations of exploiting algorithms designed for other memory models in order to obtain good performance for the internal memory hierarchy. Section 8.7 is a case study of designing an algorithm for internal memory. The algorithm considered is for the problem of sorting 32-bit integers.

8.2 Caches and TLB

In this section we give a brief introduction to caches and TLB, highlighting the features which are generally considered to have the most impact on algorithm design and analysis. The contents of this section are based on [392, 378], and the reader is referred to these books for further details.

8.2.1 Overview of Caches

When data is brought into cache, decisions have to be made about where to place the data. Some mechanism must be used to identify which main memory blocks are held in cache. Since caches have limited capacity, data may have to be *evicted* from cache in order to accommodate new data. All of this is strictly under the control of the cache hardware, even at the assembler level the programmer cannot make any of these decisions. As we will see later, this lack of programmer control can have significant implications on the performance of algorithms.

In this section we review how caches control where main memory blocks are placed, how they are found and when they are evicted.

Blocked Memory. Data is transferred between main memory and cache as a block of consecutive words. Typical block sizes are 16, 32 or 64 bytes. A cache may hold as few as 512 or as many as 64K main memory blocks. We use B to denote the number of words (data elements) in a block and M/B to denote the total number of main memory blocks the cache can hold.

Block Placement. There are three main ways of determining where to place a main memory block in cache. The data at memory address x is said to be in memory block $y = x \text{ div } B$.

In a *direct mapped* cache the data at memory address x can only be placed in cache block $(x \text{ div } B) \bmod (M/B)$.

In an a-way *set-associative* cache the blocks are organised into sets, where each set holds a blocks. The cache has $(M/B)/a$ sets and the value held at memory address x can be placed in any block in the set $(x \text{ div } B) \bmod ((M/B)/a)$. To bring a main memory block into cache, we first determine the set in which to place it and then determine the actual cache block within the set to use.

In a *fully associative* cache the value held at memory address x can be placed in any block in the cache. Direct-mapped caches are 1-way set-associative and fully-associative caches are (M/B)-way set-associative. Most caches are direct-mapped, 2-way, 4-way or 8-way set-associative. A 2-way set-associative cache is shown in Fig. 8.2.

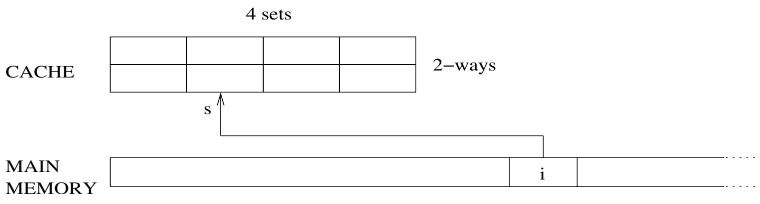


Fig. 8.2. A 2-way set-associative cache, with 4 sets. Memory block i maps to cache set s. If all blocks in set s are occupied the cache selects a random or the least-recently used block in set s for eviction.

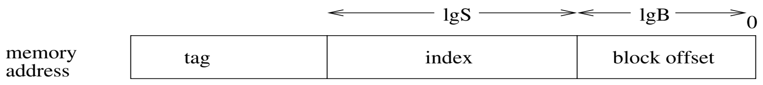


Fig. 8.3. The tag, index and block offset fields of a main memory address. $S = (M/B)/a$.

Block Identification. A memory address can be viewed as consisting of a *block address* and a *block offset*. A block address can be further divided into a *tag* and an *index*. If the block size is B bytes and the number of sets is $S = (M/B)/a$, then the lowest $\lg B$ bits of the address are the block offset, the next $\lg S$ bits are the index and the remaining bits of the address are the tag. Fig. 8.3 shows how the bits in a memory address are partitioned into the tag, index and block offset.

The index of a memory address is used to select the set that the memory block can be placed in. The cache stores the tag of the memory address along with the data.

If the CPU accesses a memory address x, we need to check if the data is in cache. The address's index is used to select the set that the block may be in and the address's tag is used to compare against all the blocks in that set for a match.

Block Replacement. If the CPU accesses memory address x which maps to set s and all blocks in the set contain valid data, then a decision needs to be made about which block to evict in order to accommodate the block at memory address x.

If the cache is direct-mapped then the decision is simple, as we just evict the contents of the single block in the set. If the cache is set-associative or fully-associative then it usually uses either a *random* replacement policy, where a randomly selected block is evicted; a *least recently used (LRU)* replacement policy, or a *pseudo-LRU* replacement policy.

Write Policy. If the CPU updates data at memory address x then, in a *write-through* cache the data is written back immediately to the next lower level of memory, which may be a cache or it may be main memory. In a *write-back* cache the data is written back to the next lower level of memory only when it is evicted from cache. Write-through caches may use write-buffers, so that the CPU does not have to wait for the update to complete. However, if an algorithm is write intensive, then the write-buffers could quickly fill and so the algorithm would start paying the penalty for access to the next lower level of memory even though data may be in cache.

Virtual Caches. Most computer systems use virtual memory and on most systems the *memory management unit*, which is responsible for translating a virtual address to a physical address, is between the CPU and cache. This means that an address that the cache sees has been translated from a virtual address to a physical memory address. Caches on such systems are termed *physical caches.*

Virtual caches, where the cache is between the CPU and the memory management unit, allow the virtual address to be used for cache accesses. On a cache hit, this eliminates the need for an address translation and so reduces the hit time. Note that on a cache miss, in order to access main memory, an address translation would still be required. Unfortunately virtual caches can considerably complicate the design of multi-tasking and mutli-processor operating systems, so most systems have physical caches.

8.2.2 TLB

A TLB entry holds the virtual to physical address translation for one virtual memory page. This translation effectively provides the translations for all memory addresses in the page. If the program accesses a memory location which belongs to (logical) page i, and the TLB holds the translation for page i, the contents of the TLB do not change. If the TLB does not hold the translation for page i, the translation for page i is brought into the TLB, and the translation for some other page is removed.

Like caches, a TLB has to address the issue of page translation placement, identification and replacement. Again, this is strictly under hardware control. The virtual page address is used to identify the page translation. TLB entries are typically fully associative and use a random, LRU or pseudo-LRU replacement policy.

8.2.3 Cache and TLB Misses

In most systems TLB misses and cache misses happen independently, in that a memory access may result in a cache miss, a TLB miss, neither, or both. This is because caches are usually *physically tagged*, i.e., the values stored in cache are stored according to their physical, and not their virtual (logical), memory addresses. Hence, when a program accesses a memory location using its logical address, the address first has to be translated to a physical address before the cache can be checked. Furthermore, a memory access which results in both a cache miss and a TLB miss pays the cache miss penalty *plus* the TLB miss penalty (there is no saving, or loss, if both kinds of misses occur simultaneously).

8.3 Memory Models

There are several ways of measuring the number of cache or TLB misses that a program makes. However any measurement is for a particular execution of an algorithm and on a particular platform, it does not tell us how an algorithm might behave on a different platform or with different parameters. Simple and precise analytical models allow us to study factors in an algorithm or in a machine platform that affect the number of cache and TLB misses and so allow us to predict behaviour as parameters vary.

Several models have been introduced to capture the memory hierarchy in a computer system and have been used to analyse algorithms and design new algorithms with improved performance. We briefly review some of these models and discuss the advantages and limitations of using them to model the internal memory hierarchy. We then introduce the *Cache Memory Model (CMM)* which has been used by several researchers to design and analyse algorithms for cache and main memory. We discuss the advantages and some of the limitations of this model and then introduce the *Internal Memory Model (IMM)* which models cache, TLB and main memory.

8.3.1 External Memory Models (EMM)

In the *Single Disk Model (SDM)* [17] there is a fast main memory, where all computation takes place, and a single slow disk, which holds data and the result of computation. Data is moved between main memory and disk as blocks. A transfer of a block of data between disk and main memory is referred to as an I/O step and the most important performance measure is the number of I/O steps an algorithm performs. In this model it is assumed that a block of data from disk can be placed at any block in main memory and the algorithm has full control over which block is evicted in order to accommodate the new block. In the *Parallel Disk Model (PDM)* [755] there are multiple disks which are accessed by multiple processors. In this chapter, we refer to the SDM or the PDM as *external memory models (EMM).*

A large number of algorithms and data structures have been analysed and designed on these models, see the survey papers [754, 55].

The advantage of the EMM is their simplicity, which ease analysis and design of algorithms. The main reason why these models cannot easily be used directly to analyse and design algorithms for internal memory is the assumption that EMM algorithms can implement their own policy for replacing data in the faster (main) memory, whereas in a cache or TLB this is strictly under hardware control.

8.3.2 Cache Oblivious Model

Algorithms designed for the EMM need to know parameters of the memory hierarchy in order to tune performance for specific memories. *Cache oblivious algorithms*, analysed on the *cache oblivious model* [321], do not have any parameters that are tuned for a specific memory level. The model was introduced to consider a single level of cache and main memory. In this model the cache, called an *ideal cache*, holds M/B blocks each of B words, is fully-associative and uses an optimal offline replacement policy. The performance measures in this model are the number of cache misses and the number of instructions. The cache oblivious model and cache oblivious algorithms are discussed in detail in Chapter 9.

8.3.3 Models for Internal Memory

In this section we describe two models, the *Cache Memory Model* , which models cache and main memory, and the *Internal Memory Model* , which models cache, TLB and main memory.

Cache Memory Model (CMM). Several researchers have used a model where there is a random-access machine, consisting of a CPU and main memory, augmented with a cache [494, 489, 654, 621, 620, 619, 685, 543, 139]. The model has the following parameters:

N = the number of data items in the problems,
B = the number of data items in a cache block,
M = the number of data items in the cache,
a = the associativity of the cache,

and the performance measures in this model are:

- the number of cache misses,
- the number of instructions.

The cache parameters are derived directly from the discussion in Section 8.2. In this chapter we will generally refer to the number of blocks in the cache, M/B, rather than the number of data items that can be held in the cache. The model assumes that blocks are evicted from a cache set on a least recently used (LRU) basis.

The advantage of counting instructions and cache misses separately is that it allows the use of the coarse O-notation for simple operations and also to analyse the number of cache misses more carefully, if necessary.

The above model simplifies the architecture of real machines considerably. For example the model considers only one level of cache and there may be multiple caches. Another example is that we do not distinguish between a read or write to memory. As discussed earlier, in a write-through cache each write would access main memory or the next lower level cache.

Internal Memory Model (IMM). The CMM is useful for virtual caches, where a virtual address can be used to index the cache. However most caches are physical, which can only be indexed using a physical address, and, as discussed in Section 8.2, these caches require an address translation before checking for data. The IMM [619] extends the CMM to take account of virtual memory and the TLB by adding the following parameters:

B' = the number of data items in a page of memory,
T = the number translations held by the TLB,

and by adding the number of TLB misses to the performance measures. For the TLB, the model considers main memory as being partitioned into equal-sized and alignedpages of B' memory words each. A TLB holds address translation information for at most T pages. If the program accesses a memory location which belongs to (logical) page i, and the TLB holds the translation for page i, the contents of the TLB do not change. If the TLB does not hold the translation for page i, the translation for page i is brought into the TLB, and the translation for some other page is removed, again on a LRU basis.

As motivated by the discussion in Section 8.2.3, the model assumes that TLB misses and cache misses happen independently.

The model makes a number of assumptions regarding the parameter values to simplify the analyses. These assumptions normally hold in practice, as explained below (see [392] for further details). The first assumption is that $B, M/B, B'$ and T are all powers of two. The remaining assumptions are:

1. $B \leq B'$
   Data is transferred from secondary memory as a page, and since secondary memory has much higher latency than main memory a page is much larger than a block. Indeed we should assume that $B \ll B'$.
2. $T \leq M/B$
   A TLB is fully-associative. This makes the hardware realisation of a TLB much more complex than a cache with limited associativity. Furthermore, a TLB hit must be serviced very quickly. These two factors imply that a TLB must be much smaller than a cache, and we should assume that $T \ll M/B$.
3. $M \leq B'T$
   If $B'T < M$ then some portion of cache will always lead to a TLB miss, thus effectively wasting part of cache. However, $B'T > M$ makes sense and does occur in practice.
4. $T \leq B'$ and $B \leq M/B$
   These are technical assumptions, which are similar to the *tall cache* assumption of [321] and appear to hold in practice (strictly speaking, we make a 'tall cache' and 'short TLB' assumption).

Fig. 8.4 shows the main (physical) memory, virtual memory, page table, cache and TLB in a computer system. The page table holds translations for all pages in virtual memory. The TLB holds translations for a subset of the pages. The cache holds a subset of the blocks of data from main memory. Note that the cache may hold blocks of data from pages for which translations are *not* held in the TLB.

Again, this model is a simplification of real TLBs. In principle a TLB miss can be much more involved than a cache miss, requiring software intervention. Hence some architectures may provide hardware support for handling TLB misses. For example, on the Sun UltraSparc-II, a TLB miss can cost any of:

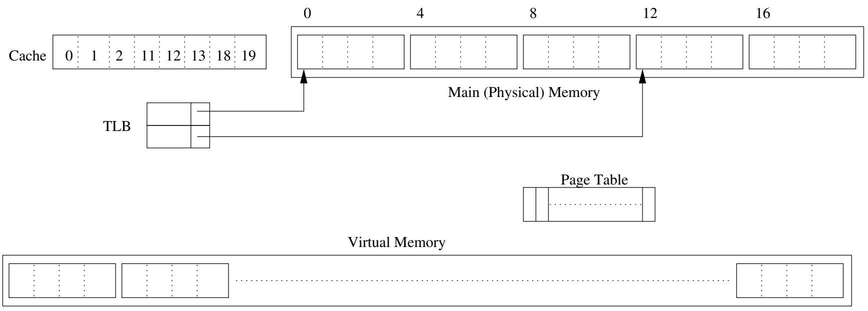


Fig. 8.4. The main memory, virtual memory, page table, cache and TLB. The TLB holds translations for pages containing the memory blocks $0, \ldots 3$ and $12, \ldots 15$. Note that the cache contains memory blocks $11, 18$ and 19, which are contained in pages for which translations are not held in the TLB.

(i) a L2 cache *hit* (2-3 CPU clock cycles) (ii) a memory access ($\approx$ 30-100 cycles) or (iii) a trap to a software miss handler (hundreds of cycles) [738, Chapter 6]. Another simplification is that TLBs almost always implement an approximation to LRU replacement, rather than a true LRU policy.

Classifying Cache Misses. In [396], cache misses are classified as *compulsory* misses, *capacity* misses and *conflict* misses. These are as follows:

- A compulsory miss occurs on the very first access to a memory block, since the block could not have been in cache.
- A capacity miss occurs on an access to a memory block that was previously evicted because the cache could not hold all the blocks being actively accessed by the CPU.
- If all a ways in a cache set are occupied then an access to a memory block that maps to that set will cause a block in the set to be evicted, even though there may be unused blocks in other cache sets. The next access to the evicted block will cause a conflict miss.

As an example, suppose we have an empty direct-mapped cache with M/B blocks, where each block holds B items and we have an array `DATA` with $N = 2M$ items. In the direct-mapped cache memory address x is mapped to cache block $(x \text{ div } B) \bmod (M/B)$. If we sequentially read all items in `DATA` then we have N/B compulsory misses. The first access to `DATA` causes B items to be loaded into a cache block, then after B accesses another B items are brought into the cache. If we sequentially read all items in `DATA` again, then we have N/B capacity misses. Now suppose the cache is empty and we read `DATA`[0] and then `DATA`[M], repeatedly $N/2$ times. `DATA`[0] and `DATA`[M] map to the same cache block, so we have 2 compulsory misses, for the first accesses to `DATA`[0] and `DATA`[M], and $N - 2$ conflict misses.

By definition a fully-associative cache does not have conflict misses. Similarly, conflict misses do not occur in the EMM.

8.3.4 Designing and Evaluating Internal Memory Algorithms

Algorithms designed in the CMM and IMM model aim to reduce the number of TLB and/or cache misses without an excessive increase in the number of instructions. We say that an algorithm is *cache-efficient* if it makes few cache misses. We say that it is *cache optimal* if the number of cache misses meet the asymptotic lower bound for I/Os in the EMM for that problem. We similarly define algorithms to be *TLB-efficient* or *TLB-optimal*.

Analyses in the CMM and IMM are usually backed up with experimental evaluations. Running times are used because:

– The relative miss penalty is much lower for caches or the TLB than for disks, so constant factors are important, we find that asymptotic analysis is not enough to determine the performance of algorithms.
– Cache misses, TLB misses and instruction counts do not tell us the running times of algorithms. The models simplify the architecture of real machines considerably. Experimental evaluations are used to validate the model.

Precise analysis in these models is often quite difficult. So an approximate analysis may be used, which again has to be validated with empirical techniques, such as measuring TLB and/or cache misses.

8.4 Algorithms for Internal Memory

In this section we first review general techniques to improve cache and TLB efficiency and then some algorithms for specific problems. Programs that exhibit good temporal and spatial locality generally have fewer cache and TLB misses. Most of the techniques and algorithms we discuss aim to improve temporal or spatial locality in a program. A few address the problem of reducing conflict misses, which may occur even if a program has good locality.

8.4.1 System Support

There are many hardware techniques for improving the cache performance of computer systems. These techniques aim to either reduce the cache miss rate or to reduce the penalty of a cache miss or to reduce the time for processing a cache hit. See [392, 378] for further details.

Several techniques have been used by compilers to improve cache performance [392]. Examples are:

– Pre-fetching is used to load data into cache before the data is needed, thus reducing CPU stalls.
– If multiple arrays are accessed in the same dimension with the same indices at the same time then this can cause conflict misses. By merging elements from each array into an individual structure, which resides in one cache block, the conflict misses are avoided. This improves spatial locality.

– If nested loops access data in a non-sequential order then spatial locality may be improved by reordering the nesting of the loops such that data is accessed sequentially.
– If the same array elements are accessed in multiple loops, then temporal locality can be improved and cache misses can be reduced by fusing the loops into one.
– In algorithms which manipulate multi-dimensional arrays, such as matrix multiplication, the technique of operating on tiles of the arrays, rather than rows or columns, can be used to reduce cache misses.

These techniques are discussed in more detail in Chapter 10. Most of these compiler techniques can also improve the TLB performance of algorithms.

Most of the above compiler techniques are for programs with regular data access patterns. More recently Chilimbi et al. [196, 195] suggest several compiler controlled techniques to improve the spatial and temporal locality of pointer-based data structures. They suggest *clustering* data, which places structure elements which are likely to be accessed in succession in the same cache block. *Colouring* data, which segregates heavily and infrequently accessed elements in non-conflicting cache regions. They also suggest *compressing* data, which reduces structure size or separates the active and inactive portions of structure elements. They also propose a cache-conscious `malloc` routine, a cache-conscious garbage collector and a run-time memory re-organiser. Garbage collection is also discussed in Chapter 11.

By considering the memory allocation algorithm used by the `gnu malloc` routine, Zhang and Martonosi [776] analyse the cache misses during the construction and traversal of linked-list and binary search tree data structures.

Romer et al. [635] have suggested using an online page size selection policy to improve TLB performance.

8.4.2 Algorithms

Matrix Algorithms. Several researchers have looked at the problems of matrix multiplication, matrix transposition, FFTs and permutations in cache memory [279, 125, 379, 178, 653, 779]. Venkatarman et al. [751] give a blocked algorithm to solve the all pairs shortest paths problem

Sorting. Nyberg et al. [588] describe the AlphaSort algorithm which sorts records stored on disk. In order to obtain good performance, they note the importance of minimising the cost of tasks completed in internal memory. They note that replacement-selection tree sort, a commonly used internal memory sorting technique, has poor cache behaviour, whereas Quicksort, due to its sequential memory accesses, has good cache behaviour. They suggest organising the nodes in the replacement-selection tree such that parent and children are in the same cache block, they report that this reduces cache misses by a factor of 2 to 3. Alphasort uses Quicksort to generate small sorted lists of data, and then merges these using replacement-selection tree sort. For large records, they state that sorting keys and pointers to records has better cache performance than sorting records.

LaMarca and Ladner [494, 493] evaluate the cache misses in Quicksort, Mergesort, Heapsort and least-significant bit (LSB) radix sort algorithms using cache simulations and analysis. To reduce cache misses in Heapsort, they suggest using d-ary heaps, where the root node occupies the last element of a cache block, rather than the normal binary heaps. For Mergesort they suggest using tiling and k-way merging. For Quicksort they suggest using a multi-partition approach and Sedgewick's technique of stopping the partitioning when the partition size is small [678]. However, in order to reduce capacity misses, they suggest insertion sorting these small partitions when they are first encountered, rather than in one single pass over the data after all small partitions have been created, as was proposed in [678]. They report that LSB radix sort has poor cache performance, even though they did not analyse an important source of conflict misses, due to concurrent accesses to multiple locations in the destination array. Their new algorithms are derived almost directly from EMM algorithms and hence reduced capacity misses, but not conflict misses. They show that their algorithms outperform non-memory tuned algorithms on their machine.

Xiao et al. [771] observe that the tiled and multi-way Mergesort algorithms presented in [494] can have in the worst-case a large number of conflict cache misses. They also observe that multi-way Mergesort can have a large number of TLB misses. They suggest padding subarrays that are to be merged in order to reduce cache misses in tiled Mergesort, and cache and TLB misses in multi-way Mergesort. Using cache and TLB simulations they show that the algorithms which use padding have fewer cache misses on several machines. They also show that these new Mergesort implementations outperform existing Mergesort implementations.

An approximate analysis of the cache misses in an $O(N)$ time distribution (bucket) sorting algorithm for uniformly random floating-point keys is given in [621]. Distribution sorting algorithms work as follows: In one pass the algorithm permutes N keys into k classes, such that all keys in class i are smaller than all keys in class $i+1$, for $i = 0, \ldots, k-1$. After one pass of the algorithm, the keys should have been permuted so that all elements of class i lie consecutively before all elements of class $i+1$, for $i = 0, \ldots, k-2$. Each class is sorted recursively and the recursion ends when a class is 'small', at which point the keys in the class may be sorted by say insertion sort. The analysis in [621] considers the cache misses in one pass of the algorithm as k varies and shows that for large k, which leads to fewer passes, the algorithm makes many cache conflict misses. The study shows the trade-offs in computation and memory access costs in a multi-pass algorithm and shows how to derive a multi-pass algorithm which out-performs a single pass algorithms, the various Mergesort and Quicksort algorithms described in [494] and the Heapsort described in [654]. An analysis of the cache misses in distribution sorting when the keys are independently and randomly drawn from a non-uniform distribution is given in [619].

Agarwal [12] notes the importance of reducing both cache and TLB misses in the context of sorting randomly distributed data. To achieve this in a bucket sort implementation, the number of buckets is chosen to be less than the number of TLB entries.

Jiménez-González et al. [434, 432] present two different algorithms for 32 and 64 bit integer keys, the Cache Conscious Radix sort and the Counting Split Radix sort respectively. Both algorithms are memory hierarchy conscious algorithms based on radix sort. They distribute keys such that the size of each class is smaller than the size of one of the levels of the cache hierarchy, the target cache level, and/or the memory that can be mapped by the TLB structure. They note that the count array used in the different steps of the algorithms must fit in the L1 cache to obtain an efficient cache-conscious implementation. They also state that the size of each class should not exceed the size of the L2 cache. For good TLB performance during distribution, they also note that the number of classes should not exceed the number of TLB entries. Their algorithms are *skew conscious.* That is, the algorithms recursively sort each class as many times as necessary or perform a sampling of the data at the beginning of the algorithm to obtain balanced classes. Each class is individually sorted by radix sort. A simple model for computing the number of bits of the key to sort on in order to obtain a good radix sort algorithm is proposed and analysed.

Rahman and Raman [620] present an extensive study of the design and implementation of integer sorting algorithms in the IMM. We discuss some of these results in Section 8.7.

Searching. Acharya et al. [3] note that in a *trie* [460], a search tree data structure, the nodes nearest the root are large (containing many keys and pointers) and those further away are increasing smaller. For large alphabets they suggest using an array which occupies a cache block for a small node. A bounded depth B-tree for a larger node. A hash table for a node which exceeds the maximum size imposed by the bounded depth B-tree. The B-trees and hash tables hold data in arrays. They also note that only one pointer to a child node, is taken out of a node, whereas several keys may be compared, so they suggest storing keys and pointers in separate arrays. For nodes with small alphabets they again suggest storing the keys and pointers in separate arrays, where the pointers are indexed using the characters in the alphabet. They experimentally evaluate their tries against non-memory tuned search trees and report significant speed improvements on several machines. Using cache simulations, they show that their data structures have significantly fewer cache misses than non-memory tuned search trees.

Rahman et al. [618] describe an implementation of very simple dynamic cache oblivious search trees. They show that these search trees out-perform B-trees. However, they also show that architecture-aware search trees which are cache and TLB optimal out-perform the cache oblivious search trees. They also discuss the problems associated with the design of dynamic architecture-aware search trees which are cache and TLB optimal.

Priority Queues and Heaps. Sanders [654] notes that most EMM priority queue data structure have high constant factors and this means that they would not perform well if adapted to the CMM. A new EMM data structure, the *sequence heap*, is described which has smaller constant factors in terms of the number of I/Os and space utilisation than other EMM priority queue data structures. The lower constant factors make sequence heaps suitable for the CMM. This data structure uses k-way merging so, before it is used in the CMM, the results from [543] are applied to select k appropriately for the cache parameters. On random 32-bit integer keys and 32-bit values and when the input is large, sequence heaps are shown to outperform implicit binary heaps and aligned 4-ary heaps on several different machine architectures.

Bojesen et al. [139] analyse the cache misses during heap construction on a fully-associative cache. They find that Floyd's method [306] of repeatedly merging small heaps to form larger heaps has poor cache performance and that Williams' method of repeated insertion [766] and Fadel et al.'s method of layer-wise construction [284] perform better. They give new algorithms using repeated insertions and repeated merging which have close to the optimal number of cache misses. They note that divide and conquer algorithms are good for hierarchical memory. They also note that by traversing a tree in depth-first order rather than breath-first order improves locality.

8.5 Cache Misses and Power Consumption

Power consumption is an important consideration in the design of computer systems, especially for mobile computing. Since caches occupy a large portion of the CPU chip, they are a good target for trying to reduce power consumption. Kaxiras et al. [453] note that a cache block has a flurry of activity when data is first brought into the cache block and then it has a period of inactivity, during which there can be power leakage. They discuss policies for switching cache blocks off when they hold data that is not likely to be reused.

Joseph et al. [440] report that power consumption on the Intel Pentium Pro increases as the data cache hit rate increases and it reaches a peak when the hit rate is approximately $80 - 85\%$. This is because as the cache hit rate increases the CPU performs more instructions per cycle which requires additional power. When considered as a trade-off between performance and power usage they report that the increased power consumption is cost effective.

This area needs further research, as possible improvements in power consumption is an additional motivation for improving the cache, and perhaps TLB, performance of algorithms.

8.6 Exploiting Other Memory Models: Advantages and Limitations

In this section we discuss some of the issues involved in exploiting techniques and algorithms from other memory models in order to obtain good performance in internal memory. Since there is a large literature on EMM algorithms, this seems like a reasonable starting point. However, due to the differing replacement policies, we find that EMM algorithms cannot always be applied directly to internal memory. We give some indications of how these techniques can be adapted for the CMM and IMM. In this context we consider problems associated with the commonly used techniques of *tiling* or *blocking* and *accessing k sequences* when applied in internal memory. We then consider why asymptotic miss analysis techniques, which follow from the analytical approach in say the EMM, may not be the most suitable for obtaining good performance in internal memory algorithms. We also discuss the use of algorithms from the EMM, cache oblivious memory model and a 3-level hierarchical memory model to reduce cache and TLB misses.

8.6.1 Tiling

EMM algorithms commonly use the idea of *tiling* or *blocking* in order to reduce the number of I/O operations in numeric algorithms, such as matrix multiplication. However applying this EMM approach directly to internal memory can give very poor cache behaviour due to the limited associativity of the cache.

In order to use an EMM algorithm which uses tiling with cache memory it is important to determine the optimal tile size which, given the matrix size and the cache and TLB parameters, minimises the number of cache and TLB misses. The optimal tile size for CMM algorithm may be considerably smaller than that appropriate for the EMM algorithm. This issue and its solutions are discussed in detail in Chapter 10.

8.6.2 Accessing k Sequences

Divide and conquer algorithms designed on the RAM model typically deal with two sets of data. For example Quicksort recursively partitions data into two sets, and Mergesort repeatedly merges two sorted lists. These techniques require $\Omega(\lg N)$ passes over the data and applying them directly to EMM algorithms can lead to an unnecessarily large number of capacity misses. To reduce the number of such misses, divide and conquer algorithms for the EMM deal with $k = O(M/B)$ sets of data, which reduces the number of passes over the data to $\Omega(\lg N/\lg k)$. For example, the analogue to Quicksort in the EMM is distribution sorting, which recursively partitions data into k sets, while an EMM Mergesort merges k sorted lists.

These EMM techniques imply that k sequences of data are accessed simultaneously and this can again cause problems in cache memory due to the limited associativity of the cache. Consider the very simple example of accessing just $k = 2$ sequences in a direct mapped cache. If the start of the two sequences map to the same cache block and there are round-robin accesses to the two sequences then, other than the first accesses to each sequence, every access will cause a cache conflict miss.

Mehlhorn and Sanders [543] analyse the number of cache misses in a set-associative cache when an adversary makes N accesses to k sequences. Their analysis assumes that the start of each sequence is uniformly and randomly distributed in the cache. They show that in order to have $O(N/B)$ cache misses, asymptotically the same as the number of misses that must be made in order just to read the data, the algorithm can use only $k = M/B^{1+1/a}$ sequences. This suggests that if EMM algorithmic techniques such as k-way merging or partitioning into k sets are used in the CMM, then the parameter k must be reduced by a factor of $B^{1/a}$.

Rahman and Raman [619] analyse the cache misses on a direct-mapped cache when distribution sort is applied to data independently drawn from a non-uniformly random distribution. Their analyses also applies to multiple sequence accesses and can be used to obtain tighter upper and lower bounds on k when the probability distribution is known.

8.6.3 Emulation of Algorithms from Other Memory Models

A large number of algorithms and data structures have been designed for the EMM, see the survey papers [754, 55], which could potentially be used on the CMM and IMM. However in the EMM block placement in fast memory is under the control of the algorithm whereas in the IMM it is hardware controlled. This can lead to conflict misses which would not have been considered in the EMM. Gannon and Jalby [324] show that cache conflict misses could be avoided by copying frequently accessed non-contiguous data into contiguous memory, this ensures that the non-contiguous data are mapped to their own cache block. Since then the technique has commonly been in use, for example, in [492] copying was used to reduce conflict misses in blocked matrix multiplication.

Using the copying technique suggested in [324], Sen and Chatterjee [685] give an emulation theorem that formalises the statement that an EMM algorithm and its analysis can be converted to an equivalent algorithm and analysis on the CMM. The same emulation theorem gives asymptotically the same number of cache and TLB misses [620].

An emulation theorem that allows an algorithm and analysis for a 3-level hierarchical memory model to be converted to an equivalent algorithm and analysis on the IMM is given in [618]. This result can be used to obtain simultaneous cache and TLB optimality by applying the emulation to optimal algorithms for 3-level hierarchical memory or to optimal cache oblivious algorithms.

8.6.4 Limitations of Asymptotic I/O Analysis

In the EMM it is generally assumed that the cost of I/Os is much greater than the cost of computation, hence the design of the algorithm is usually motivated by the need to minimise the number of I/Os. However in the CMM and the IMM the relative miss penalties are far smaller and we have to consider computation costs. We will consider the implications of this in the context of distribution sorting.

Using the analyses in [543, 619] and following the practice in the I/O model for selecting the number of classes k such that the number of I/Os are minimised would suggest that, on a direct-mapped cache, in one pass of distribution sorting, the algorithm should use $k = O(M/B^2)$ classes. However, in practice, a fast multi-pass distribution sorting algorithm designed for the CMM may use $O(M/B)$ classes [621]. A very brief explanation for this is demonstrated by the following example. On the Sun UltraSparc-II machine the cost of a L2 cache miss is ≈ 30 CPU cycles and there are ≈ 30 computations per key during one pass of the distribution sorting algorithm. An algorithm which switches from one to two passes has a minimum additional cost of N/B capacity misses, and 30 computations per key. The computation cost translates to roughly 1 cache miss per key. So, on this machine, it is only reasonable for the algorithm to switch from one to two passes, if the number of cache misses in the first pass is more than $N + 2N/B$ misses. The asymptotic analysis suggests the use of $k = O(M/B^2)$ classes in order to obtain $O(N/B)$ misses in each pass, but clearly, given the cache miss penalty and computation costs in this problem, this would have been non-optimal for the overall running time.

This example demonstrates that, even if conflict misses are accounted for, an algorithm designed for the EMM may not be directly applicable in internal memory as the asymptotic I/O analysis would not lead to the optimal parameter choices. The EMM algorithm could offer a good starting point but would need further analysis to obtain good performance.

8.6.5 Minimising Cache and TLB Misses

Algorithms designed for the EMM minimise the number of I/Os between two levels of memory, disk and main memory. IMM algorithms have to minimise misses between three memories. So an optimal EMM algorithm may not be optimal on the IMM.

8.7 Sorting Integers in Internal Memory

This section is a brief case study of how to tune a standard *random access machine (RAM)* algorithm for internal memory. We will see the tradeoff in improving performance first for the cache then for the cache and TLB. We will discuss the problems associated with making the algorithm optimal for both the cache and TLB. The initial techniques used give performance improvements by increasing the temporal locality of the algorithm. We will see how further performance improvements can be obtained by modifying the algorithm to improve spatial locality.

We consider the problem of sorting integers using *least-significant bit (LSB) radix sort* In this discussion, we assume that the keys are 32-bit integers drawn independently from a uniformly random distribution. This case study is based on results in [620]. For our discussion we assume the system is a 300MHz Sun UltraSparc-II machine, where the cache is direct mapped, $B = 16$, $M/B = 8192$, $T = 64$ and $B' = 2048$. Here B is the number of 32-bit integers in a cache block and B' is the number of 32-bit integers in a page.

In this section we will use the following default terminology. The term *key* will refer to a single digit, and the term *record* will denote the integer which was input to the sorting algorithm plus any associated information.

In LSB radix sorting we view w-bit integer keys as $\lceil w/r \rceil$ consecutive r-bit *digits*. The records are sorted in $\lceil w/r \rceil$ passes: in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$ we sort the records according to the i-th least-significant digit. LSB radix sort has an overall running time of $O(\lceil w/r \rceil (N + 2^r))$ for w-bit keys.

We now review one pass of LSB radix sort. There are a number of ways of implementing a single pass in $O(n + 2^r)$ time, the best one in practice being *counting sort* [215]. The array containing the records at the start of the pass is called the *source* array. The records into which the keys are sorted is called the *destination* array. In addition, one or more *count* arrays are used to keep auxiliary information. Counting sort comprises of a *count* phase, a *prefix sum* phase, and a *permute phase* [215, pp. 175–177]. During the count phase the algorithm counts the numbers of keys with the same *class*, where the class of a key is the value of the digit being sorted on in this pass. During the prefix sum phase the starting locations in the destination array are marked off for keys with the same class. During the permute phase, each record is moved to its new location, using the count array to determine the new location for the record. In the remaining part of this section we will generally focus on one pass of radix sorting.

8.7.1 Parameter Choices in the RAM Model

For a given values of N and w, the only parameter that can be varied in the algorithm is r. By increasing r we reduce the number of passes over the data. The RAM model assumes unit cost for arithmetic operations and for accesses to memory, so when selecting parameters the motivation is to reduce the number of passes. We select r such that w/r is minimised and that $k = 2^r = O(N)$. For large values of N the algorithm would select $r = 16$, such that there are two passes over the data.

8.7.2 Parameter Choices in the CMM

During the count phase, LSB radix sort makes the following memory accesses:

1. A sequential read access to the source array, to move to the next key to count.
2. One or more random read/write accesses to the count array(s) to increment the count values for one or more passes.

During the permute phase, the algorithm makes the following accesses:

1. A sequential read access to the source array, to find the next record to move.
2. A random read/write access to the count array, to find where to move the next record and to increment the count array location just read.
3. A random write to one of 2^r *active locations* in the destination array, to actually move the record.

Since many more random memory blocks are accessed at any one time during the permute phase than during the count phase, the permute phase leads to far more cache misses. Therefore, our parameter choices concentrate on obtaining a cache efficient permute phase. Using the analysis in [619, 543] we can determine that on a direct-mapped cache we should select $k = 2^r$ such that $k = O(M/B^2)$. For larger values of k the number of conflict misses in each pass would be asymptotically more than the number of compulsory misses. Smaller values of k would increase the number of passes and so unnecessarily increase the number of capacity misses.

On the Sun UltraSparc-II, when tuned for the CMM, the algorithm is about 5% faster than when tuned for the RAM model. However the algorithm does not out-perform a CMM tuned implementation of Quicksort.

8.7.3 Parameter Choices in the IMM

The *working set* of pages is the set of pages an algorithm accesses at a particular time. If the program makes random accesses to these pages, and the working set size is much larger than the size of the TLB, then the number of TLB misses will be large. During the count phase, the algorithm accesses the following *working* set of pages: (i) one active page in the source array and (ii) $W = \lceil 2^r/B' \rceil$ count array pages. During the permute phase, the algorithm accesses the following *working* set of pages: (i) one active page in the source array, (ii) $W = \lceil 2^r/B' \rceil + \min\{2^r, \lceil N/B' \rceil\}$ count and destination array pages. Since the working set of pages is larger for the permute phase and since it makes more memory accesses, this phase will have more TLB misses than the count phase. Thus, again our parameter choices concentrate on the permute phase.

We now heuristically analyse the permute phase to calculate the number of TLB misses for $r = 6, \ldots, 11$. For all these values, the count array fits into one page and we may assume that the count page and the current source page, once loaded, will never be evicted. We further simplify the process of accesses to the TLB and ignore disturbances caused when the source or one of the destination pointers crosses a page boundary (as these are transient). With these simplifications, TLB misses on accesses to the destination array may be modelled as uniform random access to a set of 2^r pages, using an LRU TLB of size $T - 2$. The probability of a TLB miss is then easily calculated to be $(2^r - (T - 2))/2^r$.

This suggests that choosing $r = 6$ on the Sun UltraSparc-II still gives a relatively low miss rate on average (miss probability 1/32), but choosing $r = 7$ is significantly worse (miss probability 1/2).

Experiments suggest that on random data, for $r = 1, \ldots, 5$ a single permute phase with radix r takes about the same time, as expected. Also, for $r = 7$ the permute time is—as expected—considerably (about 150%) slower than for $r \leq 5$. However, even for $r = 6$ it is about 25% slower than $r \leq 5$. This is probably because in practice T is effectively 61 or 62—it seems that the operating system reserves a few TLB entries for itself and locks them to prevent them from being evicted. Even using the simplistic estimate above, we should get a miss probability in the 1/13 to 1/16 range.

The choice $r = 5$ which guarantees good TLB performance turns out not to give the best performance on random data: it requires seven passes for sorting 32-bit data.

On the Sun UltraSparc-II, when tuned for the IMM, using $r = 6$, the algorithm is about 55% faster than CMM tuned Quicksort and LSB radix sort algorithms.

Note that for the algorithm to make an optimal $O(N/B)$ cache misses and $O(N/B')$ TLB misses in each pass requires that T is not too small, i.e., we need $\log T = \Theta(\log M/B)$. See [620] for a more detailed discussion.

8.7.4 Pre-sorting LSB (PLSB) Radix Sort

PLSB radix sort [620] is a variant of LSB radix sort which pre-sorts the keys in small groups to increase locality and hence improves cache and TLB performance. One pass of PLSB radix sort with radix r works in two stages. First we divide the input array of N keys into contiguous segments of $s \leq N$ keys each. Each segment is sorted using counting sort (a *local* sort) after which we sort the entire array using counting sort (a *global* sort). In each pass the time for sorting each of the $\lceil N/s \rceil$ local sorts is $O(s + 2^r)$ time and the time for the global sort is $O(N + 2^r)$, so the running time for one pass of PLSB radix sort is $O(N + 2^r N/s)$.

The intuition for the algorithm is that each local sort groups keys of the same class together and during the global sort we move sequences of keys to successive locations in the sorted array, thus reducing TLB and cache conflict misses between accesses to the destination array. The algorithm has good temporal locality during the local sorts and good spatial locality during the global sorts.

The segment size s is chosen such that the source and destination arrays for a local sort both fit in cache, and map to non-conflicting cache locations. The radix is chosen such that $s/2^r = O(B)$, this ensures that there are an optimal $O(N/B)$ cache misses and $O(N/B)$ TLB misses in each pass.

On the Sun UltraSparc-II, using $r = 11$ and $s = M/2$ the algorithm is twice as fast as CMM tuned Quicksort and LSB radix sort algorithms. The algorithm is also 30% faster than IMM tuned LSB radix sort.

Chapter 16 discusses applying local sorting techniques in the context of parallel sorting algorithms.

8.7.5 Experimental Results

Table 8.1 summaries the running times obtained on the UltraSparc-II when sorting 32-bit random integers using the various tuning techniques and parameter choices discussed above.

Table 8.1. Overall running times when sorting 32-bit unsigned integers using pre-sorting LSB radix sort with an 11 bit radix (PLSB 11); cache and TLB tuned LSB radix sort (LSB 6); cache tuned LSB radix sort (LSB 11); LSB radix sort tuned for the RAM model (LSB 16); cache tuned Quicksort.

Timings(sec)					
n	PLSB 11	LSB 6	LSB 11	LSB 16	Quick
1×10^6	0.47	0.64	0.90	0.92	0.70
2×10^6	0.92	1.28	1.86	1.94	1.50
4×10^6	1.82	2.56	3.86	4.08	3.24
8×10^6	3.64	5.09	7.68	7.90	6.89
16×10^6	7.85	10.22	15.23	15.99	14.65
32×10^6	15.66	20.45	31.71	33.49	31.96

9. Cache Oblivious Algorithms[*]

Piyush Kumar[**]

9.1 Introduction

The cache oblivious model is a simple and elegant model to design algorithms that perform well in hierarchical memory models ubiquitous on current systems. This model was first formulated in [321] and has since been a topic of intense research. Analyzing and designing algorithms and data structures in this model involves not only an asymptotic analysis of the number of steps executed in terms of the input size, but also the movement of data optimally among the different levels of the memory hierarchy. This chapter is aimed as an introduction to the "ideal-cache" model of [321] and techniques used to design cache oblivious algorithms. The chapter also presents some experimental insights and results.

A dream machine would be fast and would never run out of memory. Since an infinite sized memory was never built, one has to settle for various trade-offs in speed, size and cost. In both the past and the present, hardware suppliers seem to have agreed on the fact that these parameters are well optimized by building what is called a memory hierarchy (see Fig. 9.1, Chapter 1). Memory hierarchies optimize the three factors mentioned above by being cheap to build, trying to be as fast as the fastest memory present in the hierarchy and being almost as cheap as the slowest level of memory. The hierarchy inherently makes use of the assumption that the access pattern of the memory has *locality* in it and can be exploited to speed up the accesses.

The locality in memory access is often categorized into two different types, code reusing recently accessed locations (*temporal*) and code referencing data items that are close to recently accessed data items (*spatial*) [392]. Caches use both temporal and spatial locality to improve speed. Surprisingly many things can be categorized as caches, for example, registers, L1, L2, TLB, Memory, Disk, Tape etc. (Chapter 1). The whole memory hierarchy can be viewed as *levels* of caches, each transferring data to its adjacent levels in atomic units called *blocks*. When data that is needed by a process is in the cache, a *cache hit* occurs. A *cache miss* occurs when data can not be supplied. Cache misses can be very costly in terms of speed and can be reduced by designing algorithms that use locality of memory access.

[*] An updated version of the chapter can be found at the webpage `http://www.compgeom.com/co-chap/`

[**] Part of this work was done while the author was visiting MPI Saarbrücken. The author is partially supported by NSF (CCR-9732220, CCR-0098172) and by the grant from Sandia National Labs.

The *Random Access Model* (RAM) in which we do analysis of algorithms today does not take into account differences in speeds of random access of memory depending upon the locality of access [215]. Although there exist models which can deal with multi-level memory hierarchies, they are quite complicated to use [13, 14, 15, 33, 34, 406, 664, 685]. It seems there is a trade-off between the accuracy of the model and the ease of use. Most algorithmic work has been done in the RAM model which models a 'flat' memory with uniform access times. The external memory model (with which the reader is assumed to be familiar, see Chapter 1) is a two level memory model, in the context of memory and disk. The 'ideal' cache oblivious model is a step towards simplifying the analysis of algorithms in light of the fact that local accesses in memory are cheaper than non-local ones in the whole memory hierarchy (and not just two-levels of memory). It helps take into account the whole memory hierarchy and the speed differences hidden therein.

In the external memory model, algorithm design is focussed on a particular level of memory (usually the disk) which is the bottleneck for the running time of the algorithm in practice. Recall (From Chapter 1), that in the external memory model, processing works almost as in the RAM model, except that there are only M words of internal memory that can be accessed quickly. The remaining memory can only be accessed using I/Os that move B contiguous words between external and internal memory. The I/O complexity of an algorithm amounts to counting the number of I/Os needed.

The cache oblivious model was proposed in [321] and since then has been used in more than 30 papers already. It is becoming popular among researchers in external memory algorithms, parallel algorithms, data structures and other related fields. This model was born out of the necessity to capture the hierarchical nature of memory organization. (For instance the Intel Itanium has 7 levels in its memory hierarchy, see Chapter 1). Although there have been other attempts to capture this hierarchical information the cache oblivious model seems to be one of the most simple and elegant ones. The cache oblivious model is a two level model (like the external memory model [17] that has been used in the other chapters so far) but with the assumption that the parameters M,B are unknown to the algorithm (see Fig. 9.1). It can work efficiently on most machines with multi-level cache hierarchies. Note that in the present volume, all external memory algorithms that have been dealt with yet, need the programmer/user to specify M,B.

This chapter is intended as an introduction to the design and analysis of cache oblivious algorithms, both in theory and practice.

Chapter Outline. We introduce the cache oblivious model in Section 9.2. In Section 9.3 we elaborate some commonly used design tools that are used to design cache oblivious algorithms. In Section 9.4 we choose matrix transposition as an example to learn the practical issues in cache oblivious algorithm design. We study the cache oblivious analysis of Strassen's algorithm in Section 9.5. Section 9.6 discusses a method to speed up searching in balanced binary search trees both in theory and practice. In Section 9.7, a theoretically optimal, randomized cache oblivious sorting algorithm along with the running times of an implementation is presented. In Section 9.8 we enumerate some practicalities not caught by the model. Section 9.9 presents some of the best known bounds of other cache oblivious algorithms. Finally we conclude the chapter by presenting some related open problems in Section 9.10.

9.2 The Model

The ideal cache oblivious memory model is a two level memory model. We will assume that the faster level has size M and the slower level always transfers B consecutive words of data to the faster level. These two levels could represent the memory and the disk, memory and the cache, or any two consecutive levels of the memory hierarchy (see Fig. 9.1). In this chapter, M and B can be assumed to be the sizes of any two consecutive levels of the memory hierarchy subject to some assumptions about them (For instance the inclusion property which we will see soon). We will assume that the processor can access the faster level of memory which has size M. If the processor references something from the second level of memory, an I/O fault occurs and B words are fetched into the faster level of the memory. We will refer to a *block* as the minimum unit that can be present or absent from a level in the two level memory hierarchy. We will use B to denote the size of a *block* as in the external memory model. If the faster level of the memory is full (i.e. M is full), a block gets evicted to make space.

The ideal cache oblivious memory model enables us to reason about a two level memory model like the external memory model but prove results about a multi-level memory model. Compared with the external memory model it seems surprising that without any memory specific parametrization, or in other words, without specifying the parameters M, B, an algorithm can be efficient for the whole memory hierarchy, nevertheless it is possible. The model is built upon some basic assumptions which we enumerate next.

Assumptions. The following four assumptions are key to the model.

Optimal replacement: The *replacement policy* refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed furthest in the future. This is known as the *optimal off-line replacement* strategy.

Two levels of memory: There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the *inclusion property* which says that data cannot be present at level i unless it is present at level $i + 1$. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level $i + 1$.

Full associativity: When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.

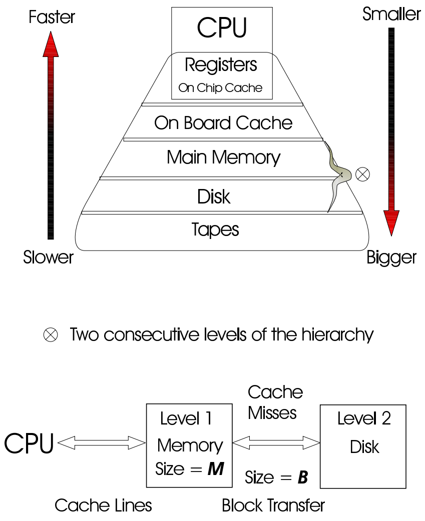


Fig. 9.1. The ideal cache oblivious model

Automatic replacement: When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

We will now examine each of the assumptions individually. First we consider the optimal replacement policy. The most commonly used replacement policy is LRU (*least recently used*). In [321] the following lemma, whose proof is omitted here, was proved using a result of [698].

Lemma 9.1 ([321]). *An algorithm that causes $Q^*(n, M, B)$ cache misses on a problem of size n using a (M, B)-ideal cache incurs $Q(n, M, B) \leq 2Q^*(n, \frac{M}{2}, B)$ cache misses on a (M, B) cache that uses LRU or FIFO replacement. This is only true for algorithms which follow a regularity condition.*

An algorithm whose cache complexity satisfies the condition $Q(n, M, B) \leq O(Q(n, 2M, B))$ is called *regular* (All algorithms presented in this chapter are regular). Intuitively, algorithms that slow down by a constant factor when memory (M) is reduced to half, are called regular. It immediately follows from the above lemma that if an algorithm whose number of cache misses satisfies the regularity condition does $Q(n, M, B)$ cache misses with optimal replacement then this algorithm would make $\Theta(Q(n, M, B))$ cache misses on a cache with LRU or FIFO replacement.

The automatic replacement and full associativity assumption can be implemented in software by using LRU implementation based on hashing. It was shown in [321] that a fully associative LRU replacement policy can be implemented in $O(1)$ expected time using $O(\frac{M}{B})$ records of size $O(B)$ in ordinary memory. Note that the above description about the cache oblivious model proves that any optimal cache oblivious algorithm can also be optimally implemented in the external memory model.

We now turn our attention to multi-level ideal caches. We assume that all the levels of this cache hierarchy follow the inclusion property and are managed by an optimal replacement strategy. Thus on each level, an optimal cache oblivious algorithm will incur an asymptotically optimal number of cache misses. From Lemma 9.1, this becomes true for cache hierarchies maintained by LRU and FIFO replacement strategies.

Apart from not knowing the values of M, B explicitly, some cache oblivious algorithms (for example optimal sorting algorithms) require a *tall cache* assumption. The tall cache assumption states that $M = \Omega(B^2)$ which is usually true in practice. Its notable that regular optimal cache oblivious algorithms are also optimal in SUMH [33] and HMM [13] models. Recently, compiler support for cache oblivious type algorithms have also been looked into [767, 773].

9.3 Algorithm Design Tools

In cache oblivious algorithm design some algorithm design techniques are used ubiquitously. One of them is a *scan* of an array which is laid out in contiguous memory. Irrespective of B, a scan takes at most $1 + \lceil \frac{N}{B} \rceil$ I/Os. The argument is trivial and very similar to the external memory scan algorithm Chapter 3. The difference is that in the cache oblivious setting the buffer of size B is not explicitly maintained in memory. In the assumptions of the model, B is the size of the data that is always fetched from level 2 memory to level 1 memory. The scan does not touch the level 2 memory until its ready to evict the last loaded buffer of size B already in level 1. Hence, the total number of times the scan algorithm will force the CPU to bring buffers from the level 2 memory to level 1 memory is upper bounded by $1 + \lceil \frac{N}{B} \rceil$.

Exercise 1 *Convince yourself that indeed a scan of an array will at most load $O(\frac{N}{B})$ buffers from the level 2. Refine your analysis to $1 + \lceil \frac{N}{B} \rceil$ buffer loads.*

9.3.1 Main Tool: Divide and Conquer

Chances are that the reader is already an expert in divide and conquer algorithms. This paradigm of algorithm design is used heavily in both parallel and external memory algorithms. Its not surprising that cache oblivious algorithms make heavy use of this paradigm and many seemingly simple algorithms that were based on this paradigm are already cache oblivious!

For instance, Strassen's matrix multiplication, quicksort, mergesort, closest pair [215], convex hulls [39], median selection [215] are all algorithms that are cache oblivious, though not all of them are optimal in this model. This means that they might be cache oblivious but can be modified to make fewer cache misses than they do in the current form. In the section on matrix multiplication we will see that Strassen's matrix multiplication algorithm is already optimal in the cache oblivious sense.

Why does divide and conquer help in general for cache oblivious algorithms? Divide and conquer algorithms split the instance of the problem to be solved into several subproblems such that each of the subproblems can be solved independentally. Since the algorithm recurses on the subproblems, at some point of time, the subproblems fit inside M and subsequent recursion, fits the subproblems into B. For instance let us analyze the average number of cache misses in a randomized quicksort algorithm. This algorithm is quite cache friendly if not cache optimal and is described in Section 8.3 of [215].

Lemma 9.2. *The expected number of cache misses incurred by randomized version of quicksort is* $O\left(\frac{N}{B}\log_2(\frac{N}{B})\right)$.

Proof. Choosing a random pivot makes at most one cache miss. Splitting the input set into two output sets, such that the elements of one are all less than the pivot and the other greater than or equal to the pivot makes at most $O(1+\frac{N}{B})$ cache misses by Exercise 1.

As soon as the size of the recursion fits into B, there are no more cache misses for that subproblem ($Q(B) = O(1)$). Hence the average cache complexity of a randomized quicksort algorithm can be given by the following recurrence (which is very similar to the average case analysis presented in [215]).

$$Q(N) = \frac{1}{N}\left[\sum_{i=1..\frac{N-1}{B}} (Q(i) + Q(N-i)) + \left(1 + \left\lceil \frac{N}{B} \right\rceil\right)\right]$$

which solves to $O(\frac{N}{B}\log_2\frac{N}{B})$.

Exercise 2 *Solve the recurrence 9.1.*

Hint 1 *First simplify the recurrence to*

$$Q(N) = \frac{2}{N}\left[\sum_{i=1..\frac{N-1}{B}} Q(i) + \Theta\left(1 + \frac{N}{B}\right)\right]$$

and then you can take help from [353].

Exercise 3 *Prove that mergesort also does* $O\left(\frac{N}{B}\log_2\frac{N}{B}\right)$ *cache misses.*

As we said earlier, the number of cache misses randomized quicksort makes is not optimal. The sorting lower bound in the cache oblivious model is also the same as the external memory model (see Chapter 3). We will see later in the chapter a sorting algorithm that does sorting in $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ cache misses.

We will now analyze another cache oblivious algorithm that is also optimal, the *median finding* as given in [215] (Section 10.3). The recurrence for finding the median of N entities is given by:

$$Q(N) = Q\left(\frac{N}{5}\right) + Q\left(\frac{7N}{10} + 6\right) + O\left(1 + \frac{N}{B}\right)$$

Solving which we get, $Q(N) = O(1 + \frac{N}{B})$. Note that again $Q(B) = O(1)$ so the recursion stops adding cost to cache misses when the instance of the problem becomes smaller than B.

9.4 Matrix Transposition

Matrix transposition is a fundamental operation in linear algebra and in fast fourier transforms and has applications in numerical analysis, image processing and graphics. The simplest hack for transposing a $N \times N$ square matrix in C++ could be:

```
for (i = 0; i < N; i++)
   for (j = i+1; j < N; j++)
       swap(A[i][j], A[j][i])
```

In C++ matrices are stored in "row-major" order, i.e. the rightmost dimension varies the fastest (Chapter 10). In the above case, the number of cache misses the code could do is $O(N^2)$. The optimal cache oblivious matrix transposition makes $O(1 + \frac{N^2}{B})$ cache misses. Before we go into the divide and conquer based algorithm for matrix transposition that is cache oblivious, let us see some experimental results. (see Fig. 9.2). The figure shows the running times of a blocked cache oblivious implementation, we stop the recursion when the problem size becomes less than a certain block size and then use the simple for loop implementation inside the block. In this experiment the block sizes we chose were 32 bytes and 16 kilobytes. Note that using different block sizes has little effect on the running time. This experiment was done on Windows NT running on 1Ghz/512Mb RAM notebook. The code was compiled with g++ on cygwin.

Here is the C/C++ code for cache oblivious matrix transposition. The following code takes as input a submatrix given by $(x, y) - (x + delx, y + dely)$ in the input matrix I and transposes it to the output matrix O. `ElementType`[1] can be any element type , for instance `long`.

[1] In all our experiments, `ElementType` was set to `long`.

```
void transpose(int x, int delx, int y, int dely,
                ElementType I[N][P], ElementType O[P][N]){

      // Base Case of recursion
      // Should be tailored for specific machines if one wants
      // this code to perform better.
      if((delx == 1) && (dely == 1)) {
              O[y][x] = I[x][y];
              return;
      }

      // Divide the transposition into two sub transposition
      // problems, depending upon which side of the matrix is
      // bigger.
      if(delx >= dely){
              int xmid = delx / 2;
              transpose(x,xmid,y,dely,I,O);
              transpose(x+xmid,delx-xmid,y,dely,I,O);
              return;
      }

      // Similarly cut from ymid into two subproblems
      ...
}
```

The code works by divide and conquer, dividing the bigger side of the matrix in the middle and recursing.

Exercise 4 *Prove that on an $N \times N$ input matrix, the above code causes at most $O(1 + \frac{N^2}{B})$ cache misses.*

Hint 2 *Let the input be a matrix of $N \times P$ size. There are three cases:*
Case I: $\max\{N, P\} \leq \alpha B$ *In this case,*

$$Q(N, P) \leq \frac{NP}{B} + O(1)$$

Case II: $N \leq \alpha B < P$ *In this case,*

$$Q(N, P) \leq \begin{cases} O(1 + N) & \text{if } \frac{\alpha B}{2} \leq P \leq \alpha B \\ 2Q(N, P/2) + O(1) & N \leq \alpha B < P \end{cases}$$

Case III: $P \leq \alpha B < N$ *Analogous to Case II.*
Case IV: $\min\{N, P\} \geq \alpha B$

$$Q(N, P) \leq \begin{cases} O(N + P + \frac{NP}{B}) & \text{if } \frac{\alpha B}{2} \leq N, P \leq \alpha B \\ 2Q(N, P/2) + O(1) & P \geq N \\ 2Q(N/2, P) + O(1) & N \geq P \end{cases}$$

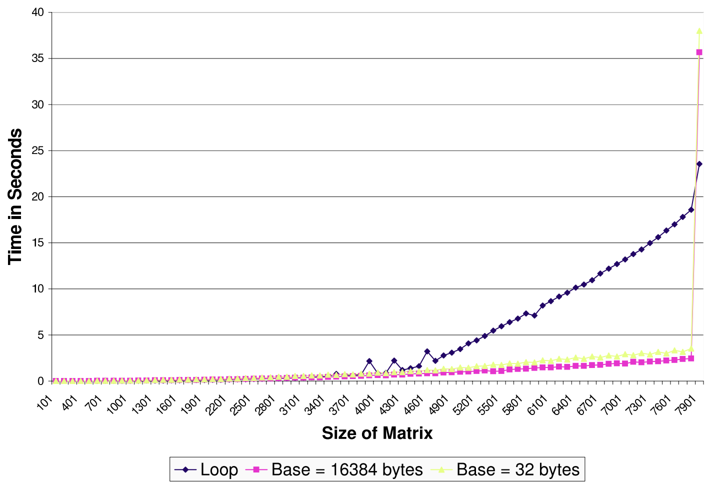


Fig. 9.2. The graph compares a simple for loop implementation with a blocked cache oblivious implementation of matrix transposition.

The above recurrence solves to $Q(N,P) = O(1 + \frac{NP}{B})$.

There is a simpler way to visualize the above mess. Once the recursion makes the matrix small enough such that $\max(N,P) \leq \alpha B \leq \beta\sqrt{M}$ (here β is a suitable constant), or such that the submatrix (or the block) we need to transpose fits in memory, the number of I/O faults is equal to the scan of the elements in the submatrix. A packing argument of these not so small submatrices (blocks) in the large input matrix shows that we do not do too many I/O faults compared to a linear scan of all the elements.
Remark: Fig. 9.2 shows the effect of using blocked cache oblivious algorithm for matrix transposition. Note that in this case, the simple for loop algorithm is almost always outperformed. This comparison is not really fair. The cache oblivious algorithm gets to use blocking whereas the naive for loop moves one element at a time. A careful implementation of a blocked version of the simple for loop might beat the blocked cache oblivious transposition algorithm in practice. (see the timings of Algorithm 2 and 5 in [178]). The same remark also applies to matrix multiplication. (Fig. 9.3)

9.5 Matrix Multiplication

Matrix multiplication is one of the most studied computational problems: We are given two matrices of $m \times n$ and $n \times p$ and we want to compute the product matrix of $m \times p$ size. In this section we will use $n = m = N$ although the results can easily be extended to the case when they are not equal. Thus, we are given two $N \times N$ matrices $x = (x_{i,j})$, $y = (y_{i,j})$, and we wish to compute their product z, i.e. there are N^2 outputs where the (i,j)'th output is

$$z_{i,j} = \sum_{k=1}^{N} x_{i,k} \cdot y_{k,j}$$

In 1969' Strassen surprised the world by showing an upper bound of $O(N^{\log_2 7})$ using a divide and conquer algorithm [708]. This bound was later improved and the best upper bound today is $O(N^{2.376})$ [209]. We will use Strassen's algorithm as given in [215] for the cache oblivious analysis.

The algorithm breaks the three matrices x, y, z into four submatrices of size $\frac{N}{2} \times \frac{N}{2}$, rewriting the equation $z = xy$ as:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \times \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Now we have 8 subproblems (matrix multiplications) of size $\frac{N}{2} \times \frac{N}{2}$ and four additions (which will lead us to a $O(N^3)$ algorithm). This can be done more cleverly by only solving 7 subproblems and doing more additions and subtractions but still keeping the number of additions and subtraction a constant (see [215] for more details). The recurrence for the internal work now becomes:

$$T(N) = 7T\left(\frac{N}{2}\right) + \Theta(N^2)$$

which solves to $T(N) = O(N^{\lg 7}) = O(N^{2.81})$. The recurrence for the cache faults is:

$$Q(N) \le \begin{cases} O(1 + N + \frac{N^2}{B}) & \text{if } N^2 \le \alpha M \\ 7Q(\frac{N}{2}) + O(1 + \frac{N^2}{B}) & \text{otherwise.} \end{cases} \tag{9.1}$$

which solves to $Q(N) \le O\left(N + \frac{N^2}{B} + \frac{N^{\lg 7}}{B\sqrt{M}}\right)$.

We also implemented a blocked cache oblivious matrix multiplication routine that works very similar to the transposition routine in the previous section. It breaks the largest of the three dimensions of the matrices to be multiplied (divides it by 2) and recurses till the block size is reached. Note that this algorithm is not optimal and does $O(1+\frac{N^2}{B}+\frac{N^3}{B\sqrt{M}})$ cache misses. Experimental results for the simpler matrix multiplication algorithm we implemented are reported in Fig. 9.3. In our experiments, we compared a blocked cache oblivious matrix multiplication implementation with simple for loop based matrix multiplication. This experiment was done on an dual processor Intel Itanium with 2GB RAM. Only one processor was being used. Note that on this platform, the blocked implementation consistently outperforms the simple loop code. The base case has little effect for large size matrices. An excellent reference for practical matrix multiplication results is [279].

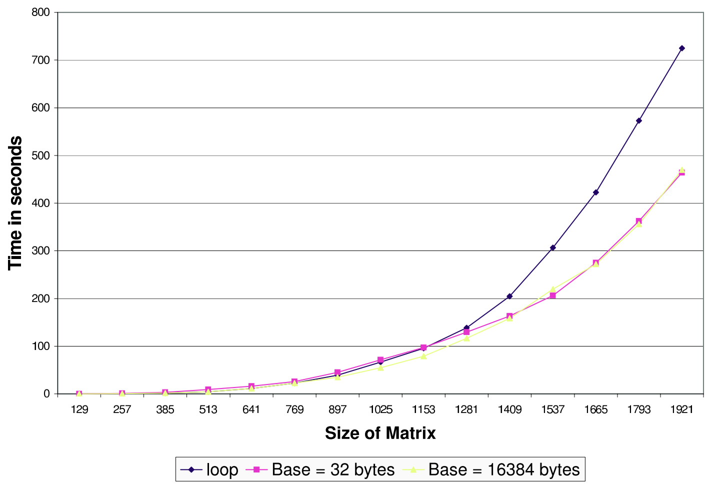


Fig. 9.3. Blocked cache oblivious matrix multiplication experiment.

Exercise 5 *Implement and analyze a matrix multiplication routine for $N \times N$ matrices.*

9.6 Searching Using Van Emde Boas Layout

In this section we report a method to speed up simple binary searches on a balanced binary tree. This method could be used to optimize or speed up any kind of search on a tree as long as the tree is static and balanced. It is easy to code, uses the fact that the memory consists of a cache hierarchy, and could be exploited to speed up tree based search structures on most current machines. Experimental results show that this method could speed up searches by a factor of 5 or more in certain cases!

It turns out that a balanced binary tree has a very simple layout that is cache-oblivious. By layout here, we mean the mapping of the nodes of a binary tree to the indices of an array where the nodes are actually stored. The nodes should be stored in the bottom array in the order shown for searches to be fast and use the cache hierarchy.

Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. Suppose the tree has N items and has height $h = \log N + 1$. Split the tree in the middle, at height $h/2$. This breaks the tree into a top recursive subtree of height $\lfloor h/2 \rfloor$ and several bottom subtrees $B_1, B_2, ..., B_k$ of height $\lceil h/2 \rceil$. There are $\sqrt{N}$ bottom recursive subtrees, each of size $\sqrt{N}$. The top subtree occupies the top part in the array of allocated nodes, and then the B_i's are laid out. Every subtree is recursively laid out.

Another way to see the algorithm is to run a breadth first search on the top node of the tree and run it till $\sqrt{N}$ nodes are in the BFS, see Fig. 9.4. The figure shows the run of the algorithm for the first BFS when the tree size is $\sqrt{N}$. Then the tree consists of the part that is covered by the BFS and trees hanging out. BFS can now be recursively run on each tree, including the covered part. Note that in the second level of recursion, the tree size is $\sqrt{N}$ and the BFS will cover only $N^{\frac{1}{4}}$ nodes since the same algorithm is run on each subtree of $\sqrt{N}$. The main idea behind the algorithm is to store recursive sub-trees in contiguous blocks of memory.

Lets now try to analyze the number of cache misses when a search is performed. We can conceptually stop the recursion at the level of detail where the size of the subtrees has size $\leq B$. Since these subtrees are stored contiguously, they at most fit in two blocks. (A block can not span three blocks of memory when stored). The height of these subtrees is $\log B$. A search path from root to leaf crosses $O\left(\frac{\log N}{\log B}\right) = O(\log_B N)$ subtrees. So the total number of cache misses is bounded by $O(\log_B N)$.

Exercise 6 *Show that the Van Emde Boas layout can be at most a constant factor of* 4 *away from an optimal layout (which knows the parameter* B*). Show that the constant* 4 *can be reduced to* 2 *for average case queries. Get a better constant factor than* 2 *in average case.*

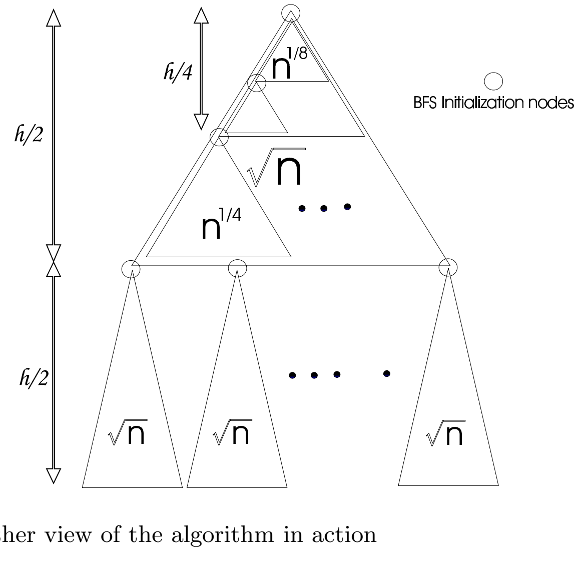


Fig. 9.4. Another view of the algorithm in action

9.6.1 Experiments

We did a very simple experiment to see how in real life, this kind of layout would help. A vector was sorted and a binary search tree was built on it. A query vector was generated with random numbers and searched on this BST which was laid out in pre-order. Why we chose pre-order compared to random layout was because most people code a BST in either pre/post/in-order compared to randomly laying it (Which incidentally is very bad for cache health).

Once this query was done, we laid the BST using Van Emde Boas Layout and gave it the same query vector. Before timing both trees, we made sure that both had enough queries to begin with otherwise, the order of timing could also effect the search times. (Because the one that is searched last, has some help from the cache). The code written for this experimentation is below 300 lines. The experiment reported in Fig. 9.5 were done on a Itanium dual processor system with 2GB RAM. (Only one processor was being used)

Currently the code copies the entire array in which the tree exists into another array when it makes the tree cache oblivious. This could also be done in the same array though that would have complicated the implementation a bit. One way to do this is to maintain pointers to and from the array of nodes to the tree structure, and swap nodes instead of copying them into a new array. Another way could be to use a permutation table. We chose to copy the whole tree into a new array just because this seemed to be the simplest way to test the speed up given by cache oblivious layouts. For more detailed experimental results on comparing searching in cache aware and cache oblivious search trees, the reader is referred to [490]. There is a big difference between the graphs reported here for searching and in [490]. One of the reasons might be that the size of the nodes were fixed to be 4 bytes in [490] whereas the experiments reported here use bigger size nodes.

9.7 Sorting

Sorting is a fundamental problem in computing. Sorting very large data sets is a key routine in many external memory applications. We have already seen how to do optimal sorting in the external memory model. In this section we outline some theory and experimentation related to sorting in the cache oblivious model. Some excellent references for reading more on the influence of caches on the performance of sorting are [494, 588], Chapter 16 and Chapter 8.

9.7.1 Randomized Distribution Sorting

There are two optimal sorting algorithms known, funnel sort and distribution sort. Funnel sort is derived from merge sort and distribution sort is a

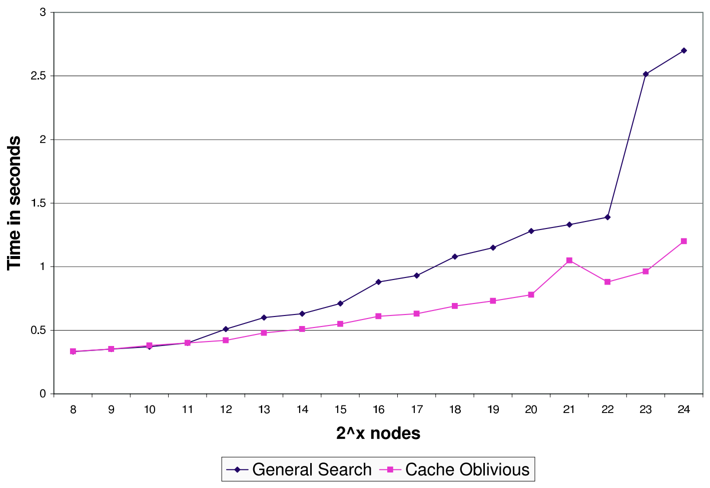


Fig. 9.5. Comparison of Van Emde Boas searches with pre-order searches on a balanced binary tree. Similar to the last experiment, this experiment was performed on a Itanium with 48 byte node sizes.

generalization of quick sort. We will see the analysis and implementation of a variant of distribution sort that is optimal and easier to implement than the original distribution sort of [321] (The original algorithm uses median finding and is deterministic). The algorithm for randomized distribution sort is presented in Algorithm 9.7.1.

Algorithm 9.7.1 procedure `Sort`(A)

```
Require: An input array $A$ of Size $N$
1: Partition A into $\sqrt{N}$ sub arrays of size $\sqrt{N}$. Recursively sort each subarray.
2: $R \leftarrow$ ComputeSplitters$(A, \sqrt{N})$
3: Sort$(R)$ recursively. $R = \{r_0, r_1, ..., r_{\sqrt{N}}\}$.
4: Calculate counts $c_i$ such that $c_i = |\{x \mid x \in A$ and $r_i \leq x < r_{i+1}\}|$
5: $c_{1+\sqrt{N}} = |A| - \Sigma_i c_i$.
6: Distribute $A$ into buckets $B_0, B_1, ...B_{1+\sqrt{N}}$ where last element of each bucket
   is $r_i$ except the last bucket. For the last bucket, the last element is maximum
   element of $A$. Note that $|B_i| = c_i$.
7: Recursively sort each $B_i$
8: Copy sorted buckets back to $A$.
```

The sorting procedure assumes the presence of three extra functions which are cache oblivious, choosing a random sample in Step 2, the counting in Step 4 and the distribution in Step 6. The random sampling step in the algorithm is used to determine splitters for the buckets, so we are in fact looking for splitters.

The function `ComputeSplitters`(X, k) in step 2 takes as input an array X and the number of splitters it needs to return as output. A very similar computation is needed in Sample Sort [131] to select splitters. We will use the same algorithm here. What we need is a random sample such that the c_i's do not vary too much from $\sqrt{N}$ with high probability. To do this, draw a random sample of $\beta\sqrt{N}$ from A, sort the sample and then pick each βth element from the sorted sample. In [131], β is referred to as the *oversampling ratio*, the more the β the greater the probability that the c_i's concentrate around $\sqrt{N}$.

The distribution and counting phases are quite analogous. Here is how we do distribution:

Algorithm 9.7.2 procedure `Distribute(int i, int j, int m)`

```
1: if $m = 1$ then
2:    CopyElements$(i, j)$
3: else
4:    Distribute$(i, j, m/2)$
5:    Distribute$(i + m/2, j, m/2)$
6:    Distribute$(i, j + m/2, m/2)$
7:    Distribute$(i + m/2, j + m/2, m/2)$
8: end if
```

In actual implementation one would also have to take care if m is odd or even and appropriately apply floor or ceiling operations for the above algorithm to work. In our code, we treated it by cases, when m was odd and when m was even, the rounding was different. Anyway, this is a minor detail and can be easily figured out. In the base case `CopyElements`(i, j) copies all elements from subarray i that belong to bucket j. If the function `CopyElements`(i, j) is replaced by `CountElements`(i, j) in `Distibute()`, the counting that is needed in Step 4 of Algorithm 9.7.1 can be performed. `CountElements()` increments the c_j values depending upon how many elements of subarray i lie between r_j and r_{j+1}. Since these subarrays are sorted, this only requires a linear scan.

Before we go to the analysis, lets see how the implementation works in practice. We report here two experiments where we compare C++ STL sort with our implementation. In our implementation we found out that as we recurse more, the performance of our sorting algorithm deteriorates. In the first graph in Fig. 9.6, Series 1 is the STL Sort and Series 2 is our sorting implementation. The recursion is only 1 level, i.e. after the problem size becomes $\sqrt{N}$ STL Sort is used. In Fig. 9.7 two levels of recursion are used. Note that doing more levels of recursion degrades performance. At least for 1 level of recursion, the randomized distribution sort is quite competitive and since these experiments are all in-memory tests, it might be possible to beat STL Sort when the data size does not fit in memory but we could not experiment with such large data sizes.

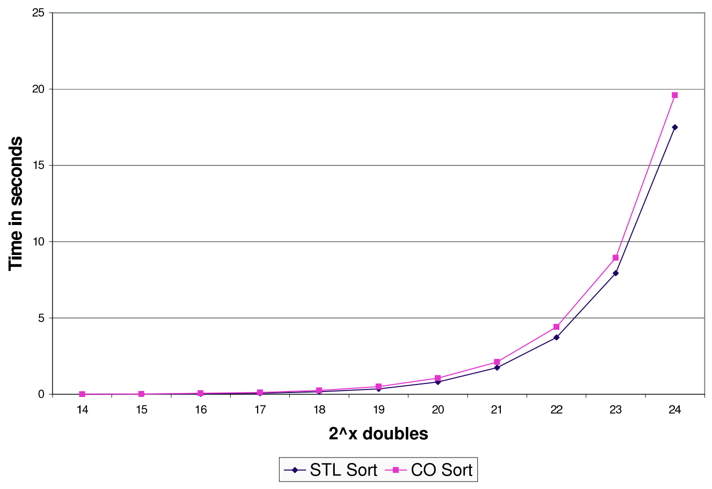


Fig. 9.6. Two Pass cache oblivious distribution sort, one level of recursion

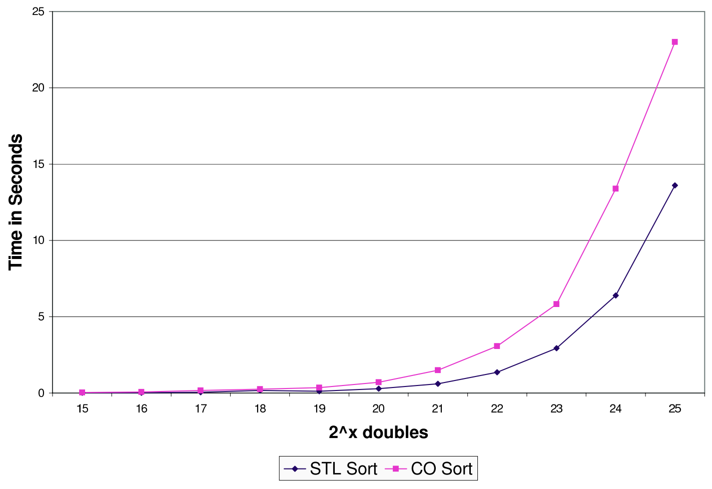


Fig. 9.7. Two Pass cache oblivious distribution sort, two levels of recursion

Lets now peek at the analysis. For a good set of splitters, $Pr(\exists i : c_i \geq \alpha\sqrt{N}) \leq Ne^{-(1-\frac{1}{\alpha})^2\frac{\beta\alpha}{2}}$ (for some $\alpha > 1, \beta > \log N$) [131]. This follows from Chernoff bound type arguments. Once we know that the subproblems we are going to work on are bounded in size with high probability, the expected cache complexity follows the recurrence:

$$Q(N) \leq \begin{cases} O(1+\frac{N}{B}) & \text{if } N \leq \alpha M \\ 2\sqrt{N}Q(\sqrt{N}) + Q(\sqrt{N}\log N) + O(1+\frac{N}{B}) & \text{otherwise.} \end{cases} \quad (9.2)$$

which solves to $Q(n) \leq O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

9.8 Is the Model an Oversimplification?

In theory, both the cache oblivious and the external memory models are nice to work with, because of their simplicity. A lot of the work done in the external memory model has been turned into practical results as well. Before one makes his hand "dirty" with implementing an algorithm in the cache oblivious or the external memory model, one should be aware of practical things that might become detrimental to the speed of the code but are not caught in the theoretical setup.

Here we list a few practical glitches that are shared by both the cache oblivious and the external memory model. The ones that are not shared are marked[2] accordingly. A reader that wants to use these models to design practical algorithms and especially one who wants to write code, should keep these issues in mind. Code written and algorithms designed keeping the following things in mind, could be a lot faster than just directly coding an algorithm that is optimal in either the cache oblivious or the external memory model.
TLB^o: TLBs are caches on page tables, are usually small with 128-256 entries and are like just any other cache. They can be implemented as fully associative. The model does not take into account the fact that TLBs are not tall. For the importance of TLB on performance of programs refer to the section on cache oblivious models in Chapter 8.
Concurrency: The model does not talk about I/O and CPU concurrency, which automatically looses it a 2x factor in terms of constants. The need for speed might drive future *uniprocessor* systems to diversify and look for alternative solutions in terms of concurrency on a single chip, for instance the hyper-threading[3] introduced by Intel in its latest Xeons is a glaring example. On these kind of systems and other multiprocessor systems, *coherence misses* might become an issue. This is hard to capture in the cache oblivious model and for most algorithms that have been devised in this model already, concurrency is still an open problem. A parallel cache oblivious model would be really welcome for practitioners who would like to apply cache oblivious algorithms to multiprocessor systems. (see Chapter 16)
Associativity^o: The assumption of the fully associative cache is not so nice. In reality caches are either direct mapped or k-way associative (typically

[2] A superscript 'o' means this issue only applies to the cache oblivious model.
[3] One physical processor Intel Xeon MP forms two logical processors which share CPU computational resources The software sees two CPUs and can distribute work load between them as a normal dual processor system.

$k = 2, 4, 8$). If two objects map to the same location in the cache and are referenced in temporal proximity, the accesses will become costlier than they are assumed in the model (also known as cache interference problem [718]). Also, $k-$way set associative caches are implemented by using more comparators. (see Chapter 8)

Instruction/Unified Caches: Rarely executed, special case code disrupts locality. Loops with few iterations that call other routines make loop locality hard to exploit and plenty of loopless code hampers temporal locality. Issues related to instruction caches are not modeled in the cache oblivious model. *Unified caches* (e.g. the latest Intel Itanium chips L2 and L3 caches) are used in some machines where instruction and data caches are merged(e.g. Intel PIII, Itaniums). These are another challenge to handle in the model.

Replacement Policy^o: Current operating systems do not page more than 4GB of memory because of address space limitations. That means one would have to use legacy code on these systems for paging. This problem makes portability of cache oblivious code for big problems a myth! In the experiments reported in this chapter, we could not do external memory experimentation because the OS did not allow us to allocate array sizes of more than a GB or so. One can overcome this problem by writing one's own paging system over the OS to do experimentation of cache oblivious algorithms on huge data sizes. But then its not so clear if writing a paging system is easier or handling disks explicitly in an application. This problem does not exist on 64-bit operating systems and should go away with time.

Multiple Disks^o: For "most" applications where data is huge and external memory algorithms are required, using Multiple disks is an option to increase I/O efficiency. As of now, the cache oblivious model does not take into account the existence of multiple disks in a system.

Write-through caches^o: L1 caches in many new CPUs is write through, i.e. it transmits a written value to L2 cache immediately [319, 392]. Write through caches are simpler to manage and can always discard cache data without any bookkeeping (Read misses can not result in writes). With write through caches (e.g. DECStation 3100, Intel Itanium), one can no longer argue that there are no misses once the problem size fits into cache! *Victim Caches* implemented in HP and Alpha machines are caches that are implemented as small buffers to reduce the effect of conflicts in set-associative caches. These also should be kept in mind when designing code for these machines.

Complicated Algorithms^o and Asymptotics: For non-trivial problems the algorithms can become quite complicated and impractical, a glaring instance of which is sorting. The speed by which different levels of memory differ in data transfer are constants! For instance the speed difference between L1 and L2 caches on a typical Intel pentium can be around 10. Using an $O()$ notation for an algorithm that is trying to beat a constant of 10, and sometimes not even talking about those constants while designing algorithms can show up in practice (Also see Chapter 8). For instance there are "constants" involved in simulating a fully associative cache on a k-way associative cache. Not using I/O concurrently with CPU can make an algorithm loose another constant. Can one really afford to hide these constants in the design of a cache oblivious algorithm in real code?

Despite these limitations the model does perform very well for some applications [178, 490, 318], but might be outperformed by more coding effort combined with cache aware algorithms [618, 178, 490, 318]. Here's an intercept from an experimental paper by Chatterjee and Sen [178].

> Our major conclusion are as follows: Limited associativity in the mapping from main memory addresses to cache sets can significantly degrade running time; the limited number of TLB entries can easily lead to thrashing; the fanciest optimal algorithms are not competitive on real machines even at fairly large problem sizes unless cache miss penalties are quite high; low level performance tuning "hacks", such as register tiling and array alignment, can significantly distort the effect of improved algorithms, ...

9.9 Other Results

We present here problems, related bounds and references for more interested readers. Note that in the table, *sort*() and *scan*() denote the number of cache misses of scan and sorting functions done by an optimal cache oblivious implementation.

Data Structure/Algorithm	Cache Complexity	Operations
Array Reversal	scan(N)	
List Ranking [192]	sort(N)	
LU Decomposition [726]	$\Theta(1 + \frac{N^2}{B} + \frac{N^3}{B\sqrt{M}})$	On $N \times N$ matrices
FFT [321]	sort(N)	
B-Trees [104, 156]	Amortized $O(\log_B N)$	Insertions/Deletions
Priority Queues [58, 155]	$O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$	Insertions/Deletions/deletemin
Tree Layout [58]	$O(\log B)$ OPT	Searches, Given probability distribution on leaves
Voronoi Diagrams in 2D [483]	sort(N)	Construction

9.10 Open Problems

Sorting strings in the cache oblivious model is still open. Optimal shortest paths and minimum spanning forests still need to be explored in the model. Optimal simple convex hull algorithms for $d-$dimensions is open. There are a lot of problems that still can be explored in this model both theoretically and practically.

One of the major open problems is to extensively evaluate how practical cache oblivious algorithms really are. For instance, our sorting experiment results don't look very promising. Is there a way to sort in the cache oblivious model that is at par with cache aware sorting codes already available? (Toy experiments comparing quicksort with a modified funnelsort or distribution sort don't count!) Currently the only impressive code that might back up "practicality" claims of cache oblivious algorithms is FFTW [318]. It looks like that the practicality of FFTW is more about coding tricks (special purpose compiler) that give it its speed and practicality than the theory of cache obliviousness. Moreover, FFTW only uses cache oblivious FFT for its base cases (Called codelets).

Matrix multiplication and transposition using blocked cache oblivious algorithms do fairly well in comparison with cache aware/external memory algorithms. B-trees also seem to do well in this model [104, 156]. Are there any other non-trivial optimal cache oblivious algorithms and data structures that are really practical and come close to cache aware algorithms, only time will tell!

For some applications, it might happen that there are more than one cache oblivious algorithms available. And out of the few cache oblivious algorithms available, only one is good in practice and the rest are not. For instance, for matrix transposition, there are at least two cache oblivious algorithms coded in [178]. There is one coded in this chapter. Out of these three algorithms only one really performs well in practice. In this way, cache obliviousness might be a necessary but not a sufficient condition for practical implementations. Another example is FFTW, where a optimal cache oblivious algorithm is outperformed by a slightly suboptimal cache oblivious algorithm. Is there a simple model that captures both necessary and sufficient condition for practical results?

9.11 Acknowledgements

The author would like to thank Michael Bender, Matteo Frigo, Joe Mitchell, Edgar Ramos and Peter Sanders for discussions on cache obliviousness and to MPI Informatik, Saarbrücken, Germany, for hosting him.

10. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*

Markus Kowarschik and Christian Weiß

10.1 Introduction

In order to mitigate the impact of the growing gap between CPU speed and main memory performance, today's computer architectures implement hierarchical memory structures. The idea behind this approach is to hide both the low main memory bandwidth and the latency of main memory accesses which is slow in contrast to the floating-point performance of the CPUs. Usually, there is a small and expensive high speed memory sitting on top of the hierarchy which is usually integrated within the processor chip to provide data with low latency and high bandwidth; i.e., the CPU registers. Moving further away from the CPU, the layers of memory successively become larger and slower. The memory components which are located between the processor core and main memory are called *cache memories* or *caches*. They are intended to contain copies of main memory blocks to speed up accesses to frequently needed data [378, 392]. The next lower level of the memory hierarchy is the main memory which is large but also comparatively slow. While external memory such as hard disk drives or remote memory components in a distributed computing environment represent the lower end of any common hierarchical memory design, this paper focuses on optimization techniques for enhancing cache performance.

The levels of the memory hierarchy usually subset one another so that data residing within a smaller memory are also stored within the larger memories. A typical memory hierarchy is shown in Fig. 10.1.

Efficient program execution can only be expected if the codes respect the underlying hierarchical memory design. Unfortunately, today's compilers cannot introduce highly sophisticated cache-based transformations and, consequently, much of this optimization effort is left to the programmer [335, 517].

This is particularly true for numerically intensive codes, which our paper concentrates on. Such codes occur in almost all science and engineering disciplines; e.g., computational fluid dynamics, computational physics, and mechanical engineering. They are characterized both by a large portion of floating-point (FP) operations as well as by the fact that most of their execution time is spent in small computational kernels based on loop nests.

* This research is being supported in part by the *Deutsche Forschungsgemeinschaft* (German Science Foundation), projects Ru 422/7–1,2,3.

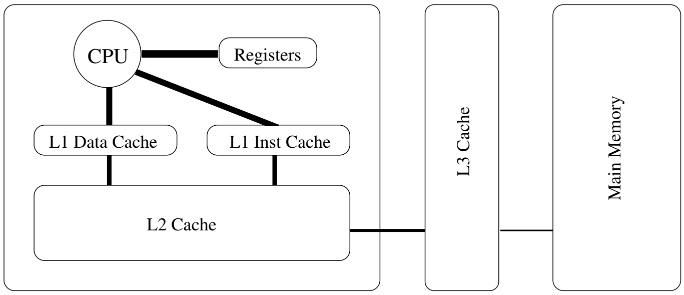


Fig. 10.1. A typical memory hierarchy containing two on-chip L1 caches, one on-chip L2 cache, and a third level of off-chip cache. The thickness of the interconnections illustrates the bandwidths between the memory hierarchy levels.

Thus, instruction cache misses have no significant impact on execution performance. However, the underlying data sets are typically by far too large to be kept in a higher level of the memory hierarchy; i.e., in cache.

Due to data access latencies and memory bandwidth issues, the number of arithmetic operations alone is no longer an adequate means of describing the computational complexity of numerical computations. Efficient codes in scientific computing must necessarily combine both computationally optimal algorithms and memory hierarchy optimizations. Multigrid methods [731], for example, are among the most efficient algorithms for the solution of large systems of linear equations. The performance of such codes on cache-based computer systems, however, is only acceptable if memory hierarchy optimizations are applied [762].

This paper is structured as follows. In Section 10.2, we will introduce some fundamental cache characteristics, including a brief discussion of cache performance analysis tools. Section 10.3 contains a general description of elementary cache optimization techniques. In Section 10.4, we will illustrate how such techniques can be employed to develop cache-aware algorithms. We will particularly focus on algorithms of numerical linear algebra. Section 10.5 concludes the paper.

10.2 Architecture and Performance Evaluation of Caches

10.2.1 Organization of Cache Memories

Typically, a memory hierarchy contains a rather small number of registers on the chip which are accessible without delay. Furthermore, a small cache — usually called *level one (L1) cache* — is placed on the chip to ensure low latency and high bandwidth. The L1 cache is often split into two separate parts; one only keeps data, the other instructions. The latency of *on-chip* caches is commonly one or two cycles. The chip designers, however, already face the problem that large on-chip caches of new microprocessors running at high clock rates cannot deliver data within one cycle since the signal delays are too long. Therefore, the size of on-chip L1 caches is limited to 64 Kbyte or even less for many chip designs. However, larger cache sizes with accordingly higher access latencies start to appear.

The L1 caches are usually backed up by a *level two (L2) cache*. A few years ago, architectures typically implemented the L2 cache on the motherboard, using SRAM chip technology. Currently, L2 cache memories are typically located on-chip as well; e.g., in the case of Intel's Itanium CPU. *Off-chip* caches are much bigger, but also provide data with lower bandwidth and higher access latency. On-chip L2 caches are usually smaller than 512 Kbyte and deliver data with a latency of approximately 5 to 10 cycles. If the L2 caches are implemented on-chip, an off-chip *level three (L3) cache* may be added to the hierarchy. Off-chip cache sizes vary from 1 Mbyte to 16 Mbyte. They provide data with a latency of about 10 to 20 CPU cycles.

10.2.2 Locality of References

Because of their limited size, caches can only hold copies of recently used data or code. Typically, when new data are loaded into the cache, other data have to be replaced. Caches improve performance only if cache blocks which have already been loaded are reused before being replaced by others. The reason why caches can substantially reduce program execution time is the principle of *locality* of references [392] which states that recently used data are very likely to be reused in the near future. Locality can be subdivided into *temporal* locality and *spatial* locality. A sequence of references exhibits temporal locality if recently accessed data are likely to be accessed again in the near future. A sequence of references exposes spatial locality if data located close together in address space tend to be referenced close together in time.

10.2.3 Aspects of Cache Architectures

In this section, we briefly review the basic aspects of cache architectures. We refer to Chapter 8 for a more detailed presentation of hardware issues concerning cache memories as well as *translation lookaside buffers (TLBs)*.

Data within the cache are stored in *cache lines*. A cache line holds the contents of a contiguous block of main memory. If data requested by the processor are found in a cache line, it is called a *cache hit*. Otherwise, a *cache miss* occurs. The contents of the *memory block* containing the requested word are then fetched from a lower memory layer and copied into a cache line. For this purpose, another data item must typically be replaced. Therefore, in order to guarantee low access latency, the question into which cache line the data should be loaded and how to retrieve them henceforth must be handled efficiently.

In respect of hardware complexity, the cheapest approach to implement block placement is *direct mapping*; the contents of a memory block can be placed into exactly one cache line. Direct mapped caches have been among the most popular cache architectures in the past and are still very common for off-chip caches.

However, computer architects have recently focused on increasing the *set associativity* of on-chip caches. An a-way set-associative cache is characterized by a higher hardware complexity, but usually implies higher hit rates. The cache lines of an a-way set-associative cache are grouped into sets of size a. The contents of any memory block can be placed into any cache line of the corresponding set.

Finally, a cache is called *fully associative* if the contents of a memory block can be placed into any cache line. Usually, fully associative caches are only implemented as small special-purpose caches; e.g., TLBs [392]. Direct mapped and fully associative caches can be seen as special cases of a-way set-associative caches; a direct mapped cache is a 1-way set-associative cache, whereas a fully associative cache is C-way set-associative, provided that C is the number of cache lines.

In a fully associative cache and in a k-way set-associative cache, a memory block can be placed into several alternative cache lines. The question into which cache line a memory block is copied and which block thus has to be replaced is decided by a (block) *replacement strategy*. The most commonly used strategies for today's microprocessor caches are *random* and *least recently used (LRU)*. The random replacement strategy chooses a random cache line to be replaced. The LRU strategy replaces the block which has not been accessed for the longest time interval. According to the principle of locality, it is more likely that a data item which has been accessed recently will be accessed again in the near future.

Less common strategies are *least frequently used (LFU)* and *first in, first out (FIFO)*. The former replaces the memory block in the cache line which has been used least frequently, whereas the latter replaces the data which have been residing in cache for the longest time.

Eventually, the *optimal* replacement strategy replaces the memory block which will not be accessed for the longest time. It is impossible to implement this strategy in a real cache, since it requires information about future cache references. Thus, the strategy is only of theoretical value; for any possible sequence of references, a fully associative cache with optimal replacement strategy will produce the minimum number of cache misses among all types of caches of the same size [717].

10.2.4 Measuring and Simulating Cache Behavior

In general, *profiling tools* are used in order to determine if a code runs efficiently, to identify performance bottlenecks, and to guide code optimization [335]. One fundamental concept of any memory hierarchy, however, is to hide the existence of caches. This generally complicates data locality optimizations; a speedup in execution time only indicates an enhancement of locality behavior, but it is no evidence.

To allow performance profiling regardless of this fact, many microprocessor manufacturers add dedicated registers to their CPUs in order to count certain events. These special-purpose registers are called *hardware performance counters.* The information which can be gathered by the hardware performance counters varies from platform to platform. Typical quantities which can be measured include cache misses and cache hits for various cache levels, pipeline stalls, processor cycles, instruction issues, and branch mispredictions. Some prominent examples of profiling tools based on hardware performance counters are the *Performance Counter Library (PCL)* [117], the *Performance Application Programming Interface (PAPI)* [162], and the *Digital Continuous Profiling Infrastructure (DCPI)* (Alpha-based Compaq Tru64 UNIX only) [44].

Another approach towards evaluating code performance is based on *instrumentation.* Profiling tools such as *GNU prof* [293] and *ATOM* [282] insert calls to a monitoring library into the program to gather information for small code regions. The library routines may either include complex programs themselves (e.g., simulators) or only modify counters. Instrumentation is used, for example, to determine the fraction of the CPU time spent in a certain subroutine. Since the cache is not visible to the instrumented code the information concerning the memory behavior is limited to address traces and timing information.

Eventually, cache performance information can be obtained by *cache modeling and simulation* [329, 383, 710] or by *machine simulation* [636]. Simulation is typically very time-consuming compared to regular program execution. Thus, the cache models and the machine models often need to be simplified in order to reduce simulation time. Consequently, the results are often not precise enough to be useful.

10.3 Basic Techniques for Improving Cache Efficiency

10.3.1 Data Access Optimizations

Data access optimizations are code transformations which change the order in which iterations in a loop nest are executed. The goal of these transformations is mainly to improve temporal locality. Moreover, they can also expose parallelism and make loop iterations vectorizable. Note that the data access optimizations we present in this section maintain all data dependencies and do not change the results of the numerical computations[1].

Usually, it is difficult to decide which combination of transformations must be applied in order to achieve a maximum performance gain. Compilers typically use heuristics to determine whether a transformation will be effective or not. Loop transformation theory and algorithms found in the literature typically focus on transformations for perfectly nested loops [19]; i.e., nested loops where all assignment statements are contained in the innermost loop. However, loop nests in scientific codes are not perfectly nested in general. Hence, initial enabling transformations like loop skewing, loop unrolling, and loop peeling are required. Descriptions of these transformations can be found in the compiler literature [30, 80, 561, 769].

In the following, a set of loop transformations will be described which focus on improving data locality for one level of the memory hierarchy; typically a cache. As we have already mentioned in Section 10.1, instruction cache misses have no severe impact on the performance of numerically intensive codes since these programs typically execute small computational kernels over and over again. Nevertheless, some of the transformations we present in this section can be used to improve instruction locality as well.

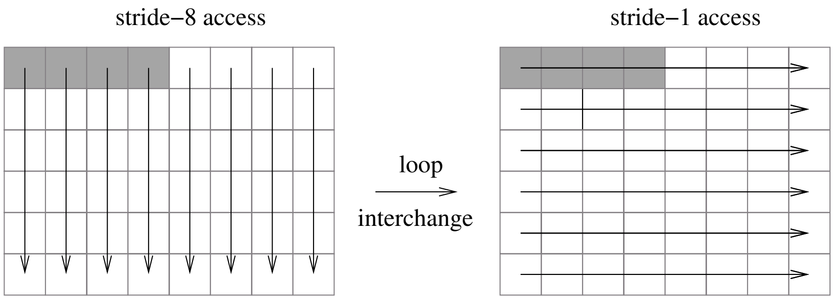


Fig. 10.2. Access patterns for interchanged loop nests.

Loop Interchange. This transformation reverses the order of two adjacent loops in a loop nest [30, 769]. Generally speaking, loop interchange can be applied if the order of the loop execution is unimportant. Loop interchange can be generalized to *loop permutation* by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

Loop interchange can improve locality by reducing the *stride* of an array-based computation. The stride is the distance of array elements in memory accessed within consecutive loop iterations. Upon a memory reference, several

[1] However, these transformations may trigger an aggressively optimizing compiler to reorder FP operations. Due to the properties of finite precision arithmetic, this may cause different numerical results.

words of an array are loaded into a cache line. If the array is larger than the cache, accesses with large stride only use one word per cache line. The other words which are loaded into the cache line are evicted before they can be reused.

Loop interchange can also be used to enable and improve vectorization and parallelism, and to improve register reuse. The different targets may be conflicting. For example, increasing parallelism requires loops with no dependencies to be moved outward, whereas vectorization requires them to be moved inward.

Algorithm 10.3.1 Loop interchange

```
double $sum$;
double $a[n, n]$;
// Original loop nest:
for $j = 1$ to $n$ do
    for $i = 1$ to $n$ do
        $sum += a[i, j]$;
    end for
end for
```

```
double $sum$;
double $a[n, n]$;
// Interchanged loop nest:
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $sum += a[i, j]$;
    end for
end for
```

The effect of loop interchange is illustrated in Fig. 10.2. We assume that the $(6, 8)$ array is stored in memory in *row major order*; i.e., two array elements are stored adjacent in memory if their second indices are consecutive numbers. The code corresponding to the left part of Fig. 10.2, however, accesses the array elements in a column-wise manner. Consequently, the preloaded data in the cache line marked with grey color will not be reused if the array is too large to fit entirely in cache. However, after interchanging the loop nest as demonstrated in Algorithm 10.3.1, the array is no longer accessed using stride-8, but stride-1. Consequently, all words in the cache line are now used by successive loop iterations. This is illustrated by the right part of Fig. 10.2.

Loop Fusion. *Loop fusion* is a transformation which takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop [238]. Loop fusion — sometimes also called *loop jamming* — is the inverse transformation of *loop distribution* or *loop fission* which breaks a single loop into multiple loops with the same iteration space. Loop fusion is legal as long as no flow, anti, or output dependencies in the fused loop exist for which instructions from the first loop depend on instructions from the second loop [30].

Fusing two loops results in a single loop which contains more instructions in its body and therefore offers increased instruction level parallelism. Furthermore, only one loop is executed, thus reducing the total loop overhead by approximately a factor of two.

Algorithm 10.3.2 Loop fusion

```
// Original code:
for $i = 1$ to $n$ do
    $b[i] = a[i] + 1.0$;
end for
for $i = 1$ to $n$ do
    $c[i] = b[i] * 4.0$;
end for
```

```
// After loop fusion:
for $i = 1$ to $n$ do
    $b[i] = a[i] + 1.0$;
    $c[i] = b[i] * 4.0$;
end for
```

Loop fusion also improves data locality. Assume that two consecutive loops perform global sweeps through an array as in the code shown in Algorithm 10.3.2, and that the data of the array are too large to fit completely in cache. The data of array b which are loaded into the cache by the first loop will not completely remain in cache, and the second loop will have to reload the same data from main memory. If, however, the two loops are combined with loop fusion only one global sweep through the array b will be performed. Consequently, fewer cache misses will occur.

Loop Blocking. *Loop blocking* (also called *loop tiling*) is a loop transformation which increases the depth of a loop nest with depth n by adding additional loops to the loop nest. The depth of the resulting loop nest will be anything from $n+1$ to $2n$. Loop blocking is primarily used to improve data locality by enhancing the reuse of data in cache [30, 705, 768].

Algorithm 10.3.3 Loop blocking for matrix transposition

```
// Original code:
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $a[i,j] = b[j,i]$;
    end for
end for
```

```
// Loop blocked code:
for $ii = 1$ to $n$ by B do
    for $jj = 1$ to $n$ by B do
        for $i = ii$ to $min(ii + B - 1, n)$ do
            for $j = jj$ to $min(jj + B - 1, n)$ do
                $a[i,j] = b[j,i]$;
            end for
        end for
    end for
end for
```

The need for loop blocking is illustrated in Algorithm 10.3.3. Assume that the code reads an array a with stride-1, whereas the access to array b is of stride-n. Interchanging the loops will not help in this case since it would cause the array a to be accessed with stride-n instead.

Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a *block* of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step. The change in the iteration space traversal of the blocked loop in Algorithm 10.3.3 is shown in Fig. 10.3.

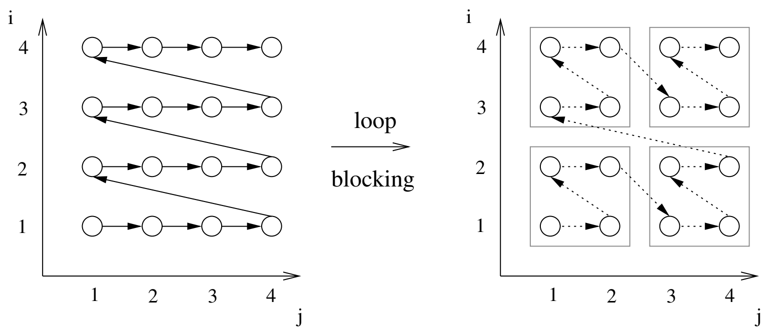


Fig. 10.3. Iteration space traversal for original and blocked code.

A very prominent example for the impact of the loop blocking transformation on data locality is matrix multiplication [127, 461, 492, 764], see also Section 10.4.2. In particular, the case of sparse matrices is considered in [577].

Data Prefetching. The loop transformations discussed so far aim at reducing the *capacity misses* which occur in the course of a computation. Misses which are introduced by first-time accesses are not addressed by these optimizations. Prefetching allows the microprocessor to issue a data request before the computation actually requires the data [747]. If the data are requested early enough the penalty of *cold (compulsory) misses* as well as capacity misses not covered by loop transformations can be hidden[2].

Many modern microprocessors implement a *prefetch* instruction which is issued as a regular instruction. The *prefetch* instruction is similar to a load, with the exception that the data are not forwarded to the CPU after they have been cached. The *prefetch* instruction is often handled as a hint for the processor to load a certain data item, but the actual execution of the prefetch is not guaranteed by the CPU.

Prefetch instructions can be inserted into the code manually by the programmer or automatically by a compiler [558]. In both cases, prefetching involves overhead. The prefetch instructions themselves have to be executed; i.e., pipeline slots will be filled with prefetch instructions instead of other instructions ready to be executed. Furthermore, the memory addresses of the prefetched data must be calculated and will be calculated again when the load operation is executed which actually fetches the data from the memory hierarchy into the CPU.

Besides *software-based* prefetching, hardware schemes have been proposed and implemented which add prefetching capability to a system without the

[2] For a classification of cache misses we refer to Chapter 8.

need of prefetch instructions. One of the simplest *hardware-based* prefetching schemes is *sequential prefetching* [702]; whenever a memory block is accessed, the next and possibly some subsequent memory blocks are prefetched. More sophisticated prefetch schemes have been invented [187], but most microprocessors still only implement stride-1 stream detection or even no prefetching at all.

In general, prefetching will only be successful if the data stream is predicted correctly either by the hardware or by the compiler and if there is enough space left in cache to keep the prefetched data together with memory references that are still active. If the prefetched data replace data which are still needed this will increase bus utilization, the overall miss rates, as well as memory latencies [166].

10.3.2 Data Layout Optimizations

Data access optimizations have proven to be able to improve the data locality of applications by reordering the computation, as we have shown in the previous section. However, for many applications, loop transformations alone may not be sufficient for achieving reasonable data locality. Especially for computations with a high degree of *conflict misses*[3], loop transformations are not effective in improving performance [632].

Data layout optimizations modify how data structures and variables are arranged in memory. These transformations aim at avoiding effects like cache conflict misses and *false sharing* [392], see Chapter 16. They are further intended to improve the spatial locality of a code.

Data layout optimizations include changing base addresses of variables, modifying array sizes, transposing array dimensions, and merging arrays. These techniques are usually applied at compile time, although some optimizations can also be applied at runtime.

Array Padding. If two arrays are accessed in an alternating manner as in Algorithm 10.3.4 and the data structures happen to be mapped to the same cache lines, a high number of conflict misses are introduced.

In the example, reading the first element of array a will load a cache line containing this array element and possibly subsequent array elements for further use. Provided that the first array element of array b is mapped to the same cache line as the first element of array a, a read of the former element will trigger the cache to replace the elements of array a which have just been loaded. The following access to the next element of array a will no longer be satisfied by the cache, thus force the cache to reload the data and in turn to replace the data of array b. Hence, the array b elements must be reloaded, and so on. Although both arrays are referenced sequentially with stride-1, no reuse of data which have been preloaded into the cache will occur since the data are evicted immediately by elements of the other array, after they have

[3] See again Chapter 8.

been loaded. This phenomenon is called *cross interference* of array references [492].

Algorithm 10.3.4 Inter-array padding.

```
// Original code:
double $a[1024]$;
double $b[1024]$;
for $i = 0$ to 1023 do
    $sum += a[i] * b[i]$;
end for
```

```
// Code after applying inter-array padding:
double $a[1024]$;
double $pad[x]$;
double $b[1024]$;
for $i = 1$ to 1023 do
    $sum += a[i] * b[i]$;
end for
```

A similar problem — called *self interference* — can occur if several rows of a multidimensional array are mapped to the same set of cache lines and the rows are accessed in an alternating fashion.

For both cases of interference, array padding [727, 632] provides a means to reduce the number of conflict misses. *Inter-array padding* inserts unused variables (pads) between two arrays in order to avoid cross interference. Introducing pads modifies the offset of the second array such that both arrays are then mapped to different parts of the cache.

Intra-array padding, on the other hand, inserts unused array elements between rows of a multidimensional array by increasing the leading dimension of the array; i.e., the dimension running fastest in memory is increased by a small number of extra elements. Which dimension runs fastest in memory depends on the programming language. For example, in Fortran77 the leftmost dimension is the leading dimension, whereas in C/C++ the rightmost dimension runs fastest.

The sizes of the pads depend on the mapping scheme of the cache, the cache size, the cache line size, its set associativity, and the data access pattern of the code. Typical padding sizes are multiples of the cache line size, but different sizes may be used as well. Array padding is usually applied at compile time. Intra-array padding can, in principle, be introduced at runtime. However, knowledge of the cache architecture is indispensable, and information about the access pattern of the program will improve the quality of the selected padding size [632, 633]. The disadvantage of array padding is that extra memory is required for pads.

Array Merging. This layout optimization technique can be used to improve the spatial locality between elements of different arrays or other data structures. Furthermore, array merging can reduce the number of cross interference misses for scenarios with large arrays and alternating access patterns, as we have introduced in the previous paragraph. The array merging technique is also known as *group-and-transpose* [429].

Array merging is best applied if elements of different arrays are located far apart in memory but usually accessed together. Transforming the data

Algorithm 10.3.5 Array merging.

```
// Original data structure:
double a[1024];
double b[1024];
```

```
// array merging using multidimensional arrays:
double ab[1024][2];
```

```
// array merging using structures:
struct{
    double a;
    double b;
} ab[1024];
```

structures as shown in Algorithm 10.3.5 will change the data layout such that the elements become contiguous in memory.

Array Transpose. This technique permutes the dimensions within multi-dimensional arrays and eventually reorders the array as shown in Algorithm 10.3.6 [202]. This transformation has a similar effect as loop interchange, see Section 10.3.1.

Algorithm 10.3.6 Array transpose.

```
// Original data structure:
double a[N][M];
```

```
// Data structure after transposing:
double a[M][N];
```

Data Copying. In Section 10.3.1, loop blocking has been introduced as a technique to reduce the number of capacity misses. Research has shown [301, 768] that blocked codes suffer from a high degree of conflict misses introduced by self interference. This effect is demonstrated by means of Fig. 10.4. The figure shows a part (block) of a big array which is to be reused by a blocked algorithm. Suppose that a direct mapped cache is used, and that the two words marked with x are mapped to the same cache location. Due to the regularity of the cache mapping, the shaded words in the upper part of the block will be mapped to the same cache lines as the shaded words in the lower part of the block. Consequently, if the block is accessed repeatedly, the data in the upper left corner will replace the data in the lower right corner and vice versa, thus reducing the reusable part of the block.

Therefore, researchers have proposed a data copying technique to guarantee high cache utilization for blocked algorithms [768]. With this approach, non-contiguous data from a block are copied into a contiguous area of memory. Hence, each word of the block will be mapped to its own cache location, effectively avoiding self interference within the block.

The technique, however, involves a copy operation which increases the total cost of the algorithm. In many cases the additional cost will outweigh

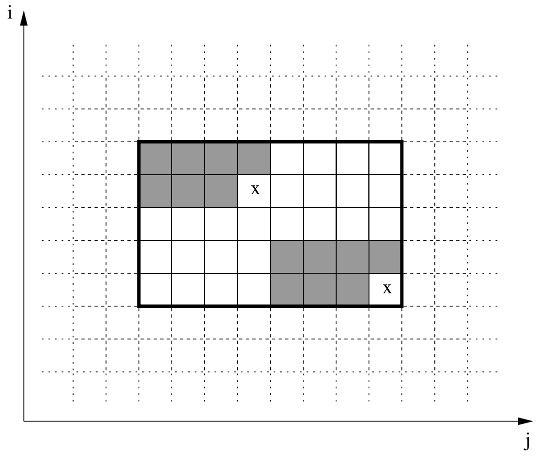


Fig. 10.4. Self interference in blocked code.

the benefits from copying the data. Hence a compile time strategy has been introduced in order to determine when to copy data [719]. This technique is based on an analysis of cache conflicts.

10.4 Cache-Aware Algorithms of Numerical Linear Algebra

10.4.1 Overview: The Software Libraries BLAS and LAPACK

The optimization of numerical algorithms is a large and multifaceted field of ongoing research. In this survey, we focus on algorithms of numerical linear algebra which play an essential role in numerical mathematics as well as in computational science. *Partial differential equations (PDEs)* which arise in almost all scientific and engineering applications, for example, are typically discretized using finite differences, finite elements, or finite volumes. This step usually yields large systems of linear equations the solution of which is only one fundamental issue of algorithms of numerical linear algebra.

These algorithms are often based on elementary kernel routines which are provided by highly optimized underlying software libraries; e.g. *BLAS*[4] and *LAPACK*[5].

BLAS provides building blocks for performing elementary vector and matrix operations [255]. In the following, we use α and β to represent scalar values, whereas x and y denote vectors, and A, B, and C represent matrices.

[4] *BLAS: Basic Linear Algebra Subprograms*, see `http://www.netlib.org/blas`.

[5] *LAPACK: Linear Algebra PACKage*, see `http://www.netlib.org/lapack`.

The BLAS library is divided into three levels. Level 1 BLAS do vector-vector operations; e.g., so-called *AXPY* computations such as $y \leftarrow \alpha x + y$ and *dot products* such as $\alpha \leftarrow \beta + x^T y$. Level 2 BLAS do matrix-vector operations; e.g., $y \leftarrow \alpha \mathrm{op}(A)x + \beta y$, where $\mathrm{op}(A) = A, A^T$, or A^H. Eventually, Level 3 BLAS do matrix-matrix operations such as $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$. Dedicated routines are provided for special cases such as symmetric and Hermitian matrices. BLAS provides similar functionality for real and complex data types, in both single and double precision.

LAPACK is another software library which is often used by numerical applications [43]. LAPACK is based on the BLAS and implements routines for solving systems of linear equations, computing least-squares solutions of linear systems, and solving eigenvalue as well as singular value problems. The associated routines for factorizing matrices are also provided; e.g., LU, Cholesky, and QR decomposition. LAPACK handles dense and banded matrices, see Section 10.4.4 below for a discussion of iterative solvers for sparse linear systems. In analogy to the BLAS library, LAPACK implements similar functionality for real and complex matrices, in both single and double precision.

10.4.2 Enhancing the Cache Performance of the BLAS Library

Our presentation closely follows the research efforts of the *ATLAS*[6] project [764]. This project concentrates on the automatic application of empirical code optimization techniques for the generation of highly optimized platform-specific BLAS libraries. The basic idea is to successively introduce source-to-source transformations and evaluate the resulting performance, thus generating the most efficient implementation of BLAS. It is important to note that ATLAS still depends on an optimizing compiler for applying architecture-dependent optimizations and generating efficient machine code. A similar tuning approach has guided the research in the *FFTW* project [320].

ATLAS mainly targets the optimizations of Level 2 and Level 3 BLAS while relying on the underlying compiler to generate efficient Level 1 BLAS. This is due to the fact that Level 1 BLAS basically contains no memory reuse and high level source code transformations only yield marginal speedups.

On the contrary, the potential for data reuse is high in Level 2 and even higher in Level 3 BLAS due to the occurrence of at least one matrix operand. Concerning the optimization of Level 2 BLAS, ATLAS implements both *register blocking*[7] and loop blocking. In order to illustrate the application of these techniques it is sufficient to consider the update operation $y \leftarrow Ax + y$, where A is an $n \times n$ matrix and x, y are vectors of length n. This operation can also be written as

[6] *ATLAS: Automatically Tuned Linear Algebra Software.* More details are provided on `http://math-atlas.sourceforge.net`.

[7] The developers of ATLAS refer to the term register blocking as a technique to explicitly enforce the reuse of CPU registers by introducing temporary variables.

$$y_i \leftarrow \sum_{j=1}^{n} a_{i,j} x_j + y_i, \quad 1 \le i \le n \ ,$$

see [764]. By keeping the current value y_i in a CPU register (i.e., by applying register blocking), the number of read/write accesses to y can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Furthermore, unrolling the outermost loop and hence updating k components of the vector y simultaneously can reduce the number of accesses to x by a factor of $1/k$ to n^2/k. This is due to the fact that each x_j contributes to each y_i. In addition, loop blocking can be introduced in order to reduce the number of main memory accesses to the components of the vector x from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ [764], see Section 10.3 for details. This means that loop blocking can be applied in order to load x only once into the cache.

While Level 2 BLAS routines require $\mathcal{O}(n^2)$ data accesses in order to perform $\mathcal{O}(n^2)$ FP operations, Level 3 BLAS routines need $\mathcal{O}(n^2)$ data accesses to execute $\mathcal{O}(n^3)$ FP operations, thus containing a higher potential for data reuse. Consequently, the most significant speedups are obtained by tuning the cache performance of Level 3 BLAS; particularly the *matrix multiply.* This is achieved by implementing an L1 cache-contained matrix multiply and partitioning the original problem into subproblems which can be computed in cache [764]. In other words, the optimized code results from blocking each of the three loops of a standard matrix multiply algorithm, see again Section 10.3, and calling the L1 cache-contained matrix multiply code from within the innermost loop. Fig. 10.5 illustrates the blocked algorithm. In order to compute the shaded block of the product C, the corresponding blocks of its factors A and B have to be multiplied and added.

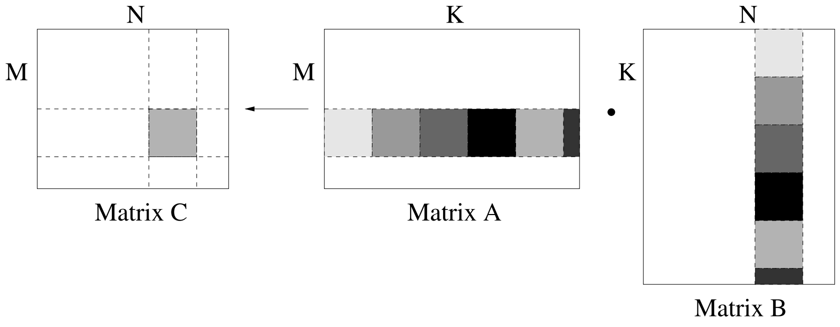


Fig. 10.5. Blocked matrix multiply algorithm.

In order to further enhance the cache performance of the matrix multiply routine, ATLAS introduces additional blocking for either L2 or L3 cache. This is achieved by tiling the loop which moves the current matrix blocks horizontally through the first factor A and vertically through the second factor B, respectively. The resulting performance gains depend on various parameters; e.g., hardware characteristics, operating system features, and compiler capabilities [764].

It is important to note that *fast matrix multiply* algorithms which require $\mathcal{O}(n^\omega), \omega < 3$, FP operations have been developed; e.g., *Winograd's method* and *Strassen's method.* These algorithms are based on the idea of recursively partitioning the factors into blocks and reusing intermediate results. However, error analysis reveals that these fast algorithms have different properties in terms of numerical stability, see [395] for a detailed analysis.

10.4.3 Block Algorithms in LAPACK

In order to leverage the speedups which are obtained by optimizing the cache utilization of Level 3 BLAS, LAPACK provides implementations of *block algorithms* in addition to the standard versions of various routines only based on Level 1 and Level 2 BLAS. For example, LAPACK implements block LU, block Cholesky, and block QR factorizations [43]. The idea behind these algorithms is to split the original matrices into submatrices (blocks) and process them using highly efficient Level 3 BLAS, see Section 10.4.2.

In order to illustrate the design of block algorithms in LAPACK we compare the standard LU factorization of a non-singular $n \times n$ matrix A to the corresponding *block LU factorization.* In order to simplify the presentation, we initially leave pivoting issues aside. Each of these algorithms determines a lower unit triangular $n \times n$ matrix[8] L and an upper triangular $n \times n$ matrix U such that $A = LU$. The idea of this (unique) factorization is that any linear system $Ax = b$ can then be solved easily by first solving $Ly = b$ using a forward substitution step, and subsequently solving $Ux = y$ using a backward substitution step [339, 395].

Computing the triangular matrices L and U essentially corresponds to performing Gaussian elimination on A in order to obtain an upper triangular matrix. In the course of this computation, all elimination factors $l_{i,j}$ are stored. These factors $l_{i,j}$ become the subdiagonal entries of the unit triangular matrix L, while the resulting upper triangular matrix defines the factor U. This elimination process is mainly based on Level 2 BLAS; it repeatedly requires rows of A to be added to multiples of different rows of A.

The block LU algorithm works as follows. The matrix A is partitioned into four submatrices $A_{1,1}, A_{1,2}, A_{2,1}$, and $A_{2,2}$. The factorization $A = LU$ can then be written as

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix} , \qquad (10.1)$$

where the corresponding blocks are equally sized, and $A_{1,1}, L_{1,1}$, and $U_{1,1}$ are square submatrices. Hence, we obtain the following equations:

[8] A *unit* triangular matrix is characterized by having only 1's on its main diagonal.

$$A_{1,1} = L_{1,1}U_{1,1} \ , \tag{10.2}$$

$$A_{1,2} = L_{1,1}U_{1,2} \ , \tag{10.3}$$

$$A_{2,1} = L_{2,1}U_{1,1} \ , \tag{10.4}$$

$$A_{2,2} = L_{2,1}U_{1,2} + L_{2,2}U_{2,2} \ . \tag{10.5}$$

According to Equation (10.2), $L_{1,1}$ and $U_{1,1}$ are computed using the standard LU factorization routine. Afterwards, $U_{1,2}$ and $L_{2,1}$ are determined from Equations (10.3) and (10.4), respectively, using Level 3 BLAS solvers for triangular systems. Eventually, $L_{2,2}$ and $U_{2,2}$ are computed as the result of recursively applying the block LU decomposition routine to $\tilde{A}_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$. This final step follows immediately from Equation (10.5). The computation of $\tilde{A}$ can again be accomplished by leveraging Level 3 BLAS.

It is important to point out that the block algorithm can yield different numerical results than the standard version as soon as pivoting is introduced; i.e., as soon as a decomposition $PA = LU$ is computed, where P denotes a suitable *permutation matrix* [339]. While the search for appropriate pivots may cover the whole matrix A in the case of the standard algorithm, the block algorithm restricts this search to the current block $A_{1,1}$ to be decomposed into triangular factors. The choice of different pivots during the decomposition process may lead to different round-off behavior due to finite precision arithmetic.

Further cache performance optimizations for LAPACK have been developed. The application of recursively packed matrix storage formats is an example of how to combine both data layout as well as data access optimizations [42]. A memory-efficient LU decomposition algorithm with partial pivoting is presented in [726]. It is based on recursively partitioning the input matrix.

10.4.4 Cache-Aware Iterative Algorithms

Iterative algorithms form another class of numerical solution methods for systems of linear equations [339, 371, 748]. LAPACK does not provide implementations of iterative methods. A typical example of the use of these methods is the solution of large sparse systems which arise from the discretization of PDEs. Iterative methods covers *basic techniques* such as *Jacobi's method, Jacobi overrelaxation (JOR)*, the *method of Gauss-Seidel*, and the *method of successive overrelaxation (SOR)* [339, 371, 748] as well as advanced techniques such as *multigrid* algorithms. Generally speaking, the idea behind multigrid algorithms is to accelerate convergence by uniformly eliminating error components over the entire frequency domain. This is accomplished by solving a recursive sequence of several instances of the original problem simultaneously, each of them on a different scale, and combining the results to form the required solution [147, 370, 731]. Typically, *Krylov subspace methods* such as the *method of conjugate gradients (CG)* and the *method of generalized minimal* *residuals (GMRES)* are considered iterative, too. Although their maximum number of computational steps is theoretically limited by the dimension of the linear system to be solved, this is of no practical relevance in the case of systems with millions of unknowns [339, 371].

In this paper, we focus on the discussion of cache-aware variants of basic iterative schemes; particularly the method of Gauss-Seidel. On the one hand, such methods are used as linear solvers themselves. On the other hand, they are commonly employed as *smoothers* to eliminate the highly oscillating Fourier components of the error in multigrid settings [147, 370, 731] and as *preconditioners* in the context of Krylov subspace methods [339, 371].

Given an initial approximation $x^{(0)}$ to the exact solution x of the linear system $Ax = b$ of order n, the method of Gauss-Seidel successively computes a new approximation $x^{(k+1)}$ from the previous approximation $x^{(k)}$ as follows:

$$x_i^{(k+1)} = a_{i,i}^{-1}\left(b_i - \sum_{j<i} a_{i,j}x_j^{(k+1)} - \sum_{j>i} a_{i,j}x_j^{(k)}\right), \quad 1 \le i \le n \ . \tag{10.6}$$

If used as a linear solver by itself, the iteration typically runs until some convergence criterion is fulfilled; e.g., until the Euclidean norm of the *residual* $r^{(k)} = b - Ax^{(k)}$ falls below some given tolerance.

For the discussion of optimization techniques we concentrate on the case of a block tridiagonal matrix which typically results from the 5-point discretization of a PDE on a two-dimensional rectangular grid using finite differences. We further assume a *grid-based* implementation[9] of the method of Gauss-Seidel using a red/black ordering of the unknowns.

For the sake of optimizing the cache performance of such algorithms, both data layout optimizations as well as data access optimizations have been proposed. Data layout optimizations comprise the application of array padding in order to minimize the numbers of conflict misses caused by the stencil-based computation [632] as well as array merging techniques to enhance the spatial locality of the code [256, 479]. These array merging techniques are based on the observation that, for each update, all entries $a_{i,j}$ of any matrix row i as well as the corresponding right-hand side b_i are always needed simultaneously, see Equation (10.6). Data access optimizations for red/black Gauss-Seidel comprise loop fusion as well as loop blocking techniques. As we have mentioned in Section 10.3, these optimizations aim at reusing data as long as they reside in cache, thus enhancing temporal locality. Loop fusion merges two successive passes through the grid into a single one, integrating the update steps for the red and the black nodes. On top of loop fusion, loop blocking can be applied. For instance, blocking the outermost loop means beginning with the computation of $x^{(k+2)}$ from $x^{(k+1)}$ before the computation of $x^{(k+1)}$ from $x^{(k)}$ has been completed, reusing the matrix entries $a_{i,j}$,

[9] Instead of using data structures to store the computational grids which cover the geometric domains, these methods can also be implemented by employing matrix and vector data structures.

the values b_i of the right-hand side, and the approximations $x_i^{(k+1)}$ which are still in cache.

Again, the performance of the optimized codes depends on a variety of machine, operating system, and compiler parameters. Depending on the problem size, speedups of up to 500% can be obtained, see [682, 762, 763] for details. It is important to point out that these optimizing data access transformations maintain all data dependencies of the original algorithm and therefore do not influence the numerical results of the computation.

Similar research has been done for Jacobi's method [94], which successively computes a new approximation $x^{(k+1)}$ from the previous approximation $x^{(k)}$ as follows:

$$x_i^{(k+1)} = a_{i,i}^{-1} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^{(k)} \right) , \quad 1 \leq i \leq n \ . \tag{10.7}$$

It is obvious that this method requires the handling of an extra array since the updates cannot be done in place; in order to compute the $(k+1)$-th iterate for unknown x_i, the k-th iterates of all neighboring unknowns are required, see Equation (10.7), although there may already be more recent values for some of them from the current $(k+1)$-th update step.

Moreover, the optimization of iterative methods on unstructured grids has also been addressed [257]. These techniques are based on partitioning the computational domain into blocks which are adapted to the size of the cache. The iteration then performs as much work as possible on the current cache block and revisits previous cache blocks in order to complete the update process. The investigation of corresponding cache optimizations for three-dimensional problems has revealed that TLB misses become more relevant than in the two-dimensional case [478, 633].

More advanced research on hierarchical memory optimization addresses the design of new iterative numerical algorithms. Such methods cover domain decomposition approaches with domain sizes which are adapted to the cache capacity [36, 358] as well as approaches based on runtime-controlled adaptivity which concentrates the computational work on those parts of the domain where the errors are still large and need to be further reduced by smoothing and coarse grid correction in a multigrid context [642, 518]. Other research address the development of grid structures for PDE solvers based on highly regular building blocks, see [160, 418] for example. On the one hand these meshes can be used to approximate complex geometries, on the other hand they permit the application of a variety of optimization techniques to enhance cache utilization, see Section 10.3 for details.

10.5 Conclusions

Cache performance optimizations can yield significant execution speedups, particularly when applied to numerically intensive codes. The investigation of such techniques has led to a variety of new numerical algorithms, some of which have been outlined in this paper. While several of the basic optimization techniques can automatically be introduced by optimizing compilers, most of the tuning effort is left to the programmer. This is especially true, if the resulting algorithms have different numerical properties; e.g., concerning stability, robustness, or convergence behavior. In order to simplify the development of portable (cache-)efficient numerical applications in science and engineering, optimized routines are often provided by machine-specific software libraries.

Future computer architecture trends further motivate research efforts focusing on memory hierarchy optimizations. Forecasts predict the number of transistors on chip increasing beyond one billion. Computer architects have announced that most of the transistors will be used for larger on-chip caches and on-chip memory. Most of the forecast systems will be equipped with memory structures similar to the memory hierarchies currently in use.

While those future caches will be bigger and smarter, the data structures presently used in real-world scientific codes already exceed the maximum capacity of forecast cache memories by several orders of magnitude. Today's applications in scientific computing typically require several Megabytes up to hundreds of Gigabytes of memory.

Consequently, due to the similar structure of present and future memory architectures, data locality optimizations for numerically intensive codes will further on be essential for all computer architectures which employ the concept of hierarchical memory.

11. Memory Limitations in Artificial Intelligence

Stefan Edelkamp

11.1 Introduction

Artificial Intelligence (AI) deals with structuring large amounts of data. As a very first example of an *expert system* [424], take the oldest known scientific treatise surviving from the ancient world, the surgical papyrus [146] of about 3000 BC. It discusses cases of injured men for whom a surgeon had no hope of saving and lay many years unnoticed until it was rediscovered and published for the New York Historical Society. The papyrus summarizes surgical observations of head wounds disclosing an inductive method for inference [281], with observations that were stated with title, examination, diagnosis, treatment, prognosis and glosses much in the sense that *if a patient has this symptom, then he has this injury with this prognosis if this treatment is applied.*

About half a century ago, pioneering computer scientists report the emergence of intelligence with machines that think, learn and create [696]. The prospects were driven by early successes in exploration. Samuel [651] wrote a checkers-playing program that was able to beat him, whereas Newell and Simon [580] successfully ran the *general problem solver* (GPS) that reduced the difference between the predicted and the desired outcome on different state-space problems. GPS represents problems as the task of transforming one symbolic expression into another, with a decomposition that fitted well with the structure of several other problem solving programs. Due to small available memories and slow CPUs, these and some other promising initial AI programs were limited in their problem solving abilities and failed to scale in later years.

There are two basic problems to overcome [539]: the *frame problem* — characterized as *the smoking pistol behind a lot of the attacks on AI* [249] — refers to all that is going on around the central actors, while the *qualification problem* refers to the host of qualifiers to stop an expected rule from being followed exactly. While [454] identifies several arguments of why intelligence in a computer is not a true ontological one, the most important reason for many drawbacks in AI are existing limits in computational resources, especially in memory, which is often too small to keep all information for a suitable inference accessible.

Bounded resources lead to a performance-oriented interpretation of the term intelligence: different to the *Turing-test* [734], programs have to show human-adequate or human-superior abilities in a competitive resource-constrained environment on a selected class of benchmarks. As a consequence even the same program can be judged to be more intelligent, when ran on better hardware or when given more time to execute. This competitive view settles; international competitions in data mining (e.g. KDD-Cup), game playing (e.g. Olympiads), robotics (e.g. Robo-Cup), theorem proving (e.g. CADE ATP), and action planning (e.g. IPC) call for highly performant systems on current machines with space and time limitations.

11.2 Hierarchical Memory

Restricted *main memory* calls for the use of *secondary memory*, where objects are either scheduled by the underlying operating system or explicitly maintained by the application program.

Hierarchical memory problems (cf. Chapter 1) have been confronted to AI for many years. As an example, take the *garbage collector problem.* Minsky [551] proposes the first copying garbage collector for LISP; an algorithm using serial secondary memory. The live data is copied out to a file on disk, and then read back in, in a contiguous area of the heap space; [122] extends [551] to parallelize Prolog based on Warren's abstract machine, and modern copy collectors in *C++* [276] also refer to [551]. Moreover, garbage collection has a bad reputation for thrashing caches [439].

Access time graduates on current memory structures: processor register are better available than pre-fetched data, first-level and second level caches are more performant than main memory, which in turn is faster than external data on hard disks optical hardware devices and magnetic tapes. Last but not least, there is the access of data via local area networks and the Internet connections. The faster the access to the memorized data the better the inference.

Access to the next lower level in the memory hierarchy is organized in pages or blocks. Since the theoretical models of hierarchical memory differ e.g. by the amount of disks to be concurrently accessible, algorithms are often ranked according to sorting complexity $\mathcal{O}(\text{sort}(N))$, i.e., the number of block accesses (I/Os) necessary to sort N numbers, and according to scanning complexity $\mathcal{O}(\text{scan}(N))$, i.e., the number of I/Os to read N numbers. The usual assumption is that N is much larger than B, the block size. Scanning complexity equals $\mathcal{O}(N/B)$ in a single disk model. The first libraries for improved secondary memory maintainance are LEDA-SM [226] and TPIE[1]. On the other end, recent developments of hardware significantly deviate from traditional von-Neumann architecture, e.g., the next generation of Intel processors have three processor cache levels. *Cache anomalies* are well known;

[1] `http://www.cs.duke.edu/TPIE`

e.g. recursive programs like Quicksort often perform unexpectedly well when compared to the state-of-the art.

Since the field of improved cache performance in AI is too young and moving too quickly for a comprehensive survey, in this paper we stick to knowledge exploration, in which memory restriction leads to a *coverage problem*: if the algorithm fails to encounter a memorized result, it has to (re)-explore large parts of the problem space. Implicit exploration corresponds to explicit graph search in the underlying problem graph. Unfortunately, theoretical results in external graph search are yet too weak to be practical, e.g. $\mathcal{O}(|V|+\mathrm{sort}(|V|+|E|))$ I/Os for *breadth-first search* (BFS) [567], where $|E|$ is the number of edges and $|V|$ is the number of nodes. One additional problem in external *single-source shortest path* (SSSP) computations is the design of performant external priority queues, for which *tournament-trees* [485] serve as the current best (cf. Chapter 3 and Chapter 4).

Most external graph search algorithms include $\mathcal{O}(|V|)$ I/Os for restructuring and reading the graph, an unacceptable bound for implicit search. Fortunately, for sparse graphs efficient I/O algorithms for BFS and SSSP have been developed (cf. Chapter 5). For example, on planar graphs, BFS and SSSP can be performed in $\mathcal{O}(sort(|V|))$ time. For general BFS, the best known result is $\mathcal{O}\left(\sqrt{|V| \cdot \mathrm{scan}(|V|+|E|)} + \mathrm{sort}(|V|+|E|)\right)$ I/Os (cf. Chapter 4).

In contrast, most AI techniques improve internal performance and include refined state-space representations, increased coverage and storage, limited recomputation of results, heuristic search, control rules, and application-dependent page handling, but close connections in the design of internal space saving strategies and external graph search indicate a potential for cross-fertilization.

We concentrate on *single-agent search*, *game playing*, and *action planning*, since in these areas, the success story is most impressive. Single-agent engines optimally solve challenges like Sokoban [441] and Atomix [417], the 24-Puzzle [468], and Rubik's Cube [466]. Nowadays domain-independent action planners [267, 327, 403] find plans for very large and even infinite mixed propositional and numerical, metric and temporal planning problems. Last but not least, game playing programs challenge human supremacy for example in Chess [410], American Checkers [667], Backgammon [720], Hex [49], Computer Amazons [565], and Bridge [333].

11.3 Single-Agent Search

Traditional single-agent search challenges are puzzles. The "fruit-fly" is the NP-complete (n^2-1)-Puzzle popularized by Loyd and Gardner [325]. Milestones for optimal solutions were [672] for $n = 3$, [465] for $n = 4$, and [470] for $n = 5$. Other solitaire games that are considered to be challenging are

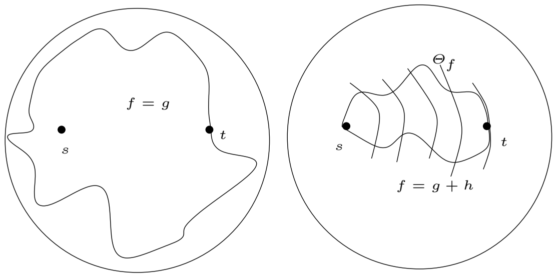


Fig. 11.1. The effect of heuristics in A* and IDA* (right) compared to blind SSSP (left).

the above mentionend PSPACE-hard computer games Sokoban and Atomix. Real-life applications include number partitioning [467], graph partitioning [291], robot-arm motion planning [394], route planning [273], and multiple sequence alignment [777].

Single-agent search problems are either given explicitly in form of a weighted directed graph $G = (V, E, w)$, $w : E \to \mathrm{I\!R}^+$, together with one start node $s \in V$ and (possibly several) goal nodes $T \subseteq V$, or implicitly spanned by a quintuple $(\mathcal{I}, \mathcal{O}, w, \mathit{expand}, \mathit{goal})$ of initial state $\mathcal{I}$, operator set $\mathcal{O}$, weight function $w : \mathcal{O} \to \mathrm{I\!R}^+$, successor generation function *expand*, and goal predicate *goal*. As an additional input, heuristic search algorithms assume an estimate $h : V \to \mathrm{I\!R}^+$, with $h(t) = 0$ for $t \in T$. Since single-agent search can model Turing machine computations, it is undecidable in general [611].

Heuristic search algorithms traverse the re-weighted problem graph. Re-weighting sets the new weight of (u, v) to $w(u, v) - h(u) + h(v)$. The total weight of a path from s to u according to the new weights differs from the old one by $h(s) - h(u)$. Function h is *admissible* if it is a lower bound, which is equivalent to the condition that any path from the current node to the set of goal nodes in the re-weighted graph is of non-negative total weight. Since on every cycle the accumulated weights in the original and re-weighted graph are the same, the transformation cannot lead to negatively weighted cycles. Heuristic h is called *consistent*, if $h(u) \leq h(v) + w(u, v)$, for all $(u, v) \in E$. Consistent heuristics imply positive edge weights.

The *A** algorithm [384] traverses the state space according to a cost function $f(n) = g(n) + h(n)$, where $h(n)$ is the estimated distance from state n to a goal and $g(n)$ is the actual shortest path distance from the initial state. Weighted A* scales between the two extremes; best-first search with $f(n) = h(n)$ and BFS with $f(n) = g(n)$. State spaces are interpreted as implicitly spanned problem graphs, so that A* can be cast as a variant of Dijkstra's SSSP algorithm [252] in the re-weighted graph (cf. Fig. 11.1). In case of negative values for $w(u, v) - h(u) + h(v)$ shorter paths to already expanded nodes may be found later in the exploration process. These nodes

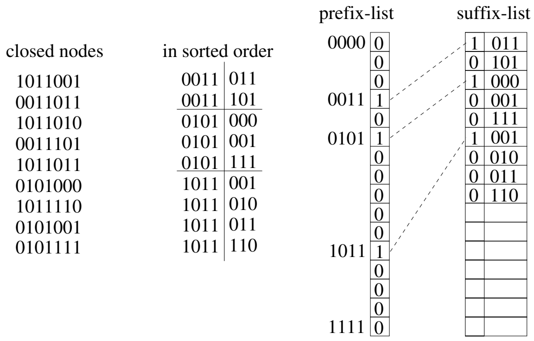


Fig. 11.2. Example for suffix lists with $p = 4$, and $s = 3$.

are *re-opened*; i.e. re-inserted in the set of horizon nodes. Given an admissible heuristic, A* yields an optimal cost path. Despite the reduction of explored space, the main weakness of A* is its high memory consumption, which grows linear with the total number of generated states; the number of expanded nodes $|V'| << |V|$ is still large compared to the main memory capacity of M states.

*Iterative Deepening A** (IDA*) [465] is a variant of A* with a sequence of bounded depth-first search (DFS) iterations. In each iteration IDA* expands all nodes having a total cost not exceeding threshold Θ_f, which is determined as the lowest cost of all generated but not expanded nodes in the previous iteration. The memory requirements in IDA* are linear in the depth of the search tree. On the other hand IDA* searches the tree expansion of the graph, which can be exponentially larger than the graph itself. Even on trees, IDA* may explore $\Omega(|V'|^2)$ nodes expanding one new node in each iteration. Accurate predictions on search tree growth [264] and IDA*'s exploration efforts [469] have been obtained at least for regular search spaces. In favor of IDA*, problem graphs are usually uniformly weighted with an exponentially growing search tree, so that many nodes are expanded in each iteration with the last one dominating the overall search effort.

As computer memories got larger, one approach was to develop better search algorithms *and* to use the available memory resources. The first suggestion was to memorize and update state information also for IDA* in form of a *transposition table* [631]. Increased coverage compared to ordinary hashing has been achieved by *state compression* and by *suffix lists*. State compression minimizes the state description length. For example the internal representation of a state in the 15-Puzzle can easily be reduced to 64 bits, 4 bits for each tile. Compression often reduces the binary encoding length to $\mathcal{O}(\log|V|)$, so that we might assume that for constant c the states u to be stored are assigned to a number $\phi(u)$ in $[1, \ldots, n = |V|^c]$. For the 15-Puzzle the size of the state space is $16!/2$, so that $c = 64/\lceil\log(16!/2)\rceil = 64/44 \approx 1.45$.

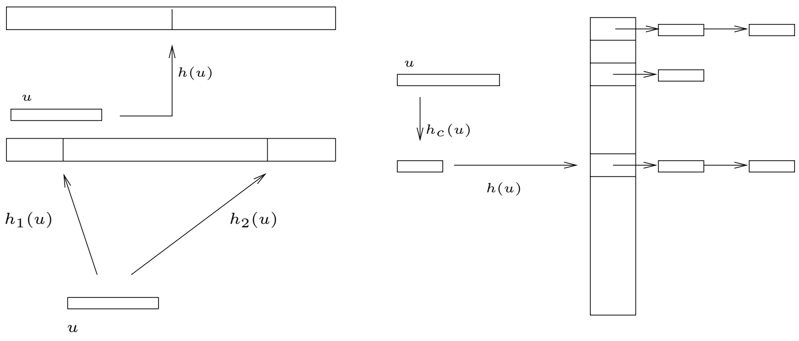


Fig. 11.3. Single bit-state hashing, double bit-state hashing, and hash-compact.

Suffix lists [271] have been designed for external memory usage, but show a good space performance also for internal memorization. Let $bin(\phi(u))$ be the binary representation of an element u with $\phi(u) \leq n$ to be stored. We split $bin(\phi(u))$ in p high order bits and $s = \lceil \log n \rceil - p$ low order bits. Furthermore, $\phi(u)_{s+p-1}, \ldots, \phi(u)_s$ denotes the prefix of $bin(\phi(u))$ and $\phi(u)_{s-1}, \ldots, \phi(u)_0$ stands for the suffix of $bin(u)$. The suffix list consists of a linear array P and of a two-dimensional array L. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in P, whereas the distinctive suffixes form a group within L. P is stored as a bit array. L can hold several groups with each group consisting of a multiple of $s+1$ bits. The first bit of each $(s+1)$-bit row in L serves as a *group bit*. The first s bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Fig. 11.2. The space performance is by far better than ordinary hashing and very close to the information theoretical bound. To improve time performance to amortized $\mathcal{O}(\log |V|)$ for insertions and memberships, the algorithm buffers states and inserts checkpoints for faster prefix-sum computations.

Bit-state hashing [224] and *state compaction* reduce the state vector size to a selection of few bits allowing even larger table sizes. Fig. 11.3 illustrates the mapping of state u via the hash functions h, h_1 and h_2 and compaction function h_c to the according storage structures. This approach of *partial search* necessarily sacrifices completeness, but often yields shortest paths in practice [417]. While hash compact also applies to A*, single and double bit-state hashing are better suited to IDA* search [271], since the priority of a state and its predecessor pointer to track the solution, are mandatory for A*.

In regular search spaces, with a finite set of different operators to be applied, *Finite state machine (FSM) pruning* [715] provides an alternative for duplicate prediction in IDA*. FSM pruning pre-computes a string acceptor for move sequences that are guaranteed to have shorter equivalents; the set of *forbidden words*. For example, twisting two opposite sides of the Rubiks cube in one order, has always an equivalent in twisting them in the opposite order.

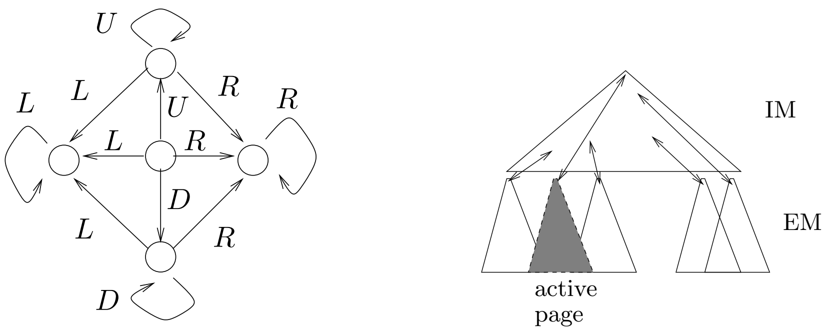


Fig. 11.4. The finite state machine to prune the Grid (left) and the heap-of-heap data structure for localized A* (right). The main and the active heap are in internal memory (IM), while the others reside on external memory (EM).

This set of forbidden words is established by hash conflicts in a learning phase prior to the search and converted to a substring acceptor by the algorithm of Aho and Corasick [20]. Fig 11.4 shows an example to prune the search tree expansion in a regular Grid. The FSM enforces to follow the operators *up* (U), *down* (D), *left* (L), and *right* (R) along the corresponding arrows reducing the exponentially sized search tree expansion with 4^d states, $d > 0$, to the optimum of d^2 states. Suffix-tree automata [262] interleave FSM construction and usage.

In route planning based on spatial partitioning of the map, the heap-of-heap priority-queue data structure of Fig. 11.4 has effectivly been integrated in a localized A* algorithm [273]. The map is sorted according to the two dimensional physical layout and stored in form of small heaps, one per page, and one being active in main memory. To improve locality in the A* derivate, *deleteMin* is substituted by a specialized *deleteSome* operation that prefers node expansions with respect to the current page. The algorithm is shown both to be optimal and to significantly reduce page faults counter-balanced with a slight increase in the number of node expansions. Although locality information is exploited, parts of the heap-of-heap structure may be substituted by provably I/O efficient data structures (cf. Chapter 2 and Chapter 3) like *buffer trees* [54] or *tournament trees* [485].

Most *memory-limited search* algorithms base on A*, and differ in the caching strategies when memory becomes exhausted. MREC [684] switches from A* to IDA* if the memory limit is reached. In contrast, SMA* [645] reassigns the space by dynamically deleting a previously expanded node, propagating up computed f-values to the parents in order to save re-computation as far as possible. However, the effect of node caching is still limited. An adversary may request the nodes just deleted. The best theoretical results on search trees are $O(|V'| + M + |V'|^2/M)$ node expansions in the MEIDA* search algorithm [260]. The algorithm works in two phases: The first phase fills the doubly-ended priority queue D with at most M nodes in IDA* man-

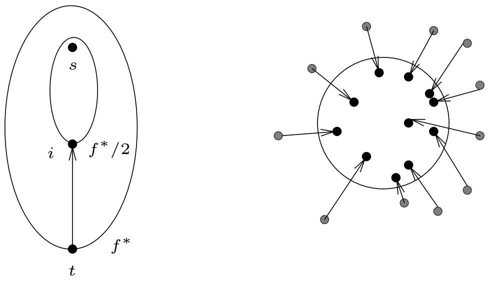


Fig. 11.5. Divide step in undirected frontier search (left) and backward arc look-ahead in directed frontier search (right).

ner. These nodes are expanded and re-inserted into the queue if they are *safe*, i.e., if D is not full and the f-value of the successor node is still smaller than the maximal f-value in D. This is done until D eventually becomes empty. The last expanded node then gives the bound for the next IDA* iteration. Let $E(i)$ be the number of expanded nodes in iteration i and $R(i) = E(i)-E(i-1)$ the number of newly generated nodes in iteration i. If l is the last iteration then the number of expanded nodes in the algorithm is $\sum_{i=1}^{l} i \cdot R(l-i+1)$. Maximizing $\sum_{i=1}^{l} i \cdot R(l-i+1)$ with respect to $R(1)+\ldots+R(l) = E(l) = |V'|$, and $R(i) \geq M$ for fixed $|V'|$ and l yields $R(l) = 0$, $R(1) = |V'| - (l-2)M$ and $R(i) = M$, for $1 < i < l$. Hence, the objective function is maximized at $-Ml^2/2 + (|V'| + 3M/2)l - M$. Maximizing for l yields $l = |V'|/M + 3/2$ and $O(|V'| + M + |V'|^2/M)$ nodes in total.

Frontier search [471] contributes to the observation that the newly generated nodes in any graph search algorithm form a connected horizon to the set of expanded nodes, which is omitted to save memory. The technique refers to Hirschberg's linear space divide-and-conquer algorithm for computing maximal common sequences [400]. In other words, frontier search reduces a $(d+1)$-dimensional search problem into a d-dimensional one. It divides into three phases: in the first phase, a goal t with optimal cost f^* is searched; in the second phase the search is re-invoked with bound $f^*/2$; and by maintaining shortest paths to the resulting fringe the intermediate state i from s to t is detected, in the last phase the algorithm is recursively called for the two subproblems from s to i, and from i to t. Fig. 11.5 depicts the recursion step and indicates the necessity to store virtual nodes in directed graphs to avoid falling back behind the search frontier, where a node v is called *virtual*, if $(v, u) \in E$, and u is already expanded.

Many external exploration algorithms perform variants of frontier search. In the $\mathcal{O}(|V'|+\text{sort}(|V'|+|E'|))$ I/O algorithm of Munagala and Ranade [567] the set of visited lists is reduced to one additional layer. In difference to the internal setting above, this algorithm performs a complete exploration and uses external sorting for duplicate elimination.

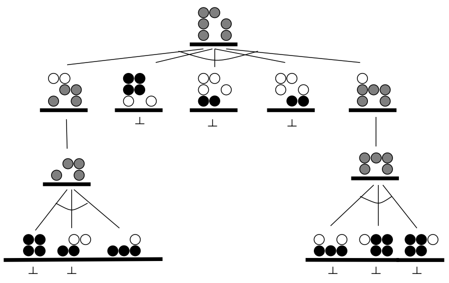


Fig. 11.6. Bootstrapping to build dead-end recognition tables.

Large *dead-end recognition tables* [441] are best built in sub-searches of problem abstractions and avoid non-progressing exploration. Fig. 11.6 gives an example of bootstrapping dead-end patterns by expansion and decomposition in the *Sokoban* problem, a block-sliding game, in which blocks are only to be pushed from an accessible side. Black ball patterns are found by simple recognizers, while gray ball patterns are inferred in bottom-up fashion. Established sub-patterns subsequently prune exploration.

Another approach to speed up the search is to look for more accurate estimates. With a better heuristic function the search will be guided faster towards the goal state and has to deal with less of nodes to be stored. Problem-dependent estimates may have a large overhead to be computed for each encountered state, calling for a more intelligent usage of memory. Perimeter Search [253] is an algorithm that saves a large table in memory which contains all nodes that surround the goal node up to a fixed depth. The estimate is then defined as the distance of the current state and the closest state in the perimeter.

11.4 Action Planning

Domain-independent action planning [29] is one of the central problems in AI, which arises, for instance, when determining the course of action for a robot. Problem domains and instances are usually specified in a general domain description language (PDDL) [312]. Its "fruit-flies" are *Blocks World* and *Logistics.* Planning has effectively been applied for instance in robot navigation [367], elevator scheduling [462], and autonomous spacecraft control [322].

A classical (grounded) *Strips planning problem* [303] is formalized as a quadruple $\mathcal{P} = <\mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G}>$, with $\mathcal{S} \subseteq 2^F$ being the set of states, 2^F being the power set of the set of propositional atoms F, $\mathcal{I} \in \mathcal{S}$, $\mathcal{G} \subseteq \mathcal{S}$, and $\mathcal{O}$ being the set of operators that transform states into states. A state $S \in \mathcal{S}$ is interpreted by the conjunct of its atoms. Operator $o = (P, A, D) \in \mathcal{O}$ is specified with its precondition, add and delete lists, $P, A, D \subseteq F$. If $P \subseteq S$, the result $S' \in \mathcal{S}$ of an operator $o = (P, A, D)$ applied to state $S \in \mathcal{S}$ is defined as $S' = (S \setminus D) \cup A$. *Mixed propositional and numerical planning* [312] take $\mathcal{S} \subseteq 2^F \times \mathbb{R}^k$, $k > 0$, as the set of states, *temporal planning* includes a special variable *total-time* to fix action execution time, and *metric planning* optimizes an additionally specified objective function. Strips planning is PSPACE-complete [167] and *mixed propositional and numerical planning* is undecidable [391].

Including a non-deterministic choice on actions effects is often used to model uncertainty of the environment. *Strong plans* [204], are plans that guarantee goal achievement despite all non-determinism. *Strong plans* are complete compactly stored state-action tables, that can be best viewed as a controller, that applies certain actions depending on the current state. In contrast, in *conformant planning* [203] a plan is a simple sequence of actions, that is successful for all non-deterministic behaviours. Planning with partial observability interleaves action execution and sensing. In contrast to the successor set generation based on action application, observations correspond to "And" nodes in the search tree [120]. Both conformant and *partial observable planning* can be cast as a deterministic traversal in *belief space*, defined as the power set $2^{\mathcal{S}}$ of the original one planning state space $\mathcal{S}$ [142]. Belief spaces and complete state-action tables are seemingly too large to be explicitly stored in main memory, calling for refined internal representation or fast external storage devices.

In *probabilistic planning* [142], different action outcomes are assigned to a probability distribution and resulting plans/policies correspond to complete state-action tables. Probabilistic planning problems are often modeled as Markov decision process (MDPs) and mostly solved either by policy or value iteration, where the latter invokes successive updates to Bellmann's equation. The complexity of probabilistic planning with partial observability is PP^{NP}-complete [568]. Different caching strategies for solving larger partial observable probabilistic planning problems are studied in [526], with up to substantial CPU time savings for application dependent caching schemes.

Early planning approaches in Strips planning were able to solve only small Strips problems, e.g., to stack five blocks, but planning graphs, SAT-encodings, as well as heuristic search have changed the picture completely. *Graphplan* [132] constructs a layered planning graph containing two types of nodes, action nodes and proposition nodes. In each layer the preconditions of all operators are matched, such that *Graphplan* considers instantiated actions at specific points in time. *Graphplan* generates partially ordered plans to exhibit concurrent actions and alternates between two phases: *graph extension* to increase the search depth and *solution extraction* to terminate the planning process. *Satplan* [452] simulates a BFS according to the binary encoding of planning states, with a standard representation of Boolean formulae as a conjunct of clauses.

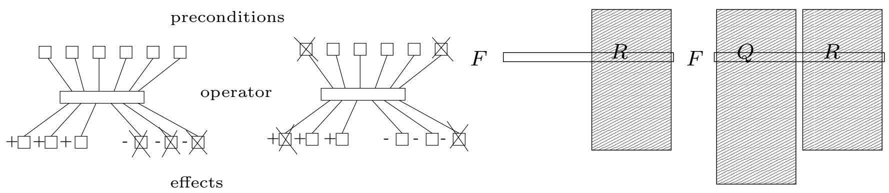


Fig. 11.7. Operator abstractions for the relaxed planning and the pattern database heuristic (left); single and disjoint PDB for subsets R and Q of all atoms F (right).

In current heuristic search planning, relaxed plans [403] and pattern databases (PDB) [266] turn out to be best (cf. Fig. 11.7). The *relaxed planning heuristic* generates approximate plans to the simplified planning problem, in which negative effects have been omitted in all operators, and is computed in a combined forward and backward traversal for each encountered state. The first phase determines a fix point on the set of reachable atoms similar to *Graphplan*, while the second phase greedily extracts the approximate plan length as a heuristic estimate.

Pattern databases (PDB) [232] are constant-time look-up tables generated by completely explored problem abstractions. Subgoals are clustered and for all possible combination of these subgoals in every state of the problem, the relaxed problem is solved without looking on the other subgoals. Fig. 11.7 illustrates PDB construction. The planning state is represented as a set of propositional atoms F, and the operators are projected to one or several disjoint subsets of F. The simplified planning problem is completely explored with any SSSP algorithm, starting from the set of goal states. The abstract states are stored in large hash tables together with their respective goal distance, to serve as heuristic estimates for the overall search. Retrieved values of different databases are either maximized or added. PDBs optimally solved the 15-Puzzle [232] and Rubik's Cube [466]. A space-time trade-off for PDB is analyzed in [405]; PDB size is shown to be inversely correlated to search time. Since search time is proportional to the number of expanded nodes N' and PDB-size is proportional to M, PDBs make very effective use of main memory. Finding good PDB abstractions is not immediate. The general search strategy for optimal-sized PDBs [405] applies to a large set of state-space problems and uses IDA* search tree prediction formula [469] as a guidance. A general bin-packing scheme to generate disjoint PDBs [469] is proposed in [263]. Recall that disjoint PDBs generate very fast optimal solutions to the 24-Puzzle.

Some planners cast planning as *model checking* [334] and apply *binary decision diagrams*(BDDs) [163] to reduce space consumption. BDDs are compact acyclic graph data structures to represent and efficiently manipulate Boolean functions; the nodes are labeled with Boolean variables with two outgoing edges corresponding to the two possible outcomes when evaluating a given assignment, while the 0- or 1-sink finally yield the determined re-

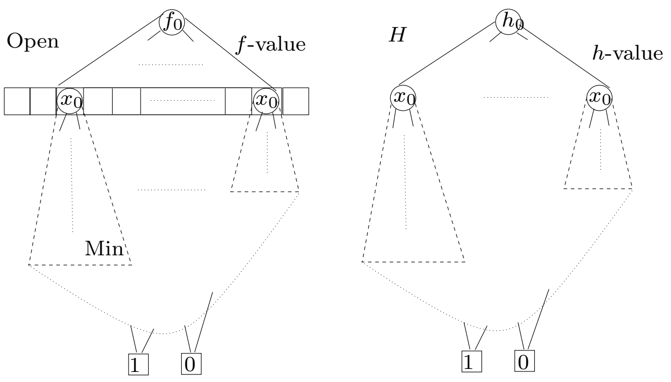


Fig. 11.8. Symbolic heuristic A* search with symbolic priority queue and estimate.

sult. *Symbolic exploration* refers to *Satplan* and realizes a BFS exploration on BDD representations of sets of states, where a state is identified by its characteristic function. Given the represented set of states $S_i(x)$ in iteration i and the represented transition relation T the successor set $S_{i+1}(x)$ is computed as $S_{i+1}(x) = \exists x\ (S_i(x) \wedge\ T(x,x'))[x/x']$. In contrast to conjunctive partitioning in hardware verification [544], for a refined image computation in symbolic planning disjunctive of the transition function is required: if $T(x,x') = \vee_{o\in\mathcal{O}} o(x,x')$ then $S_{i+1}(x) = \exists x\ (S_i(x) \wedge\ \vee_{o\in\mathcal{O}} o(x,x'))[x/x'] = \vee_{o\in\mathcal{O}}(\exists x\ (S_i(x) \wedge\ o(x,x')))[x/x']$.

Symbolic heuristic search maintains the search horizon *Open* and the heuristic estimate H in compact (monolithical or partitioned) form. The algorithm BDDA* [272] steadily extracts, evaluates, expands and re-inserts the set of states *Min* with minimum f-value (cf. Fig. 11.8). For consistent heuristics, the number of iterations in BDDA* can be bounded by the square of the optimal path length.

Although *algebraic decision diagrams* (ADDs), that extend BDDs with floating point labeled sinks, achieve no improvement to BDDs to guide a symbolic exploration in the 15-Puzzle [380], generalization for probabilistic planning results in a remarkable improvement to the state-of-the-art [292]. Pattern databases and symbolic representations with BDDs can be combined to create larger look-up tables and improved estimates for both explicit and symbolic heuristic search [266]. In conformant planning BDDs also apply best, while in this case heuristics have to trade information for exploration gain [119].

Since BDDs are also large graphs, improving memory locality has been studied e.g. in the breadth-first synthesis of BDDs, that constructs a diagram levelwise [412]. There is a trade-off between memory overhead and memory access locality, so that hybrid approaches based on context switches have been explored [772]. An efficiency analysis shows that BDD reduction of a decision diagram G can be achieved in $\mathcal{O}(\text{sort}(|G|))$ I/Os, while Boolean operator application to combine two BDDs G_1 and G_2 can be performed in $\mathcal{O}(\text{sort}(|G_1| + |G_2|))$ I/Os [53].

Domain specific information can control forward chaining search [78]. The proposed algorithm progresses first order knowledge through operator application to generate an extended state description and may be interpreted as a form of parameterized FSM pruning.

Another space efficient representation for depth-first exploration of the planning space is a *persistent search tree* [78], storing and maintaining the set of instantiations of planning predicates and functions. Recall that persistent data structures only store differences of states, and are often used for text editors or version management systems providing fast and memory-friendly random access to any previously encountered state.

Mixed propositional, temporal and numerical planning aspects call for plan schedules, in which each action is attached to a fixed time-interval. In contrast to ordinary scheduling the duration of an action can be state-dependent. The currently leading approaches[2] are an interleaved plan generator and optimal (PERT) scheduler of the imposed causal structure (MIPS), and a local search engine on planning graphs, optimizing plan quality by deleting and adding actions of generated plans governed by Lagrange multipliers (LPG).

Static analysis of planning domains [311] leads to a general efficient state compression algorithm [268] and is helpful in different planners, especially in BDD engines. *Generic type* analysis of domain classes [515] drives the design of hybrid planners, while different forms of symmetry reduction based on object isomorphisms, effectively shrink exploration space [313]: generic types exploit similarities of different domain structures, and symmetry detection utilizes the parametric description of domain predicates and actions.

Action planning is closely related to *error detection* in software [269] and hardware designs [629], where systems are modeled as state transition graphs of synchronous or asynchronous systems and analyzed by reasoning about properties of states or paths. As in planning, the central problem is overcoming combinatorial explosion; the number of system states is often exponential in the number of state variables. The transfer of technology is rising: *Bounded model checking* [124] exports the *Satplan* idea to error detection, *symbolic model checking* has lead to BDD based planning, while *directed model checking* [269] matches with the success achieved with heuristic search planning.

One approach that has not yet been carried over is *partial order reduction* [509], which compresses the state space by of avoiding concurrent actions, thus reducing the effective branching factor. In difference to FSM pruning, partial ordering sacrifices optimality, detects necessary pruning conditions on the fly, and utilizes the fact that the state space is composed by the cross product of smaller state spaces containing many local operators.

[2] www.dur.ac.uk/d.p.long/competition.html

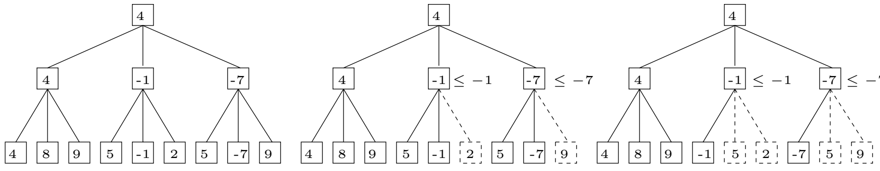


Fig. 11.9. Mini-max game search tree pruned by $\alpha\beta$ and additional move ordering.

11.5 Game Playing

One research area of AI that has ever since dealt with given resource limitations is game playing [666]. Take for example a *two-payer zero-sum game* (with perfect information) given by a set of states $\mathcal{S}$, move-rules to modify states and two players, called Player 0 and Player 1. Since one player is active at a time, the entire state space of the game is $\mathcal{Q} = \mathcal{S} \times \{0, 1\}$. A game has an initial state and some predicate *goal* to determine whether the game has come to an end. We assume that every path from the initial state to a final one is finite. For the set of goal states $\mathcal{G} = \{s \in \mathcal{Q} \mid goal(s)\}$ we define an evaluation function $v : \mathcal{G} \rightarrow \{-1, 0, 1\}$, -1 for a lost position, 1 for a winning position, and 0 for a draw. This function is extended to $\hat{v} : \mathcal{Q} \rightarrow \{-1, 0, 1\}$ asserting a game theoretical value to each state in the game. More general settings are *multi-player games* and *negotiable games* with incomplete information [628].

DFS dominates game playing and especially computer chess [531], for which [387] provides a concise primer, including mini-max search, $\alpha\beta$ pruning, minimal-window and quiescence search as well as iterative deepening, move ordering, and forward pruning. Since game trees are often too large to be completely generated in time, static evaluation functions assert numbers to root nodes of unexplored subtrees. Fig. 11.9 illustrates a simple mini-max game tree with leaf evaluation, and its reduction by *$\alpha\beta$ pruning* and move ordering. In a game tree of height h with branching factor b the minimal traversed part tree reduces from size $\mathcal{O}(b^h)$ to $\mathcal{O}(\sqrt{b^h})$. Quiescence search extends evaluation beyond exploration depth until a quiescent position is reached, while *forward pruning* refers to different unsound cut-off techniques to break full-width search. *Minimal window search* is another inexact approximation of $\alpha\beta$ with higher cut-off rates.

As in single-agent search, *transposition tables* are memory-intense containers of search information for valuable reuse. The stored move always provides information, but the memorized score is applicable only if the nominal depth does not exceed the value of the cached draft.

Since the early 1950s, from the "fruit-fly"-status, *Chess* has advanced to one of the main successes in AI, resulting in the defeat of the human-world champion in a tournament match. DeepThought [532] utilized IBM's Deep-Blue architecture for a massive-parallelized, hardware-oriented $\alpha\beta$ search scheme, evaluating and storing billions of nodes within a second, with a fine-tuned evaluation function and a large, man-made and computer-validated opening book.

Nine-Men-Morris has been solved with huge *endgame databases* (EDB) [326], in which every state after the initial placement has been asserted to its game-theoretical value. The outcome of a complete search is that the game is a draw. Note that the DeepBlue chess engine is also known to have held complete EDB on the on-chip memories.

Four Connect has been proven to be a win for the first player in optimal play using a knowledge-based approach [31] and mini-max-based *proof number search* (PNS) [32], that introduces the third value *unknown* into the game search tree evaluation. PNS has a working memory requirement linear in the size of the search tree, while $\alpha\beta$ requires only memory linear to the depth of the tree. To reduce memory consumption [32], solved subtrees are removed, or leveled execution is performed. PNS also solved GoMoku, where the search tree is partitioned into a few hundred subtrees, externally stored and combined into a final one [32]. *Proof Set Search* is a recent improvement to PNS, that saves node explorations for a more involved memory handling [564].

Hex is another PSPACE complete board game invented by the Danish mathematician Hein [161]. Since the game can never result in a draw it is easy to prove that the game is won for the first player to move, since otherwise he can adopt the winning strategy of the second player to win the game. The current state-of-the-art program *Hexy* uses a quite unusual approach electrical circuit theory to combine the influence of sub-positions (virtual connections) to larger ones [49].

Go has been addressed by different strategies. One important approach [562] with exponential savings in some endgames uses a divide-and-conquer method based on *combinatorial game theory* [116] in which some board situations are split into a sum of local games of tractable size. *Partial order bounding* [563] propagates relative evaluations in the tree and has also been shown to be effective in Go endgames. It applies to all mini-max searchers, such as $\alpha\beta$ and PNS.

An alternative to $\alpha\beta$ search with minimax evaluation is *conspiracy number search* (CNS) [537]. The basic idea of CNS is to search the game tree in a manner that at least $c > 1$ leaf values have to change in order to change the root one. CNS has been successfully applied to chess [516, 665].

Memory limitation is most apparent in the construction of EDBs [743]. Different to analytical machine learning approaches [673], that construct an explanation why the concept being learned works for positive learning examples — to be stored in operational form for later re-use in similar cases — EDBs do not discover general rules for infallible play, but are primary sources of information for the game-theoretical value of the respective endgame positions. The major compression schemes for positions without pawns use symmetries along the axes of the chess board.

EDB can also be constructed with symbolic, BDD-based exploration [89, 265], but an improving integration of symbolic EDBs in game playing has still to be given. Some combinatorial chess problems like the total number of 33,439,123,484,294 complete Knight's tours have been solved with the compressed representation of BDDs [513].

For EDBs to fit into main memory the general principle is to find efficient encodings. For external usage run-length encoding suits best for output in a final external file [491]. Huffman encodings [215] are further promising candidates. Thereby, modern game playing programs quickly become I/O bound, if they probe external EDBs not only at the root node. In checkers [666] the distributed generation of a very large EDBs has given the edge in favor to the computer. Schaeffer's checker program *Chinook* [667] has perfect EDB information for all checker positions involving eight or fewer pieces on the board, a total of 443,748,401,247 positions generated in large retrograde analysis using a network of workstations and various high-end computers [491]. Commonly accessed portions of the database are pre-loaded into memory and have a greater than 99% hit rate with a 500MB cache [481]. Even with this large cache, the sequential version of the program is I/O bound. A parallel searching version of *Chinook* further increased the I/O rate such that computing the database was even more I/O intensive than running a match.

Interior-node recognition [697] is another memorization technique in game playing that includes game-theoretical information in form of score values to cut-off whole subtrees for interior node evaluation in $\alpha\beta$ search engines. Recognizers are only invoked, if transposition table lookups fail. To enrich the game theoretical information, material signatures are helpful. The memory access is layered. Firstly, appropriate recognizers are efficiently detected and selected before a lookup into an EDB is performed [387].

11.6 Other AI Areas

An apparent candidate for hierarchical memory exploitation is *data or information mining* [644]; the process of inferring knowledge from very large databases. *Web mining* [474] is data mining in the Internet where *intelligent internet systems* [502] consider user modeling, information source discovering and information integration. Classification and clustering in data mining links to the wide range of *machine learning* [553] techniques with decision-tree and statistical methods, neural networks, genetic algorithms, nearest neighbor search and rule induction. *Association rules* are implications of the form $X \Rightarrow I$ with I being a binary attribute. Set X has support s, if s% of all data is in X, whereas a rule $X \Rightarrow I$ has confidence c, if c% of all data that are in X also obey I. Given a set of transactions D, the problem is to generate all association rules that have user-specified minimum support and confidence.

The main association rule induction algorithm is *AIS* [18]. For fast discovery, the algorithm was improved in *Apriori.* The first pass of the algorithm simply counts the number of occurrences of each item to determine itemsets of cardinality 1 with minimum support. In the k-th pass the itemset with $(k-1)$ elements and minimum support of phase $(k-1)$ are used to generate a candidate set, which by scanning the database yields the support of the candidate set and the k-itemset with minimum support. The running time of Apriori is $\mathcal{O}(|C| \cdot |D|)$, where $|D|$ is the size of the database and $|C|$ the total number of generated candidates. Even advanced association rule inference requires substantial processing power and main memory [757]. An example to hierarchical memory usage is a distributed rule discovery algorithm [189].

Case-based reasoning (CBR) [449] systems integrate database storage technology into knowledge representation systems. CBR systems store previous experiences (cases) in memory and in order to solve new problems, i) retrieve similar experience about similar situation from memory ii) complete or partial re-use or adapt the experience in the context of the new situation, iii) store new experience in memory. We give a few examples that are reported to explicitly use secondary memory. Parka-DB [706] is a knowledge base with a reduction in primary storage with 10% overhead in time, decreasing the load time by more than two orders of magnitude. Framer [369] is a disk-based object-oriented knowledge based system, whereas Thenetsys [609] is a semantic network system that employs secondary memory structure to transfer network nodes from the disk into main memory and vice versa.

Automated theorem proving procedures draw inferences on a set of clauses $\Gamma \rightarrow \Delta$, with Γ and Δ as multisets of atoms. A top-down proof creates a proof tree, where the node label of each interior node corresponds to the conclusion, and the node labels of its children correspond to the premises of an inference step. Leaves of the proof tree are axioms or instances of proven theorems. A *proof state* represents the outer fragment of a proof tree: the top-node, representing the goal and all leaves, representing the subgoals of the proof state. All proven leaves can be discharged, because they are not needed for further proof search. If all subgoals have been solved, the proof is successful. Similar to action planning, proof-state based automated theorem proving spans large and infinite state spaces. The overall problem is undecidable and can be tackled by user invention and implicit enumeration only. While polynomial decision procedures exists for restricted classes [538], first general heuristic search algorithms to accelerate exploration have been proposed [270].

11.7 Conclusions

The spectrum of research in memory limited algorithms for representing and exploring large or even infinite problem spaces is enormous and encompasses large subareas of AI. We have seen alternative approaches to exploit and memorize problem specific knowledge and some schemes that explicitly schedule external memory. Computational trade-offs under bounded resources become increasingly important, as e.g. a recent issue of *Artificial Intelligence* [381] with articles on recursive conditioning, algorithm portfolios, anytime algorithms, continual computation, and iterative state space reduction indicates. Improved design of hierarchical memory algorithms, probably special-tailored to AI exploration, are apparently needed.

Nevertheless, there is much more research, sensibility, and transfer of results needed, as two feedbacks of German AI researchers illustrate. For the case of external algorithms, Bernhard Nebel [578] mentions that current memory sizes of 256MB up to several GB make the question of refined secondary memory access no longer that important. This argument neglects that even by larger amount of main memory the latency gap still rises, and that with current CPU speed, exploration engines often exhaust main memory in less than a few minutes.

For the case of processor performance tuning, action execution in robotics has a high frequency of 10-20 Hz, but there is almost no research in improved cache performance: Wolfram Burgard [165] reports some successes by restructuring loops in one application, but has also seen failures for hand-coded assembler inlines to beat the optimized compiler outcome in another.

The ultimative motivation for an increased research in space limitations and hierarchical memory usage in AI is its inspirator, the human brain, with an hierarchical layered organization structure, including ultra short time working memory, as well as short and long time memorization capabilities.

Acknowledgments. The work is supported by DFG in the projects *heuristic search and its application to protocol verification* and *directed model checking.*

12. Algorithmic Approaches for Storage Networks

Kay A. Salzwedel*

12.1 Introduction

Persistent storage in modern computers is usually realized by magnetic hard disk drives. They form the last, and therefore the slowest level in the memory hierarchy. Disk drive technology is very sophisticated and complex, making an accelerated growth in disk capacity possible. Nowadays, a single of-the-self disk drive is capable of storing up to 180 GB and this number is doubled every 14 – 18 month. Nevertheless, this is not sufficient to manage the ever growing data volumes. The fast evolving processing power of modern computers, the global availability of information, and the increasing use of mixed-media contents demand for more flexible storage systems. The easiest way of providing almost unlimited storage capacity is the use of many disk drives in parallel. Unfortunately, a straightforward solution does not exploit the full potential of storage networks. As an example, suppose there is the need to add further disk drives to an existing system due to changed capacity demands. Without a redistribution of already stored data the capacity problem is solved but the addition of new (and usually faster) disks does not improve the overall performance of data accesses. There might exist very popular data on an 'old' disk resulting in a heavy load for that disk and leading to a bottleneck in the system.

When it comes to very large data sets the best access strategies are given by *External Memory Algorithms*, i.e. organize the data in such a way, that the number of I/O operations is minimized. In general, storage networks are on a different abstraction level and motivated by the growing amount of data, but external memory algorithms usually define a data layout and, hence, are one possibility to manage a number of disk drives in parallel. So, why not always use the data layouts imposed by external memory algorithms? Looking at the previous chapters two drawbacks of external memory algorithms become apparent. First, one has to find efficient and practical algorithms for all the sub-problems given in an application, and that is not always possible (see Chapter 4). And second, and more important, one has to be better then the underlying operating system. It is one of the key features of common operating systems to handle accesses to the memory hierarchy very efficiently. This is done in a general purpose way by asynchronous I/O operations, virtual

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

memory techniques, or the use of buffer caches. Hence, a possible implementation needs to bypass the disk access strategies of the operating system and implements its own external memory behavior. This is a rather complex task and very seldom acceptable. As far as we know, only large data base systems (see Chapter 14) implement their own data access model.

As an alternative, it is possible to use specialized kernels (like micro kernels or exokernals) which allow the easy implementation of any memory policy. But this approach has the major disadvantage that all the already implemented software has to be adapted to this kernel which might result in a very complex task. Because of this, researchers implementing external memory algorithms stick to widespread, open-source kernels like LINUX.

One might think that there should be libraries to ease the implementation of external memory algorithms. As mentioned in Chapter 15 there exists some support but these libraries suffer from the same problem. They need to provide data access algorithms which might be broken by the access strategy of the operating system.

This all adds up to the need for more general management of a assemblage of disk drives. A storage network is a collection of disk drives which are connected by a network. Each of the disks is characterized by its capacity and its bandwidth (see Fig. 12.1). The underlying topology or architecture of the network is unrestricted. Hence, we are not concerned with network properties like congestion or latency. It may vary from disks attached to an SCSI bus up to very complex networks like peer-to-peer (P2P) networks where the disks drives are associated with server nodes. In such an overlay network, a number of server work together to solve special tasks like searching. There is no central instance to control the behavior and server might enter or leave the network at any time.

Having a closer look at the above mentioned abstraction, two crucial observations become apparent. First, larger demands for capacity are easily solved by attaching more disks to the network. Nevertheless, the data need to be distributed in an intelligent way to provide an even utilization of disks. Only a good distribution can ensure the scalability in terms of data requests when the number of disk drives is increased. Second, a storage network offers the possibility of parallel data accesses provided that the data resides on different disk drives. This improves not only the access time but also the disk utilization because large data requests are served by more then one disk and, hence, are more balanced.

But using storage networks also imposes an inherent requirement – availability. Every disk has an estimated operation time during which an error is rather unlikely to occur. This entity is called *mean time to/between failure* (MTTF or MTBF, respectively) and for modern drives it is approximately 1,000,000 hours. But for a whole storage network, the MTBF decreases significantly. This is due to the fact, that it depends on the failure rate λ_{sys}

[693, 331] which is simply the sum of the failure rates λ_i of each participating disk[1].

The observations made above can be formalized to the *data distribution problem*. It has to give an answer to the following question: Where should the data be stored so that any data request can be answered as fast as possible? Furthermore, the data distribution has to ensure *scalability*. The more disks are in a storage network the more data requests should be answered or the faster should the data be delivered. This can only be done if the data is evenly distributed over all disks.

In recent years, modern storage systems move towards distributed solutions. Such systems usually consist of common components connected by standardized technology like Ethernet or FibreChannel. Hence, it is rather unlikely that all components have the same characteristic concerning capacity or performance. This leads to the *heterogeneity* problem. Not exploiting the different properties of each component results in a waste of capacity and performance.

As noted above, the data volume is annually growing. This implies a new challenge to storage networks because they should easily *adapt* to changing capacity/bandwidth demands. Allowing the addition of disks would be an easy task, if the space/access balance is ignored. One simply extends with new disks and maps new data to the added disks. This will result in a distribution that cannot increase the access performance for all data even when newer and faster disks are added. The only way to achieve both goals, space balance and the addition of disks, is to redistribute some data. This property is measured by the *adaptivity*. Naturally, the less data is redistributed the more adaptive is the underlying strategy.

The above discussion can be summarized by defining a number of requirements a storage network has to meet.

1. **Space and Access Balance:** To ensure good disk utilization and provide scalability the data should be evenly distributed over all disk drives. The data requests are usually generated by a file system above the storage network. Hence, the storage network has no knowledge about the access pattern and must handle any possible request distribution.
2. **Availability:** Because of the increased sensitivity to disk failures storage networks need to have some redundancy and implement mechanisms to tolerate the loss of data.
3. **Resource Efficiency:** Redundancy implies a necessary waste of resource. Nevertheless, systems should use all its resources in an useful way. Some problems, e.g. adaptivity, are easily solved if large space resources are available. But especially for large networks it is infeasible to provide these resources.

[1] This is only true under the assumption that the failure rate is constant which is equal to an exponentially distributed time to failure and the failures are independent of each other.

4. **Access Efficiency:** Stored data that cannot be accessed is lost data. Access efficiency defines *time* and *space* requirements needed to access an arbitrary data element. This property is of crucial importance when large networks are concerned.
5. **Heterogeneity:** Heterogeneity describes the capability to handle disks of different size or different bandwidth.
6. **Adaptivity:** The data volume is constantly growing and even generous planned system can quickly run out of space. This property measures the ability to adapt quickly to a changed situation like the addition of a new disk drive.
7. **Locality:** Locality describes the degree of communication required to answer any data request. One can distinguish centralized and distributed approaches. In this article, only distributed systems are of interest.

It seems impossible to meet all the above requirements optimally, especially heterogeneity and adaptivity are challenging if space/access balance and resource efficiency must be guaranteed. Nevertheless, we will introduce some randomized techniques that come very close to optimal solutions.

12.1.1 Organization of the Paper

In the next section we define a storage network and introduce all the relevant notations. The remainder of the paper will give an overview of techniques and algorithms used to achieve the above mentioned properties. We will focus on rather new techniques handling heterogeneity and adaptivity because these are the most challenging questions and they will become more important in the future. The last chapter will summarize the paper and give a short conclusion.

12.2 Model

A *storage network* is a collection of n disks labeled $D_1, \dots, D_n$, where each disk is characterized by a *capacity* C_i describing the storage capabilities and a *bandwidth* b_i stating the average data transfer time. For simplification reasons, lets assume that the maximal number of disks in one system is bounded by N. The capacity of the whole network can be expressed by $S = \sum_{1 \leq i \leq n} C_i$. The communication network may be arbitrary as long as all disks are connected. Network specific performance measures like congestion and latency are not considered. A storage network is called *homogeneous* (or *uniform*) if all disks have the same characteristic. Otherwise, the system is *heterogeneous* (or *non-uniform*).

The data to be stored is given by the set $U = \{1, \dots, p\}$ of *data elements* of equal size. The size of the universe out of which data elements are drawn is denoted by M. In general, we call all operations *data accesses* that read or

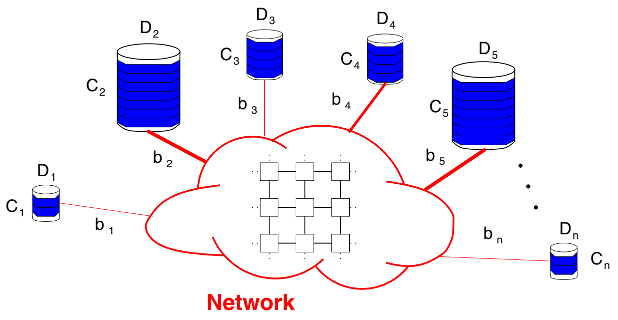


Fig. 12.1. An arbitrary storage network consists of a number of disks $D_1, \ldots, D_n$ and a connecting network. Each of the disks D_i is characterized by its capacity C_i and its bandwidth b_i.

write data elements. As mentioned in the introductory Chapter 1 the data in each storage hierarchy is accessed in chunks of size B. This modeling is even more accurate when it comes to disk drives because data access involves mechanical components. The time to access a data item is dominated by the time of moving the mechanical read/write head to the right track on the disk. The actual data access is very fast as soon as the head is settled and, furthermore, depends only on rotation speed (modern disk drives spin with 5k up to 15k rotation per minute). In our model, the size of the accessed data chunk B is insignificant. Hence, to avoid confusion, the data on disk is measured in *data units* of equal size (or size B to be precise). For simplification reason, assume w.l.o.g. that the size of a data element equals the size of a data unit. Furthermore, the data elements are ranked by an arbitrary order (e.g. the block address). Any of the p data units may be completely or partly *replicated*. Let $r \in \mathbb{R}$ denote the replication factor by which all data units are extended, e.g. a factor of 1 indicates that every data element has another copy.

The parallel access of data units is modeled by the *stripe* concept. A stripe is a collection of l consecutive data units (consecutive in the defined order of data elements). If the data units are partially replicated, the stripe also contains redundant data units (like parity units). The access to a stripe is fully parallel if all its units reside on different disks.

The number of data units and replication units to be placed in the system is defined by $m = \sum_{1 \le i \le p} i + r \cdot i$. A *data distribution strategy* for a storage network defines the replication, the mapping to the n disks, and the access strategy to data units.

For the description of randomized solutions we use the common balls-into-bins model [617]. Here, the data elements are associated with balls and the disks withs bins, respectively.

12.3 Space and Access Balance

One of the major goals of storage networks is the decrease of overall access time to data elements. In general, this time is strongly correlated to disk utilization if the network behavior is neglected. When a disk gets too many requests the track search time will be increased significantly resulting in disk head threshing in the worst case. Thus, the data elements and the data accesses must be balanced over all participating disks. By the *space balance* of a storage network we mean the property of always mapping even loads to the disks. This will also ensure a complete use of the systems capacity as long as the system is homogeneous. Furthermore, the space balance must guarantee that data units in a stripe are hosted by different disks to ensure parallel access and fault tolerance. The *access balance* property describes the capability of distributing the data accesses evenly over the disk drives. It ensures an even disk utilization, if all data elements are equally likely to be accessed. Space and access balance are mutually dependent.

Furthermore, the data access patterns are usually generated by the underlying file system (see Chapter 13). These patterns can only be predicted if there exists knowledge about the file system itself and the access pattern to each data file. Because this knowledge is hard to predict and may be subject to rapid changes a data distribution cannot take this into account. Hence, a good layout tries only to distribute the data elements as evenly as possible.

Nowadays, disk storage is rather cheap so the question may arise why one is concerned with space balance at all. There are many reasons to grapple with space balance. First of all, the trend of cheaper disks is easily compensated by the annually growing data volume. Furthermore, when the data is evenly balanced over the disk drives the number of accesses served in a given time can be increased because more disks are participating and can be kept busy.

The techniques to achieve space balance are quite simple. They can be grouped into deterministic and randomized approaches. The main deterministic technique is *striping* [435, 607, 650]. Here, the ith data element is put on disk $D_{i \bmod n}$ on position $i \operatorname{div} n$. Giving this simple scheme, full stripes (one data unit from each disk) can always be accessed in parallel. Furthermore, the total capacity can be exploited as long as all disks have the same capacity. There are a number of variations of disk striping. Especially in multimedia systems, it might be of interest not to stripe over the whole array because one wants to achieve a predefined accumulated bandwidth (e.g. the streaming rate of a media object). The natural extension to simple striping would be to group the disks into clusters of stripe length l and use simple striping within the cluster.

But how can the optimal length of a stripe and the size of a data unit be determined for a given demand? In [186] the authors synthesize rules for deriving these parameters. They could show that the size of a data unit only depends on the number of outstanding data accesses to be served and the average performance of a disk (positioning time and transfer rate).

There are a number of variations of simple striping which try to break the very regular structure of this approach. One generalization is *staggered striping* [118] in which a logical clustering of disk drives is implemented. Instead of arranging the data units into clusters of disk drives according to a needed bandwidth they are logically distributed over the whole network. Staggered striping defines a stride s by which consecutive data units are separated, e.g. $s = 1$ defines simple striping. This gains disk utilization if data elements of different bandwidth requirements are accessed.

A different approach is followed by *randomized distribution schemes* [27, 128, 660]. In general, a data unit b is placed on a randomly chosen disk D_i at a randomly chosen position. In practice, randomization is implemented by pseudo-random hash functions or pre-computed random distributions [659]. This makes not only randomization feasible but gives an efficient access scheme, too. A bound for the load of each disk can then be determined by the balls-into-bins model (e.g. see [617] for an overview). For the case that the number of data units is greater than $n \log n$ the load of each bin has the order of the mean value (with high probability). In particular, there exists a bin that receives $m/n + \Theta(\sqrt{m \ln n/n})$ balls with high probability (w.h.p.). Surprisingly, the multiple-choice case (each ball has the random choice of $d \geq 2$ bins) behaves differently [115], i.e. the load does not decline with increasing m. The authors have shown by using coupling Markov Chains that the load is independent from the number of balls m. Their simple algorithm works as follows. A number of m balls are sequentially processed. Each ball has the choice of $d \geq 2$ different bins and picks the one with the smallest load.

A slightly different approach is followed by Sanders [656]. His goal is to access any t disk units as fast as possible, i.e. with only t/n parallel access operations. Given a buffer space of size $\mathcal{O}(n/\epsilon)$ (up to $(1-\epsilon) \cdot n$ requests have to be handled in every access step) which hosts n writing queues and given that every data unit is replicated and put on a randomly chosen disk, any t data units can be accessed with only $\lceil t/n \rceil + 1$ access operations. This is done by deriving an optimal parallel schedule using maximum flow computations.

It can be said that the simple striping techniques are sufficient to achieve an optimal space and access balance as long as the system is static (the number of disks does not change over time) and the access pattern is balanced (no hot spots). Nevertheless, the randomized approaches are closing the gap on deterministic approaches in static settings fast and they possess many advantages when it comes to dynamic systems and irregular access patterns. Unfortunately, most of the randomized solutions work with ideal randomization which is impossible to implement. So, there is a large demand for practical implementations of randomized approaches. Due to their simple structure, deterministic solutions are currently favored in practice.

12.4 Availability

Availability describes the property to retrieve the data units even in the case of failed disk drives and, hence, lost data units. It is a crucial feature of storage networks because the probability of a failure in a collection of disks is scaled by its size [607].

The problem of availability can be solved by using *redundancy*. The easiest way is to store c copies of all data units like in an RAID 1 system (with $c = 1$, which is called mirroring in the storage community) [607] or the PAST storage server [641, 259]. Naturally, the distribution scheme has to ensure that the copies are hosted by different disks. The system can tolerate up to $c - 1$ failures and is still operational (only with degraded performance) so that the faulty disks can be replaced. The data that has been hosted by the faulty disk can then be rebuild using one of the redundant copies.

Because a full replication scheme results not only in the necessity to update all copies in order to keep the scheme consistent but also in a large waste of resources, more space efficient methods are sought. One of the common approaches is the use of *parity information* like in RAID level 4/5 [607, 185], Random RAID [128], Swarm [569, 385], and many video servers [118, 659, 725, 752, 27]. The idea behind is the use of redundant information to secure more then just one data unit. This can be done by deriving the bit-wise parity information (parity means taking an bit-wise XOR operation) of a whole stripe and storing it in an extra parity unit. Assuming the units reside on different disks, one disk failure can be tolerated. When a disk failure occurs the faulty disk has to be replaced and the unavailable data units must be rebuild by accessing all other units in its stripe. During this reconstruction phase, the next failure would be hazardous. Obviously, this reconstruction should be done as fast as possible aiming to minimize the maximum amount of data one has to read during a reconstruction [404]. This can be done by choosing a stripe length less then n. The resulting reconstruction load for each disk is given by the declustering ratio $\alpha = \frac{l-1}{n-1}$. If $\alpha = 1$ (like in the RAID 5 layout) each of every surviving disks participates in the reconstruction.

Using redundancy imposes a new challenge to the data distribution because it has not only to ensure an even distribution of data units but also an even distribution of parity information. The reason for this lies in the different access pattern for data units and parity units. First of all, we have to distinguish between read and write accesses. A read can be done by getting any l units. A write operation to any data unit (call it *small write*) has not only to access the written data unit but also the parity unit because the parity information after a write may have changed. It follows that the access pattern to the parity unit is different from all other data accesses. There are a number of different approaches to handle this situation. In RAID level 4 [607] the parity units are all mapped onto the same disk. As long as the whole stripe is written (call it *full write*) this does not impose any problems.

Unfortunately, this is not always the case. Small writes will put a lot of stress on the disk storing the parity information. A solution to this is provided by RAID level 5. Here, the parity units are spread over all participating disks by permutating the position of the parity unit inside each stripe. In the ith stripe the parity unit is at position $i \bmod l$.

As long as the number of disks is equal to the stripe length l this approach distributes the parity units evenly. The more general case, allowing the length to differ from n, calls for another distribution scheme like *parity declustering* [404]. The authors propose the use of complete and incomplete block designs. A *block design* is the arrangement of ν distinct objects into b blocks or tuples, each containing k elements, such that each object appears in exactly t blocks, and each pair of objects appears in exactly λ_p blocks. For a complete block design one includes all combinations of exactly k distinct elements from a set of ν objects. Note, that there are $\binom{\nu}{k}$ of these different combinations. Furthermore, only three variables are free because $b \cdot k = \nu \cdot r$ and $r(k-1) = \lambda_p(\nu - 1)$ are always true.

If we now associate the blocks with the disk stripes and the objects with disks the resulting layout (called block design table) distributes the stripes evenly over the n disks (i.e. ν equals the length of a stripe l and k equals the number of disks n). But such a block design table gives no information about the placement of the redundant information. To balance the parity over the whole array we build l different block design tables putting the parity unit in each of them at a different position in the stripe. The full (or complete) block design is derived by fusing these block design tables together. Obviously, this approach is unacceptable if the number of disks becomes large relative to the stripe length l, e.g. a full 41 disk array with stripe length 5 has 3,750,000 blocks. Hence, we have to look for small block designs of n objects with a block size of l, called balanced incomplete block designs (BIBDs), which do not need full tables to balance the parity units. There is no known way to derive such designs algorithmically and for all possible combinations of parameters a BIBD might not be known. Hall [376] presents a large number of them and some theoretical techniques to find them. In this case, the use of complete designs or the use of the next possible combination is recommended.

The approach of using parity information can be extended to tolerate more then one disk failure. As an example the EVENODD layout [130] can survive two disk failures. This is done by taking the regular RAID 5 layout and adding parity information for each diagonal on a separate disk. Such an approach can be even extended to tolerate t arbitrary simultaneous disk failures. Gibson et al. [390] showed that in this case, $t \cdot l^{1-\frac{1}{t}}$ disks worth of parity information are needed. Taking fast reconstruction into account, these schemes are only of theoretical interest.

The above mentioned techniques try to solve different problems, like availability and space balance. Combining them leads to new and more general data distributions. In [675, 674] the EVENODD layout is used together with the parity declustering technique. First, an appropriate BIBD is chosen. The data units in each stripe in this layout are then replaced by a complete EVEN-ODD layout. To do so, the data units of the BIBD are partitioned such that they will host a full column of an EVENODD layout. Such a data distribution can tolerate two disk failures and allows faster reconstruction compared to RAID 5 layouts. Furthermore, the authors reduce the problem of finding a data distribution scheme to the partitioning of the storage network into parity stripes (defining the length l and the replication factor r). This is done by using a network flow based method. The idea is further extended in [676] by defining approximately balanced distribution schemes. The major gain lies in the construction of such layouts. The authors propose three different construction methods: randomization, simulated annealing, and perturbation of exact layouts. They implemented a simulation environment and tested the behavior in degraded mode (when one disk is faulty) and reconstruction mode. As a result, the performance of approximately balanced layouts is very close to the performance of BIBDs based layouts.

The redundant information used to protect the storage network against disk losses can also be exploited to decrease the data access time significantly. In the case of $c \geq 1$ copies a simple protocol might work as follows. Suppose, there exists information about the queue length of the disk drives., e.g. gathered by probing. Because each data element has at least two copies, we always access the disk with the smallest queue length. Using the same arguments as in [617] it can be shown, that accesses are very evenly balanced.

In [128] the same technique is applied to parity extended random distribution schemes. As noted above, any $c - 1$ units suffice to reconstruct the complete stripe. Hence, a data access might choose those data units which reside on the least loaded disks. By similar arguments as in [617] this should improve the load substantially. Note, that some further processing is needed if the parity information is accessed. This approach is especially interesting for random distributions because it clips the tail of the disk access queue-length distribution. The author has implemented some simulations and gives evidence that such a strategy improves the response time significantly.

12.5 Heterogeneity

Heterogeneity describes the ability to handle disks with different characteristics efficiently, i.e. use the disk according to its capacity or its bandwidth. This problem is easy to solve if all other properties are neglected but is rather challenging if the space and access balance must also be guaranteed.

Heterogeneity is becoming more and more common in recent years. First of all, with the growth of the data volume, storage systems quickly run out of space. If a complete exchange of the system is unacceptable (due to the higher cost) one has to expand the existing configuration. It is rather unlikely that disks with the same characteristics can be found which results in a heterogeneous storage network. Furthermore, the larger the range of the underlying network (ranging form SCSI buses to LAN and WAN) the more likely heterogeneity becomes. As an example, look at clusters of workstations in a company. Suppose, there is a dedicated workstation pool that is connected via a fast network. As a result to modernization, the fast network may now span the whole company expanding the capabilities of the cluster to every single computer. To use the full potential of such an investment, the cluster must be capable of handling different workstations with different characteristics. Furthermore, it is rather unlikely that a cluster of workstations is completely exchanged by a new one. This too, leads to a heterogeneous system.

A further field of interest are *peer-to-peer* networks (P2P). Such a network consists of a changing number of servers (peers) which work together to manage special tasks like storing of, distributing of, and searching for data elements. These tasks are solved without a centralized server. Hence, each peer only knows its next neighbors (or some part of the network). When some new server wants to join the network, it contacts some server inside and joins its neighborhood. P2P networks usually use some established network to communicate. In other words, their are overlay networks that provide their services on top of basic communication. Because there is no central instance to supervise the joining or leaving of nodes, the general structure of a P2P network is very likely to be heterogeneous concerning capacity and bandwidth.

Until recently, little attention was paid to the heterogeneity problem, e.g. implementations of the RAID distribution schemes can work over different sized disks but they simply treat each disk as if it has the size of the smallest disk which results in a waste of space capacity.

Given a number of different disk drives, an easy and quick solution to the problem is the clustering of disks according to their characteristics [328]. Especially in multimedia systems, this is a common technique due to the bandwidth requirements in such systems. Within a cluster, the placement of data units can be done optimally with aforementioned techniques (namely striping). The big advantage of this approach is its simple extendibility. When new disks with new characteristics enter the system they simply form a new cluster. But the clustering approach has a main disadvantage. Adding new (and usually faster) disks does not improve the response time of all data units. Only the access to data elements hosted by newer disks will be significantly faster. This implies also that response time depends on the cluster a data element is hosted by. Assuming that the access pattern is known in advance, this could be used to avoid hot spots by putting popular data elements on faster disks. But unfortunately, the access pattern is usually unknown and may change significantly over time.

Another approach to handle heterogeneous disks was proposed by Sanders [655]. Here, the aim is to design optimal scheduling algorithms for arbitrary data requests. Each data element is replicated and randomly mapped to separated disks. Then, the scheduling problem can be transformed into a network flow problem. The requests are on the left hand side (source) and have an edge (with weight ∞) to all disks which store a copy of the requested data element. On the right hand side of the flow network (sink), the disks might be served by disk controllers and I/O-busses each of which is modeled by an edge with an appropriate weight to the disks/controllers it connects. Now, the flow problem can be solved resulting in a scheduling algorithm for the data requests. Heterogeneity can be introduced by adjusting the edge weights of the flow network according to the given capacities. The disadvantage of this approach lies in its batch-like behavior. For any collection of new requests a new flow problem must be solved. Especially for large systems the number of needed requests is considerable.

In the next four sections we will introduce new approaches that try to solve the heterogeneity problem.

12.5.1 AdaptRaid

AdaptRaid [221, 222] is a distribution scheme that extends the general RAID layout so that heterogeneity (corresponding to different capacities) can be handled. The basic idea is very simple. Larger disks are usually newer once and, thus, should serve more requests. Nevertheless, the placement of whole stripes should be kept as long as possible to gain from parallelism. So far, there are extensions to handle the case without data replication (RAID level 0) and with fully distributed parity information (RAID level 5).

AdaptRaid Level 0. The initial idea of AdaptRaid level 0 [221] is to put as many stripes of length $l = n$ on all disks as possible. As soon as some k disks cannot store any more data units, stripes of length $n - k$ are mapped onto the remaining $n - k$ disks. This process is repeated until all disks are full (see Fig. 12.2).

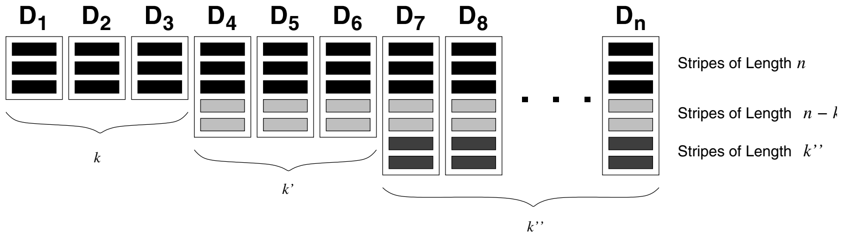


Fig. 12.2. The initial idea is to place as many stripes over all disks as there is capacity in the k smallest ones. When they saturate this strategy is repeat with the $n - k$ disks.

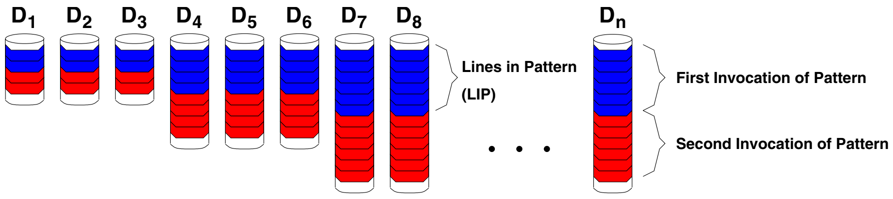

D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_n
Lines in Pattern (LIP)
First Invocation of Pattern
Second Invocation of Pattern

Fig. 12.3. Using the pattern defined by the initial idea, one distributes longer lines over the whole disk by fusing many invocations of the pattern on the disk.

This approach has the major drawback that the access to data stored in the upper part of the disk is faster then the access to data in the stored in the lower part of the disk drives because of the different stripe length. Furthermore, the assumption that newer disks are faster is in general not true. The following generalization will cope with both problems. First, we define the utilization factor $UF \in [0, 1]$ to be the ratio between the capacity of a disk and its bandwidth. This factor is determined on a per disk basis, where the largest disk has always $UF_i = 1$ and all other disks get a UF according to their capacity and relative to the largest one. These values should be set by the system administrator. To overcome the access problem, data patterns, which capture the overall capacity configuration, are defined. These patterns have a similar structure to the pattern of the initial idea, but are kept smaller such that many of them can fit on a single disk. The size of these patterns is measured by the second parameter – lines in a pattern (LIP). In general, this parameter is an indicator for the distribution quality of the layout and is measured by the number of units hosted on the largest disk. The overall data layout is defined by repeating these data patterns until the disks are full (see Fig. 12.3).

The access to data elements is twofold. In the first phase, the correct pattern has to be found which can easily be done because the size of a pattern (in number of units) is known. Then, the requested data unit is found by accessing the pattern itself. The space resources needed to access any data element is proportional to the size of the pattern.

The performance of AdapRaid0 was tested using a disk array simulator. The tests have been made with varying ratios of fast and slow disks[2]. Two different scenarios have been tested. First, the performance is compared to a RAID level 0. Naturally, AdaptRaid0 can shift the load from slower disks to the faster ones resulting in a performance gain between 8% and 20% for read accesses and 15% up to 35% for write accesses. The second scenario compares AdaptRaid0 to a configuration that contains only fast disks. This is reasonable because it gives an evaluation of the usefulness of keeping older disks in the system and therefore of the effect of parallel disk accesses. The

[2] The faster disks possess roughly doubled performance parameters, like average seek time, short seek time, and local cache.

experiments show that the performance gain is more pronounced in read accesses, e.g. in an array with 2 fast and 6 slow disks the read accesses can be answered 10% faster then in a fast disk only configuration.

It can be concluded that the exploitation of heterogeneity gains a significant performance speed-up.

AdaptRaid Level 5. AdaptRaid level 5 [222] is the heterogeneous variant of RAID level 5. It uses the same simple idea as AdaptRaid0, mainly to put more data units on larger and usually faster disks. But here, the distribution problem is more complex because to guarantee a fast reconstruction time the parity information must to be evenly distributed, too.

Following the same approach as in AdaptRaid0 and keeping the parity distribution of RAID 5 will result in a small write problem. As can be seen in Fig. 12.4, such a layout has stripes of different length. For the distribution scheme this is not a problem, but when it comes to performance the situation is different. Note, that the parity unit needs to be updated every time a data unit is written. Hence, file systems (see Chapter 13) should group the data requests such that full stripes could be written as often as possible. Such a technique will avoid many small writes and works well when the stripe size is fixed. Therefore, such optimizations would be useless for the proposed data layout.

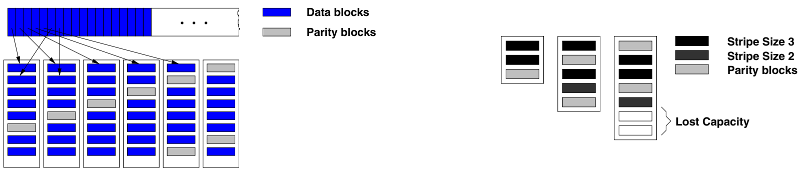


Fig. 12.4. The left hand side shows the regular RAID 5 layout. Parity units are distributed over the whole array by letting every stripe start on a different disk in a round-robin manner. The right hand side shows the initial layout in AdaptRaid5. Here, the RAID 5 strategy is applied on stripes of varying size.

This problem can be avoided by restricting the stripe length of each stripe to integer divisors of the length of the largest stripe (see the leftmost layout in Fig. 12.5). This leads to a heterogeneous layout which may waste some capacity. But a careful distribution of the free units can significantly improved this scheme. First, the layout should be independent of the number of disks still participating. This can be achieved by letting each stripe start on a different disk (see the center layout of Fig. 12.5). The nice effect of this transformation is the even distribution of the wasted free space over all disks instead of concentrating them on a few disk drives. Now, we can get rid of the wasted capacity by using a 'Tetris' like algorithm. The holes of free units are just filled with subsequent data units from stripes further below (data units

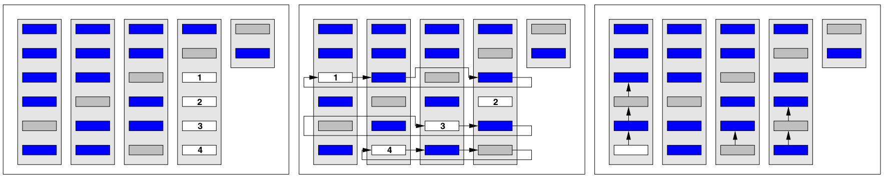


Fig. 12.5. The left figure shows a placement, so that the length of any stripe is a divisor of the length of the largest stripe. Clearly, some disk may have unused capacity. In the next step, the free space is distributed over all disks of this stripe length as shown in the center. This opens the possibility to apply an 'Tetris'-like algorithm. The free spaces are erased by moving the subsequent data units on each disk upwards. The resulting layout is depict in the right figure. The arrows indicate the movement of data units.

form these stripes 'move' upwards). Because the free spaces are distributed over all disks whole stripes can be added at the end of the pattern. Still the layout is regular and only the size of one pattern is need to locate any data unit (see the rightmost picture in Fig. 12.5). As in the AdaptRaid0, we will generalize the solution by defining two parameters. As before, UF is the utilization factor describing the load of a disk related to the largest disk. Similar to LIP the parameter SIP – stripes in pattern – controls the distribution of pattern over the whole array.

For the practical evaluation, the same simulator as for AdaptRaid0 has been used. The new approach was tested against a RAID 5 and a homogeneous configuration of only fast disk[3]. Obviously, both approaches waste some capacity in one way or the other. The performance was measured for reads, small writes and full writes (writing a full stripe). It was observed [222], that AdaptRaid5 is almost always the best choice and scales gradually with an increasing number of fast disks. Only when large chunks of data are read, the method is outperformed by the homogeneous configuration of only fast disks. This is due to the fact that the slower disks cannot deliver the data units appropriately. When full stripes are read they are the bottleneck of the system. Nevertheless, when simulations of real workloads are used, the performance gain of the new approach can be around 30% if only half of the disks are fast disks.

12.5.2 HERA – Heterogeneous Extension to RAID

A different kind of heterogeneity is used in the disk merging approach [783]. Here, the aim is not only to provide heterogeneous bandwidth distributions but also to increase the reliability of the system.

The disk merging approach constructs a homogeneous, logical view out of a physical collection of disks. Each physical disk D_i is partitioned into $k_i \in \mathbb{R}$ equal sized *extents*. These extents are arranged into G parity groups – the

[3] In the experiments, the number of fast disks varies from 2 to 9 disks.

logical view of the system. If the parity groups are fused together they form a homogeneous system where the stripe length l is the number of extents in a parity group. It is possible that an extent gets capacity and bandwidth from different physical disks. The data distribution scheme simply stripes the data units over the parity groups (see Fig. 12.6).

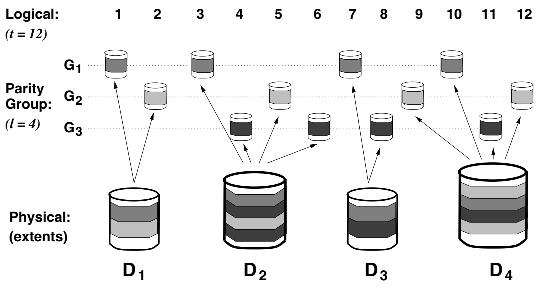


Fig. 12.6. HERA divides the capacity of a disk into logical extents 1, ... , D^t. Each physical disk D_i has capacity for k_i many logical disks. The logical disks are arranged in G parity groups each of which appears homogeneous.

How many parity groups are possible? In order to ensure the fault tolerance of the system enforced by the used striping strategy, each extent in every parity group has to come from a different physical disk. Let D^t denote the number of logical extents in the system. Then, the number of parity groups has to fulfill the following inequality:

$$G \leq \left\lfloor i \frac{D^t}{k_i} \right\rfloor \qquad 0 \leq i \leq n$$

To show the reliability of the system, the behavior is modeled by different processes like the failure process and the reconstruction process. Because the mean time to failure of any disk can be estimated, the time of these processes can be estimated, too. Using a Markov model [331], the mean time to service loss (MTTSL) can be derived for such a heterogeneous system. It was shown that the reliability (measured in MTTSL) is only a factor of approximately 10 away from the best possible configuration, namely the clustering of identical disks.

Nevertheless, this approach still needs the experienced hand of an administrator and the performance strongly relies on his decisions.

12.5.3 RIO – Random I/O Mediaserver

The RIO Mediaserver [658, 659] was build as a generic multimedia storage system capable of efficient, concurrent retrieval of many types of media objects. It defines a randomized distribution strategy and supports real-time data delivery with statistic delay guarantees. The used randomized distribution scheme is very simple. The data units are placed on a randomly chosen disk at a randomly chosen position[4] (see Fig. 12.7). Heterogeneity may occur in different disk capacities and varying bandwidth properties. As noted before randomization ensures a good balance in the long run, especially when it is used in conjunction with replication. In addition, RIO also exploits redundancy to improve the short time balance by carefully scheduling the accesses to data units.

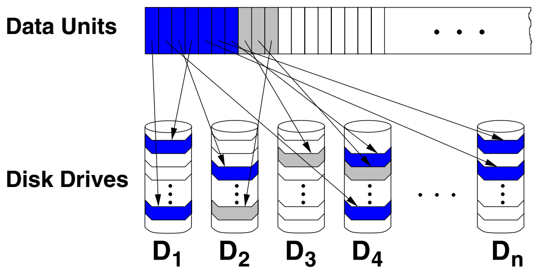


Fig. 12.7. RIO assigns every data unit b to a random position on a randomly chosen disk.

But randomization can also be used to handle heterogeneity. The initial idea is to put more data units on larger disks. This can be done as follows. Let $d_j = \frac{C_j}{S}$ be the relative (normalized) capacity of disk D_j. Note, that $\sum_{1 \le j \le n} d_j = 1$. Now d_j is used as the probability of putting a data unit on disk D_j. In the long run this would fill the disks according to their capacity. Nevertheless, there might be short time imbalances and the bandwidth problem is not taken into account.

How can both, capacity and bandwidth, be optimized simultaneously? The simplest way is a coupling of bandwidth and capacity by defining the bandwidth to space ratio $BSR = \frac{b_i}{C_i}$. The higher the BSP the faster the disk can deliver all its data units, hence, the more data units should be mapped to these disks. Obviously, the disks with low BSR present the bottleneck in the system. The major idea for a better balance is to shift some load from disks with low BSR towards disks with high BSR. This can be done using replication, and putting the copies on disks with high BSR. But how much redundancy is needed to sustain a given load? This question can be answered by using the following approach. First, the disks are grouped into t clusters according to their BSR and sorted in ascending order. Let N_i be the number of disks in cluster i. Further let $S_i = \sum_{1 \le j \le N_i} C_i$ and $B_i = \sum_{1 \le j \le N_i} b_j$ be the accumulated capacity and bandwidth in cluster i, respectively. Let S and

[4] The latter is only important for the theoretical analysis. A practical implementation can only ensure pseudo-random behavior.

B define the total capacity and bandwidth in the storage network and let $S'_i = S_i/S$ and $B'_i = B_i/B$ be the relative capacity and relative bandwidth of cluster i, respectively (see Fig. 12.8). The relative bandwidth space ratio can then be defined as $BS_i = B'_i/S'_i$. For a homogeneous system, $BS_i = 1$. The cluster with the lowest BS_i will be the most stressed cluster, not only because its disks have the slowest bandwidth but also because they host a large chunk of the data units. The number of data units that can be stored in the system is defined by $U = \frac{S}{1+r}$ where r denotes the replication rate. In formal terms, we want to use replication to shift the stress from clusters with $BS_i < 1$ to clusters with $BS_i > 1$. But how much replication is needed to sustain a certain maximal load $\lambda_{max} \leq B$?

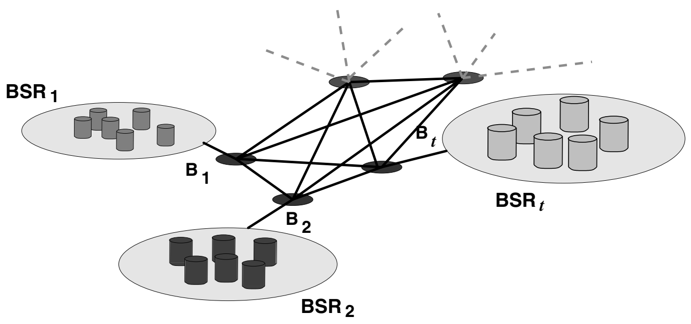


Fig. 12.8. RIO groups the disks according to their BSR. Each of the t groups is homogeneous in itself and has a collected capacity of S_i and a collective bandwidth of B_i.

To answer this question, we have to distinguish between two different cases. Let $\rho = \lambda_{max}/B$ be the average disk utilization. If $\rho = 1$ we have full utilization and if $\rho < 1$ the disks are only partially utilized. Lets consider full utilization first. It can only be achieved, if the load directed to a cluster i is in order of its bandwidth B_i. The maximum load that can be directed to any cluster i is reached when all data units stored in this cluster are distinct and all accesses to these data units are send to i. Hence, the maximal load can be computed by $\lambda_i = \lambda_{max} \cdot \frac{S_i}{U} = \rho \cdot B \cdot (1+r) \cdot S'_i$. To use all the available disk bandwidth on the system, we have to fulfill $\lambda_{i_{max}} \geq B_i$. This is satisfied if $BS_i \leq (1+r)$ for all $1 \leq i \leq m$. It follows that the replication rate can be expressed by $r \geq \max_i\{BS_i\} - 1 = BS_m - 1$.

In a similar way, one can determine the minimum replication rate to ensure partial bandwidth utilization. Furthermore, it can be shown that this replication rate is sufficient to achieve the desired utilization by defining a distribution scheme mapping replicated data units to clusters according to their BSR.

12.6 Adaptivity

One motivation for the heterogeneity in Sect. 12.5 was the addition of newer disks due to a growing space demand. But how does the introduction of new disks effect the still operational system? Obviously, if we want to have better performance when new disks are added, the data units must be evenly distributed over all disks. Hence, some portion of the existing data units has to be redistributed. The efficiency of this process is described by *adaptivity.* A distribution scheme is adaptive if it only has to redistribute a small number of data units in case of a changing system. Such a change can be the entering or failing of disks or the introduction of newly accessed data units. To capture heterogeneous requirements, we define the property of *faithfulness.* A distribution scheme is faithful if after a change in the system it can always ensure that the number of data units stored on a disk is according to its capacity/bandwidth requirement. In the homogeneous case each disk should get the same number of data units. For the more general case, lets denote by d_i the relative capacity of each disk D_i (relative to the total capacity S). Then a distribution scheme is faithful if it can ensure that any disk D_i gets $(d_i + \epsilon) \cdot m$ data units for an arbitrary small ϵ. Faithfulness uses the ϵ-term to allow randomized approaches to possess this property. Note, that m denotes the number of data units currently in the system.

The adaptivity can be measured by *competitive analysis.* For any sequence of operations σ that represent changes in the system, we intend to compare the number of (re-)placements of data units performed by the given scheme with the number of (re-)placements of data units performed by an optimal strategy that ensures faithfulness. A placement strategy will be called *c-competitive* if for any sequence of changes σ, it requires the (re-)placement of (an expected number of) at most c times the number of data units an optimal adaptive and perfectly faithful strategy would need.

Looking at the distribution techniques of the last chapters, e.g. the striping algorithm and the random distribution, none of them is adaptive. Consider a stripe of length n. Introducing a new disk literally changes the whole distribution. If only the $(n + 1)$ fraction of data units are redistributed, the system needs to keep track of the redistributions and, in the long run, to store every position of any data unit to access it. A similar argument can be used for random distributions.

12.6.1 Consistent Hashing

Consistent hashing [446, 504] was developed for distributed web caching but can also be applied here. The algorithm overcomes the drawback of conventional hashing schemes by defining a hash function that is consistent in case of a changing number of servers. We use the balls-into-bin analogy for the description. The algorithm works as follows.

Suppose we are given a random function f_B and a set of independent, random functions $g_1, \ldots, g_k$, where k may depend on n, the number of bins. The function $f_B : \{1, \ldots, M\} \to [0,1)$ maps the balls uniformly at random to real numbers in the interval $[0,1)$ and each function $g_i : \{1, \ldots, N\} \to [0,1)$ maps the bins uniformly at random to real numbers in the interval $[0,1)$. A ball i is assigned to the bin closest to it, regarding to this mapping and viewing $[0,1)$ as a ring, i.e. it is assigned to bin D_i that minimizes $\min_j \min[|f_B(i) - g_j(D_i)|, 1 - |f_B(i) - g_j(D_i)|]$.

From the proofs in [446] it follows that this strategy is faithful and 2-competitive (in the expected sense, if renaming of bins is not allowed) and that it can be implemented in a way that the location of a ball can be determined in expected constant time. It requires k to be in order of $\Omega(\log N)$ to ensure that the number of balls differs only by a constant factor from the expected number with high probability. This results in a space consumption of around $\mathcal{O}(n \log N)$ words (assuming that one word can hold $\log n$ bits and not including the hash functions).

This strategy does work well in an homogeneous system. One might think that it can be easily extended to cover the heterogeneous case by allowing disks with more capacity to have more random points in $[0,1)$. However, this would require $\Omega(\min[C_{\max}/C_{\min}, m])$ points to be faithful, where $C_{\max}$ is the maximum capacity and $C_{\min}$ is the minimum capacity of all disk. Thus, in the worst case, the number of points could be as much as $\Theta(m)$, violating severely the space consumption property. On the other hand, restricting the total number of points to something strictly below m cannot guarantee faithfulness under any capacity distribution (just consider two disks with capacities c/m and $(m-c)/m$ for some constant $c > 1$).

12.6.2 Cut-and-Paste Strategy

A two-phase approach is used by the cut-and-paste strategy [152]. We will stick to the balls-into-bins model (data units are represented by balls and disk drives are represented by bin, respectively) to describe the strategy.

Due to the fact that the number of data units is not fixed in advance[5] we define an address space of $\{1, \ldots, p\}$ for the balls. The parameter p is arbitrarily large but fixed (for current systems 2^{64} will suffice). The first phases maps all possible balls M to a $[0,1)$ interval by using a random hash function $h : M \to [0,1)$ (see the first picture in Fig. 12.9). This has the effect that any number of m balls are evenly distributed over the $[0,1)$ interval. Hence, it is sufficient to assign equal sized shares of the interval to the bins to ensure a certain even distribution. Initially, the hash value $h(b)$ denotes the height (in the $[0,1)$ interval) of that ball. The second phase now makes

[5] The adaptivity allows the a storage system to grow dynamically. Hence, the possible number of accessed data units is not restricted by the overall capacity of the network.

certain that all bins get an even share. This is done by cutting the $[0, 1)$ interval into ranges and these ranges (and therefore all the balls that fall into them) are mapped to bins by an *assimilation function*. If more then one range is assigned to a bin, they are fused together and adjusted such that the range on any bin is continuous and starts with 0. The height of the balls falling into the ranges are adjusted accordingly. Let $[0, 1/n]_i$ denote the range mapped to bin i when n bins are currently in the system. The assimilation function maintains the invariant that the (accumulated) ranges have the same size. Furthermore, it ensures that only a small portion of every range is remapped if the number of bins changes. The definition of the assimilation function is repetitive in the number of bins. It starts by assigning the interval $[0, 1)$ to bin 1. Then, the change from n to $n + 1$ bins is defined by cutting off the range $[1/(n + 1), 1/n]_i$ from every bin i with $i \in \{1, \ldots, n\}$ and fuse them together such that they are in reversed order (the range coming from bin 1 is topmost). This will be the new range for bin $n+1$ (see also second picture in Fig. 12.9). Due to the fact that each bin gets an equal share of the $[0, 1)$ interval it gets also an almost equal number of balls, accuracy depending only on the hash function. Furthermore, the number of balls moved to a new bin comes evenly from all other bins and is in the order of the balls needed to be moved to ensure an even distribution.

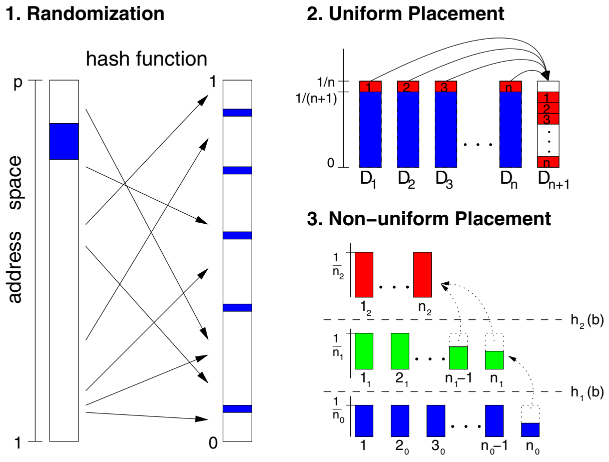


Fig. 12.9. The cut-and-paste strategy. First, the address space is hashed into a $[0, 1)$ interval. Then the uniform placement is done according to an assimilation function which defines the transition from n bins to $n + 1$ bins. Heterogeneity is achieved by defining a number of levels in which only disks with free capacity participate.

This strategy will not only ensure the fast adaptation to a changed number of bins but also provide an efficient way to access the balls. It suffices to trace the replacements performed by a ball for the consecutive addition of 1 to n bins. This can be done by determining the condition whenever a ball is replaced. In other words, when the height of the ball in the range will be greater then $1/(n+1)$. It can be shown that only $\log n + 2$ replacements will occur and therefore only a logarithmic number of operation is necessary to access any ball.

Unfortunately, this approach can not be easily extended to the heterogeneous case. The authors propose the introduction of levels in which only the bins with free capacity participate. But unfortunately, the competitive ratio depends logarithmically on the number of balls currently in the system which is rather inefficient.

Recently, both of the above strategies were extended to meet either the heterogeneity and the adaptivity requirements [153]. They propose two new strategies, called SHARE and SIEVE, that are perfectly faithful and 2-competitive. Both strategies are based on random hashing and need only a logarithmic number of steps to access any data element (w.h.p.).

12.7 Conclusions

The constantly growing amount of data calls for flexible and efficient storage systems. It has been shown that storage networks can keep up to this expectation. There is a large number of different techniques to exploit the inherent parallelism of such networks to gain better performance. Especially smaller systems like RAID arrays or multimedia data servers can be used very efficiently.

Nevertheless, the systems are growing rather rapidly. This introduces new problems like heterogeneity, adaptivity, or the resource efficiency considerations. For these problems there exist only a few approaches presented in this paper. Most of them have not been practically tested, yet.

In the last few year two very interesting application fields emerged. The first is the area of Storage Area Networks (SAN). The fast growing data volume especially in enterprises leads to very high cost of data maintenance. This cost can be significantly reduced if a centralized storage system is used. Nevertheless, such a storage system must allow for a growing data volume and they are very likely to be heterogeneous. Carefully managed storage networks are capable of providing these features.

The second new field of interest arepeer-to-peer networks (P2P). First of all, these networks are highly heterogeneous (with respect to space and connection requirements). Furthermore, availability is a big issue in these overlay networks. Data will be stored in them and the main task will be to retrieve the data. Furthermore, a P2P network is highly dynamic. Server will join and leave it without further notice calling to efficient adaptivity.

For all these reasons, techniques used in storage networks will become more popular in the near future.

13. An Overview of File System Architectures

Florin Isaila

13.1 Introduction

The ever increasing gap between processor and memory speeds on one side and disk systems on the other side has exposed the I/O subsystems as a bottleneck for the applications with intensive I/O requirements. Consequently, file systems, as low-level managers of storage resources, have to offer flexible and efficient services in order to allow a high utilization of disks.

A storage device is typically seen by users as a contiguous linear space that can be accessed in blocks of fixed length. It is obvious that this simple uniform interface can not address the requirements of complex multi-user, multi-process or distributed environments. Therefore, *file systems* have been created as a layer of software that implements a more sophisticated disk management. The most important tasks of file systems are:

- Organize the disk in linear, non-overlapping *files*.
- Manage the pool of free disk blocks.
- Allow users to construct a logical name space.
- Move data efficiently between disk and memory.
- Coordinate access of multiple processes to the same file.
- Give the users mechanisms to protect their files from other users, as for instance access rights or capabilities.
- Offer recovery mechanism for the case the file system becomes inconsistent.
- Cache frequently used data.
- Prefetch data predicted to be used in the near future.

In this chapter we investigate how different types of file systems address these issues. The next section describes the access patterns of sequential and parallel applications, as reported by several research groups. The access patterns are interesting, because they are often used to motivate file system design choices. Section 13.3 details some tasks of file systems, as enumerated above. We continue by explaining general issues of distributed file systems in the first part of Section 13.4. In the second part of Section 13.4, we address parallel (13.4.5), shared (13.4.6) and grid file systems (13.4.7) and show how file systems may handle operations for mobile computers(13.4.8). We summarize in Section 13.5.

13.2 File Access Patterns

Applications have various needs for accessing files. Therefore, in order to design a efficient file system it is very important to understand the file *access patterns*. A large number of studies have been devoted to this goal. In this section we will summarize some of their conclusions.

13.2.1 File Access Patterns of Sequential Applications

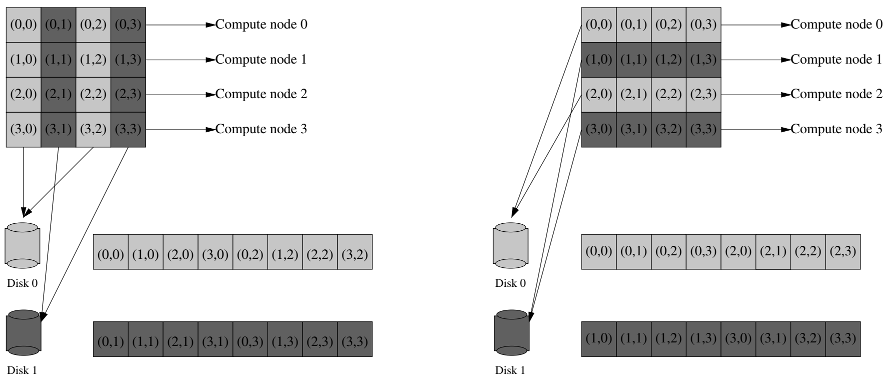
Several studies [663, 87, 598] analyzed the file access patterns of applications running on uniprocessor machines and accessing files belonging to either a local or a distributed file system. Their results were used either for implementing an efficient local or distributed file systems.

– Most files are small, under 10K [663, 87]. Short files are used for directories, symbolic links, command files, temporary files.
– Files are open for a short period of time. 75% of all file accesses are open less than 0.5 seconds and 90% less than 10 seconds [663, 87, 598].
– Life time of files is short. A distributed file system study [87] measured that between 65% and 80% of all files lived less than 30 seconds. This has an important impact on the caching policy. For instance, short-lived files may eventually not be sent to disk at all, avoiding unnecessary disk accesses.
– Most files are accessed sequentially [663, 598, 87]. This suggests that a sequential pre-fetching policy may be beneficial for the file system performance.
– Reading is much more common than writing. Therefore, caching can bring a substantial performance boost.
– File sharing is unusual. Especially the write sharing occurs infrequently [87]. This justifies the choice for a relaxed consistency protocols.
– File access is bursty [87]. Periods of intense file system utilization alternate with periods of inactivity.

13.2.2 Parallel I/O Access Characterization

Several studies of parallel I/O access patterns are available [582, 701, 695, 225]. Some of them focus on parallel scientific applications that are typically multiprocess applications, which typically access a huge amount of data.

Some of their results are summarized below, illustrated by the parallel access example from Figure 13.1. The figure shows two different way of physical partitioning of a two-dimensional 4x4 matrix, over two disks attached two different I/O nodes: (a) by striping the columns and (b) by striping the rows. The matrix is logically partitioned between four compute nodes, each process accessing a matrix row. For instance , this kind of access can be used by a matrix multiplication algorithm.



(a) Column striping (b) Row striping

Fig. 13.1. Parallel file access example

– Unlike in the uniprocessor applications, file sharing between several compute nodes is frequent. However, concurrent sharing between parallel applications of data within the file is rare [582, 701]. In the example, it is obvious that the file is shared among the four compute nodes. The accesses of the individual processes do not overlap, and are therefore, non-concurrent.
– The files are striped over several disks in order to increase the access throughput. In Figure 13.1(a) the matrix columns 0 and 2 reside on disk 0 and columns 1 and 3 on disk 1.
– Individual compute nodes often access the file non-contiguously. For instance, writing the first matrix line by compute node 0 from Figure 13.1(a) results in two non-contiguous disk accesses: (0,0) and (0,2) at disk 0, and (0,1) and (0,3) at disk 1. Non-contiguous accesses cause costly disk head seek movements, and therefore, have a negative influence on performance. However, with the different physical partitioning from Figure 13.1(b), the contiguous access of each process translates into contiguous disk access.
– The compute nodes frequently access a file in interleaved access patterns. This may result in high global inter-process spacial locality of data at I/O nodes [582], but also in a poor intra-process spatial locality [695]. In our example from 13.1(a), the accesses of the four compute nodes translate into interleaved accesses at disk 0 and 1. If they occur approximatively at the same time, the global disk access pattern is sequential, and therefore optimal. Otherwise, unnecessary seek times may drastically affect the application performance. This problem was addressed by different collective I/O operations designs and implementations [247, 475]. For the disk layout from Figure 13.1(b), the disk accesses of individual processes are not interleaved.
– In MIMD systems there is a high number of small I/O requests [582]. To a large extent this is the result of the non-contiguous and interleaved

accesses, as described earlier. In Figure 13.1(a), the process 0 access results in sending four small requests, two to each disk. However, the layout from 13.1(b) permits the row access with a single request.
– Parallel I/O is bursty, periods of intensive I/O activity alternate with computation [701]. It has been shown that the intensive I/O periods benefit from a collective approach rather than from one that treats the parallel requests independently [475, 247].
– Parallel applications accesses cause sometimes an unbalanced use of disks. For example, suppose that the four compute nodes in the Figure 13.1 show a bursty access by repeatedly executing the following two steps: reading one element from the file in the first step and performing some computation in the second step. In case (a), the two disks are used alternatively, and therefore, the parallel access of the compute nodes is serialized at the disks, with a direct impact on the speedup, according to Amdahl's law. Nevertheless, in the second case, the maximum disk parallelism degree may be achieved for each access, both disks being equally loaded.
– Some applications cause large contention of compute nodes' requests at the disks. In the extreme case, each compute node has to send requests to all disks, due to an unfortunate file data distribution. This may have a negative impact on both computing and I/O scalability: incrementing the number of compute nodes, may increase the number of requests of a collective access by the number of disks; similarly, incrementing the number of disks, may increase the number of requests by the number of compute nodes. In our example from Figure 13.1(a), in order to read the matrix, each of the four compute nodes has to access both disks. However, for the physical partitioning from Figure 13.1(b), each compute node has to send requests to a single disk.
– Parallel applications frequently use simple and nested strided access patterns. This denotes the use of multidimensional arrays, partitioned across the compute nodes [582].

13.3 File System Duties

The storage space has to be managed in a way that responds to the needs of different applications. In an operating system this is usually the role of the file system. This section outlines the most important duties of a file system.
File abstraction. A disk is used to store data belonging to different users or processes and having various contents. These justify the separation of the linear physical space of the disk into several logical units, called files. A file is a logical linear-addressable sequence of bytes that is mapped onto physical disk blocks.

A separate part of the disk is typically reserved for system information, called *metadata*. Each file is associated with a metadata structure. This structure contains the file attributes like file creation, modification and access times, owner, rights, mapping information, etc. In traditional UNIX system this structure is called *i-node*.

Mapping a file on the disk. There are various data structures used for mapping logical file offsets to physical disk addresses. The MS-DOS file system uses a linked list containing a direct logical to physical mapping. The structure is not scalable for random accesses because a linear search is used. In the traditional UNIX file systems, the logical blocks are multiples of disk blocks. The mapping structure consists of a tree whose root has thirteen children: the first ten contain direct pointers to disk blocks and the next three simple-, double- and triple-indirect pointers [79]. The disadvantage of this scheme is that large file access may result in several disk accesses only for mapping a file block.

Recently developed file systems like JFS [420], ReiserFS [571] and XFS [686] use more efficient schemes. First, instead of mapping blocks they map contiguous sequences of blocks called *extents*. Second, the extents are indexed by their file offset and organized in B-trees. The mapping consists of a tree descent having the goal of locating the disk address corresponding to a given file offset. For efficiency reasons, the first extents of a file are not kept in the B-tree, but are directly mapped onto disks. Therefore, small files can quickly find the disk addresses without a tree lookup. There are two more possible small file optimizations, which reduce the number of disk accesses. First, the metadata and data of small files may be put on the same disk block. Second, several small files may be placed on the same disk block.

Name space. Another important role of a file system is to provide a *naming* mechanism that allows users to differentiate among the set of files. Each file may be assigned a string of characters as a name. The names are organized into a tree structure by means of special files called *directories*. Directories represent the inner nodes of the tree and act as repositories for other directories or files. The files containing actual user data are the leaf nodes. The path from the root to the file's node uniquely identifies the file in the name space.

The UNIX file system stores the directory content into a linked list, linearly stored on the disk. Therefore, a linear scan has to be used for file name lookup. This affects the scalability for directories containing a large number of files. Recent file systems use B-trees for providing efficient lookup. XFS and JFS use a B-tree for each directory, whereas ReiserFS keeps a B-tree for the whole name space.

Disk space management. Files are normally stored on disks, therefore *disk space management* is an important part of a file system. Usually the disk space is managed by structures maintaining information about the free blocks. Traditional UNIX file systems keep a bitmap structure stored on the disk. A bit has the value '0' if the corresponding block is free and '1' otherwise. The main disadvantage of these scheme is that the bitmap size increases linearly with the size of the disk. These may cause a significant performance penalty for large disks, due to large times devoted to linearly scanning the bitmap for a free block.

Newer file systems use mainly two techniques for making disk space management more efficient: extents and B-tree structures. The main advantage of extents is that they allow finding several free blocks at a time. The space overhead for storing extents is lower than for bitmaps, as long as the file system is not very fragmented. B-trees allow a quicker lookup for a free block than lists, which have to be sequentially scanned. Extents and B-trees can be used in conjuction. Indexing by extent size makes possible to quickly allocate chunks of contiguous blocks of a certain size. Indexing by extent position allows to quickly locate blocks by their addresses. XFS is an example of a file system that employs all these techniques.

Caching. When performing a read from a file, the data is brought block-wise from the disk into the memory. A later file access may find parts of the data in memory. This technique is called *caching.* We will refer to the memory region used for this purpose in a file system as *file cache.* Caching improves the performance of the applications which exhibit temporal locality of access, i.e. in a program, once a block has been accessed, it is highly probable that it will be accessed again in the near future. Performance measurements show that this is the case with most applications. In the local file systems, caching is used to improve local disk access times, providing copies of the low-speed disks in the faster memory.

Prefetching. *Prefetching* is the technique of reading ahead from disk into cache data blocks probable to be accessed in the near future. The prefetching is motivated by several factors:

– the predictability of file access patterns of some applications
– the availability of file access hints
– the poor utilization of disk parallelism
– the large latencies of small non-sequential disk requests

On the other hand, prefetching may hurt performance by causing the eviction of valuable disk blocks from the cache. In this respect, it is very important to take prefetching decisions at the right time.

Prefetching can be done manually or automatically. The programmers can manually insert prefetching hints into their programs. This approach supposes that the source code is available. Additionally, this involves a high cost of program understanding and modification.

There are several automatic prefetching approaches.

– **Sequential read-ahead.** This is the simplest type of prefetching. It is based on the file access studies that have found out that the most frequent type of access is sequential. This type of automatic prefetching is implemented by the most file systems.

- **History-based prefetching.** The file system keeps information about past file accesses and tries to predict the future access pattern [476, 235, 357, 482, 500].
- **Statical analysis.** In this approach a compiler analyzes the program and inserts prefetching hints into the code [559, 212, 721, 602].
- **Hint-based.** The applications should generate informing hints that disclose their future accesses allowing the file system to make optimal global prefetch decisions [608, 170].
- **Speculative Execution.** The idle processor times due to blocking I/O are used for speculative pre-execution that generates prefetching hints [176].

Failure recovery. An important duty of file systems is to recover from failure. Traditional file systems used a recovery mechanism that searches the whole disk in order to correct corrupted blocks. As the storage devices are increasing in capacity this approach may let a file system unavailable for a significant time.

The *log-structured file systems* (LFS) [637] uses a combination between checkpointing and roll-forward in order to allow quick recovery. *Checkpointing* is the technique of saving the state of a system on a permanent storage device. The checkpointing may be done on a regular basis, when certain conditions are met, by request. The saved state can be subsequently used for a full recovery of the system at the last checkpoint. For more recovery accuracy a roll-forward technique may be used. *Roll-forward* consists of replaying all the changes recorded on the disk after the last checkpoint, such that the consistency of the system is preserved. LFS uses both recovery techniques. The main idea of LFS is to gather all the data and metadata updates into a memory segment called log and to store it to disk in a single operation when it becomes full or by need.

The disk in LFS is seen as a circular buffer of logs. When a file block is modified, its previous disk position is invalidated and the block is written anew on the log. This approach causes holes within the logs residing on the disks. Therefore, a garbage collector is used for rearranging the blocks on the disk in order to create new free segments for the log. The garbage collector becomes very complex, because it has to deal with both metadata and data scattered on the disk that has to be rearranged. In the *journalled file systems* [420, 571, 686], the design is simplified. The log is renamed journal and it is used only for metadata updates. Data updates have to be done manually or by another external checkpointing module. The advantage of these scheme resides in a much cleaner design by eliminating the need for garbage collection.

13.4 Distributed File Systems

The main goal of *distributed file systems* is to make a collection of independent storage resources appear as a single system. Distributed file systems have different characteristics depending on the resources they are built on or on their utilization goals. The following main factors influence the architecture and terminology of distributed file systems:

Storage attachment. The storage may be computer-attached or network-attached. In the computer-attached case, a single computer has exclusive access to the storage and acts as a server that serves disk requests, either directly from the disk or through its local file system. A distributed file system using network-attached devices is called *shared file system* [589], as described in Subsection 13.4.6.

Network connectivity. The connectivity plays an important role for the type of services a distributed system may offer. There are file systems for tightly-connected networks as those for supercomputers like IBM SP's or cluster of computers using high-performance networking technologies. In this case the file systems can be used by scientific applications that process a huge amount of data and are typically parallel. They need a high throughput of disk systems and a tight interconnection of their parallel running processes.

Distributed systems may also span several different administrative domains. In this case the network connectivity varies due to different reasons as resource heterogeneity or traffic variations. Security is also an important issue that has to be dealt with. These tasks are implemented by *grid file systems*, as we show in Subsection 13.4.7.

The ever increasing usage of mobile computers has brought the need to address changing network characteristics ranging from high connectivity to no connectivity. Subsection 13.4.8 explains specific issues related to file systems for mobile computers.

Parallel access. A distributed file system does not necessarily offer parallel file access. One such example is NFS. NFS servers serialize all the accesses from clients, even though they refer to different files or different parts of the same file. However, as seen in the access pattern section, parallel application performance may suffer drastically from such a limitation. This is one of the main reasons why *parallel file systems* decluster a file over different disks managed by independent servers. Some systems also distribute metadata in order to allow parallel access to different files or different parts of the directory tree. More details can be found in Subsection 13.4.5.

13.4.1 Location Transparency and Location Independence

A distributed file system typically presents the user a tree-like name space that contains directories and files distributed over several machines that can be accessed by a path. A file's location is *transparent* when the user cannot tell, just by looking at its path, if a file is stored locally or remotely. NFS [652] implements location transparency. A file location is *independent* if the files are accessed with the same path from all the machines using the distributed file system. It can be noticed that location independence entails location transparency, but not vice-versa. AFS [409] files are location-independent.

13.4.2 Distributed File System Architectures

The traditional distributed file system architecture was based on the *client-server* paradigm [652, 409, 145]. A file server manages a pool of storage resources and offers a file service to remote or local clients. A typical file server has the following tasks: stores file data on its local disks, manages metadata, caches metadata and data in order to quickly satisfy client requests, and eventually manages data consistency by keeping track of clients that cache blocks and updating or invalidating stale data.

Examples of file systems using a client-server architecture are NFS [652] and AFS [409]. A NFS server exports a set of directories of a local file system to remote authorized client machines. Each client can mount each directory at a specific point in its name tree. Thereafter, the remote file system can be accessed as if it is local. The mount point is automatically detected at file name parsing. AFS's Vice is a collection of file servers each of which stores a part of the system tree. The client machines run processes called Venus that cooperate with Vice for providing a single location-independent name space and shared file access.

A centralized file server is a performance and reliability bottleneck. As an alternative, a *server-less* architecture has been proposed [45]. The file system consists of several cooperating components. There is no central bottleneck in the system. The data and metadata are distributed over these components, can be accessed from everywhere in the system and can be dynamically migrated during operation. This gives also the opportunity of providing available services by transferring failed component tasks to remaining machines.

13.4.3 Scalability

Even though a distributed file system manages several resources in a network, it does not necessary exploit the potential for parallelism. We have seen that in a typical client-server architecture [652, 409] the data and metadata of a particular file are stored at a single server. Therefore, if a file system is accessed in parallel, even if there are several disks in the network, only the server disks are utilized. Additionally, when the number of clients increases, the file server becomes a bottleneck for both data and metadata requests.

Data *scalability* can be achieved by file replication or distribution. *File replication* allows the distribution of requests for the whole file over distinct disks. For instance, it is suitable for read-only access in a Web file system, in order to avoid hot spots [641]. The main drawback of replication is the cost of preserving consistency among copies. A file may also be *declustered* or striped over several disks by using for instance a software RAID technique [607]. The primary advantage of this approach is the high throughput due to disk performance aggregation.

13.4.4 Caching

Distributed file system caches have two main roles: the traditional role of caching blocks of local disks and providing local copies of remote resources (remote disks or remote file caches). In our discussion, we will use the attribute *local* for a resource (i.e. cache) or entity (i.e page cache) that is accessible only on a single machine and *global* for a resource that is accessible by all machines in a network.

Cooperative Caches. In order to make a comparison between caching in client-server and server-less architectures we consider a hybrid file caching model in a distributed file system. In this model "server" represents a file server for the client-server architecture and a storage manager, in the sense of disk block repository, for the server-less architecture. "Client" is also used in a broader sense meaning the counterpart of a server in the client-server architecture, and a user of the file system in the server-less architecture.

In this hybrid model we identify six broad file caching levels of the disk blocks, from the perspective of a client:

1. client local memory
2. server memory
3. other clients' memory
4. server disk
5. client disk
6. other clients' disks

In the client-server design only four levels are used: 1,2,4,5. For instance, in order to access file data, NFS looks up the levels 1, 2 and 4 in this order and AFS 1, 5, 2 and 4. The client has no way of detecting if the data is located in the cache of some other client. This could represent a significant performance penalty if the machines are inter-connected by a high-performance network. Under the circumstances of the actual technologies, remote memory access can be two to three orders of magnitude quicker than disk access. Therefore, cached blocks in other clients' memory could be fetched more efficiently.

Coordinating the file caches of many machines distributed on a LAN in order to provide a more effective global cache is called *cooperative caching.* This mechanism is very suitable to the cooperative nature of the server-less architecture. Dahlin and al. [237] describe several cooperative caching algorithms and results of their simulations.

There are three main aspects one has to take into consideration when designing a cooperative caching algorithm. First, cooperative caching implies that a node is able to look up for a block not only in the local cache but also in remote caches of other nodes. Therefore, a cooperative caching algorithm has to contain both *a local and a global lookup policy.* Second, when a block has to be fetched into a full file cache, the node has to chose a another block for eviction. The eviction may be directed either to the local disk or to the remote cache/disk of another node. Therefore, an algorithm has to specify both *a local and a global block replacement policy.* Finally, when several nodes cache copies of the same block, an algorithm has to describe *a consistency protocol.* The following algorithms are designed to improve cache performance for file system reads. Therefore, they do not address consistency problems.

Direct client cooperation. At file cache overflow, a client uses the file caches of remote clients as an extension of its own cache. The disadvantage is that a remote client is not aware of the blocks cached on behalf of some other client. Therefore, he can request anew a block he already caches resulting in double caching. The lookup is simple, each client keeps information about the location of its blocks. No replacement policy is specified.

Greedy forwarding. Greedy forwarding considers all the caches in the system as a global resource, but it does not attempt to coordinate the contents of these caches. Each client looks up the levels 1,2,3,4 in order to find a file block. If the client does not cache the block it contacts the server. If the server caches the block, it returns it. Otherwise, the server consults a structure listing the clients that are caching the block. If it finds a client caching the block it instructs him to send the block to the original requester. The server sends a disk requests only in the case the block is not cached at all. The algorithm is greedy, because there is no global policy, each client managing its own local file cache, for instance by using a local “least recently used” (LRU) policy, for block replacement. This can result in unnecessary data duplication on different clients. Additionally, it can be noticed that the server is always contacted in case of a miss and this can cause a substantial overhead for a high system load.

Centrally coordinated caching. Centrally coordinated caching adds coordination to the greedy forwarding algorithm. Besides the local file caches, there is a global cache distributed over clients and coordinated by the server. The fraction of memory each client dedicates to local and global cache is statically established. The client looks up the levels 1,2,3,4 in order to find a file block, in the same way as greedy forwarding does. Unlike greedy forwarding, centrally coordinated caching has a global replacement policy. The server keeps lists with the clients caching each block and evicts always the least recently used blocks from the global cache. The main advantage of centrally coordinated caching is the high global hit rate it can achieve due to the central coordinated replacement policy. On the other side it decreases the data locality if the fraction each client manages greedily is small.

N-Chance forwarding. This algorithm is different from greedy forwarding in two respects. First, each client adjusts dynamically the cache fraction it manages greedily based on activity. For instance, it makes sense that an idle client dedicates its whole cache to the global coordinated cache. Second, the algorithm considers a disk block that is cached at only one client as very important and it tries to postpone its replacement. Such a block is called a *singlet*. Before replacing a singlet, the algorithm gives it n chances to survive. A recirculation count is associated with each block and is assigned to n at the time the replacer finds out, by asking the server, that the block is a singlet. Whenever a singlet is chosen for replacement, the recirculation count is decremented, and, if it is not zero, the block is sent randomly to another client and the server is informed about the new block location. The client receiving the block places the block at the end of its local LRU queue, as if it has been recently referenced. If the recirculation count becomes zero, the block is evicted. N-Chance forwarding degenerates into greedy forwarding when $n = 0$. There are two main advantages of N-Chance forwarding. First, it provides a simple trade-off between global and local caches. Second, favoring singlets provides a better performance, because evicting a single is much expensive as evicting a duplicate, because a duplicate can be later found in other client's cache. The N-Chance forwarding algorithm is employed by xFS distributed file system.

Hash-distributed caching. Hash-distributed caching differs from centrally coordinated caching in that each block is assigned to a client cache by hashing its address. Therefore, a client that does not find a block in its cache is able to contact directly the potential holder, identified by hashing the block address, and only in miss case the server (lookup order: 1,3,2,4). The replacement policy is the same as in the case of centrally coordinated caching. This algorithm reduces significantly the server load, because each client is able to bypass the server in the first lookup phase.

Weighted LRU. The algorithm computes a global weight for each page and it replaces the page with the lowest value/cost ratio. For instance, a singlet is more valuable than a block cached in multiple caches. The opportunity cost of keeping an object in memory is the cache space it consumes until the next time the block is referenced.

Semantics of File Sharing. Using caching comes at the cost of providing consistency for replicated file data. Data replication in several caches is normally the direct consequence of file sharing among several processes. A consistency protocol is needed when *at least* one of the sharing processes writes the file. The distributed file systems typically guarantee a *semantics of file sharing.*

The most popular model is UNIX semantics. If a process writes to a file a subsequent read of any process must see that modification. It is easy to implement in the one-machine systems, because they usually have a centralized file system cache which is shared between processes. In a distributed file system, caches located on different machines can contain the copy of the same file block. According to UNIX semantics, if one machine writes to its copy, a subsequent read of any other machine must see the modification, even if it occurred a very short time ago. Possible solutions are invalidation or update protocols. For instance, xFS uses a token-based invalidation protocol. Before writing to its locally cached block copy, a process has to contact the block manager, that invalidates all other cached copies before sending back the token. Update or invalidation protocols may incur a considerable overhead. Alternatively the need for a consistency protocol can be eliminated by considering all caches in the distributed system as a single large cache and not allowing replication [220]. However, the drawback of this approach is that it would reduce access locality.

In order to reduce the overhead of a UNIX semantics implementation, relaxed semantics have been proposed. In the *session semantics*, guaranteed by AFS, all the modifications made by a process to a file after opening it, will be made visible to the other processes only after the process closes the file.

Transaction semantics guarantees that a transaction is executed atomically and all transactions are sequentialized in an arbitrary manner. The operations not belonging to a transaction may execute in any order.

NFS semantics guarantees that all the modification of a client will become visible for other clients in 3 seconds for data and 30 seconds for metadata. This semantics is based on the observation of access patterns studies that file sharing for writing is rare.

13.4.5 Parallel File Systems

Files are rarely shared by several uni-process applications as we have shown in subsection 13.2.1. Distributed file systems were often designed starting from this premise. For instance, file data and metadata in NFS are stored at a single server. Therefore, all parallel accesses to a certain file are serialized. However, in subsection 13.2.2, we have seen that the parallel applications typically access files in parallel. The distributed file systems have to be specialized in order to allow efficient parallel access to both data and metadata. In this case they are called parallel file systems.

File Physical Distribution. In order to allow true parallel access the files have to be physically stored on several disks. The nodes of supercomputers are typically split into *compute nodes* running the application and *I/O nodes* equipped with disks that store the files. It is also possible to use a node for I/O and computation (part-time I/O). The parallel file systems for clusters have also adopted this design [422, 423, 670].

The files are typically declustered over several I/O nodes or disks by simple striping [670, 422] or by more advanced declustering techniques [241, 415, 211, 581, 423]. As the characterization studies have shown, declustering has an important impact on performance. For instance, an unfortunate file declustering may turn parallel logical file access into sequential disk access as the example from subsection 13.2.2 has shown.

For instance, files in GPFS [670] and PVFS [422] are split into equally-sized blocks and the blocks are striped in a round-robin manner over the I/O nodes. This simplifies the data structure used for file physical distribution, but it can affect the performance of a parallel application due to a poor match between access patterns and data placement, as we have shown in subsection 13.2.2.

Other parallel file systems subdivide files into several subfiles. The file is still a linearly addressable sequence of bytes. The subfiles can be accessed in parallel as long as they are stored on independent devices. The user can either rely on a default file placement or control the file declustering. The fact that the user is aware of the potential physical parallelism helps him in choosing a right file placement for a given access pattern.

In the Vesta Parallel File System [211, 210], a data set can be partitioned into two-dimensional rectangular arrays. The nCube parallel I/O system [241] builds mapping functions between a file and disks using address bit permutations. The major deficiency of this approach is that all sizes must be powers of two. A file in Clusterfile parallel file system [423] can be arbitrarily partitioned into subfiles.

Logical Views. Some parallel file systems allow applications to logically partition a file among several processors by setting views on it. A *view* is a portion of a file that appears to have linear addresses. It can thus be stored or loaded in one logical transfer. A view is similar to a subfile, except that the file is not physically stored in that way. When an application opens a file it has by default a view on the whole file. Subsequently, it might change the view according to its own needs.

Views are used by parallel file systems (Vesta [211, 210] , PVFS [422], Clusterfile [423]) and by libraries like MPI-IO [548]. MPI-IO allows setting arbitrary views by using MPI datatypes. With Vesta, the applications may set views only in two dimensional rectangular patterns, which represents obviously a limitation. Multidimensional views on a file may be defined in PVFS. Like MPI-IO, Clusterfile allows for setting arbitrary views. An important advantage of using views is that it relieves the programmer from complex index computation. Once the view is set the application has a logical sequential view of the set of data it needs and can access it in the same manner it accesses an ordinary file.

Setting a view gives the opportunity of early computation of mappings between the logical and physical partitioning of the file. The mappings are then used at read/write operations for gathering/scattering the data into/from messages. The advantage of this approach is that the overhead of computing access indices is paid just once at view setting.

Views can also be seen as hints to the operating system. They actually disclose potential future access patterns and can be used by I/O scheduling, caching and pre-fetching policies. For example, these hints can help in ordering disk requests, laying out of file blocks on the disks, finding an optimal size of network messages, choosing replacement policies of the buffer caches, etc.

File Pointers. Another typical task of file systems is file pointer management. Unix file system assigns a file pointer to each open file. The pointers of the same file are always manipulated independently. As the parallel I/O characterization studies found out, the parallel applications often share files. If the file access is independent, Unix-style independent pointers can be used.

However, the parallel processes of an applications need sometime to cooperate in accessing a file. A shared file pointer may help implementing coordinated access. For instance, shared pointers can be used by a self-scheduling policy. Each process of a parallel application needs to access a portion of a file and then let the others know about this access through the shared pointer. A shared pointer is not proper for non-coordinated access, because it represents a centralized entity and therefore hinders parallelism when independent access is needed.

Collective I/O Operations. *Collective I/O* is a technique that merges several small I/O requests of several compute nodes. It is based on two well known facts. First, the overhead of sending small messages over the network is high. Second, non-contiguous disk access is inefficient. The collective I/O targets the minimization of the number of requests that are sent over the network and to the disk. There are two well known collective I/O methods: *two-phase I/O* [247] and *disk-directed I/O* [475].

The two phases of the first one are the shuffle and the I/O operation. In the shuffle phase the data is permuted by the compute nodes into a distribution that matches the physical distribution of data over the I/O nodes. In the second phase the data is redistributed according to the user defined distribution. The advantage of this scheme is that it decouples the logical distribution selected by the user from the physical distribution. Therefore, the data access phase performs always good, regardless of the logical distribution. The disadvantage is that the data is sent typically twice over the network, once in each phase and that additional memory is needed for permutation phase.

In the disk directed I/O the compute nodes send the requests directly to the I/O nodes. I/O nodes merge the requests and then arranges them in an order that minimizes disk head movement. The disk-directed I/O has several advantages over two-phase I/O: the disk performance is improved by merging and sorting requests, data is sent only once over the interconnect, communication is spread throughout disk transfer, not concentrated in the shuffle phase, no additional memory is needed at compute node for permuting the data.

13.4.6 Shared File Systems

Originally the storage devices were typically attached to computers. Direct disk access from remote hosts was possible only indirectly by contacting the machine they were attached to. Therefore, if the machine becomes unavailable due to crash, overloading or any other reason, the disks also become unaccessible. This may happen even though the unavailability reason had nothing to do with the disk to be accessed. The advent of network-attached storage (NAS) has allowed the separation of storage from computers. The disks can be attached to the network and accessed by every entitled host directly.

There are several advantages of this approach. First, the NAS allows the separation of file data and metadata [332]. A server running on one machine takes care of metadata, whereas the data is kept on NAS. The clients contact the server when opening a file and receive an authorization token, which can be subsequently used for accessing the disks bypassing the server. This makes the servers more scalable, because they are relieved from data transfer duties. Second, this may eliminate the need for expensive dedicated servers, because a lighter load allows a machine to be used also for other purposes. Third, the performance is boosted , because a client requests doesn't have to go through expensive software layers at the server (e.g. operating system), but it can contact the disks directly. Forth, in a server-attached disk, the data has to go thorough two interconnects: the server internal I/O bus and the external network. The server-attached disks reduce the number of network transits from two to one. Fifth, if a host in the network fails, the disks continue to be available for the remaining machines.

The challenge is how to manage concurrent access by several clients to the disks. If the disks are smart (have a dedicated processor), the concurrent access may be implemented at disks. Otherwise, an external lock manager may be needed. Each client has to acquire a lock for a file or for a file portion before effectively accessing the disks.

13.4.7 Grid File Systems

The Internet is a heterogeneous collection of resources, geographically distributed over a large area. It spawns several administrative domains, each of which running specific software. A single organization may have its resources distributed over the Internet and need a unified management, as for instance the meteorological stations. Or distinct organizations may need to cooperate for a common goal, for instance academic institutions that work in a common project.

Grid file systems [590] offer a common view of storage resources distributed over several administrative domains. The storage resources may be not only disks, but also higher-level abstractions as files, or even file systems or data bases. The grid file system gathers them into a unified name space and allows their shared usage.

Grid file systems are usually composed of independent administrative domains. Therefore, they must allow the smooth integration or removal of resources, without affecting the integrity of neither the individual independent domains nor the system as a whole.

13.4.8 Mobility

The increasing development of mobile computing and the frequent poor connectivity have motivated the need for weakly connected services. The file system users should be able to continue working in case of disconnection or weak connectivity and update themselves and the system after reintegration.

A mobile computer typically pre-fetches (hoards) data from a file system server in anticipation of disconnection [145]. The data is cached on the local disk of the mobile computer. If disconnection occurs the mobile user may continue to use the locally cached files. The modification of the files can be sent to the server only after reconnection. If several users modify the same file concurrently, consistency problems may occur. The conflicts may be resolved automatically at data reintegration by trying to merge the updates from different sources. If the automatic process fails, the data consistency has to be solved manually. However, as we have shown in the access pattern section, the sequential applications rarely share a file for writing. Therefore, it is likely that the conflicts occur rarely and the overhead of solving them is small.

13.5 Summary

This chapter provided an overview of different file system architectures. We showed the influence of I/O access pattern studies results on file system design. We presented techniques, algorithms and data structures used in file system implementations. The paper overviewed issues related to both local and distributed file systems. We described distributed file system architectures for different kinds of network connectivity: tightly-connected networks (clusters and supercomputers), loosely-connected networks (computational grids) or disconnected computers (mobile computing). File systems architectures for both network-attached and computer-attached storage were reviewed. We showed how the parallel file systems address the requirements of I/O bound parallel applications. Different file sharing semantics in distributed and parallel file systems were explored. We also presented how efficient metadata management can be realized in journaled file systems.

14. Exploitation of the Memory Hierarchy in Relational DBMSs*

Josep-L. Larriba-Pey**

14.1 Introduction

The exploitation of locality has proven to be paramount for getting the most out of today's computers. Both data and instruction locality can be exploited from different perspectives, including hardware, base software and software design.

From the hardware point of view, different mechanisms such as data victim caches or hardware prefetch [392] have been proposed and implemented in different processors to increase the locality of the reference stream, or to hide the latency of the accesses to distant memory levels.

From the base software point of view, compilers are able to extract locality with the help of heuristics or profiles of the applications, both from the instruction stream [627] and from the data references [392].

From the software design point of view, it is possible to design applications so that both the spatial and temporal locality of the data and instruction streams are exploited. However, while the programmer can influence the data access pattern directly, it is more difficult for her/him to influence the instruction stream.

Database management systems (DBMSs) are complex applications with special characteristics such as large executables, very large data sets to be managed, and complex execution patterns. The exploitation of locality in such applications is difficult because they access many different types of data and routines to execute a simple transaction. Thus, one simple transaction in a DBMS may exercise all the levels of the memory hierarchy, from the first level of cache to the external permanent storage.

Our objective is to understand how the exploitation of locality can be exercised in the execution kernel of a DBMS on computers with a single processor. We focus on the study of DBMS locality for the execution of large read queries coming from Decision Support Systems and Data Warehousing workloads. We analyse the different layers of the execution kernel of a DBMS, the base software and the hardware. Furthermore, we study not only the basic

* This work was supported by the Ministry of Education and Science of Spain under contracts TIC-0511/98 and TIC2001-0995-C02-01, CEPBA, the CEPBA-IBM Research Institute, and an IBM Research grant.

** To my wife Marta and son Ferran who fill my life with joy, and to my caring parents.

operations of DBMSs, such as joins or indexed data accesses, but also how they influence the execution of the database in general.

Aspects related to logging, locking or the On-Line Transaction Processing environment will not be considered, because the objective is to understand how read queries on large amounts of relational data can be enhanced with the exploitation of locality at different levels in a DBMS.

This chapter is organized in the following way. In Sect. 14.2, we start by considering what to expect from the chapter and the minimum knowledge assumed when writing it. In Sect. 14.3, we describe the internal structure of a Database Management System and how it executes read queries. In Sect. 14.4, we give evidence of locality in DBMSs, and in Sect. 14.5 we describe the basic software techniques that can be used to expose such locality. Later, in Sects. 14.6, 14.7, and 14.8, we explain how the techniques explained in Sect. 14.5 can be used to exploit locality in each of the horizontal DBMS layers. In Sects. 14.9 and 14.10, we give some insight into the the hardware and compilation issues. Finally, in Sect. 14.11, we summarize.

14.2 What to Expect and What Is Assumed

A crucial aspect in the improvement of large applications like DBMSs is the reduction of their execution time. One of the most important factors contributing to the execution time of applications is the distance between the processor and its memory. The memory hierarchy seeks to reduce the effect of this distance on the execution time by exploiting the locality of data and instructions.

Locality shows up in two different ways [392]. When an application tends to re-use data and instructions that have been used recently, there is temporal locality. When it re-uses data and instructions that are physically close-by, there is spatial locality.

However, locality does not necessarily show up naturally in applications for many different reasons: for example the use of complex data structures, complex program structure, bad programming habits, etc. When that happens, it is either necessary to expose locality by changing the program explicitly, or to hide the latency of the memory accesses.

The exploitation of locality can be done at different levels (programmer, compiler, data, instructions) and aims at the reduction of the number of accesses to main memory by changing the application so that the different cache levels capture more memory references. Hiding the latency can also be done at different levels but its principal aim is to execute as many instructions as possible during the idle time caused by a memory reference to a distant memory level.

In This Chapter, We Look into how the different layers of the execution kernel of a DBMS can be implemented to exploit memory locality in modern uni-processor computers. That is, we look into ways of reducing the number of times a data item has to be brought from any of the levels of the memory hierarchy (including main memory and disk) to a level closer to the processor.

We do not consider disk technology or how disks are implemented to exploit locality during their access to physical disk sectors. Neither we consider techniques that help to hide the latency of the memories. This is because hiding the latency does not fall into what we understand as algorithmic changes to improve locality.

In the survey, and wherever possible, we only give the number of data reads from the most significant levels of the memory hierarchy. We do so because data reads are usually in the critical path of operations (some operations have dependences on a read), while data writes may be delayed because they do not cause dependences.

Basic DBMS Concepts. We assume the reader has a basic knowledge of the Relational algebra operators like *Restriction, Projection, Join* etc.[1] Reading [694] may help to understand these concepts. We also assume the reader knows how these operations map onto their implementations. For instance, how a join can be implemented as a *Hash Join*, a *Nested Loop Join* or a *Merge Join*. For an understanding of the different implementations of relational operators we refer the reader to [347, 552].

To enable the reader to understand the implementation of a couple of join operators, we describe the Nested Loop Join and the Hash Join implementations. Let us suppose relations R and S with joining attributes $R.r$ and $S.s$ and a join operation $R \bowtie S$ with join condition $R.r > S.s$. The Nested Loop Join takes every record of relation R and checks its join attribute with each and every join attribute of the records of relation S returning as result the pairs of records that fulfil the join condition. Relation R is called outer relation and relation S is called inner relation because they are traversed as in the execution of a nested loop where the traversal of R is governed by the outer loop and the traversal of S is governed by the inner loop.

The execution time of the Nested Loop Join operation is proportional to the cardinality (number of records) of relation R (which we denote by $|R|$) times the cardinality of relation S, $|S|$.

For the Hash Join implementation we assume that the join condition is $R.r = S.s$[2]. The Hash Join implementation takes all the instances of at-

[1] The application of a *Restriction* operation on a relation eliminates all the records of that relation that do not fulfill the Restriction condition imposed. A *Projection* operation obtains a new relation with the same amount of records but only with the attributes desired. A *Join* operation on two relations R and S, $R \bowtie S$, and a join condition on one or more attributes of each of those relations, obtains as a result the pairs of records that fulfill the join condition.

[2] Given the nature of the algorithms, the join condition of a Hash Join can only be of the type $R.r = S.s$, while the Nested Loop Join condition can be of any type (larger than, smaller than or equal to, etc.). Therefore, the Hash Join can only be used in the case of equality or natural joins, and the Nested Loop Join can be

tribute, say, $R.r$ and builds a hash table using hash function H. This *build* phase has an execution time proportional to the cardinality of the build relation R. Using the same hash function H, all the instances of attribute $S.s$ are probed against the hash table. The execution time of this *probe* phase is proportional to the cardinality of the probe relation S.

14.3 DBMS Engine Structure

The basic structure of the DBMS query engine is simple in concept, and can be explained from the point of view of how a query is executed with the help of Figs. 14.1 and 14.2. The explanation in this section is based on [356] and the structure of the PostgreSQL DBMS [612].

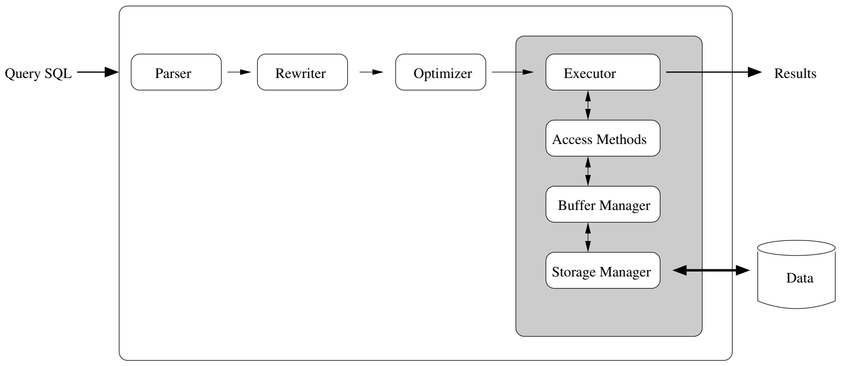
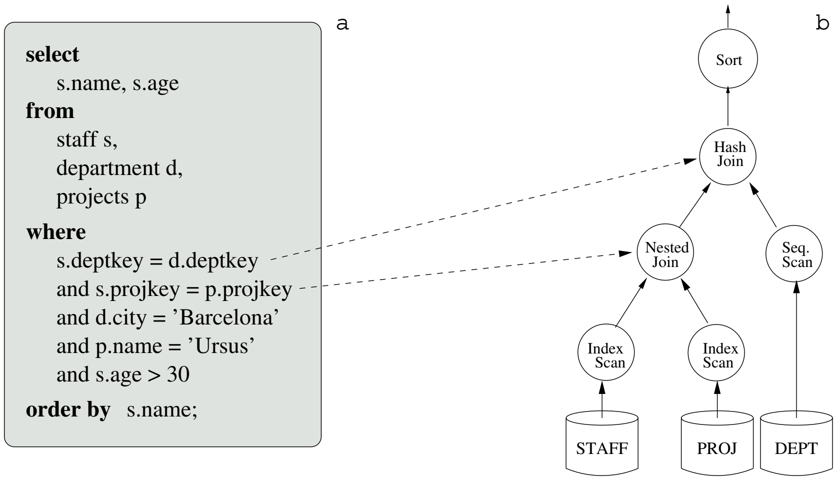


Fig. 14.1. Layered structure of the query engine of a Database Management System. Layers involved in the execution of read only queries.

When an *Structured Query Language* (SQL) query (Fig. 14.2.a) is launched to the engine, the Parse-Rewrite-Optimization Engine (Fig. 14.1) takes care of it. First, the query is parsed for syntactic correctness. Then, the SQL statement is rewritten generating a new SQL version of the query in canonical form. Finally, the optimizer generates the query execution plan. The *plan* (Fig. 14.2.b) may be represented as a data structure that describes a data flow graph with the specific implementations of the database operations to be used by the execution engine. Each *node* in the graph is an operator descriptor and the directed arcs determine the flow of data. Among other things, the query optimizer decides the specific implementations that have to be used in each of the nodes of the plan (Hash Join, Nested Loop Join, Merge

used in any case. However, the execution time of the Hash Join is significantly shorter than that of the Nested Loop Join, making the Hash Join the preferable implementation whenever possible.

Join, Index Scan, Sequential Scan, etc.) and the order in which they have to be executed. Some current optimizers obtain the execution plans based on estimates of the cardinality of the tables and the costs of the different implementations of those operations, among others. Some of the information used by the optimizer is stored in the *database catalog* or *data dictionary*. The database catalog holds information about the database, such as the structure of the relations, the relations among tables, the number of attributes of each record in a table, the name of the attributes on which an index has been created, the characteristics of the results of database operations in previous executions, etc.



```
select
    s.name, s.age
from
    staff s,
    department d,
    projects p
where
    s.deptkey = d.deptkey
    and s.projkey = p.projkey
    and d.city = 'Barcelona'
    and p.name = 'Ursus'
    and s.age > 30
order by   s.name;
```

Fig. 14.2. SQL statement (a) and final query execution plan (b).

The query *Execution Engine* is now responsible for the execution of the Plan obtained by the optimizer. The components of the Execution Engine used in read only queries are shown in the shaded box in Fig. 14.1. These are the Executor, the Access Methods, the Buffer Manager, and the Storage Manager modules. Therefore, after allocating memory for the data structures required by the query and its nodes[3], and after initializing those data structures, the Executor module takes control. The memory allocated for each node is called the *heap* of that node. The size of the heap is usually fixed during the execution of a query, and it can be decided at optimization time by the user or by the database administrator.

Fig. 14.3 shows a sample of the routines that each layer of the Execution Engine of the DBMS offers to the upper layers, and what type of data those

[3] Those data structures may be, for instance, the hash structure for the Hash Join node, space to store intermediate results, etc.

layers are expected to obtain from the layers immediately below. Fig. 14.3 serves to illustrate the following explanation.

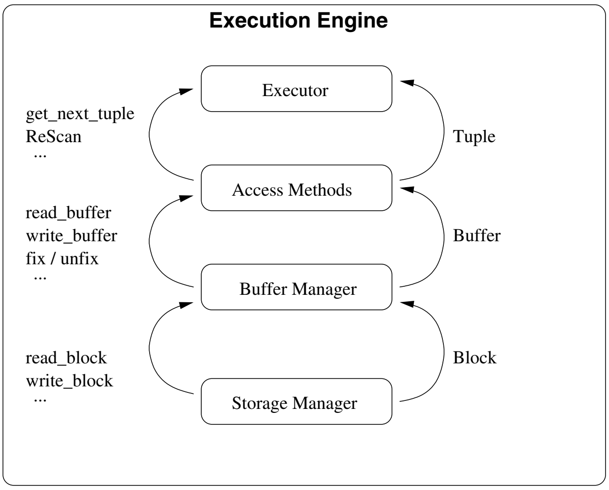


Fig. 14.3. Functionalities and data offered to the upper layer of a DBMS query engine.

The Executor module processes the plan in a record by record data driven execution form, starting from the topmost node of the plan. This is called pipelined execution: when one node requires a record to process, it invokes its immediate lower node or nodes and so on. For instance, the Nested Loop Join operation invokes the inner relation node to retrieve all its records, one record at a time, for each record of the outer relation node. In this case, the first result may be obtained and passed to the upper node before reading the whole inner relation. On the other hand, the Hash Join node requires all the records of the build relation to create the hash data structure, which will be stored in the heap of the Hash Join node. Only when the hash data structure has been fully created, can the probe relation be invoked record by record to check for joining records against the hash structure [4].

[4] As we will show later, there are cases like the *Hybrid Hash Join* algorithm that partition the build and probe relations. This happens when the records of the build relation do not fit in the heap. After partitioning the relations, only the records of one partition (that supposedly fit in main memory) are required to create the hash structure. Moreover, only the records of the corresponding probe partition are required to be probed against a build partition.

Other nodes, like the Sort nodes, require the full input relation before producing the first result of their operations[5]. In this case, it is said that the Sort nodes are materialized operations, as opposed to pipelined ones.

With the pipelined execution strategy, it is possible (i) to execute a full plan with the least amount of main memory space to store intermediate results, and (ii) to obtain the first result as soon as possible.

Only the Scan nodes access records from the relations of the database. Each time a parent node invokes a Scan, the Scan serves a record. However, the Scan nodes cannot access the relations directly. Instead, they can only call an Access Methods interface routine to obtain a record.

The Access Methods module abstracts the physical organization of records in file pages from the Executor module. The Access Methods module offers functionalities that allow the Executor module to read, delete, modify, or insert records, and to create or destroy relations.

The generic routines offered by the Access Methods module for sequential accesses are of the type "get_next_record(Relation)", which returns a record in sequential order from "Relation". The generic routines for indexed accesses are of the type "get_next_indexed_record(Attribute, Relation)", which returns a record from "Relation" in the order dictated by the index on "Attribute", or "get_next_indexed_record_for_value(Attribute, attr_value, Relation)", which returns a record from "Relation" with "Attribute" value "attr_value" through an index on "Attribute".

The Access Methods module accesses the database catalog in order to obtain knowledge of the type of data file or index file used for the relation being accessed[6]. The Access Methods module also holds structures in memory that describe the relations like the identifiers of the pages of those relations, etc.

Whenever the Access Methods module needs to access a record, it calls the Buffer Pool Manager to get hold of the page that contains the record, uses the page, and finally releases it.

The Buffer Pool Manager, also called the Buffer Manager; is similar to an operating system virtual memory manager, it has a Buffer Pool with slots to store file pages. The objective of the Buffer Pool is to create the illusion to the Access Methods module that all the database is in main memory.

The Buffer Pool Manager offers a set of functionalities to the Access Methods module. Those functionalities allow the Access Methods module to state the intention of working with a page and to release the intention to use the page. This is done with "Buffer_Fix" and "Buffer_Unfix" primitives, respectively. With a fix on it, a page cannot be removed from the Buffer Pool.

[5] Note that sorting can be performed in batches and then the batches are merged, as we will see later. Each batch can be processed in a pipelined manner, but in order to start the merge process, the last data item to be sorted has to be inserted in a batch.

[6] These can be record or multimedia files, B-trees, R-trees or other data structures.

It is the responsibility of the Access Methods module to unfix a page after having used it.

When the Buffer Manager is called to fix a page that is not resident in main memory, it is necessary to bring it from disk. There may not be free buffer slots to store the new page. In this case, it is necessary to replace an already memory resident page. The page to be replaced is selected by the page replacement policy routines of the Buffer Manager.

The Buffer Manager accesses the Storage Manager in order to obtain file blocks from disk. The Storage Manager abstracts the drivers of the physical storage devices from the Buffer Manager.

14.4 Evidences of Locality in Database Workloads

The analysis of locality is very useful for understanding the capability of the software to expose it. The characterization of Decision Support System workloads shows different aspects of the behavior of DBMSs, such as how the architecture is able to capture the locality of references both in the instruction stream and the data structures or how the execution time of queries is distributed in the different tasks necessary to execute an instruction.

Some papers analyze Decision Support System workloads on present DBMSs [23, 91, 411]. While [23] analyzes several undisclosed DBMSs, [91] focuses on Oracle and [411] focuses on DB2. We concentrate on the latter.

For instance, in a 30GB TPC-D [223] workload running on DB2/UDB [411][7], only 66% of all the data, including indexes and database relations are ever touched during the execution of the benchmark. Of this 66%, a little more than half (50.8%) are data pages and 43.1% are index pages. The rest are other data types. In other words, only 73% of the database data pages and 56% of the database index pages are ever touched.

In addition, only 5.4% of all the TPC-D references are made to database data, and a very high 92.7% of the references are made to database indexes. This translates into 35.5 references per referenced index page, and a little less than 2 references per referenced data page, telling us about the importance of locality exploitation in database index structures.

In terms of the instruction stream, there is evidence about the locality of instruction execution. For instance, PostgreSQL only executes a total of 75K instructions of the almost 600K instructions in its binary [626]. Similar figures for Oracle go up to a footprint of 500KB out of an executable of size 27MB [625].

[7] We base the figures in this discussion on the results shown in Table 6 on page 795 of this reference.

14.5 Basic Techniques for *Locality Exploitation*

Different basic techniques can be used to expose the locality of data in algorithms. The particular structure of the DBMS Execution Engine makes it feasible to use some of these techniques in certain layers of the engine.

In this section, we describe a number of basic techniques for the exploitation of locality. Each technique is described, the layers of the DBMS Execution Engine where it can be used are detailed, and an example of its use in the database area is given. In Sects. 14.6 and 14.7, a full explanation of the application of those basic techniques is given.

The basic techniques are as follows:

– Blocking restructures an algorithm to expose the temporal locality of groups of records that fit in a level of the memory hierarchy. With this Blocking, the memory traffic between different levels of the memory hierarchy is reduced. Blocking can be implemented in the Executor module of the DBMS.
  For instance, Blocking can be applied to the Nested Loop Join [687]. For the in-memory case, instead of checking all the inner relation records against each and every record of the outer relation, the DBMS can check a record of the inner relation against a block of records of the outer relation. This divides the number of loads of the whole inner relation by the size of the block. The size of the block, called the Blocking Factor, has to accommodate the size of the target cache level. Blocking can also be used in the external memory implementation of the Nested Loop Join, as we will see later.
– Horizontal Partitioning distributes all the records to be processed into subsets, with the aim that every record subset should fit in a certain level of the memory hierarchy. The criterion for distributing the records is some type of hash function on the values of an attribute or attributes of the relation to be partitioned. An advantage of Horizontal Partitioning is that it does not usually require post-processing after processing each record subset. A disadvantage of Horizontal Partitioning is that the exact size of the partitions is uncertain and depends on skew, the number of duplicate keys, etc. Horizontal Partitioning can be implemented in the Executor module.
  For instance, Horizontal Partitioning can be used in the Hash Join algorithm when the build relation does not fit in the heap. In that case, the build and probe relations are partitioned with the same hash function. The number of different partitions is decided by the DBMS with the help of a heuristic and depends on factors like the size of the memory, the expected skew in the data set, etc. After the partitioning, pairs of partitions are joined (one from each joining relation). The partitioning phase is expected to chop the build relation so that each partition fits in the heap. This is the Grace Hash Join algorithm [459].

- Run Generation is similar to Horizontal Partitioning. However, the partitioning criterion of Run Generation is to take consecutive records to form a subset that fits in the target memory level [460]. Run Generation usually requires some type of post-processing after dealing with each record subset. Run Generation is usually implemented in the Executor module.
  Merge Sort is a typical example of Run Generation [460]. With Merge Sort, runs are sorted using any classic technique such as Quick Sort. A final Merging process is necessary.
- Grouping complements Blocking, Horizontal Partitioning and Run Generation. It is implemented in the Scan operations either at the Access Methods or the Executor layers. The objective of Grouping is to prepare groups of records in the Scan operations for the upper nodes in the execution plan that require them to perform Blocking. It enables sets of records to be processed in a single instantiation of a Scan node.
  For instance, when a Blocking Nested Loop Join has to be implemented, it is necessary for the Scan operation to produce sets of records. The amount of records to be passed to the Join node depends on the Join operation and the cache or heap size. However, the ability to read a set of records in a single call of the Scan operation depends on the Scan operation itself.
- Extraction of Relevant Data takes parts of the data of each record to be processed into an independent data structure and creates some type of pointer to the record. The aim is to achieve a larger degree of spatial locality. Extraction of Relevant Data can be implemented in the Executor layer.
  Sorting operations in DBMSs usually extract the sorting key from each record and create a pointer to the record [347]. Sorting is performed on the set of keys and pointers and, when it is finished, the records are used or stored in the order dictated by the sorted keys. Hash Joins also benefit from this technique during the build phase, when the joining attribute value and a pointer to the record are inserted in the hash structure.
  Bit map data structures for join indexing are another example of this technique. Bit maps are created when attributes have a small amount of different values. In this case, each bit map indicates the presence of a value [592].
- Vertical Partitioning takes the instances of each attribute of a record and stores them in a separate table [22]. This increments the spatial locality, but also fragments records, and may force extra I/O. Vertical Partitioning may be used to partition files at the Access Methods layer.
  Clustering can be applied to Vertical Partitioning [687]. This implies that more than one attribute is stored together. Thus, attributes that are frequently used together live together, reducing fragmentation and I/O.
- Operator Fusion (or loop fusion) merges the bodies of different nodes of the plan in a single operator, in such a way that fewer routine calls are performed and data structures are traversed with higher temporal locality [687].

For instance, in some circumstances, Sort and Aggregate operators may be grouped in the Executor module [601].

– Pointer Elimination is used in linked lists in order to eliminate pointer indirections [194]. For instance, a set of elements of a linked list are stored contiguously, and a single pointer to the next set of elements in the list is necessary. This increases the spatial locality at the cache level.
  This technique may be used in any data structure and, in particular, in the hash structures used in Hash Join algorithms of the Executor module.

14.6 Exploitation of Locality by the Executor

In this section, we explore the space of some basic database operations such as Scan, Join, Aggregate and Sort. For each operation, we detail some of the proposals that have been used to exploit data locality in the Executor layer, using any of the techniques above. Locality is not only important at the record level in this layer of the DBMS, but also at the heap level.

For instance, once the records are obtained from the lower nodes of the execution plan, part of the heap stores those records and the rest of the heap is devoted to store the data extracted. As already stated, this is the *extraction of relevant data* and may be applied to the operations explained below.

For the models below, we assume a cache level able to accommodate a total of B_1 records, a disk page able to accommodate B_2 records, and a node heap able to accommodate the equivalent to B_3 disk pages.

14.6.1 Scan Operations

Scan operations receive one record at a time from the Access Methods layer and pass one record at a time to upper nodes of the plan. In order to allow Scan operations to pass more than one record at a time, *Grouping* can be implemented in these operations. In this case, a loop that iterates as many times as records wanted in upper nodes, calls routines of the Access Methods module, such as "get_next_record(Relation)", to build a group. In this case, Scan operations need a number of record slots equal to the number of records passed.

Grouping at this level has to perform a number of calls to the Access Methods routines. This implies that Grouping may not exploit locality as it could at this level. For instance, Index Scan routines traverse the index for every Scan routine call, which results in as many page accesses as the depth of the tree structure. This pages are accessed in consecutive calls of the Index Scan routines, which improves the locality of the non Blocking approach. However, one only traversal of the index pages would be enough if Grouping was implemented at the Access Methods level, which we prefer (see Sect. 14.7 for details).

14.6.2 Join Operations

The different implementations of the join operation can be improved if data locality is exploited. This is possible for the in-memory versions [687] as well as for the external memory versions [347, 552]. Both cases will be studied for the Nested Loop Join, the Hash Join, and the Merge Join implementations.

Nested Loop Join. Nested Loop Join is the universal join implementation and is normally used when the joining condition is different from "=".

Blocking can be applied to the Nested Loop Join implementation both in the case of in-memory [687] and external memory [552] implementations. In-memory Blocking saves traffic at the cache hierarchy level and is limited by the size of the target cache level, as explained above.

In this case, the number of records that should be moved from main memory to the target cache level of the hierarchy would be $|R|$ for relation R plus $\frac{|R|}{B_1}|S|$ for relation S, for a cache level able to accommodate a total of B_1 records of R. The non Blocking Nested Loop Join would require a total of $|R|$ movements of records of R plus $|R||S|$ movements of records of S.

The external memory implementation reads blocks of records of the outer relation until the heap assigned to the operation is full. Every time the heap is filled with records from the outer relation R, the inner relation S has to be fully traversed. External memory Blocking is limited by the size of the heap.

The total number of pages read from disk for the external memory Blocking Nested Loop Join is $\frac{|R|}{B_2}$ for relation R plus $\frac{|R|}{B_2 B_3}\frac{|S|}{B_2}$ for relation S, as opposed to $\frac{|R|}{B_2}$ for relation R plus $|R|\frac{|S|}{B_2}$ for relation S for the external memory non Blocking Nested Loop Join.

Blocking can be implemented in a nested fashion if it is used both at the in-memory and external memory levels. However, the larger benefit will be obtained from the external memory implementation.

One important circumstance when executing the Nested Loop operation is whether the inner relation node has a sub-plan below. In that case, it is possible to execute the sub-plan once, materialize its result, and then execute the Nested Loop Join. This strategy implies one only execution of the sub-plan, as opposed to the $\frac{|R|}{B_2 B_3}$ executions for the Blocking version of the Nested Loop Join. In a sense, this is a strategy for increasing the locality of execution of the operations below the Nested Loop Join node.

The external memory indexed Nested Loop Join is suitable for equi joins and natural joins. Blocking can be applied to this algorithm. For every record of the outer relation R, the Grouping Index Scan operation is invoked to retrieve records from the inner relation S, which returns a group of records of S that join with the record of R. If the total number of joining records of S is larger than the blocking factor, the Index Scan has to be called several times. However, in the model below we assume one only Index Scan call.

The number of page reads in this case depends on the Index Scan routines used and on whether the Grouping is implemented in the Executor module or in the Access Methods module. If Grouping is implemented in the Executor module, the indexed Nested Loop Join generates a total number of reads from disk storage equivalent to $\frac{|R|}{B_2}$ for relation R plus a minimum of $|R|K_2\frac{K_1|S|}{K_3}$ for the index pages, and a minimum of $|R|K_1\frac{|S|}{B_2}$ for relation S, where K_1 stands for the fraction of records of relation S involved in the join, K_2 stands for the number of levels of the index, and K_3 stands for the average number of records in relation S with the same joining attribute value. When we give a minimum number of pages for both the index and relation S, we understand that (i) the index pages are only read once for each joining value because locality is exploited, and (ii) there is such a record clustering that places together all the tuples of relation S that join.

Hash Join. This is an important operator because it is a very time and memory efficient algorithm. Hash Join is only used for equi joins and natural joins.

Hash Loops would be the simplest, straight-forward algorithm for the external memory Hash Join. In this case, one iterate of an outer loop creates a hash structure in the heap with as many records as possible from the build relation. Then, all the probe records are probed against the records in the hash structure. The following iterates of the outer loop repeat this operation with different sets of build records until all the records of the build relation have been visited. However, this algorithm reads the probe relation as many times as pieces of the build relation fit in the heap, which is not efficient in terms of locality exploitation.

The two most efficient generic external memory Hash Join algorithms are based on the *Horizontal Partitioning* technique explained above. These are the Grace Hash Join [459] and the Hybrid Hash Join [250] algorithms.

As already stated, the Grace Hash Join[8] partitions the probe and build relations by applying a hash function H on the joining attribute. Once those partitions are materialized in external memory, each pair of build/probe partitions is processed with the in-memory Hash Join. The Hybrid Hash Join algorithm is an optimization of the Grace Hash Join algorithm where the Join on partition zero is performed during the partitioning phase.

The number of pages read from disk for the three algorithms are as follows. The total amount of pages read from disk for the Hash Loops algorithm is $\frac{|R|}{B_2}$ for relation R plus $\frac{|R|}{B_2 B_3}\frac{|S|}{B_2}$ for relation S. For the Grace Hash Join and Hybrid Hash Join, we assume a random distribution of the values of the joining attribute and a perfect hash function H that does a balanced Horizontal Partitioning. In the Grace Hash Join case, relation R is read once to be partitioned and once more to be joined, $2\frac{|R|}{B_2}$. The same happens with relation S. The number of pages for the Hybrid Hash Join algorithm is $2\frac{|R|}{B_2} - \frac{|R_0|}{B_2}$

[8] The name Grace Hash Join comes from the name of the computer where it was applied for the first time, the Grace Machine [459].

for relation R plus $2\frac{|S|}{B_2} - \frac{|S_0|}{B_2}$ for relation S, where $|R_0|$ and $|S_0|$ stand for the number of records of partition 0 of relations R and S.

Blocking could be used in the Hash Join operator. Its benefits would come principally from the number of routine calls performed but no significant reduction in the memory traffic among the different memory layers would be obtained.

Pointer Elimination can be implemented in the hash structure stored in the heap. With this strategy, the joining attributes stored in a hash bucket are packed into arrays of a certain number of attributes. One only pointer per array of attributes is stored. With that Pointer Elimination, the probe phase has higher locality. This is particularly effective for in-memory Hash Join algorithms.

A specific implementation of Pointer Elimination for in-memory databases is the radix-cluster Join algorithm proposed in [141]. The proposed strategy implements a hash structure with semi-sorted buckets using a radix sort technique. The idea is to completely eliminate the pointers in the hash structure of the heap. This is an extreme implementation of the Pointer Elimination technique.

Hash Teams. This algorithm [348] and the Generalized Hash Teams [455] perform what could be called, a clever *Horizontal Partitioning* of the relations involved in a query that leads to some type of *Blocking.* The important aspect is that the relations involved with each node of the plan are partitioned in such a way that the possible I/O of intermediate results is avoided. We discuss the Hash Teams implementation and leave the Generalized Hash Teams for further reading.

Hash Teams are used when multiple neighboring Hash Join nodes in a query plan join on the same attribute. In this case, the joining attribute is used to perform Horizontal Partitioning on all the relations involved, using hash function H as partitioning criterion for all the joins as shown in Fig. 14.4. Once all the relations have been partitioned, the processing is easy because it saves the partitioning of intermediate results: as soon as an intermediate result is generated, it may only join with the corresponding partition of other joins. As shown in Fig. 14.4, partitions 0 (dashed boxes) of tables R, S, and T are processed simultaneously. The result of node HJ1 for partition 0 ($J1R0$) need not be stored in external storage because $PT0$ is the build relation and node HJ2 consumes the result of node HJ1 as soon as it is generated as the probe relation.

Therefore, for the two join query shown in Fig. 14.4 with relations R, S, and T involved, their corresponding cardinalities $|R|$, $|S|$, and $|T|$, and a common joining attribute u, we have the following number of page reads from disk. Relation R is read once to be partitioned and again once to perform the build phase of the Hash Join, $2\frac{|R|}{B_2}$. Relations S and T are also read the same amount of times as R. The intermediate results do not require partitioning, as we said.

A Hybrid Hash Join algorithm would generate the following number of reads for the query: The same amount of reads for each of the relations involved plus one read for the intermediate relation $J1R$ of Hash Join $HJ1$, $\frac{|J1R|}{B_2}$.

This approach can be regarded as a kind of Blocking where, once a data subset of the intermediate results is in memory, it is fully processed and will not be required again.

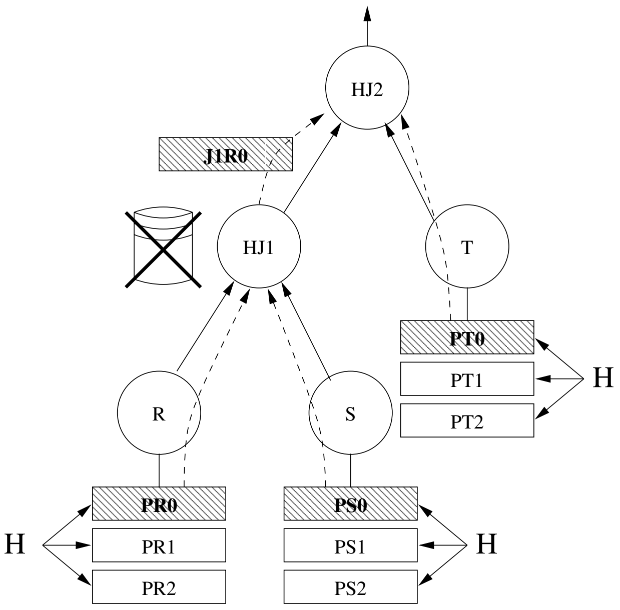


Fig. 14.4. Implementation of Hash Teams for a two join query on a common join attribute. Pxy stands for partition y of relation x. J1R0 stands for join 1 result of partition 0.

Merge Join. The joining process with Merge Join starts with the sorting of one or both of the joining relations. When one of the joining relations has an index on the joining attribute, only the non-indexed relation has to be sorted [347].

In the non-indexed implementation, when the sorting of the relations has finished, it is necessary to traverse the relations in a sequential way to join them. The indexed implementation is similar to the Nested Loop Join with indexes except for the sorting process of the non indexed relation, which has to be sorted here.

Therefore, the weight of the Join operation is in the Sort nodes. Aspects related to locality in the Sort operation are explained in Sect. 14.6.4. The rest of the implementation is just a sequential read of the two joining relations for the non-indexed Merge Join.

One other possible implementation of the Merge Join for non-indexed relations can be as follows: *Horizontal Partitioning* is applied to the joining relations. Each partition is processed with an in-memory Merge Join implementation. This is similar to the Grace Hash Join algorithm, and the total number of disk page reads would also be the same.

Bit Mapped Joins. As mentioned above, the *Extraction of Relevant Data* can be implemented with bit mapped joins [506, 592]. This can be done when the number of different values in the joining attribute is very small (i.e., male and female for a gender attribute). In this case, a bit map is created for each join attribute value (i.e., one for male and another one for female). Each bit in the vector is associated to a record of the joining relation. A one in a position of the bit vector means that the value is present in the record.

This technique has the advantage of reducing the amount of data traffic between disk and main memory when the join is performed. Only one read of the bit maps has to be performed to execute the Join operation. The presence of a one in each of the joining relations implies a match and the two Record Identifiers (RIDs) have to be saved. Once the join operation is performed, the pairs of RIDs obtained can be sorted in some way, and a read of the records that join has to be performed to obtain the final result with the RIDs stored.

The amount of page reads from disk is reduced from $2\frac{|R|}{B_2} - \frac{|R_0|}{B_2}$ for relation R plus $2\frac{|S|}{B_2} - \frac{|S_0|}{B_2}$ for relation S for the Hybrid Hash Join to $K_3\frac{|R|}{B_2}$ for relation R, plus $K_4\frac{|S|}{B_2}$ for relation S, plus $K_5|R|$ and $K_6|S|$ for the bit maps involved in the Join operation. Here, K_3 and K_4 stand for the fraction of records of R and S that join, while K_5 and K_6 stand for the fraction of R and S occupied by the corresponding bit maps[9].

A disadvantage of this technique is that it is costly when the records of the relations involved in the join change often, which implies changes in the bit maps.

14.6.3 Aggregate and Expression Evaluation Operations

The observation is that some queries perform a large amount of arithmetic and aggregation operations on the records of the largest table involved in a query plan [601]. Usually, each of those operations is associated to a query plan node. This implies the use of a significant amount of data structures, the checking of different conditions and calls to routines for every single database record. In other words, not exploiting locality.

In order to reduce the amount of routine calls, it is possible to do *Grouping* of records so that a group is processed in one single call, and *Operator Fusion*,

[9] Note that for 128 byte records only one bit would be necessary. This means a $\frac{1}{1024}$ total extra storage required for the bit maps.

which may also reduce the amount of calls since it collapses several nodes of the plan in one node.

14.6.4 Sorting

A Sort node in a query plan usually obtains the records to be sorted in a pipelined manner. This means that, with some sort of cadence, records keep arriving to the Sort node coming from its lower execution plan node. There are also cases where the relations to be sorted are obtained in full by the Sort node.

For the case of records obtained in a pipelined manner, *Run Generation* can be performed using Replacement Selection [460] as the sorting method of choice [496] to sort each of the runs generated. With Replacement Selection, and assuming an ascending sorting order, two runs are generated at the same time as records arrive to the Sort node, the *previous run* and the *present run*. The *previous run* just keeps track of the largest value of the run and is stored in external memory, except for one page that is stored in the heap. The *present run* is kept in full in the heap. A new key larger than the largest key of the *previous run* is added to that *previous run*. A new key smaller than the largest value of the *previous run* is included in the *present run*. The *previous run* is finished when the data items of the *present run* fill the heap. In this case, the *present run* starts to play the role of the *previous run*; that is, all its items are stored in permanent storage except for one page. The process starts again with the new *previous run* and a new empty *present run*. This leads to runs that are larger than the heap[10], which also means a smaller number of runs. This has the advantage that the Merge phase is performed on a smaller number of runs, which reduces the fan-in of the Merge phase and increases the chances of performing one only set of reads for the Merge [496].

Therefore, there is only one data read from disk, that of the Merge phase. This is so because data items are passed from a different node, and thus, data are in main memory. However, if the fan-in of the Merge phase is larger than the number of pages that fit in the heap, the Merge phase will be performed in several steps, incurring more data reads.

If the records to be sorted are not obtained in a pipelined manner, there are two approaches to do the sorting. First, sort runs that fit in the heap and do the Merge with those sorted runs [778]. Second, perform a Horizontal Partitioning of the records so that each partition contains keys in a range independent from other partitions and sort each partition independently [12]. In both cases, there is a minimum number of two data reads from disk.

Several methods can be used to sort the runs in memory. Among the fastest, we find variants of Radix sort for 32 and 64 bit integers [430, 431].

[10] Knuth analyzed this problem [460] and found that the average size of a run using Replacement Selection was twice the size of a run created with the Run Generation technique.

14.7 Access Methods

The Access Methods module implements different routines to allow the Executor module to access records of the Database either sequentially or with indexes. As stated above, routines for accessing records like "get_next_record(Relation)" or "get_next_indexed_record(Attribute, Relation)" are typical.

The Access Methods layer follows different steps to return a record to the Executor layer. For example, in an indexed access, the Access Methods module checks the catalog for the type of index used with the specific attribute in the relation to be read, searches the routine that performs the task for the index type and traverses the index structure to obtain the record.

The key issue about the Access Methods layer is how data are stored in physical pages. This is something that the Executor layer does not have to be aware of, because it can only receive records. The physical storage determines the patterns of access (hence the locality) by the Access Methods and the Buffer Manager. We give further details of these aspects for data files and index files below.

14.7.1 Record Layout Strategies for Data Files

The three basic models for the storage of records in a file page are: the N-ary Storage Model (NSM), the Decomposition Storage Model (DSM) and the Partition Attributes Across (PAX) model. Fig. 14.5 shows the data layout for the three strategies.

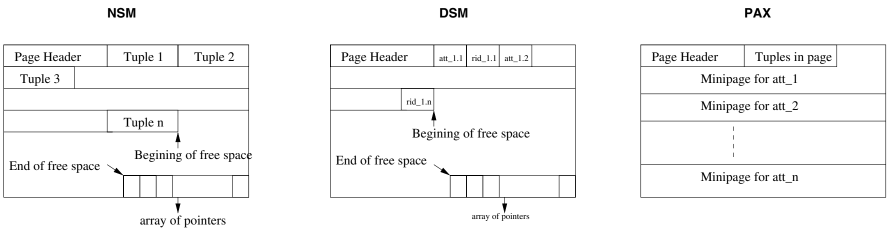


Fig. 14.5. Different record layout strategies for data files.

The N-ary Storage Model stores records contiguously starting from the beginning of each disk page. In order to identify the offset to each record in the page, NSM holds a pointer table starting at the end of the page and growing towards the beginning. In this way, record and pointer table storage do not interfere until the page is full. Entry X of the offset or pointer table points to record X in the page. This is the model used by most DBMSs [22].

NSM aims at the exploitation of spatial locality at the record level, in the belief that a significant number of attributes in a record are accessed in one single record access, exploiting spatial locality. However, in some cases this model is not efficient in terms of the cache hierarchy. For example, when a query only processes one attribute per record, the attributes next to it are also fetched, with the cache line, up to the cache level closest to the processor. Memory bandwidth is wasted and unnecessary data are stored in caches along the cache hierarchy [22].

In order to prevent this from occurring, the Decomposition Storage Model [208] performs a *Vertical Partitioning* of the relation into a number of vertical sub-relations equal to the number of attributes. Each sub-relation holds two values, a record identifier and the attribute itself. A pointer table is used to identify the record to which each attribute belongs within a page.

DSM aims at the exploitation of locality at the attribute level. This strategy is useful when a small amount of attributes have to be fetched from each record. However, a buffer page per attribute has to be maintained in the Buffer Pool and the cost of reconstructing a record is rather expensive. As shown in [22], DSM deteriorates significantly when more than 4 attributes of the same relation are involved in a query. As an alternative to this problem, the proposal in [217] performs a clustered partitioning of each relation based on an attribute affinity graph, placing together attributes that often appear together in queries.

As an alternative to the previous two models, the Partition Attributes Across [22] model partitions records vertically within each page. The objective is that one record is stored within a page while the instances of each attribute are clustered in mini-pages. Each mini-page holds a presence bit for fixed size attributes and a pointer for variable size attributes.

PAX aims at exploiting register and attribute locality at the same time. By storing in a page the same data as NSM, but having each attribute in separate mini-pages, PAX solves the cache hierarchy problem and, at the same time, reduces the amount of pages to be maintained in the Buffer Manager. This gives a performance similar to that of NSM, when external memory queries are executed but outperforms significantly NSM and DSM for in-memory queries [22].

14.7.2 Layout Strategies for B-tree Index Files

In this section, we concentrate on the physical layout characteristics of B-trees in relational Databases. For an in-depth account of B-trees in general, we refer the reader to Chapter 2. Other papers dealing with the detailed description and use of B-trees on databases and the performance analysis of B-trees are [356] and [436] respectively.

Let us recall that, a B-tree indexes a relation by an attribute. A B-tree is formed by indexing nodes and leaf nodes, as shown in Fig. 14.6. Each node is stored in a physical disk page. The indexing nodes form a balanced tree of depth h with a maximum of B keys or pointers fitting in a disk page. An indexing node N_j on a fixed size attribute (e.g., integer) contains an array of records (IA, P), sorted by the indexing attribute IA. The pointer P_i of a record in node N_j points to a successor node N_k, such that $IA_i \leq IA_l < IA_{i+1}$, for all the attribute values IA_l of node N_k. A leaf node contains an array of records (IA, RID), sorted by the indexing attribute IA. The Record Identifier RID of each record is a record descriptor and points to the target record.

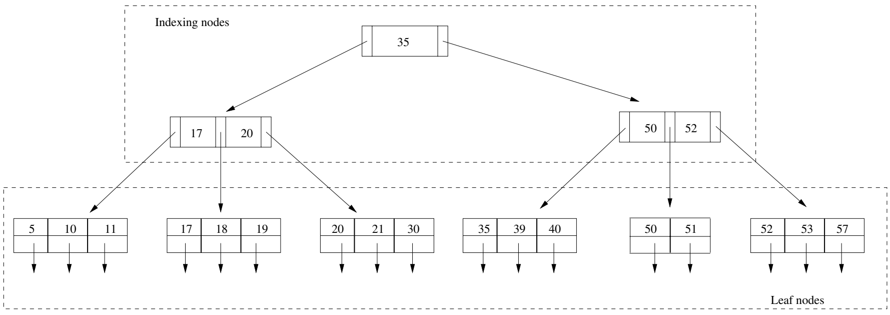


Fig. 14.6. Structure of a B-tree in a DBMS.

The use of B-trees in DBMSs has the following important features regarding locality:

– The page structure of a B-tree raises the question of what pages of the B-tree should be kept in main memory to minimize I/O.
– Each record access through a B-tree implies a number of page accesses equal to the depth of the B-tree, h, plus one access to the page where the record is stored in the database relation. In the example shown in Fig. 14.6, there would be 4 page accesses.
  The larger the number of records per indexing node, the smaller the potential depth of the B-tree. A possible means of incrementing the number of records per indexing node is compression, both in the case of fixed size attributes and in the case of variable sized attributes.
– The complexity of the search for an attribute instance in a physical page depends on the length and size variability of the attributes stored in a page. Therefore, searching is also a key feature and may depend on how compression is implemented.

In this section, we only focus on compression of the indexing attributes [514, 349]. The first feature, locality at page level, is a Buffer Manager responsibility and is explained in Sect. 14.8.

The third feature, searching for an attribute instance among the set of items of a node, can be performed with a binary search algorithm. Binary search is discussed in [85].

We discuss a few compression techniques that have been proposed in the literature. Those techniques were designed for variable length keys, but in some cases may also be applied to fixed length keys. Fig. 14.7, which shows the uncompressed original attribute vector on its left hand side, helps in the explanation of these techniques:

- When the keys of a node have a set of common most significant bits, a common prefix of y bits may be factored out from the attributes, as shown in Fig. 14.7.a. In this case, the rest of the attribute will be smaller and the binary search will be faster [97].
- When there are almost no common most significant bits, it is possible to extract a few bits of each key, say z, forming a prefix vector so that comparisons can be made on smaller attribute sizes, as shown in Fig. 14.7.b [514]. A tie is resolved by searching on the attribute extensions that have the same value in the prefix vector.
- A hybrid between the common prefix and the vector prefix can be implemented to save comparisons. This may be done for long indexing keys that have a common set of most significant bits plus a few different second most significant bits. This is not shown in Fig. 14.7.
- The *Vertical Partitioning* approach, where the indexing attributes and pointers are separated into two vectors, can also be considered, see Fig. 14.7.c. This can be applied to any of the above compression techniques, and leads to cache conscious algorithms since spatial locality can be exposed during the binary search on the attribute instances [638].

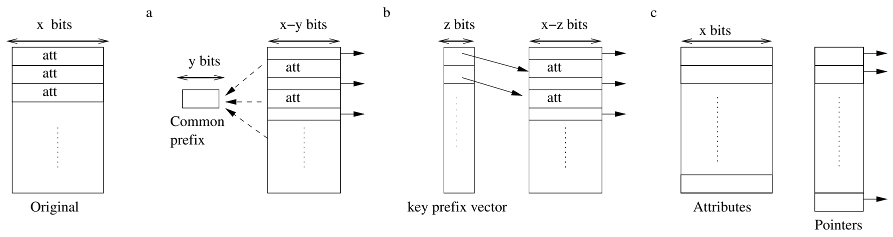


Fig. 14.7. Compression techniques for attributes in a B-tree.

Among other aspects that can be taken into account to improve the data locality for main memory B-trees, we also have data alignment with cache lines or implementation of B-trees based on the cache line size [194, 349].

14.7.3 Grouping Records in the Access Methods Layer

The usual way for the Access Methods module to communicate with the Executor module is record by record, as already stated. However, it is possible to communicate by groups of records, which increases the locality at the different levels now discussed:

– For a group of n records, there is only one Access Methods routine call, as opposed to the n routine calls that would be necessary in the case of a record by record access.
– For sequential accesses to relations, locality increases because consecutive records are accessed in an atomic operation, thus increasing spatial locality. A routine "get_next_record(Relation, n)" would be necessary.
– For accesses to indexed relations, locality also increases because the index is traversed only once for a group of record reads. This increases the temporal locality of the indexing nodes and the spatial locality of the leaf nodes.
  In this case, a couple of calls are necessary. First, when the indexed relation is traversed completely in the order established by the index, a routine like "get_next_indexed_record(Attribute, Relation, n)" with the same added parameter as above, would return a block of records of size n. Second, when a direct access is required to a set of records with the same value for the indexing attribute, a routine like "get_next_indexed_records_for_value(Attribute, attr_value, Relation, n)" would return a block of maximum size n records.

Grouping at the Access Methods layer may be used for the Blocking, Run Generation and Horizontal Partitioning techniques. In these three cases, Grouping allows the three techniques to process records in groups at the Executor layer reducing the number of routine calls in Scan operations.

One last comment is necessary: let us recall that an index is intended for the consecutive access of records that are not sorted by the indexing attribute in the actual relation. So, in this case, blocking can be of special interest at the Access Methods layer because it may save additional I/O if it is divided into the following phases. First, n accesses to the index structure are performed to collect the RIDs of those records. Second, the RIDs collected are sorted. Third, the records in the sorted RID order are accessed. With this strategy, and in the case that there is some clustering of records with similar indexing attributes in physical pages, there may be some additional I/O reduction.

14.8 Exploitation of Locality by the Buffer Pool Manager

One of the objectives of the Buffer Pool Manager is to exploit data locality between the disk and main memory. The Buffer Pool is managed to reduce the number of pages read from and written to disk. The ideal situation would be one where only one read per page was made (compulsory reads). Thus, the objective of a Buffer Pool Manager is to use a policy that minimizes the number of reads per page. In order to give a hint about the behavior of locality at the Buffer Pool, some figures are given in Sect. 14.4.

There are different approaches to manage the Buffer Pool [277]. A first approach is to consider the Buffer Pool as a unique, global space where different users[11] compete for memory pages. Global strategies have one only replacement policy associated. We discuss some basic replacement strategies in Sect. 14.8.2.

A second approach is to consider that the type of page or file is critical for the locality behavior of the reference stream. Hence, the Buffer Pool is structured in local pool areas that are exercised by different users.

A third approach is to consider the transaction as a key feature in locality exploitation. Similar to the previous case, the Buffer Pool is separated in different local areas, one per transaction. The areas are managed in an independent way. This approach may use any of the two previous approaches to manage the local transaction space. Therefore, we only concentrate on the first two approaches.

There are three important aspects to the local strategies. First, the page replacement strategy to be used for the local pool of each user. This is the same problem as for the global strategies, but in this case a different page replacement strategy may suit each local buffer area better. Second, how many pages are assigned to each local area and whether those pages are assigned statically or dynamically. Finally, in the case of dynamic page assignation, what must be done in the case of page starvation by one user. These aspects are covered in Sect. 14.8.3.

One final aspect to be considered is how the Buffer Pool interacts with the operating system. This is addressed in Sect. 14.8.4.

14.8.1 Data Structures for the Buffer Pool

The management of the pages stored in the Buffer Pool requires a control of (i) the pages that are fixed by the upper layers, and (ii) the pages that are not being used and can be substituted by newly referenced pages.

The list or lists of pages being used and the list of candidates to be replaced can be implemented in one data structure combining both types of lists or in several data structures, one for each list. The data structures used may be chained lists or hash structures.

14.8.2 Global Replacement Strategies for the Buffer Pool

A replacement strategy is aimed at keeping in memory the pages with more chances to be used in the future. The replacement strategy is used when, after a number of references, it is necessary to displace a page in the Buffer Pool in order to free a slot for a referenced page.

The global replacement strategies detailed here are Random, First In First Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU), and OPT as explained and analyzed in [277]. We explain the LRU-K and A0 as

[11] As user of the buffer space, we consider either a transaction or a database relation.

proposed in [591], and finally CLOCK, Generalized CLOCK (GCLOCK) [277] and the concept of correlated references discussed in [634], which can be used in combination with some of the previous basic algorithms.

Basic Replacement Strategies

For the following explanation, the victim pages are those that are not fixed by a transaction when a referenced page has to displace a page from the Buffer Pool.

The Random replacement algorithm takes one page at random as victim for replacement. This strategy only requires a structure that maintains all the possible victims.

The FIFO algorithm takes as victim the page with the first reference of all those pages present in the buffer pool. This algorithm requires a list of all the page descriptors sorted by the historic reference order. The page at the top of the list is the one to be substituted if it is not fixed.

The LRU strategy takes the page that has been Least Recently Used (referenced) as victim. This algorithm requires a linked list of page descriptors so that when a page is unfixed, it is placed at the bottom of the list. When a victim is necessary, the first page in the list is chosen.

The LRU-K strategy [591] modifies the LRU strategy in the following way. The victim page is the one whose backward K-distance is the maximum of all the pages in the Buffer Pool. The backward K-distance of a page is defined as the Kth most recent reference to that page. As explained in [591], this strategy requires a significant amount of memory to store the K-distance history of all the pages in the buffer, and the added problem of having to traverse the vector of page distances to find the maximum distance among the candidate victims. The data structures of LRU-K can be improved to reduce data manipulation, as shown in [437] for the $2Q$ algorithm.

The LFU strategy takes the page with a smallest number of references as victim. This requires a set of counters to keep track of the number of references to the pages of the database. The strategy can be implemented in two different ways. First, by only keeping track of the pages that are present in the Buffer Pool. Second, by keeping track of all the pages including those that have been cached in the Buffer Pool in the past.

Apart from the previous algorithms, the OPT and A0 algorithms are used as a reference by several authors [277, 591]. The OPT algorithm replaces the page with the longest forward reference distance. This is the optimum algorithm, but it is impossible to implement because the knowledge of future references would be needed. The A0 algorithm replaces the page whose probability of access is lowest. It is impossible to know this probability beforehand, but it can be estimated using the previous reference behavior.

A Second Chance for Victims

The CLOCK strategy makes use of a used bit per page to give a second chance to a candidate victim page. The used bit of a page is set to one when the page is referenced. Candidate pages are found in a round robin traversal of all the page descriptors. Finding a used bit equal to one, forces a reset to zero. The first page found with a used bit equal to zero is the victim of choice.

The CLOCK strategy can be combined with the Random, FIFO and LFU strategies. Every time a page is referenced, the bit is set to one. When a victim has to be found, the list of potential victims is traversed in the order determined by the strategy (instead of the round robin traversal), displacing a page as explained above.

The CLOCK strategy is similar to the LRU strategy by virtue of the fact that when a page is used, a recency bit gives it a second chance. Note that the use of the CLOCK strategy combined with the LRU strategy makes little sense.

The GCLOCK strategy generalizes the CLOCK strategy to the use of a used counter instead of a used bit. In this way, different types of pages can increment the used counter in different quantities, depending on how important it is to keep them in main memory.

Correlation of References

One important aspect of the reference stream is that references are usually bursty [503]. The burstiness of references comes from the fact that, when a record is referenced, for instance, a few of its attributes may be referenced in a short lapse of time.

One possibility is to consider the references made to the same page in a certain lapse of time as a single reference [634]. The work presented in that paper is based on the LRU replacement strategy, and although it is proposed for operating systems' file managers, it can be useful in DBMSs.

14.8.3 Local Behavior Strategies for the Buffer Pool

The Buffer Pool may be structured in separate pool areas that are exercised by different users. The pool areas are managed in different ways and employ different replacement policies depending on the needs of each user.

The global replacement strategies do not take into account that different data structures and database operations have different access patterns on the database pages. A very good account of different local behavior strategies can be found in [199]. In this section, we explain two of the strategies described in that paper, the Hot Set algorithm, proposed in [646], and the DBMIN Buffer Management algorithm, proposed in [199].

The Hot Set algorithm relies on a model with the same name. This model calls *hot sets* those groups of pages that show a looping behavior within a query. If all the hot sets of a query fit in the Buffer Pool, its processing will be efficient because all the references that iterate on a set of pages will fit in memory.

For instance, the hot set of a Nested Loop Join algorithm would be the number of pages of the inner relation plus one for the outer relation.

The Hot Set algorithm provides an independent set of buffer slots for each query. Each of these sets are managed by the LRU replacement strategy, and the number of slots is decided according to the Hot Set model of the specific query. The amount of slots assigned to a query varies dynamically as a function of the need for new pages.

On the other hand, the DBMIN algorithm relies on the Query Locality Set Model (QLSM). The QLSM classifies the page reference patterns based on the reference behavior of basic database operations such as the Nested Loop Join or the Index Scan. DBMIN makes use of different replacement strategies depending on the type of pattern shown by the operation. The partitioning scheme of the algorithm dynamically assigns the amount of Buffer Pool slots needed at each moment.

The QLSM distinguishes among three types of reference patterns, Sequential, Random and Hierarchical. Sequential references are those performed by (*i*) a sequential Scan, (*ii*) local re-scans[12] and (*iii*) the Nested Loop Join. Random references are those performed in independent accesses to a relation to obtain (*i*) a single record or (*ii*) a clustered set of records. Finally, Hierarchical references are caused by index structures and include those performed in the access to (*i*) a single record through an index, (*ii*) an Index Scan, (*iii*) a clustered Scan after an index traversal and (*iv*) a Join where the inner relation is indexed.

The DBMIN algorithm allocates and manages buffers on a per file instance basis. The buffer pages linked to a file instance are referred to as the locality set of that file instance. When a requested page is in the Buffer Pool but in the locality set of a different file instance than that requested, the page is given to the requester but remains in the locality set of its host. If it is not in memory or it is in memory but not assigned to a locality set, it is assigned to the requester's locality set. Each locality set is assigned a replacement strategy based upon the type of reference pattern incurred. For instance, when a file is scanned with a Nested Loop pattern, its locality set is managed with a Most Recently Used replacement policy, which takes as candidate for replacement the last referenced page in the locality set. Additionally, when a file is scanned through an index, the root of the index is the only page worth keeping in memory because the fan-out of an index node is usually large.

[12] Like those performed by a Merge Join when records within the inner relation are repeatedly scanned and matched with those that have the same value for the joining attribute of the outer relation.

Comparison of Replacement Strategies

In this section, we use the results of the papers mentioned above to compare the different replacement strategies described in this paper. We compare all the algorithms because they are all meant for the same objective, i.e. the improvement of locality in the Buffer Pool of a DBMS.

The following conclusions can be extracted from those papers:

- Among the FIFO, Random, LRU, CLOCK, and GCLOCK global replacement strategies, the LRU and the CLOCK strategies have a more satisfactory overall behavior [277].
- The best performance/cost LRU-K strategy is LRU-2. It behaves better than the plain LRU strategy and does not have a significantly worse performance than the LRU-3 strategy, which requires additional memory space [591].
- Factoring out recent references improves the LRU strategy in Unix operating system environments [634].
- Compared to FIFO, Random, CLOCK, and the Hot Set algorithm, DBMIN obtains the best results for different query mixes and degree of data sharing among those mixes [199].
- The previous points lead to the conclusion that replacement strategies that specialize in the type of access pattern, such as DBMIN, may lead to better locality of reference exploitation.

Apart from the basic, static, buffer replacement policies explained above, there are dynamic policies, based on Data Mining [733], to adaptively capture the best replacement strategy depending on the most recent page references, and policies based on queuing models [286].

14.8.4 Virtual Memory Interactions

An important aspect in the execution of DBMSs is their interaction with the underlying operating system. Although the Buffer Manager of some DBMSs can manage permanent storage directly, without the need for an operating system, it is the case that most DBMS executables run on top of an operating system and make use of the Virtual Memory Manager of the operating system. The interaction between the DBMS and the operating system may reduce the exploitation of locality in queries to the database, as we explain below. Several papers have dealt with this problem [191, 495, 707, 760].

The most important problem from the point of view of the type of queries we are addressing is the double paging anomaly [191]. For instance, a reference to a database page P is made, but the page is not in the Buffer Pool. Then a page, say M, has to be selected and displaced to make room for page P. Page M, however, has been displaced by the Virtual Memory Manager to permanent storage. This means that Page M has to be brought to memory by the OS, and then it will be written back to memory by the Buffer Manager of the DBMS. There is an excess of a read and a write of a page to bring M to the Buffer Pool.

There are different solutions to this problem [191]:

1. One possibility is to displace a page different from M from the DBMS Buffer Pool. This implies that the Buffer Manager and the Virtual Memory Manager have to communicate in some way to avoid double paging of those pages referenced solely with the purpose of replacement. However, this does not ensure the complete elimination of the double paging anomaly.
2. Another possibility is to make sure that the complete Buffer Pool is always resident in main memory. This can be achieved by (i) assuring enough main memory so that there is no need to page out pages from the Buffer Pool, (ii) making the Buffer Pool small enough so that it fits in main memory, or (iii) fixing into main memory the Buffer Pool pages from the Virtual Memory Manager itself.
3. A final approach would be to use the files that contain the database relations as the external storage of the Virtual Memory. This would be called a memory mapped system. With this system, and some communication between the Buffer Manager and the Virtual Memory Manager, it is possible to avoid the double paging anomaly.

14.9 Hardware Related Issues

The role played by the hardware of modern computers is important for increasing the locality of DBMSs. Different research works have focused on ways of improving the exploitation of locality on in-memory DBMSs.

Among different aspects, the cache size and, more importantly, the cache line size play a significant role both in PostgreSQL [730] and in Oracle [91] for Decision Support Systems workloads. In some cases, the execution time of complex queries can be penalized by more than 20%. In general, the optimum cache line size ranges from 64 to 128 bytes.

Simultaneous Multithreading[13] (SMT) processors have also been analyzed in the DBMS context [510]. In this case, the results show that an execution time improvement of more than 35% can be achieved with SMTs compared to superscalar processors. The reason for this is the large amount of low level parallelism that can be found in DBMSs, and the ability of SMTs to exploit this type of parallelism and to hide the memory latency.

[13] The ability of one processor to execute several program threads simultaneously.

14.10 Compilation for Locality Exploitation

One final consideration is how the compiler can collaborate to expose the locality of DBMSs. Results in this area show that the execution performance of code-reordered DBMSs for Decision Support Systems workloads achieve as good results with 32K byte first level instruction caches as non-reordered codes with 128K bytes or larger [572].

The optimizations that help to improve the performance of DBMS codes reorganize the binary in two different ways. First, groups of instructions that are executed in sequence are stored together, and the most frequently executed instructions are stored in a special part of the binary in such a way that no other instructions conflict with them in the instruction cache [627].

14.11 Summary

In this chapter, we have explored the structure and the different means of exploiting locality in Relational Database Management Systems. This has been done for read queries on large data sets arising from the Data Warehousing and Decision Support Systems areas.

DBMSs are complex codes with a layered structure. The Engine of a DBMS is divided into the Executor, the Access Methods, the Buffer Management, and the Storage Management layers. We concentrated in the former three layers and analyzed different means of exploiting data locality.

The Executor layer performs operations at a record level. The pipelined execution of queries is implemented in this layer and has a direct influence on the exploitation of locality for basic operations like Scan, Join, Aggregate and Sort. We have seen that techniques like Blocking or Horizontal Partitioning can achieve significant reductions in I/O and memory traffic in this layer. These techniques provide significant improvements in the different implementations of the Join operation; Nested Loop (Blocking) and Hash Join (Hybrid Hash Join, Hash Teams).

The Access Methods module provides records to the Executor layer. The Access Methods module is capable of managing the physical structure of the data and index pages in a DBMS. By using variants of Vertical Partitioning, it is possible to organize data files in such a way that I/O is reduced and locality at the memory hierarchy is exploited better. In terms of index structure, a key aspect is compression and how the index nodes of B-trees are structured to reduce the search time for attributes.

The Buffer Management layer provides the pages that store data and index structures to the Access Methods layer. The Buffer Manager has the aim of reducing the amount of page misses for a given amount of memory to manage. We have seen that there are different page replacement techniques that range from global policies to local policies. We have also explained that the local replacement policies are more effective than the global policies in reducing the miss ratio.

Finally, we have given hints about how the hardware and base software influence the execution of DBMSs. On one hand, compilers can restructure the code of DBMSs so that more instruction locality can be obtained. On the other hand, the hardware may be enhanced (larger cache lines/blocks) so that more locality can be extracted from Database Codes running Decision Support Systems and Data Warehousing workloads.

Acknowledgements

The author would like to thank students Josep Aguilar, Roque Bonilla, Daniel Jiménez-González, and Victor Muntés for their discussions on Database Management; Carlos Navarro, Alex Ramirez, and Xavi Serrano for the seminal work they did in the Database group; Calisto Zuzarte, Adriana Zubiri, and Berni Schiefer for their discussions on DBMS structure; and Christian Breimann, Peter Sanders, and Martin Schmollinger for their useful comments to improve this chapter.

15. Hierarchical Models and Software Tools for Parallel Programming

Massimo Coppola and Martin Schmollinger

15.1 Introduction

Hierarchically structured architectures are becoming more and more pervasive in the field of parallel and high performance computing. While memory hierarchies have been recognized for a long-time, only in the last years hierarchical parallel structures have gained importance, mainly as a result of the trend towards cluster architectures and high-performance application of computational grids.

The similarity among the issues of managing memory hierarchies and those of parallel computation has been pointed out before (see for instance [213]). It is an open question if a single, unified model of both aspects exists, and if it is theoretically tractable. Correspondingly, a programming environment which includes support for both hierarchies is still lacking. We thus need well-founded models and efficient new tools for hierarchical parallel machines, in order to connect algorithm design and complexity results to high-performance program implementation.

In this chapter we survey theoretically relevant results, and we compare them with existing software tools and programming models. One aim of the survey is to show that there are promising results with respect to the theoretical computational models, developed by merging the concepts of bulk-parallel computational models with those from the hierarchical memory field. A second goal is to investigate if software support has been realized, and what is still missing, in order to exploit the full performance of modern high-performance cluster architectures. Even in this case, solutions emerge from combining results of different nature, those employing hardware-provided shared memory and those explicitly dealing with message passing. We will see that both at the theoretical level and on the application side, combination of techniques from different fields is often promising, but still leaves many open questions and unresolved issues.

The chapter is organized in three parts. The first one (Sect. 15.2) describes the architectural background of current parallel platforms and supercomputers. The basic architectural options of parallel architectures are explained, showing that they naturally lead to a hierarchy concept associated with the exploitation of parallelism. We discuss the technological reasons for, and future expectations of current architectural trends.

The second part of the chapter gives an overview of parallel computational models, exploring the connection among the so-called *parallel bridging models* and external memory models, here mainly represented by the *parallel disk model* (PDM) [754]. There is a similarity between problems in bulk parallelism and block-oriented I/O. Both techniques try to efficiently exploit locality in mapping algorithmic patterns to a hierarchical structure. We discuss the issues of parallel computation models in Sect. 15.3. In Sect. 15.4 we get to discuss parallel bridging models. We survey definitions and present some models of the class. We describe their extensions to hierarchical parallelism, and survey results on emulating their algorithms using sequential and parallel external-memory models. At the end of Sect. 15.4 we describe two results that exploit parallel hierarchical models for algorithm design.

The third part of the chapter shifts toward the practical approach to hierarchical architectures. Sect. 15.5 gives an overview of software tools and programming models that can be used for program implementation. We consider libraries for parallel and external-memory programming, and combined approaches. With respect to parallel software tools, we focus on the existing approaches which support hierarchy-aware program development. Section 15.6 summarizes the chapter and draws conclusions.

15.2 Architectural Background

In the following, we assume the reader is familiar with the basic concepts of sequential computational architectures. We also assume the notions of process and thread[1] are known. Table 15.1 summarizes some acronyms used in the chapter.

Parallel architectures are made up from multiple processing and memory units. A network connects the processing units and the memory banks (see Fig. 15.1a). We refer the reader to [484], which is a good starting point to understand the different design options available (the kind of network, which modules are directly connected, etc.). We only sketch them here due to lack of space.

Efficiency of communication is measured by two parameters, *latency* and *bandwidth*. Latency is the time taken for a communication to complete, and bandwidth is the rate at which data can be communicated. In a simple world, these metrics are directly related. For communication over a network, however, we must take into account several factors like physical limitations, communication startup and clean-up times, and the possible performance penalty from many simultaneous communications through the network. As a very general rule, latency depends on the network geometry and implementation, and bandwidth increases with the length of the message, because of the lesser

[1] Largely simplifying, a *process* is a running program with a set of resources, which includes a private memory space; a *thread* is an activity within a process. A thread has its own control flow but shares resources and memory space with other threads in the same process.

and lesser influence of fixed overheads. Communication latency and bandwidth dictate the best *computational grain* of a parallel program. The grain is the size of the (smallest) subproblem that can be assigned to a different processor, and plays a main role in the trade-off between exploited parallelism and communication overheads.

Extreme but realistic examples of networks are the *bus connection* and the $n \times n$ *crossbar*. The former is a common hardware channel which can link only a pair of modules at a time. It has the least circuital complexity and cost. The latter is a square matrix of switches that can connect up to n non-conflicting pairs of modules. It achieves the highest connectivity at the highest circuital cost. A lot of systems use structures that are in between these two, as a compromise between practical scalability and performance.

Different memory organizations and caching solutions have been devised to improve the memory access performance (see Fig. 1b, 1c).

Almost all modern parallel computers belong to the MIMD class of parallel architectures. This means that processing nodes can execute indepent programs over possibly different data. The MIMD class is subdivided in [744] according to the characteristic of the physical memory, into *shared memory* MIMD (SM-MIMD) and *distributed memory* MIMD (DM-MIMD).

The memory banks of a shared memory MIMD machine form a common address space, which is actively supported by the network hardware. Different processors can interfere with each other when accessing the same memory module, and race conditions may show up in the behaviour of the programs. Therefore, hardware lock and update protocols have to be used to avoid inconsistencies in memory and among the caches[2]. Choosing the right network structure and protocols are critical design issues, which drive the performance of the memory system. Larger and larger shared memory machines lead to difficult performance problems.

Multi-stage crossbars (networks made of smaller interconnected crossbars) are getting more and more important with the increasing number of processors in shared memory machines. If the processors are connected by a multistage crossbar and each processor has some local memory banks, there is a memory hierarchy within the shared memory of the system.

[2] We do not analyze in depth here either cache coherence issues (see Chapter 16), multi-stage networks or cache-only architectures [484].

Table 15.1. Main acronyms used throughout the chapter.

MIMD	multiple instruction, multiple data	EM	external memory
SM-MIMD	shared-memory MIMD	PDM	parallel disk model
DM-MIMD	distributed-memory MIMD	HMM	hierarchical memory model
SMP	symmetric multiprocessor	PBM	parallel bridging model
		BSP	see Sec. 15.4, page 329
SDSM	software distributed shared memory	CGM	see Sec. 15.4, page 330
LAN	local area network	QSM	see Sec. 15.4, page 331
		LogP	see Sec. 15.4, page 331

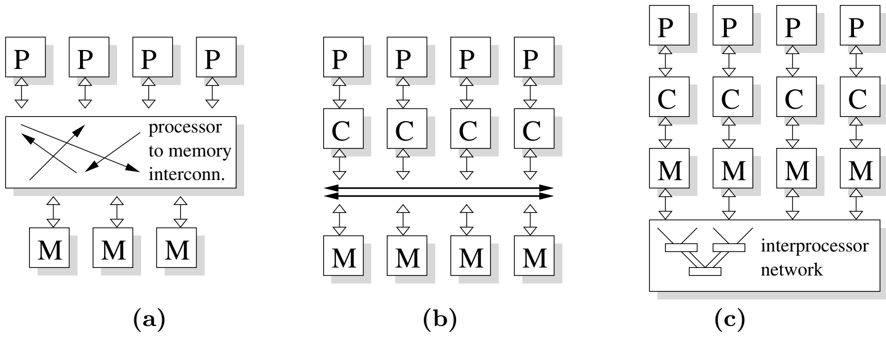


Fig. 15.1. Overall structure of DM and SM-MIMD architectures. (a) Generic MIMD machines have multiple processors and memory banks (not necessarily the same number) – (b) example of SMP with multiple memory banks, processor caches and a bus interconnection – (c) example of a MIMD architecture composed of single-processor nodes and a more sophisticated interconnection network. Depending on the network implementation, this can be either a DM-MIMD or a NUMA SM-MIMD architecture.

Systems in which the access from a processor to some of the memory banks, e.g. its local one, is faster than access to the rest of the memory are called non-uniform memory access systems (NUMA), in contrast with uniform memory access systems (UMA). Shared memory architectures, both uniform and non-uniform ones, are often called Symmetric Multiprocessors (SMP), because the architecture is fully symmetric from the point of view of the running programs.

In contrast to the shared memory machines, in a distributed memory machine each processing node has its own address space. Therefore, it is up to the user to define efficient data decompositions and explicit data exchange patterns for the applications. Even distributed memory machines have to confront with issues concerning the interconnection structure among the processing nodes. Popular networks for distributed memory machines are the hypercube, the 2D or 3D meshes, and multi-stage crossbars. Each processing node has its own local memory, so distributed memory architectures obviously are non-uniform memory access architectures. The network is inherently slower than the local memory, hence we have a memory hierarchy in distributed memory MIMD machines too. However, distributed memory architectures are less demanding with respect to the interconnection network[3], so they are much more scalable than the shared memory ones.

In recent years, a strong trend has emerged in the field of high performance computers towards two kinds of architectures, (1) clusters of vector computers and (2) clusters of scalar uni- and multiprocessors. Looking at the list of the

[3] For instance, coherence and locking problems are not dealt with at the hardware level. This removes some design constraints, and reduces communication overheads.

fastest 500 supercomputers in the world, the majority of them belongs to these two classes [103, 549], with the latter steadily gaining more share.

Especially clusters of SMP nodes (*SMP clusters*, in the following) are a more and more popular architecture for building supercomputers. At different scales, these systems can be classified both as distributed memory and as shared memory architectures, because SM-MIMD processing nodes are connected together to form a larger distributed memory machine. The result is a powerful parallel architecture, which combines the high effectiveness of small shared-memory computing nodes with the scalability of the distributed memory parallelism among the nodes.

In principle, we could classify SMP clusters either as shared memory or as distributed memory MIMD depending on the existence of a common address space abstraction for all the processors. A shared space can be provided by the network hardware and firmware, or only by software means on top of a general purpose network (see Section 15.5.3). However, in the latter case algorithms which exploit memory locality within SMP nodes incur much lesser communication overheads. We can thus enhance performance if we explicitly consider SMP clusters as architectures with a *parallel hierarchy* of at least two levels. The number of levels may actually be higher, depending on the topology of the intra- and inter-node networks.

Grid or metacomputing technologies [309], where supercomputers or clusters of workstations are connected with each other to run applications, result in even more levels and a less regular parallel hierarchy. Broadband connec-

Table 15.2. Parameters for different levels in a hierarchical parallel architecture. The reference commodity architecture is an Intel IA32 microprocessor core from a Pentium4/Xeon. Network bandwidth and latency are measured w.r.t. a node.

Layer	Peak Bandwidth	Latency Best	Latency Worst	Notes
CPU and caches	10-100 GB/s	< 1 ns	200 ns	(1)
SM communication	< 1 GB/s	200 ns	$\sim$1000 ns	(2)
DM communication	1-400 MB/s	$\sim 2\ \mu$s	2 ms	(3,4)
Local I/O	10-100 MB/s	4 ms	30 ms	(4)
DM communication (WAN)	10-100 MB/s	2 ms	1s	(5)

1. Worst-case latency is that of reading a block from memory into the external cache. In the IA32 architecture, a cache line is 64 bytes.
2. Processors share the same memory bus through separate external caches. A shared memory communication implies at least a L2 cache fault. The worst-case accounts for other issues like acquiring hardware and software locks, and thread scheduling.
3. From slow Ethernet up to Gbit Ethernet and Myrinet [744].
4. We do not include here using in parallel several disks, and multiple network interfaces per node.
5. Multi-Gigabit geographic networks are being already built, but wide area network (WAN) have higher latencies [309, sec. 21.4].

tions, ranging from local area networks to geographic ones, add more levels to the hierarchy, with different communication bandwidth and latency [309, chapter 2].

Summing up, in modern parallel architectures we have the following hierarchy of memory and communication layers.

- shared memory
- distributed memory
- local area network
- wide area network

Each one of these layers may exhibit hierarchical effects, depending on its implementation choices.

The effects on latency and bandwidth of the parallel hierarchy are similar and combine with those of the ordinary memory hierarchy. A crucial observation is that there is no strict order among the levels of these two hierarchies, which we can easily exploit to build a unitary model. For instance, we can see in Table 15.2 that the communication layers (both shared memory and distributed memory based ones) provide a bandwidth lower than main memory, and in some cases lower than that of local I/O. However, their latency is usually much lower than that of mechanical devices like disks. Different access patterns thus lead to different relative performances of communication and I/O.

Assessing the present and future characteristics of the parallel hierarchy [193] and devising appropriate programming models to exploit it are among the main open issues in modern parallel/distributed computing research.

15.2.1 Motivation and Technological Perspective

As we explained in the last section, parallel computing architectures employ memory and parallel hierarchies. In the following we summarize arguments that explain the trend towards even more hierarchical architectures than SMP clusters, and we discuss possible future developments.

In [438] some main advantages of SMP cluster architectures are found, most of them being technological and economical considerations.

- Standard off-the-shelf processors are getting faster and faster, even with respect to special purpose architectures. Because of mass production, the performance/price ratio of commercial parts is consistently better than that of special purpose processors (e.g. vector processors). Architectures made from commodity processors are going to be increasingly preferred to build SMPs, massively parallel and cluster machines.
- A similar effect shows up for network and other architecture components. Clusters ans small SMP, which employ standard components, will take advantage of this phenomenon and will become cheaper and more scalable. As soon as the performance advantages of the special purpose networks

for massive parallelism disappear, SMP clusters will supersede massively parallel architectures.
– SMP clusters are scalable and expandable. Their architecture is intrinsically more scalable, and it is moreover practically expandable by adding more nodes and/or upgrading processing nodes. While it is often not possible to add processors in a monolithic SMP or in a special purpose connection, it is easy to build a SMP cluster step by step.
– The size of memory, disk subsystem capacity and bandwidth are critical resources in a supercomputer. SMP clusters are characterized by a greater total memory size and number of disks. Hence, it is possible to have more active jobs, which even have larger data storage available.
– Still according to [438], most software for massively parallel processors or SMPs can easily be ported to SMP clusters achieving similar efficiency. With the knowledge of software for SMP machines and the already existing software for the massively parallel processors, it should be possible to provide a powerful environment for parallel software development and execution. We will give an overview of the efforts in this direction in Sect. 15.5.

Some of the preceding considerations have been recognized years ago, while others are a more recent discovery. According to Bell and Gray [103], a similar trend will last for more than a while. They depict a scenario of the evolution of parallel computers and computing grid architectures, which implies a change in the role played by Beowulf architectures and supercomputing centers.

Beowulf clusters are parallel machines built using commodity hardware and software, where the hardware can even be a mixture of uniprocessor workstations and small SMPs. Companies and research institutes, formerly users of supercomputers and proprietary software, will gradually start to build their own large Beowulf machines, which are more cost-effective than remotely hosted super-computers. Moreover, thanks also to the increased Internet bandwidth, cluster and Grid [309] technologies will merge. It will then become possible to merge Beowulf computational resources into larger, geographically distributed clusters. Applications that tolerate high communication latencies will easily be able to exploit a processing power measured in Teraflops.

In this perspective, applications which need a large amount of shared memory are seen as a weak point of Beowulf clusters. Computing centers will still exist and host large vector machines and exotic architectures (like processor-in-memory and cellular supercomputers) for the sake of running these applications. In addition, centers will have the role of resource brokers for computational grids, managing net-distributed clusters, and will provide storage for peta-scale data sets. According to this analysis [103], hierarchical parallel systems will be the principal computing structure in the future. Therefore, research on programming environments and in understanding hierarchical parallelism has an essential role.

15.3 Parallel Computational Models

We have already seen in this book that the classical random access machine (RAM) sequential model, the archetype of the von Neumann computer, does not properly account with the cost of memory access within a hierarchy of memories. The PDM model [754], and more complex multi-level computational models have been developed to increase prediction accuracy with respect to the practical performance of algorithms.

The same has happened in the field of parallel algorithms. The classical parallel random access machine (PRAM) computational model is made up of a number of sequential RAM machines, each one with its local memory. These "abstract processors" compute in parallel and can communicate by reading and writing to a global memory which they all share. Several precise assumptions are made to keep the model general.

- Unlimited resources: no bound is put on the number of processors, the size of local or global memory.
- Parallel execution is fully synchronous, all active processor always complete one instruction in one time unit.
- Unitary cost of memory access, both for the local and global memory.
- Unlimited amount of simultaneous operations in the global memory is allowed (though same-location collisions are forbidden).

These assumptions are appropriate for a theoretical model. They allow to disregard the peculiarities of any specific architecture, and make the PRAM an effective model in studying abstract computational complexity. However, they are not realistic for the majority of physical architectures, as practical bandwidth constraints, network traffic constraints and locality effects are completely ignored.

Considering the issues discussed in Sect. 15.2, we see that real MIMD machines are much more complex. Synchronous parallel execution is usually impossible on modern parallel computers, as well as to ensure constant, uniform memory access times independently of machine size, amount of exchanged data and exploited parallelism. Indeed, optimal PRAM complexity is often misguiding with respect to real computational costs.

Several variants of the PRAM model have been devised with the aim of reconciling theoretical computational costs with real performance. They add different kinds of constraints and costs on the basic operations. A survey on these derived models is given in [354]. We do not even discuss the research on communication and algorithmic performance of models which use a fixed network structure (e.g. a mesh or hypercube). Despite the results on network cross-simulation properties, network-specific algorithms are often too tied to the geometry of the network, and show a sub-optimal behavior on other kinds of interconnection.

We focus on different research track, which has started in the last years and involves the class of parallel bridging computational models. Parallel bridging models aim at exploiting a higher degree of hardware independence, and some of them explicitly consider the grain of the computation as a key factor in devising efficient and practical algorithms. In the following section we concentrate on this approach to modeling performance. We start from a description of the approach and of the most widely adopted models of this class, and then we survey some of the results about the connections with hierarchical memory models for sequential programming.

15.4 Parallel Bridging Models

Computational models should exhibit the right balance of abstraction and detail, to reflect the actual behavior of parallel algorithms while keeping the analysis tractable. As in the sequential case, ideal parallel models should satisfy both efficiency and universality requirements, so that most results about computational costs can be applied in practice and that they are essentially unaffected by the underlying architecture.

The need to develop such a "reference" parallel model has lead in the 1990s to the concept of a *parallel bridging model* (PBM). A bridging model should act as a standard, aiming at the best separation of algorithm and software development from the architecture. Several models of this class have been developed. They use a more abstract approach in modeling the interconnection architecture.

One of the pioneering works in the field is [742] by Valiant, where the following goals stemming from the initial definition are stated.

Cost measure. A PBM should have a cost measure to guide algorithm development, detailed enough for accurate performance prediction. The model should be independent of a specific architecture and technology, but it should reflect the fundamental constraints of parallel machines.

Efficient universality. Mapping high-level PBM algorithms onto actual machines should not lead to reduced efficiency. We wish to avoid logarithmic simulation losses, and aim at constant bounded inefficiency, so that we can afford developing algorithms for the PBM model only.

Neutrality and Portability. A PBM should be neutral with respect to the number of processors, i.e. results should be applicable not only asymptotically, but also for very small parallel machines. It should also allow the programmer to write fully portable programs, avoiding explicit memory management, low-level communications and synchronizations. We are thus requiring that our computational model is a good *programming* model too.

Parallel slackness. This is the amount of excess parallelism in the algorithm needed to achieve optimal execution. A PBM program written for v virtual processors should be optimally simulated on p physical processors if p is rather smaller than v (e.g. $v = p \log p$). The approach ensures that there is a

p	number of processors
L	message latency / synchronization
g	cost parameter for message routing
for processor i in superstep t	
$w_{i,t}$	local computation
$\lambda_{i,t}$	num. sent messages
$\mu_{i,t}$	num. received messages
$w_t = \max_i w_{i,t}$	global work in t
$h_t = \max_i \max\{\lambda_{i,t}, \mu_{i,t}\}$	global routing in t
$w_t + g \cdot h_t + L$	cost of superstep t

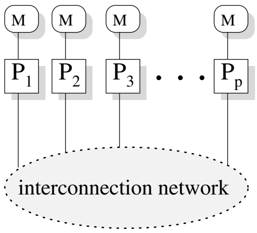


Fig. 15.2. BSP symbols and parameters (*left*). The BSP abstract architecture (*right*).

chance to superimpose computation and communication of different virtual processors on a wide range of interconnection networks. The higher degree of asynchronism introduced in PBMs helps to avoid that fine-grain parallelism negatively affects the algorithm execution.

15.4.1 The Bulk Synchronous Parallel Model

The bulk synchronous parallel model (BSP) that Valiant proposes (as described in [75]) is made up of a set of p processing nodes with local memories and a complete interconnection network which delivers messages between pairs of nodes (Fig. 15.2). Three parameters are used to specify the underlying hardware: the number p of processors, a latency parameter L, which is the maximum latency of a message or synchronization in the network, and a gap parameter g, which is the amount of time that a processor must wait after a send operation before sending a new message.

A BSP computation consists of *supersteps*. During a superstep processors compute using their local memory and exchange a certain amount of messages with each other. Messages sent during superstep t are received only at the beginning of superstep $t + 1$. The components of a superstep, represented in Fig. 15.3a, are thus a computation phase (the grey stripes), a varying routing relation (which expresses the pattern of message exchange) and the constant-bounded synchronization time L (the white vertical stripes). The table of Fig. 15.2 summarizes the essential notation of the BSP model, with respect to a superstep t and a processing node i.

We define $h_{i,t} = \max(\lambda_{i,t}, \mu_{i,t})$ as the largest number of messages sent or received by processor i during current superstep. The routing relation among the nodes (the relation among sender and receiver processors for the set of all messages) has size $h_t = \max_i h_{i,t}$ for superstep t. It is usually called h-relation, to emphasize the fact that we look at its h parameter.

With these parameter definitions, w_t and h_t are the largest w and h values in superstep t. If we imagine that supersteps are globally synchronized

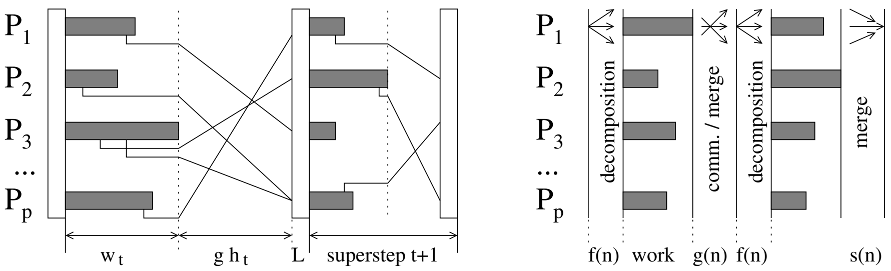


Fig. 15.3. (a) BSP superstep execution (b) CGM supersteps

by barriers (i.e. all units must complete a superstep before any of them can proceed to the next superstep), we can estimate the length of a superstep in time units as $w_t + g \cdot h_t + L$. This value becomes an upper bound if we assume instead that synchronization is only enforced when actually needed (e.g. before receiving a message).

We can analyze a BSP algorithm by computing w_t and h_t for each superstep. If the algorithm terminates in T supersteps, the *local work* $W = \sum_t w_t$ and the *communication volume* $H = \sum_t h_t$ of the algorithm lead to the cost estimate $W + g \cdot H + L \cdot T$. A more sound evaluation compares the performance of the algorithm with that of the best known sequential algorithm. Let T_{seq} be the sequential running time, we call *c-optimal* a BSP algorithm that solves the problem with $W = c \cdot T_{\text{seq}}/p$ and $g \cdot H + L \cdot T = o(T_{\text{seq}}/p)$ for a constant c.

Other parallel bridging models have been developed. Among them we mention the CGM model, which is the closest to the BSP, the LogP and the QSM models, which we describe below.

The Coarse-Grained Multicomputer. The *coarse-grained multicomputer* (CGM) model [245] is based on supersteps too. The communication phase in CGM is different from that of BSP, as it involves all processors in a global communication pattern, and $O(n)$ data are exchanged at each communication phase. In Fig. 15.3b the different patterns are the f, g, s functions. Only two numeric parameters are used, the number of nodes p and the problem size n. Each node has thus $O(n/p)$ local memory.

In the original presentation the model was parametric, as the network structure was left essentially unspecified. The communication phases were allowed to be any global pattern (e.g. sorting, broadcasts, partial sums) which could be efficiently emulated on various interconnection networks. To get the actual algorithmic cost, one should substitute the routing complexity of the parallel patterns on a given network (e.g. $g(n,p)$ may be the complexity of exchanging $O(n/p)$ keys in a hypercube of diameter $\log_2 p$).

The challenge in the CGM model is to devise a coarse-grain decomposition of the problem into independent subproblems by exploiting a set of “portable” global parallel routines. The best algorithms will usually require the smallest possible number of supersteps. During the years, in the common use CGM has been simplified and became close to BSP. In recent works [243, 244], the network geometry is no longer considered. CGM algorithms are defined as a special class of BSP algorithms, with the distinguishing feature that each CGM superstep employs a routing relation of size $h = \Theta(n/v)$.

LogP Model. In the LogP [233] model, processors communicate through point-to-point messages, ignoring the network geometry like in BSP. Unlike BSP and the other PBMs, there are no supersteps in LogP.

LogP models physical communication behavior. It uses four parameters: l, an upper bound on communication latency; o, the overhead involved in a communication; g, a time gap between sending two messages; and the physical parallelism P. Messages are considered to be of small, fixed length, thus introducing the need to split large communications. There are two flavors of the model, *stalling* LogP, which imposes a network capacity constraint (a processor can have no more than $\lceil l/g \rceil$ messages in transit to it at the same time, or senders will stall), and *non-stalling* LogP, which has no constraint.

Because of the unstructured and asynchronous programming model, of the need to split messages into packets, and to deal with the capacity constraint, algorithm design and analysis with LogP is more complex than with the other PBMs. There are comparably fewer results with LogP, even if most basic algorithms (broadcasts, summing) have been analyzed.

QSM Model. The *queuing shared memory* (QSM) [330] model can be seen both as a PRAM evolution and as a shared-memory variant of the BSP. Like a PRAM, a set of processors with private memories communicate by means of a shared memory. Like in the BSP, QSM computation is globally divided into phases. Read and write operation are posted to the shared memory, and they complete at the end of a phase. Concurrent reads or writes (but not both) to a memory location are allowed.

Each processor must also perform a certain amount of local computation within each phase. The cost of each phase is defined as $\max(m_{op}, g \cdot m_{rw}, \kappa)$, where m_{op} is the largest amount of local computation in the phase, m_{rw} is the largest number of shared reads and writes from the same processor, and the gap parameter g is the overhead of each request. Latency is not explicitly considered, and it is substituted by the *maximum contention* κ of the phase, i.e. the maximum number of colliding accesses on any location in that phase. A large number of algorithms designed for variants of the PRAM can be easily mapped on the QSM.

A Comparison of Parallel Bridging Models. Several results about emulation among different parallel bridging models can be found in the literature. Emulations are *work-preserving* if the product $p \cdot t$ (processors per execution time) on the emulating machine is the same as that on the machine being emulated, to within a constant factor. Work-preserving emulations typically increase the amount of parallel slackness (the emulating machine has fewer processor than the emulated one), and are characterized by a certain *slowdown*. The slowdown is $O(f)$, when we are able to map an algorithm, running in t time on p processors, to one running on $p' \leq p/f$ processors in time $t' = O(t \cdot (p/p'))$. An ideal slowdown of 1 means that the emulation introduces at most a constant factor of inefficiency. Table 15.3 summarizes some asymptotic slowdown results taken from a recent survey by Ramachandran [622]. The fact that a collection of work-preserving emulations with small slowdown exists, suggests that these models are to a good extent equivalent in their applicability as cost models to real parallel machines.

However, some of the bridging models are better suited than others for the role of programming models, as a more abstract view of the algorithm structure and communication pattern allows easier algorithm design and analysis.

From this point of view, LogP is probably the hardest PBM to use. It leads to difficult, low-level analysis of communication behavior, and thus it has been rarely used to evaluate complex algorithms. The QSM can be used to evaluate the practical performance of many existing PRAM algorithms, but is a low-level model too. QSM is a “flat” model, which disregards the hierarchical structure of the computation, and it has an abstract but fine-grain approach to communication cost.

The bulk parallel models (BSP and CGM) have been used more extensively to code parallel algorithms. They proved to be easier to use when designing algorithms, and actually several software tools have been designed to directly implement BSP algorithms. In the same direction there are even more simplified model, like the one used in [690]. Aggregate computation cost is measured in terms of total work, total network traffic (sum of messages) and total number of messages. Thus the three “weights” of these operations are the parameters of this model. It can be seen as a flat, close relative of BSP and CGM, and at least a class of algorithms based on computation phases with limited unbalancing can be analyzed using this model.

Both the CGM, and extensions of the original BSP model, allow to represent hierarchically structured networks. Finally, the concepts of parallel slackness, medium-grain parallelism and supersteps have been exploited to develop efficient emulation of BSP and CGM algorithms in external memory,

Table 15.3. Slowdown of work-preserving emulation between PBMs. Most of these results concern randomized emulation algorithms. See [622] for details and references.

	Emulating model		
Emulated model	BSP	LogP (stalling)	QSM
BSP		$\log^4 p + (L/g)\log^2 p$	$\lceil (g \log p)/L \rceil$
LogP (non stalling)	L/l (det.)	1 (det.)	$\lceil (g \log p)/l \rceil$
QSM	$(L/g) + g \log p$	$\log^4 p + (l/g)\log^2 p + g \cdot \log p$	

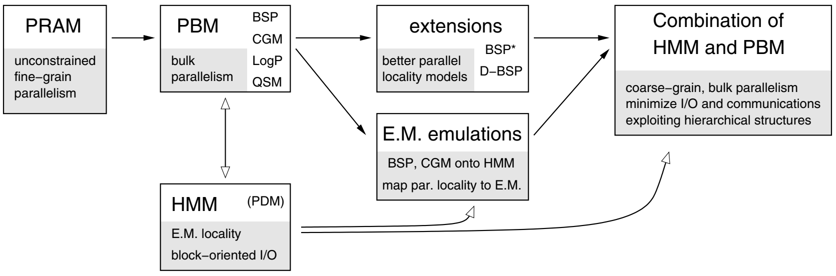

PRAM
unconstrained fine–grain parallelism
PBM
BSP
CGM
LogP
QSM
bulk parallelism
extensions
BSP*
D–BSP
better parallel locality models
E.M. emulations
BSP, CGM onto HMM
map par. locality to E.M.
Combination of HMM and PBM
coarse–grain, bulk parallelism
minimize I/O and communications
exploiting hierarchical structures
HMM
(PDM)
E.M. locality
block–oriented I/O

Fig. 15.4. Development path of parallel bridging models and their relationship with hierarchical memory models

showing up the connections among the design of parallel algorithms and that of external-memory algorithms.

15.4.2 Parallel Bridging Models and Hierarchical Parallelism

As earlier noted by Cormen and Goodrich [213], the bulk-processing nature of external memory algorithms is close to the bulk-parallel approach of most parallel bridging models. Both kinds of models aim at properly accounting in an abstract way for a kind of access locality. Moreover, most practical problems involving large-scale data structures are definitely a target both for parallel computing and for secondary memory computing techniques. Cormen and Goodrich fostered the development of a single computation model which included bulk-like measures for computation, communication and I/O. This will be our main topic in the following.

Two different research paths, shown in Fig. 15.4, have been developed from parallel bridging models by introducing a hierarchy concept. One path aims at enhancing the performance accuracy of parallel bridging models. This is done by exploiting the concepts of processor locality and block-oriented communications. In doing this, solution are exploited which mimic those of hierarchical memory models. A different path, discussed in Sect. 15.4.3, explores the relationships among PBMs and HMMs, by developing equivaleces and emulation procedures that turn bulk-parallel algorithms into external memory ones. As we will see in Sect. 15.4.4, there are some results which show a possible convergence of the two approaches.

Because of the initial aim of modeling parallel computation, the main line of development of PBMs is directed towards a more accurate model of parallel locality effects. Most parallel architectures exploit regular, hierarchical structures which reward local data access. Even without making explicit geometric or implementation assumptions about the network, parallel bridging models can be refined by including parameters that indirectly reflect actual communication behavior.

The BSP communication model is simple and abstract enough to be refined this way. We describe two different extensions of BSP, which account for (i) effects due to message length and (ii) for the relationship between network size and parallel overhead in communications.

The BSP* Model. In real interconnection networks, communication time is not independent of message length. The combination of bandwidth constraints, startup costs and latency effects is often modeled as a linear affine function of message length. BSP disregards this aspect of communication. The number of exchanged messages roughly measures the congestion effects on the network.

Counting non-local accesses is a first-order approximation that has been successfully used in external memory models like the PDM. On the other hand, PDM uses a block size parameter to measure the number of page I/O operation. To model the practical constraint of efficiency for real communications, the BSP* model [75] has been introduced in 1996.

BSP* adds a *critical block size* parameter b, which is the minimum size of data for a communication to fully exploit the available bandwidth. The cost function for communications is modified to account both for the number h_t of message start-ups in a phase, and for the communication volume s_t (the sum of the sizes of all messages). Each message is charged a constant overhead, and a time proportional to its length in blocks. Superstep cost is defined as $w_t + g(s_t/b + h_t) + L$, often written as $w_t + g^* \cdot (s_t + h_t \cdot b) + L$, where $g^* = g/b$. The effect on performance evaluation is that algorithm that pack information when communicating are still rewarded like in BSP, but high communication volumes and long messages are not. Thus the BSP* model explicitly promotes both block-organized communications and a reduced amount of data transfers, the same way external memory models do for I/O operations.

D-BSP Model. The BSP model inherits from the PRAM the assumption that model behavior is independent of its size. While we can easily account for a general behavior by changing parameter values (e.g. adding processors to a bus interconnection leads to larger values of g and L), there is no way we can model more complex situations where the network properties change according to the part of it that we are using. This is an intentional trade-off of the BSP model, but it can lead to inaccurate cost estimates in some cases. We mention two examples.

– Networks with a regular geometry, like meshes or hypercubes, can behave quite differently if most of the communication traffic is local, as compared to the general case.
– Modern cluster of multiprocessors and multiple-level interconnections cannot be properly modeled with any value of g, L, as shared memory and physical message-passing communications among different kinds of connections imply very different bandwidths and overheads.

De La Torre and Kruskal [240] introduce the *decomposable* BSP (D-BSP), which rewards locality of computation by allowing hierarchical decomposition of the machine into smaller BSP-submachines. The g and L parameters of BSP are replaced with two functions, $\mathcal{G}_m$ and $\mathcal{L}_m$, of the submachine size m.

During algorithm execution, the computation can be recursively split, and smaller subproblems can be assigned to different submachines. The subcomputations are still D-BSP computations, which proceed independently until they merge together again. Their computational cost is the maximum of the costs, each one being evaluated with appropriate g, L values. The actual shape of $\mathcal{G}_m, \mathcal{L}_m$ controls the advantage of decomposing the computation into local subcomputations.

This abstract way of accounting for parallel locality avoids directly dealing with the geometry of the interconnection structure, which appears only by means of its characteristic functions. D-BSP thus adds the power to explicitly evaluate architectural effects on communications due to the geometry of the network, or due to its implementation.

A first comparison among the D-BSP and BSP models can be found in [111]. Meyer auf der Heide and Wanka [75] investigated the relationships among the BSP* and the D-BSP models. Bilardi and others [126] also examine the D-BSP model, concluding that it offers the same design advantages of BSP, but has higher effectiveness and portability over realistic parallel architectures. They show results for the family of functions $\mathcal{G}_m, \mathcal{L}_m$ of the form $C \cdot (n/2^i)^\alpha$, where $m = 2^i, \quad (0 \le i \le \log n)$ and $(0 < \alpha < 1)$. These functions capture a wide family of commonly used interconnection networks with n nodes, including multidimensional arrays.

15.4.3 Emulation in External Memory

A classical result of emulation of parallel algorithms using external memory techniques is the PRAM emulation algorithm in [192]. The result is described in Chapter 3. It is an asymptotically good emulation, but the asymptotic complexity hides quite large constant factors, as an external-memory sorting of the whole memory is required at each PRAM execution step.

It is clearly a fundamental issue to distinguish those emulations that are only asymptotically good, from those that can be practically exploited.

The emulation of PBM algorithms in external memory models has been shown to improve over known external memory algorithms in some cases, at least from the point of view of the abstract I/O complexity. The topic of external memory algorithm design is addressed in other parts of the book. Here we want to underline the connections among the two fields and the option to merge the two approaches into a more general, hierarchy-aware one.

Sequential BSP-like emulation. A simple simulation algorithm is presented in [692] by Sibeyn and Kaufmann. The emulation is performed by a sequential external-memory machine, simulating a number v of virtual BSP processors.

Once for each superstep, the computational context of each virtual processor is loaded into main memory in turn. A computational context consists of the memory image and message buffers of a virtual processor. Local computation and message exchange are emulated before switching to next processor.

Efficient simulation needs picking the right number v of virtual BSP processors with respect to the emulating machine. By implementing communication buffers using external memory data structures, we can derive efficient external memory algorithms from a subclass of BSP algorithms, those that require limited memory and communication bandwidth per processor. There are interesting points to note.

– the use of a *bandwidth gap* G parameter, measuring the ratio between the instruction execution speed and the I/O bandwidth,
– the introduction of a notion of x-optimality, close to that of c-optimality, which relates the number of I/O operations of a sequential algorithm with those of an emulated parallel algorithm for the same problem,
– the fact that the BSP* messages have a cost depending on their length in blocks helps in determining a relationship among the parallel algorithm and its external memory emulation.

Parallel emulation. In [242] the emulation results hold under more general assumptions. To evaluate the cost of *parallel emulation*, the EM-BSP* *model* is defined. EM-BSP* is a BSP* model extended with a secondary memory which is local to the processing nodes, see Fig. 15.5. Alternatively, we can see it as a PDM model augmented with a BSP* interconnection and a superstep cost function. In addition to the L, g, b, p parameters of BSP*, we find also local memory size M, the number (per processor) of local disks D, the I/O block transfer size B (which is borrowed from the PDM model) and the computational to I/O capacity ratio G as in the simpler simulation.

The emulation of BSP* algorithms proceeds by supersteps, but each emulating processor loads from disk a set of the virtual processors (with their needed context data) at the same time, instead of a single one. The emulation procedure can run sequentially in external memory, or in parallel, where the emulating machine is modeled using EM-BSP*. A reorganization algorithm is provided to perform BSP message routing in the external memories using an optimal amount of I/O.

Like in [692], the result in [242] exploits the BSP* cost function to simplify the emulation algorithm. BSP*, BSP and CGM algorithms (by reduction to BSP*) can be emulated if they satisfy given bounds on message sizes and of the memory used by the processors.

The c-optimality criterion is refined, taking into account I/O, computation and communication time of the emulated algorithm. We thus have a metric to compare EM-BSP* algorithms with the best sequential algorithms known.

Model		additional parameters and features
BSP*	b	critical block size
	$g*$	reduced message cost (g/b)
D-BSP	$\mathcal{G}_m$	g as function of submachine size m
	$\mathcal{L}_m$	L as function of submachine size m
CGM		*constraint on communication steps*
EM (PDM)	M	(node local) memory size
	D	(per node) number of disks
	B	block size for disk I/O
all models	G	*ratio of I/O and computation*

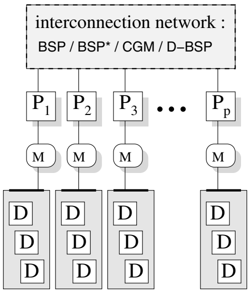


Fig. 15.5. The common structure of the combined parallel and external-memory models, and a summary of parameters used in the models, beyond those from BSP.

Parallel and serial external-memory emulation of bulk parallel algorithms is still under development [243, 244]. New results focus more on the CGM model, which is seen as a submodel of BSP with a known kind of routing relation (see Sect. 15.4.1), which allows a more efficient emulation. Despite technical improvements, the overall structure of the model and of the simulation is essentially unchanged from those of Fig. 15.5. These new works also identify an interesting subspace of algorithmic parameters where parallel external memory execution is efficient. Simulation has thus been used to obtain new or improved complexity results for several external memory problems.

15.4.4 Algorithms Designed for Parallel Hierarchical Models

In this section, we briefly survey two works which exploit mixed models of computation to develop parallel-external memory algorithms. The first work pre-dates most of the results we have previously presented. Aggarwal and Plaxton [16] define a multi-level storage model, made up of a chain of hypercubes of increasing dimension $0 \leq d \leq a$. A set of four primitive operations is defined on such a structure, which includes a scan operation, two routings and a shift of sub-cube data. A hypercube of dimension $b > a$ is then made up of the smaller ones, using bounded-degree networks to connect them through a subset of their nodes. The networks are supposed to compute prefix and scan operations in $O(b) = O(\log p)$ time. Due to its complexity, the model was never further developed despite a promising result on sorting.

Apart from the details of the model and of the sorting algorithm, in [16] it is indeed interesting to note how a parallel, hierarchical data space is defined on which to compute. The authors choose a set of primitive operation that can be practically implemented both in external memory and in parallel. This choice allows a certain degree of flexibility in choosing which levels of the computation to map to external memory, and which ones to perform in parallel.

Newer approaches, following the path of Fig. 15.4 (page 333), are based on external-memory extensions of BSP-like models. Dehne and others [246] use a D-BSP interconnection structure in the configuration of Fig. 15.5, thus taking into account the hierarchy of the communication network and two levels of physical memory. This kind of models is of great practical interest for cluster and grid computing. The set of parameters in the composite model is essentialy the same of the sequential and parallel emulation approaches (table in Fig. 15.5). In particular, we now have a pair of g, G parameters relating computation time respectively with communication and I/O costs.

In [246], a set of basic primitives for sorting, merging and broadcasting is developed on the EM parallel model, by carefully composing D-BSP and PDM algorithms. The other result shown is a geometric algorithm following the distribution sweeping approach (see Chapter 6). It performs input partitioning to execute in parallel local external-memory computations. Intermediate results of local computations need to be exchanged, and composing the local and parallel parts of the algorithm is not simple, as the sweep paradigm does not allow a simple, one-shot input decomposition.

The solution devised is recursive in nature, and it exploits an orthogonal partitioning of the input and of the intermediate results of the recursion, such that the algorithm takes maximum advantage of parallel locality, and all the generated subproblems are independent and can be assigned to smaller subclusters.

15.5 Software Tools

In this section we will survey a set of software tools and programming models that can be used to exploit parallel hierarchical architectures. We classify them according to two main principles, the kind of hierarchy they exploit (either the parallel hierarchy, the memory hierarchy or both), and the number of levels they actually manage, which is two or three. Corresponding to the need for a unifying hierarchical model, we will see that there is a lack of software tools which span across multiple levels and different hierarchies. As a consequence, writing programs that fully exploit hierarchical parallel architectures is a difficult and error-prone task, where a large part of the effort is spent in tuning and debugging activities. In Sect. 15.5.1 we survey tools targeted at one of the two hierarchies, and spanning across two levels. Sect. 15.5.2 presents software systems that exploit parallelism and secondary memory, but only deal with a 2-level parallel hierarchy. Sect. 15.5.3 reports about recent proposals and experiments about extending existing parallel programming models, and developing new ones, which can cope with architecture hierarchy at least within SMP clusters.

15.5.1 Tools for Managing Parallelism or External Memory

Software tools of this first kind manage only two levels of a hierarchy. Since the mangement of parallelism and I/O are in principle completely separate, these tools can be combined within the same environment to exploit architectures which correspond to the EM parallel models of Sect. 15.4.3 and 15.4.4. The work in the two separate fields of external memory and parallel programming is mature enough to have already produced some widely recognized standards.

Parallel Programming Libraries. There are two main parallel programming paradigms, which fit the two extremes of the MIMD architectural class, the distributed memory paradigm and the shared memory paradigm.

In the *message passing paradigm* each process has its local data, and it communicates with other processes by exchanging messages. This *shared nothing* approach corresponds to the abstraction of a DM-MIMD architecture, if we map each process to a distinct processor.

In the *shared-memory programming paradigm*, all the data is accessible to all processes, hence this *shared-everything* approach fits perfectly the SM-MIMD class of architectures. The programmer however has to formulate race-conditions to avoid deadlocks or inconsistencies.

For both paradigms, there is one official or de facto standard library, respectively the *message-passing interface* (MPI) standard, and the OpenMP programming model for shared memory programming. In both, MPI and OpenMP, possible hierarchies in the parallel target machine are not considered. They assume independent processors, either connected by an interconnection network or by a shared memory. Of course, there are approaches to incorporate hierarchy sensitive methods in both libraries. We will present some of them in Section 15.5.3.

Message-Passing-Interface MPI. In 1994, the MPI-Forum unified the most important concepts of message-passing-based programming interfaces into the MPI standard [547]. The current, upward compatible version of the standard is known as MPI-2 [548], and it specifies primitive bindings for languages of the C and Fortran families.

In its simplest form, an MPI program starts one process per processor on a given number of processors. Each process executes the same program code, but it operates on its local data, and it receives a *rank* (a unique identifier) during the execution, that becomes its address w.r.t. communications. Subject to the rank, a process can execute different parts of the program. This single program multiple data (*SPMD*) model of execution actually allows a generic MIMD programming model.

There are MPI implementations for nearly all platforms, which is the prerequisite for program portability. Key features of the MPI standard include the following, and those described on page 342 about the I/O.

Point-to-point communication: The basic MPI communication mechanism is to *exchange messages between pair of endpoint processes*, regardless of the actual network structure that delivers the data. One process initiates a send operation and the other process has to start a receive operation in order to start the data transfer.

Several variants of the basic primitives are defined in the standard, which differ in the communication protocol and the synchronous/asynchronous behavior. For instance, we can choose to block or not until communication set-up or completion, or to use a specific amount of communication buffers.

These different options are needed both to allow optimized implementation of the library and to allow the application programmer to overlap communication and computation.

Collective operations: *Collective communications* involve a group of processes, each one having to call the communication routine with matching arguments, in order for the operation to execute.

Well-known examples of collective operations are the *barrier synchronization* (processes wait for each other at a synchronization point), the *broadcast* (spreading a message to a group of processes) or the scan operation.

One-sided Communications: With *one-sided communication* all communication parameters for both, the sender and the receiver side, are specified by one process, thus avoiding explicit intervention of the partner in the communication. This kind of remote memory access separates communication and synchronization. Remote write, read and update operations are provided this way, together with additional synchronization primitives.

OpenMP. The *OpenMP-API* [593] is a standard for parallel shared memory programming based on compiler directives. Directives are a way to parameterize a specific compiler behavior. They preserve program semantics, and have to be ignored when unknown to a compiler. Thus they are coded as `#pragma` statements in C and C++, and are put within comments in Fortran. OpenMP directives allow to mark parallel regions in a sequential program. This approach facilitates an incremental parallelization of sequential programs.

The sequential part of the code is executed by one thread (master thread) that forks new threads as soon as a parallel region starts and joins them at the end of the parallel region (*fork-join model*). OpenMP has three types of directives.

Parallelism directives mark parallel regions in the program.

Work sharing directives within a parallel region divide the computation among the threads. An example is the *for/DO* directive (each thread executes a part of the iterations of the loop).

Data environment directives control the sharing of program variables that are defined outside a parallel region (e.g. shared, private and reduction).

Synchronization directives (barrier, critical, flush) are responsible for synchronized execution of several threads. Synchronization is necessary to avoid deadlocks and data inconsistencies.

External Memory Programming Libraries. There are libraries designed to simplify processing of external-memory data structures. While the main reference model, PDM, is parallel, these libraries usually only support sequential algorithm operation over parallel disks. To use these libraries in a parallel setting, we can either use independent physical disks, or use independent data structures on a shared device (with a possible performance loss).

TPIE. TPIE [57] is a library developed as a programming tool to simplify the implementation of algorithms based on the PDM model. The assumption of the TPIE approach is that all operations and access methods exploit the best known EM algorithm for the problem, and that any TPIE-based program can immediately benefit of theoretical and practical improvements in EM algorithms, as soon as they are propagated to the library.

TPIE initially provided data structures and algorithms to solve batched problems, providing a strongly stream-oriented interface to the data. A recent work by Arge and others [66] presents an extension of TPIE to deal with random-access data structures.

Common operations on streams are provided (e.g. sort, merge, distribution, permutation) as well as on simple external-memory data types (matrices, stacks). A flexible, general support for external memory trees allows to code several different tree management strategies.

The fundamental components of TPIE are the memory manager, which controls in-core memory utilization, and the *block transfer engine* (BTE), the software kernel that moves blocks of data from physical devices to main memory and back. Two separately designed BTEs deal respectively with streamed and random accesses to the disk, interacting with a common memory manager. The BTEs have different and interchangeable implementations based on the UNIX *stdio* functions, on blocked read/write calls and on memory-mapped I/O.

The current implementation of the library is completely sequential, as each BTE manages a single physical disk, and BTEs on different processors do not cooperate. Interprocess coordination is left to the user like it is in PDM algorithms.

LEDA-sm. LEDA [570] is a commercial library of data structures and algorithms for combinatorial and geometric computation. LEDA-sm is a secondary memory extension [229] which is publicly available under the GPL software license. It provides external data structures (e.g. arrays, queues, trees, strings) with basic operations, and algorithms that work on these structures. Like TPIE, the implementation relies on a library kernel, the EM manager, which implements a PDM abstraction and programming interface over concrete disk devices.

15.5.2 Tools for Parallel-External Programming

In this section we present two tools that allow to implement external-memory aware, parallel programs.

VIC*. The VIC* compiler [214] is a compiler for the C* language, an extension of C with virtual-memory data parallel constructs. The VIC* compiler produces code which interfaces to an implementation of the PDM model.

It allows to fully exploit the PDM model, using parallel disks and processing elements. The user can control the number of computing units and data-server processes that are set up on the target machine. The run-time support of VIC* has been implemented over a set of different sequential and parallel architectures, exploiting existing conventional, networked and parallel file systems. Widely available libraries, including MPI, are used to support the communication.

VIC* has been used to evaluate the actual performance of sequential and parallel PDM algorithms. It shows the effectiveness of the PDM model, as the tested external-memory algorithms are mainly computation bound, whereas their main-memory counterparts are severely I/O bound at the same problem size [214].

While VIC* programs exploit available parallel resources, there is still no model for the communication part. An interesting result in this view is reported in [90] about the problem of external memory, parallel FFT. In [90], the problem is solved by using the dimensional decomposability of the FFT to devise a parallel partitioning of the out-of-core computation.

MPI-IO. The MPI-2 standard includes the specification of a parallel I/O programming interface. Programs written using MPI-IO can exploit message passing parallelism and a shared disk space, while remaining largely portable. A full discussion of parallel file systems is not appropriate here, so we summarize the MPI-IO approach and its rationale.

A typical parallel I/O scenario is that of multiple processors in a MIMD machine (Fig. 15.1b,c) trying to access different parts of a single large file. Shared memory architectures often use centralized I/O subsystems, and the target is to minimize contention due this bottleneck. Distributed memory architectures, on the other hand, usually have local disks in each processing node. In order to exploit these disks as a single storage support within a parallel program, data blocks are sent through the network from hosting nodes to the requesting ones. In both cases, shared and distributed memory, the solution to the performance problem of I/O lies in aggregating several requests to serve them efficiently.

The UNIX-like semantics of most file systems does not allow this transformation [722]. Indeed, the gain is significant if the program explicitly gives information about collective I/O (parallel, logically synchronized I/O requests from a set of processors). MPI-IO provides this interface. MPI *derived datatypes* are a portable mechanism to specify the memory layout of a data structure. They allow MPI functions to minimize communication overheads, and to automatically compact non-contiguous data structures. MPI-2 has extended the use of MPI datatypes from communication to parallel I/O. Since MPI-IO also offers collective and asynchronous I/O functions, there is plenty of room for optimizations.

ROMIO [722, 723] is a public domain implementation of the standard. It is based on a virtual device interface, ADIO, which connects to most sequential, networked and parallel file systems in current use. [722, 723] show that the following two optimizations are fundamental in boosting parallel I/O performance.

Data sieving: Separate asynchronous requests are reordered and merged into bigger ones. In doing this we can afford to read a certain amount of extra data in the "holes", in order to reduce the number of separate I/Os.

Collective request merging: Parallel I/O requests to the same file often address small different regions of it, according to complex patterns which depend on the application (e.g. processor i reads blocks $i+k\cdot j, j=0,1,\ldots$). Merging together all of these patterns we can identify a much simpler I/O pattern at the hosting nodes, which is used to satisfy all the requests by means of a data reorganization phase.

We note two features of the library which are also of more general relevance. The first one is that the same concepts are used for communication and I/O programming interfaces: data types, contexts, asynchronous versus synchronous operations. The second one it that most of the optimizations that are possible with MPI-IO rely on exploiting a form of global architecture-level caching. Data from the external memory level are loaded into memory buffers which are shared by hardware and software means.

15.5.3 Tools for Parallel-Hierarchical Programming

In this section we survey some proposed approaches to the problem of producing efficient programs on hierarchical parallel architectures like SMP clusters. Starting from the shared memory and distributed memory programming standard, we can choose one and try to develop optimizations and extensions, we can try to merge the two, or we can develop new, different programming models. Several approaches are still proposals, but, looking at the available experimental results, those solutions that try to hide the architecture hierarchy to the programmer do not produce the expected performance gain.

Hierarchical Optimizations for MPI. The message-passing paradigm does not consider the hierarchical architecture of SMP clusters. In the following, we present two approaches for adapting MPI to SMP clusters that try do avoid this inefficiency.

Shared-Memory Communication. This approach improves the communication between processors that reside in the same node by using the shared-memory for point-to-point communication. When a message is sent, the system detects if the target process works on a processor that resides in the same node. If this is the case, the message will be delivered through shared memory, instead that over the network. Reducing the number of message copy operation is a well known issue to reduce the host overhead and latency of message-passing communication. For inner-node communication, in-memory copying is the largest part of the message delivery cost, so such an optimization is even more important.

In [712] optimizations are presented of inter-node and inner-node communication for a special MPI implementation, that works on PC-based SMP clusters. Performing an inner-node communication initially requires two message copies, to go from the memory space of one process to that of another one by means of the UNIX kernel primitives. Single-copy operation is achieved by building a dedicated kernel primitive, that writes directly into the receiver's memory.

In order to test the library, the authors performed experiments using the *NAS Parallel Benchmark 2.3* [86]. This benchmark suite is a set of 8 programs designed to help evaluate the performance of parallel supercomputers.

In [712], NAS results on an SMP cluster are compared with those on a cluster of uni-processors with the same number of processors. The SMP cluster achieved 70-100% of the performance of the uni-processor cluster. Intuitively, the SMP clusters should perform better, thanks to the inner-node communication. Several differences between the two clusters can all together explain the results. Communication latency is higher for a process on the SMP, as there is a single, shared network interface per node. Cluster-level synchronization mechanisms are realized by means of messages, and their cost is dominated by inter-node communication delay. Thus, for all applications which do a lot of synchronizations, like those of the NAS benchmark, the inner-node communication performance is wasted.

In conclusion, the approach can improve the average point-to-point communication time, but it it is not guaranteed to improve the overall performance of a program, when compared to that of a cluster of single-processor machines. Indeed, the problem is that the programmer is not forced to consider the SMP cluster architecture at all during the design of an application. The SMP cluster is seen as *flat* parallel machine, thus there is no way to match the program structure with the real architecture.

Threads Only MPI. The *threads only MPI* (*TOMPI*) [248] is an MPI for uni-processor and SMP workstations. The aim is to make the development of MPI programs on workstations less time-consuming. Most standard MPI implementations, for the sake of portability, use UNIX processes and UNIX domain sockets. When working on a single workstation, this method is resource-inefficient and involves an unneeded overhead. TOMPI uses a source code translator to rewrite MPI program into thread-based programs, which can be executed on an SMP workstation.

This approach seems to have the potential to be more efficient than the one based only on shared memory communication. It improves the speed of communications and avoids the large memory overhead due to processes. On the one hand, with TOMPI it is possible to execute MPI programs with hundreds of MPI processes on a single workstation, without bringing the system down. On the other hand, there is no extension of the approach to SMP clusters yet, even if converting processes to threads is an interesting opportunity for this kind of architectures.

Distributed Shared-Memory Programming with OpenMP. In the following, we present two approaches that try to adapt OpenMP to SMP clusters. The main issue for this approach is that there is either a need for a global shared memory in physical distributed environment or OpenMP has to be extended with data distribution facilities.

Software Distributed Shared Memory. Software distributed shared memory (*SDSM*) systems are libraries that provide a global address space for physically distributed memory machines. We can translate OpenMP directives into appropriate calls to the SDSM system. An example of this approach is described in [413]. The result of the source-to-source translation is a standard C/C++ or Fortran program which can be compiled and linked with the SDSM system TreadMarks [41].

Even in this case, the communication system is modified to exploit the hardware shared memory within the SMP nodes. Experimental testing with several algorithms showed that the performance of the modified software distributed shared memory was much better than the original ThreadMarks library. Nonetheless, performance was still worse than that achieved by an MPI implementation of the programs. The speedups obtained were only 7-30% of those achieved by the MPI versions. The reasons are the overhead from coherence-maintenance network traffic, and the fact that the SDSM system does not exploit application-specific data access patterns, because only at run-time it is known whether communication will happen through the network or not. Again, the issue is that the method does not allow the programmer to explicitly address the hierarchical structure of the machine at design and at compile time.

A more promising approach is the *compiler directed SDSM* one [662], which is a two-step optimization.

In a first step, the OpenMP compiler inserts memory coherence code primitives, called *check code*, to keep the node-distributed memory consistent. There are three types of check codes. Two of them ensure that the data is valid before a read or write of shared data, the third is responsible to inform the other nodes that data has been changed by a shared write.

In the second step, the compiler analyzes parallel regions in order to optimize communication and synchronization by removing unnecessary check codes. The following optimization strategies are applied.

Parallel extent detection. Memory coherence code only has to be used in parallel regions. Therefore, the compiler can remove the check codes outside parallel regions and in the static extent of parallel regions.

Redundant check code elimination. Flush directives are responsible for giving all threads a consistent view of the memory. They are executed implicitly at barrier synchronizations, at the end of work sharing constructs and at references to volatile variables. Therefore, check codes after a write may be delayed until the thread reaches a flush directive and check codes before a read or write may be redundant if the data is already available by the preceding read check at the same location. The compiler performs a data-flow analysis of the statements in the parallel regions to determine the earliest possible read check code, and the latest possible write check code. All others are redundant and can be removed.

Merging multiple check codes. Arrays are very often accessed contiguously within a loop structure. The corresponding check codes may be moved outside the loop and simultaneously converted into a single one. This reduces the number of check code calls. In the following example `a` and `b` are shared arrays.

```
for (i=0; i<n;i++) a[i]=c*b[i];
```

The compiler inserts the check codes into the loop as follows.

```
for (i=0; i<n;i++) {
    check_before_read(&b[i], size);
    check_before_write(&a[i], size);
    a[i]=c*b[i];
    check_after_write(&a[i], size);
}
```

Since the loop does not contain any flush directive, the check codes can be moved outside the loop.

```
check_before_read(&b[0], n*size);
check_before_write(&a[0], n*size);
for (i=0; i<n;i++)  a[i]=c*b[i];
check_after_write(&a[0], n*size);
```

Data-parallel communication optimization. It is also possible to improve the program by using data-parallel compilation techniques. For example, the compiler should determine data mappings of arrays that accessed within a loop in such a way that iterations of the loop can be done locally, on the nodes where the data is stored. Since the number of threads is not known at compile time, calls to a data mapping runtime library are inserted to compute loop bounds and data that must be communicated. In this setting, the data is stored locally and check codes can be removed.

Collective communication optimization. Inter-node communication is necessary to implement a reduction operation on variables defined in the data scope attribute of a parallel region. It can be performed efficiently using a collective communication library. The execution starts after the local reduction at the end of parallel regions or after work-sharing directives.

Distributed OpenMP. A different approach to adapt OpenMP to SMP clusters is suggested in [546]. The authors propose the *distributed OpenMP*. This extension of OpenMP with data locality features provides a set of new directives, library routines and environment variables. One data-distribution extensions is the `distribute` directive with which it is possible to partition an array over the node memories. For performance reasons, the threads should work on local array elements. Hence, the user must distribute the data in order to minimize remote data accesses. Another proposed extension is the `on home` directive in a parallel region. With this directive, it is possible to perform a parallel loop over a distributed array without redistributing the array. The threads of a node perform the iterations for the array elements that reside in their local memory. Further extensions are library routines and environment variables that provide specific numbers of the run-time instance of the SMP cluster, like for example the number of involved nodes or processors per node. Disadvantages are that programs get more complex, and the user has to take care about an efficient data decomposition. Since we are providing more information to the compiler, after adding the new directives to an OpenMP program a redesign step, and a performance tuning phase have to be performed.

Hybrid Programming with MPI and OpenMP. The idea of the *hybrid programming model* is to use message passing between the SMP nodes, and shared memory programming inside the SMP nodes. The structure of this model fits exactly to the architecture, therefore, the model has potential to produce programs with significant performance improvement. But it is also obvious that the model is more complicated to use, and that there may arise unpredicted performance problems, because of the simultaneous usage of the two programming models. There are several possibilities for choosing libraries for each model, but it is straightforward to combine the de facto standards *MPI and OpenMP*. In the following we give an overview of the different approaches to the production of hybrid programs, with no emphasis on technical details. We also survey some performance evaluations that compare hybrid programs with pure MPI ones.

The general execution scheme uses one process in each node, to handles communications by means of MPI primitives. Inside the process, multiple threads compute in parallel. The number of threads in a node is equal to the number of processors in that node. The base for the design of an efficient hybrid program is an efficient MPI program. According to [171], there are two approaches to incorporate OpenMP directives into MPI programs, the fine-grain and the coarse-grain approach.

Fine-Grain Parallelization. The hybrid *fine-grain parallelization* is done incrementally. The computational part of an MPI program is examined, and the loop nests are parallelized with OpenMP directives. Therefore, the approach is also called loop-level parallelization. Clearly, the loops must be profiled, and only loop nests with a significant contribution to the global execution time are selected for OpenMP parallelization.

Some loop-nests can not be parallelized directly. If they are non negligible, the developer can try to transform them into parallelizable loops. Techniques like loop exchange and loop permutations, and introduction of temporary variables, can often avoid false sharings and reduce the number of synchronizations.

Performance of Fine-Grain Hybrid Programs. In [171, 172, 173, 200] investigations to measure performance of fine-grain hybrid programs are presented. A comparison is shown of the performance achieved by a hybrid and a pure MPI version of the NAS benchmark [86] on a SMP cluster. An important subject of [171, 172, 173, 200] is the interpretation of the performance measurements, in order to understand the behavior of the hybrid programs and their performance. Experiments were made on a PC-based SMP cluster with two processors per node and on IBM SP cluster systems with four processors per node.

The comparison among the two kinds of models for SMP clusters shows no general advantage of one over the other. Depending on the characteristics of the application, some benchmarks perform better with the hybrid version, others perform better with the pure MPI version. The following aspects have influence on the performance of the models.

Level of shared memory parallelization. The more of the total computation can be parallelized, the more interesting is the hybrid approach. The size of the parallelized sections (OpenMP) compared to the whole computation section must be significant.

Communication time. It depends on the communication pattern of an application, and on the differences between the two models concerning latency, bandwidth, and synchronization time. If more processes share one network interface, then the latency for network accesses increases, but the per process bandwidth increases too. If there is only one process per node, the latency is low, but the process cannot transfer data fast enough to the network interface to fully exploit the maximum network bandwidth. Therefore, the pure MPI approach performs better if the application is bandwidth limited, and it is worse for latency limited applications.

Memory access patterns. The memory access patterns are different for the two models. Whereas MPI allows to express multi-dimensional blocking, OpenMP does not. To achieve the same memory access patterns, rewriting of loop nests is necessary, which may be very complex.

Performance balance of the main components. (processors, memory and network) can offset the communication/computation tradeoff. If the processors are so fast that communication becomes the bottleneck, then the actual communication pattern decides which model is best. If, on the other hand, computation is the bottleneck, then MPI seems to be always the best.

Coarse-Grain Parallelization. In this approach a single program multiple data style is used to incorporate OpenMP threads into MPI programs. OpenMP is used to spawn threads immediately after the initialization of the MPI processes in the main program. Each thread itself is acting similar to an MPI process. For threads there are several issues to consider:

– The *data distribution between the threads* is different from that of MPI processes. Because of the shared memory, it is only necessary to calculate the bounds of the arrays for each thread. There has to be a mapping from array regions to threads.
– The *work distribution between the threads* is made according to the data distribution. Instead of an automatic distribution of the iterations, some calculations of the loop boundaries depending of the thread number define the schedule.
– The *coordination of the threads* means managing critical sections by either the usage of OpenMP directives, like `MASTER` or thread library calls like `omp_get_thread_num()`, to construct conditional statements.
– *Communication* is still done by only one thread.

As far as we know, the coarse-grain approach has been proposed, but there are no results yet. We can compare it with TOMPI, as both methods convert MPI processes to threads. However, TOMPI programs on SMP clusters do not share data structures common to all the processes, as they would do in a coarse-grain parallelization.

High-Level Programming Models. Besides the programming libraries and paradigms above, there are some programming models for SMP clusters that try to build a higher level of abstraction for the programmer. All these models are based on the hybrid programming paradigm where threads are used for the internal computation and message passing libraries are used to perform communication between the nodes.

SIMPLE Model. The significant difference between *SIMPLE* [82] and the manual hybrid programming approach above lies in the provided primitives for communication and computation.

The computation primitives comprise data parallel loops, control primitives to address threads or nodes directly, and memory management primitives.

Data parallel loops: There are several parallel loop directives for executing loops concurrently on one or more nodes of the SMP cluster, assuming no data dependencies. The loop is partitioned implicitly to the threads without need for explicit synchronization or communication between processors. Both block and a cyclic partitioning is provided.

Control: With this class of primitives, it is possible to control which threads are involved in the computation context. The execution of a block of code can be restricted to one thread per node, all threads in one node, or to only one thread in the SMP cluster.

Memory management: A heap for dynamic memory allocation is managed in each processing node, and can be used by the threads of that node via the `node_malloc` and `node_free` primitives.

SIMPLE provides three libraries for communication. There is an internode-communication library, an SMP node library for thread synchronization, and a SIMPLE communication library build on top of both. The SMP node library implements the three primitives *reduce*, *barrier* and *broadcast*. It is based on POSIX threads. Together with the functionality of the internode-communication library, it is possible to implement the primitives *barrier, reduce, broadcast, allreduce, alltoall, alltoallv, gather, and scatter* that are assumed to be sufficient for the design of SIMPLE algorithms. The use of these top-level primitives means using message passing between nodes and shared memory communication within the nodes.

Hybrid-Parallel Programming with High Performance Fortran. High Performance Fortran (*HPF*) is a set of extensions to Fortran that enables users to develop data-parallel programs for architectures where the distribution of data impacts performance. Main features of HPF are directives for data distribution within distributed memory machines and primitives for data parallel and concurrent execution. HPF can be employed on both distributed memory and shared memory machines, and it is possible to compile HPF programs on SMP clusters. However, HPF does not provide primitives or directives to exploit the parallel hierarchy of SMP clusters. Most HPF compilers just ignore the shared memory within the nodes and treat the target system as a distributed memory machine.

One exception is presented in [106]. Therein, HPF is extended with the concept of *processor mappings* and the concept of *hierarchical data mappings*. With these two concepts, it is possible for the programmer to consider the hierarchical structure of SMP clusters. A product of this approach is the Vienna Fortran Compiler [105]. It creates *fine-grain hybrid programs* using MPI and OpenMP, starting from programs in an enriched HPF syntax.

Processor mappings: Beside the already existing *abstract processor array* that is used as the target of data distribution directives, *abstract node arrays* are defined. Together with an extended version of the `distribute` directive it is possible to construct the structure of an SMP cluster.

Hierarchical data mapping: In addition to the processor mappings it is necessary to assign data arrays to nodes and processors. The `distribute` directive is extended in the way that node arrays may appear as distribution targets. This defines an explicit inter-node mapping of the data. In contrast, the `share` directive is introduced in order to define an explicit intra-node mapping. The intra-node mapping controls the work sharing between the processors within a node.

Intrinsic functions: Two new functions are provided, that return the number of nodes and the number of processors in the SMP cluster. They are provided in order to support abstract node arrays whose sizes are determined at program startup.

The following is a sample code fragment for the use of the new directives and mappings. It defines a SMP cluster with four processors per node and distributes an array A equally over the nodes and processors.

```
!hpf$ processors P(2,8)                  !abstract processor array
      real, dimension (32,16) :: A       !array of real
!hpfC nodes N(4)                         !abstract node array
!hpfC distribute P(*, block) onto N      !processor mapping
!hpfC distribute A(*, block) onto N      !inter-node mapping
!hpfC share A (block,*)                  !intra-node mapping
...
```

`block` is a standard HPF distribution format and divides the concerned dimension into equal parts with respect to the distribution target. The asterisk defines that the whole dimension of the array will be mapped to the target elements.

KeLP2 Model. The *Kernel Lattice Parallelism 2 Model (KeLP2)* [81] is a C++ framework for implementing (irregular) block-structured numerical applications on SMP clusters. KeLP2 provides mechanisms to coordinate data decomposition, data motion and parallel control flow similar to HPF. Like HPF, it hides from the programmer low-level details like message-passing, processes, threads, synchronization and memory allocation. In contrast to HPF, KeLP2 performs no analysis of the code to make high-level restructuring, and it provides no automated data decomposition.

The underlying assumption is that the programmer knows best the structure of his (irregular) data and algorithm. KeLP2 provides him a framework and a methodology to define the decomposition, facilitating the construction of partitioning libraries. With respect to HPF, KeLP2 allows to overlap computation and communication.

KeLP2 supports three levels of control, the collective level (SMP cluster), the node-level, and the processor-level. Parallelism in programs is expressed at the node and the processor levels, while communication takes place in the collective (cluster) and node levels. The collective and the node levels have their own data layout and data motion plan. Three classes of KeLP2 programming abstractions help to manage this mechanisms.

1. The *Meta-Data* represents the abstract structure of some facet of the calculation. It describes the data decomposition and the communication patterns.
2. *Instantiators* execute the program according to the information contained in the meta-data.
3. The primitives for *parallel control flow* are iterators, which iterate over all nodes, or over all processors of a specified node.

When comparing KeLP2 with SIMPLE, the latter provides lower-level primitives, does not support data-decomposition, and does not overlap communication and computation. KeLP2 is of narrower scope concerning the application domain, but it nevertheless enables a parallel specification that is less dependent on the implementation.

15.6 Conclusions

It is clear from Sect. 15.2 that we need theoretical models and new software tools to fully exploit hierarchical architectures like large clusters of SMP and future Computational Grid super-clusters.

The interaction among parallel bridging models and external memory models has produced several results, which we surveyed in Sect. 15.3, 15.4. The exploitation of locality effects in these two classes of models employs very similar solutions, that involve block-oriented cooperation and abstract modeling of the hierarchical structure. The intuition underlying theoretical and performance results on bulk parallel models, and their theoretical lesson, is that the simple exploitation of fine-grain parallelism at the algorithm level is not the right way to obtain portable parallel programs in practice.

However, there is still a lot of work to do in order to meet the need of appropriate computational models. Hierarchical-parallel models like those of Fig. 15.5 are already close to the structure of modern SMP clusters, and they are relatively simple to understand, yet algorithms can be quite hard to analyze. Composed models usually employ the full set of parameters of their parallel part, those of their memory part, and at least another one to asses the relative cost of I/O and communication operations. For the BSP derivatives we have described, this leads to seven or eight parameters.

Excessive complexity of the analysis and poor intuitive understanding are a limiting factor for the diffusion of computation models, as it was pointed out in [213]. There is no answer yet to the questions "can all these parameters be merged in some synthesis?" and "what are the four or five most important parameters?"

For these reasons the impact of sophisticated parallel computational models is still limited, while simple disk I/O models like PDM have been quickly adopted to evaluate a corpus of external memory algorithms widely used in practice.

The situation is the same for what concerns software tools. Indeed, most of the efforts in parallel software development are spent in maintaining existing programming interfaces and optimizing them for new architectures. This approach is not effective or efficient for hierarchical parallel architectures. As we have reported, SMP cluster-enabled implementations of both MPI and OpenMP libraries are quite far from exploiting the potential performance of these machines. When programmers completely disregard the existence of a hierarchy, and we try to hide all optimizations in the library, it is often impossible to achieve program implementations with optimal performance.

A complementary approach is to compose existing libraries which manage a portion of the hierarchy. Most standard libraries only exploit two level of memory or parallel structure (e.g. MPI, OpenMP, TPIE). While it is possible to compose libraries that manage well-separated hierarchies (e.g. main memory/disks and main memory/shared memory), we have seen in Sect. 15.5.3 that the hybrid programming model, resulting from the empirical combination of MPI and OpenMP, leads to much harder problems.

Nevertheless, the hybrid model is the only parallel and hierarchical programming model accepted in practice, because it has the driving advantage of exploiting the new cluster architectures using existing, available tools. It has numerous drawbacks, though. Programs are more complex to design, implement, debug and maintain. Implementation and debugging are also complicated by the need for extensive performance analysis and tuning. The complexity and the amount of the interactions among the architecture, the algorithm and the software tools make it quite difficult to devise performance models for the resulting programs.

According to us, two main directions for future research are now still open. A first research and development track aims at simplifying the management of hybrid parallel programming, by extending existing flat approaches with ad-hoc optimizations, and with improvements to the compilation tools. Code translators, e.g. from MPI to hybrid structured code, and semi-automatic restructuring tools are two feasible solutions to ease and speed up hybrid software development.

Another option is to devise simpler parallel hierarchical models that are sound and intuitive, and thus can be used both for theory and as the base for a programming environment. Too cumbersome models are ruled out, as they fail in providing that kind of intuitive guidance that, even if not fully trustworthy, is essential for the acceptance of a programming model.

Looking at the literature, an important resource in this perspective is a set of common, basic operations that can be efficiently performed both in parallel and exploiting the memory hierarchy. More freedom in choosing the implementation level of the basic operations simplifies design and analysis of more complex algorithms, as well as the implementation of software tools.

Systems like SIMPLE and KeLP2 are close to this research path. Abstract, high-level operations simplify program writing, while still providing the tools with information about the best mapping to the architecture hierarchy.

Acknowledgments

We wish to thank all the participants to the GI-Dagstuhl-Forschungsseminar "Algorithms for Memory Hierarchies". In particular we would like to thank Florin Isaila and Kay Salzwedel for their help in the bibliographic research, and Peter Sanders, Jop Sibeyn, Ulrich Meyer, Rasmus Pagh, and Daniel Jimenez for several discussion which contributed to improve the quality of the paper.

16. Case Study: Memory Conscious Parallel Sorting

Dani Jiménez-González, Josep-L. Larriba-Pey, and Juan J. Navarro

16.1 Introduction

The efficient parallelization of an algorithm is a hard task that requires a good command of software techniques and a considerable knowledge of the target computer. In such a task, either the programmer or the compiler have to tune different parameters to adapt the algorithm to the computer network topology and the communication libraries, or to expose the data locality of the algorithm on the memory hierarchy at hand.

This chapter analyzes different techniques related to the exploitation of *data locality*, which may be taken into account in the design of efficient parallel algorithms. These techniques depend on the programming model and the memory organization of the target parallel computer. Some of the techniques we explain may have been implemented by a few up to date compilers, but our experience shows that automatic detection and implementation of parallelism is a technology that still has a long way to run in order to obtain efficient parallel implementations [377, 729, 732].

In order to exploit data locality in parallel algorithms, it is important to (i) reduce the number of misses when accessing local caches, and (ii) reduce the time spent by a processor in accessing the remote or shared caches and memory. The impact of local accesses can be addressed from the design of the sequential algorithm, taking into account the capacity, line size and miss penalty of the different levels of the local memory hierarchy. See Chapters 15, 16, and 8 for more details. The impact of remote accesses depends on the design of the parallel algorithm, taking into account the latency of the remote memories, the hardware techniques implemented to preserve data coherence, etc.

With the objective of understanding how to exploit data locality, we present a discussion of software techniques that target the exploitation of data locality on *sorting* algorithms, with special emphasis on the parallel aspects of the problem. We concentrate on the well known and important problem of in-memory sorting.

In the chapter, we start by explaining non-efficient algorithms and their implementations for didactic purposes. Then, we focus on the different techniques that can be applied to improve the data locality of those non-efficient algorithms and explain the techniques in detail. Finally, we efficient algorithms and analyze the results obtained by their implementations.

We analyze two different radix sort based solutions on the SGI Origin 2000 parallel computer; the Straight Forward Parallel version of radix sort, SF-Radix sort, and Communication and Cache Conscious Radix sort, *CCC*-Radix or $C3$-Radix [430]. SF-Radix sort is not a very efficient algorithm but a very good algorithm for didactic purposes. $C3$-Radix sort is a memory conscious algorithm at both the sequential (cache conscious) and the parallel (communication conscious) levels. $C3$-Radix shows a good speed-up[1] of 7.3 for 16 processors and 16 million keys, while previous work only achieved a speed-up of 2.5 for the same problem size and set up [430]. Furthermore, the sequential algorithm used in $C3$-Radix has proved to be faster than previous algorithms found in the literature [433].

The data sets we use to test our algorithms are N records formed by one 32-bit integer key and one 32-bit pointer.[2] The keys we sort are generated at random with a uniform distribution and have no duplicates. The parallel algorithms explained here may not be efficient for data distributions with duplicates and/or with skew. In this case, we recommend reading [431, 432] which focus on the specific problem of skew and duplicates.

In order to motivate this chapter, we advance some results in Figs. 16.1 and 16.2 using the visualization and analysis tool Paraver [604]. We use this tool throughout this chapter in order to show the executions of the different implementations of the sorting problem. Each window in Figs. 16.1 and 16.2 shows the time spent in sorting 16 million records on 8 R10000 processors (one processor per horizontal line) of the SGI Origin 2000. The execution time in those figures (as well as in the Paraver based figures below) starts at the first small flag above each horizontal line and ends with the last flag. Note that all the processors start at the same time and end at different times.

The results in Fig. 16.1 are for SF-Radix sort (picture at the top) and for $C3$-Radix sort (picture at the bottom) using MPI. The most important aspect of that figure is that the *parallel algorithm chosen* has a significant impact on the overall performance. Data communication is shown with the lightest shaded areas between marks in the figure. In both cases, the *2.00 s* vertical dotted line falls in the middle of the first data communication. We can see that SF-Radix sort performs four communication steps, while $C3$-Radix sort performs just one.

The results in Fig. 16.2 are for two different implementations of $C3$-Radix sort using OpenMP. We can see that the implementation at the top takes more time than that at the bottom. However, both implementations only differ slightly; we have laid special emphasis on the memory aspects of the

[1] We measure the speed-up as the ratio between the execution time of the fastest sequential algorithm at hand and the parallel execution time.

[2] We sort key-pointer records because, for instance, in database applications, sorting is done on records that may be very large. Therefore, to avoid moving large sets of data, the key is extracted from the record, a pointer is created to that record and they are both copied to a new vector of key and pointer tuples that is sorted by the key [588, 687].

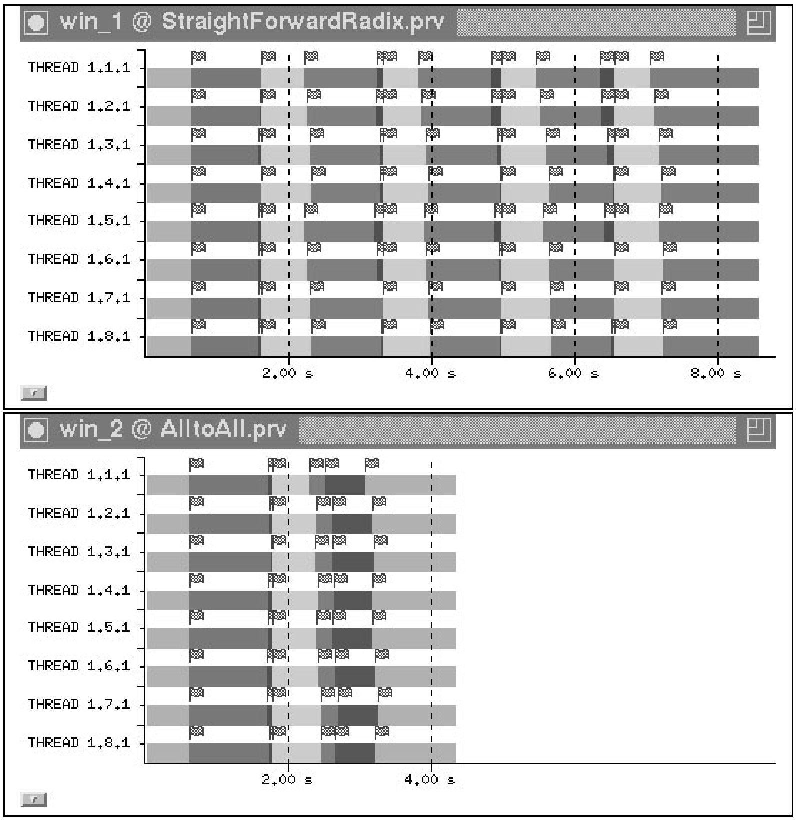


Fig. 16.1. Comparison of two MPI implementations of SF-Radix sort *(top)* and $C3$-Radix sort *(bottom)*

fastest algorithm at the bottom of the figure. Therefore, we can say that *a memory conscious implementation* of an algorithm may also have a significant impact on the performance. Thus, Figs. 16.1 and 16.2 give insight into the results obtained by the techniques addressed in this chapter.

The chapter is structured as follows. In Sect. 16.2, we discuss the main computer architecture aspects that affect the cost of memory accesses or communication. In Sect. 16.3, we discuss and analyze the SF-Radix sort algorithm for shared and distributed memory machines. Then, as a better solution for the same problem, the shared and distributed versions of $C3$-Radix sort are discussed and analyzed in detail in Sect. 16.4. In both Sects. 16.3 and 16.4, we discuss different implementations of the algorithms to understand the influence of some implementation details on the exploitation of data locality, and thus, on the performance of those implementations. Finally, in Sect. 16.5, we set out our conclusions.

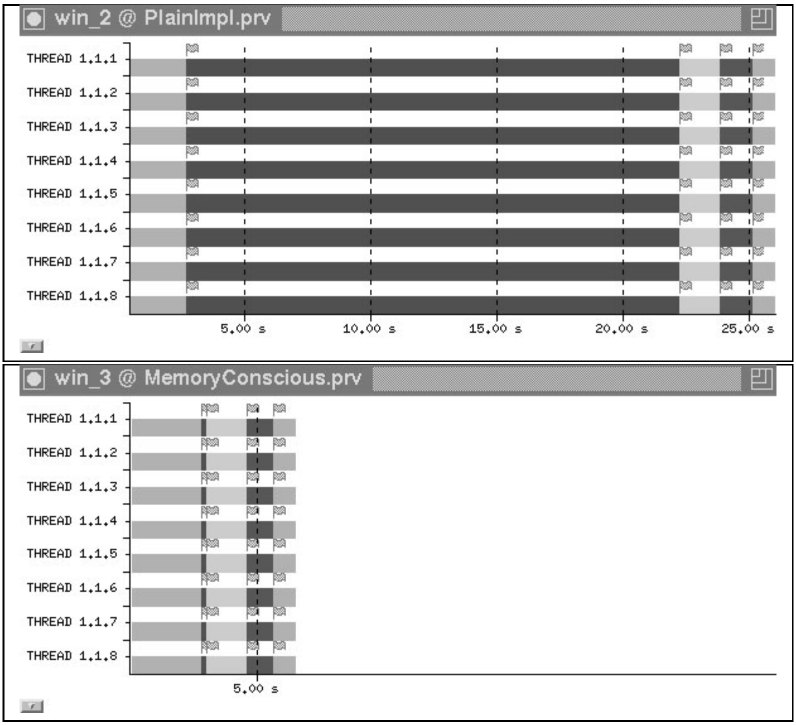


Fig. 16.2. Comparison of two OpenMP implementations of $C3$-Radix sort

16.2 Architectural Aspects

Here, we explain some parameters affecting the communication between processes when working with Multiple Instruction, Multiple Data stream, for shared and distributed memory computers; SM MIMD and DM MIMD respectively. The SM MIMD and DM MIMD architecture models are explained in detail in Chapter 15.

In general, if we want to communicate a certain load of data within a message-passing system, the dominant costs are fixed and determined by the operating system overhead, the DMA programming, and the interrupt processing. In a shared-memory system, the overhead is primarily caused by the consumer reading the data, since it is then that data moves from the global memory to the local memory hierarchy of the consuming processor [501]. In [198], the authors provide insight into the relative effectiveness of various communication mechanisms for several applications over a modest range of communication latencies and bandwidth.

16.2.1 SM MIMD

In order to reduce the memory latency in shared memory systems it is necessary to have several levels of cache memories conforming a complex memory hierarchy. From the point of view of a processor, there is a hierarchy of local cache memories and a hierarchy of remote cache memories. This causes some cache coherence problems among the cache hierarchies of the different processors.

The Cache Coherence Problem

The *Cache Coherence Problem* arises when there are several copies of the same data item in caches belonging to different processors. So, whenever a processor modifies this data item in its cache, some action has to be taken on the copies of the remote caches to preserve coherence, otherwise there would be processors working on stale data. There are two cache coherence protocols for solving this problem:

1. The *update coherence protocol.* Whenever a processor modifies a data item stored in its cache, it is updated in the rest of the processor's caches that have a copy of the item, and in main memory. Updates are done at an individual data item level. When two or more processors share the same individual data item, this is called *true sharing.*
2. The *invalidate coherence protocol.* Whenever a processor modifies a data item stored in one of its cache lines, it invalidates the cache lines that also contain the previous valid value of this data item in the rest of processor's caches. Invalidations are done at a cache line level. Therefore, the access to invalidated lines will cause misses. These misses are called *false sharing* misses if the processors access a data item of the same cache line but different from the data item that caused the invalidation. If the access is to the same data item, it is a true sharing miss.

There are several hardware implementations of those two protocols. The most common implementations are the snoopy and the directory-based cache coherence protocols. They only differ in the way processors carry out an update or invalidation. In the snoopy protocol, updates or invalidations are broadcast to all processors through a bus. So, each processor in the machine reads that information from the bus and has to figure out if it has a copy of the data to be updated or invalidated. Then it updates or invalidates if necessary. In the directory-based protocol, the information of who has copies of data is stored in a memory structure. Updates and invalidations are sent only to those processors with copies of the data being accessed. See [73, 234, 363, 392, 427, 501] for details about the problems and the hardware implementations of cache coherence protocols.

True and *false sharing* misses have been studied extensively in the literature [140, 278, 355, 505, 520, 560, 728]. An extended survey of different techniques for reducing these two kinds of misses can be found in [605].

16.2.2 DM MIMD

In the distributed memory MIMD computers, processors communicate by means of explicit messages. Each explicit message has a cost that depends on the interconnection network, as well as the operating system and the software communication assistant. Part of this cost is fixed (sender and receiver overhead, and time of flight) and the rest of the cost depends on the size of the message (data transfer time). See [392] for more details.

We can distinguish two types of communication: synchronous and asynchronous. With synchronous communication, processors have to wait until communication has completely finished. With asynchronous communication, processors do not have to wait until communication has completely finished, so that the performance of an application may not be hindered by communication. In this case, processors can overlap communication with computation. However, asynchronicity may introduce some overheads, such as the cost of buffering data, and/or waiting until user resources are not needed by the asynchronous operation.

16.2.3 The SGI Origin 2000

The algorithms we analyze and compare in this paper are tested on the SGI Origin 2000 at CEPBA[3].

The SGI O2000 is a directory based NUMA parallel computer [234, 498]. It allows both the use of the shared and distributed memory programming models. The O2000 is a distributed shared memory (DSM) computer, with cache coherence maintained via a directory-based protocol.

The building block of the SGI O2000 is the 250 MHz MIPS R10000 processor that has a memory hierarchy with private on-chip 32Kbyte first level data and instruction caches and external 4Mbyte second level combined instruction and data cache.

One node card of the SGI O2000 is formed by 2 processors with a 720MB/s peak bandwidth, 128Mbyte shared memory. Groups of 2 node cards are linked to a router that connects to the interconnection network. In our case, the interconnection network is formed by 4 routers connected as a 160GB/s global peak bandwidth 2-d hypercube network. This hypercube has two express links between the groups of two routers that are not connected directly.

[3] CEPBA stands for "Centre Europeu de Paral.lelisme de Barcelona". More information on CEPBA, its interests and projects can be found at `http://www.cepba.upc.es`.

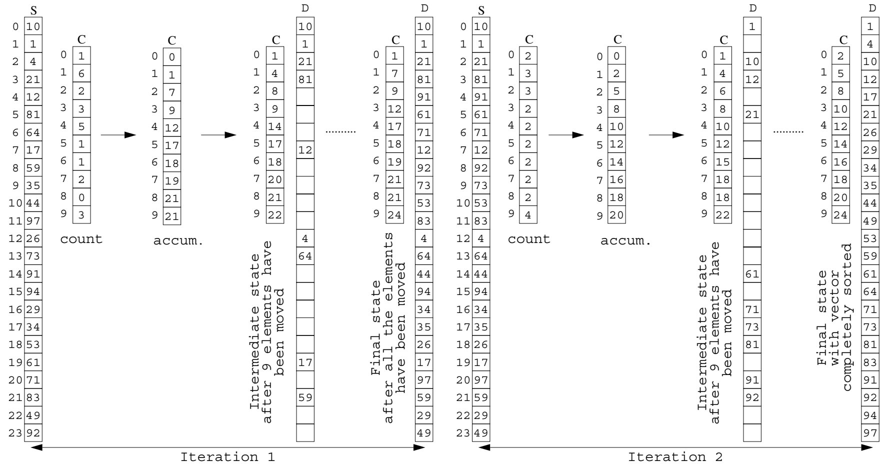

S
C
D
count
accum.
Intermediate state after 9 elements have been moved
Final state after all the elements have been moved
Final state with vector completely sorted
Iteration 1
Iteration 2

Fig. 16.3. Example for radix sort for 2 decimal digit keys on the O2000 machine

16.3 Sequential and Straight Forward Radix Sort Algorithms

The radix sort algorithm is a simple sequential algorithm that can easily be parallelized. We first explain the sequential radix sort, and then we present two straight forward parallel solutions of radix sort, one for shared memory and another one for distributed memory computers.

16.3.1 Sequential Radix Sort

The idea behind radix sort is that b-bit keys are sorted on a per digit basis, iterating from the least significant to the most significant digits. The b-bits of a key can be grouped forming m digits of b_i consecutive bits where $\sum_{i=0}^{m-1} b_i = b$.

To sort the set of keys for each digit, radix sort may use any sorting method. Our explanation relies on the counting algorithm [460] for that purpose. Alg. 1 is an example of this version of radix sort.

The counting algorithm performs three steps for each digit: count, accumulation and movement steps, which correspond to the three code sections within the *for* loop starting at line 1 of Alg. 1.

Next, we explain the procedure of the radix sort algorithm for the sorting of the least significant digit of a 2-decimal digit key with the help of Alg. 1, and the example in Fig. 16.3. The processing of this digit corresponds to iteration 1 of Alg. 1 and the left hand side of Fig. 16.3.

First, the *count step* computes a histogram of the possible values of the least significant digit of vector S, on vector C (lines 3 to 6). In the example, it is necessary to have a vector with 10 counters, one for each possible value

Algorithm 1: Sequential Radix sort

```
 1: for $dig$ = least significant digit to most significant digit do
 2:    initialize (C, nbuckets)

 3:    for $i = 0$ to $N - 1$ do
 4:       $value \leftarrow get\_digit\_value(S[i], dig)$
 5:       $C[value] \leftarrow C[value] + 1$
 6:    end for

 7:    $tmp \leftarrow C[0]$
 8:    $C[0] \leftarrow 0$
 9:    for $i = 1$ to $nbuckets - 1$ do
10:       $accum \leftarrow tmp + C[i - 1]$
11:       $tmp \leftarrow C[i]$
12:       $C[i] \leftarrow accum$
13:    end for

14:    for $i = 0$ to $N - 1$ do
15:       $value \leftarrow get\_digit\_value(S[i], dig)$
16:       $D[C[value]] \leftarrow S[i]$
17:       $C[value] \leftarrow C[value] + 1$
18:    end for

19:    swap_addresses ( S, D )
20: end for
```

(0 to 9) of one digit (in general, 2^{b_i} counters for b_i-bit digits). The counters show that, for instance, the least significant digit of the key has 6 and 2 occurrences of values 1 and 7 respectively.

Second, the *accummulation step* computes a partial sum of the counters (lines 7 to 13). We show this sum in a different instance of vector C with label *acumm* in Fig. 16.3.

Third, the *movement step* reads vector S, writing each key to vector D using the values of vector C (lines 14 to 18). During that process, 10 buckets (in general, 2^{b_i} for a b_i-bit digit), one for each possible value of this digit, are formed. In the example of Fig. 16.3, we show two different situations of vector C after moving the first 9 keys, and after moving all the keys from vector S to vector D. Subsequently, vectors S and D exchange their role (line 19). Then the same procedure is performed on the most significant digit of the key during iteration 2. The final sorted vector is also shown in Fig. 16.3.

In general, note that after sorting digit k, we say that the N keys are sorted for the $\sum_{i=0}^{k-1} b_i$ least significant bits.

A discussion about how to improve this sequential algorithm with the objective of exploiting data locality can be found in Chapter 8. Here we use the fastest version known to us [433]. However, we focus on its parallelization, and we distinguish between the shared and distributed memory versions of the algorithm.

16.3.2 Shared Memory SF-Radix Sort

The shared parallel version of radix sort is very similar to the sequential version adding some parallelization directives. Alg. 2 is an OpenMP version of SF-Radix sort. The parallel directives used are quite intuitive and we do not explain them here. More information can be found in [595].

Algorithm 2: Shared memory SF-Radix sort

```
 1: #pragma omp parallel shared (S, D, $Gl$) private ($LCacc$)
 2: for $dig$ = least significant digit to most significant digit do
 3:    for $i = 0$ to $nbuckets - 1$ do
 4:       $Gl[i][pid] \leftarrow 0$
 5:    end for

 6:    #pragma omp for private (i, value) schedule (static)
 7:    for $i = 0$ to $N - 1$ do
 8:       $value \leftarrow get\_digit\_value(S[i], dig)$
 9:       $Gl[value][pid] \leftarrow Gl[value][pid] + 1$
10:    end for

11:    partial_sum ($Gl$, $LCacc$)

12:    #pragma omp for private (i, value) schedule (static)
13:    for $i = 0$ to $N - 1$ do
14:       $value \leftarrow get\_digit\_value(S[i], dig)$
15:       $D[LCacc[value]] \leftarrow S[i]$
16:       $LCacc[value] \leftarrow LCacc[value] + 1$
17:    end for

18:    #pragma omp single
19:    swap_addresses ( S, D )
20: end for
```

Fig. 16.4 shows the parallel procedure for sorting the same previous example with two processors. Processors P_0 and P_1 work with the white and grey vector parts of vector S and D respectively in each iteration of Alg. 2, lines 2 to 20 of the code. For instance, for the least significant digit, each processor P_{pid}:

1. Performs the initialization of its counters in the global counter matrix Gl (lines 3 to 5).
2. Computes a histogram (lines 6 to 10) of the number of keys for each possible value of the least significant digit. Then, all processors synchronize in a barrier before the next step. The *omp for* directive in line 6 of the algorithm inserts a barrier after the corresponding piece of code by default.
3. Computes the local accumulation vector $LCacc$ with the *partial_sum* call (using the global counters Gl) as follows:

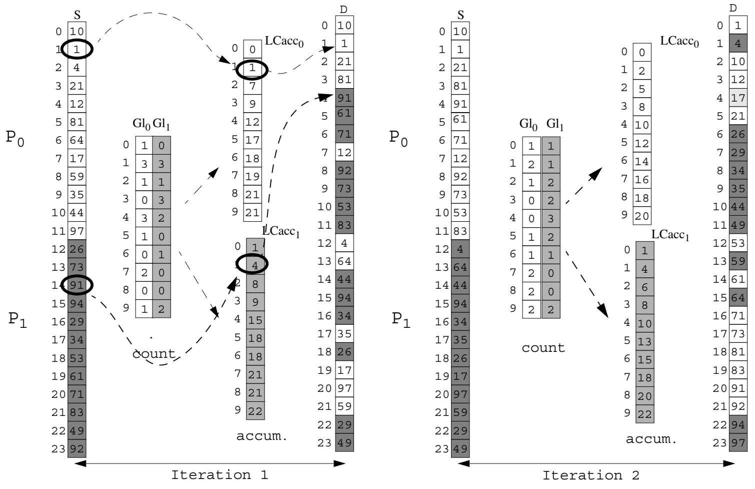


Fig. 16.4. Example of the shared memory SF-Radix sort for 2 decimal digit keys. $LCacc_{pid}$ indicates the local accumulation vector of processors P_{pid}

$$LCacc[i] = \sum_{ip=0}^{P} \sum_{j=0}^{i-1} Gl[j][ip] + \sum_{ip=0}^{pid-1} Gl[i][ip] \;\;,\;\; 0 \le i \le nbuckets - 1 \;\;.$$

With this formula, processor P_{pid} knows the first place where the first key of each of its buckets should be written. So, the first key belonging to bucket i of processor P_{pid} is placed in the position of vector D just after all the keys belonging to buckets 0 to $i-1$, and after all the keys of the same bucket i belonging to processors P_0 to P_{pid-1}. For instance, in the loop in lines 12 to 17, processor P_1 writes key 91 in position 4 of vector D, as its $LCacc_1[1]$ indicates in Fig. 16.4, and processor P_0 writes the key of vector S with value 1 in position 1 of vector D.

4. Moves keys from its part of vector S to vector D using its local counters $LCacc$ (lines 12 to 17). Each processor writes keys belonging to the same bucket in independent places of vector D. For instance, during iteration 1 of the example of Fig. 16.4, and for bucket 1, processor P_0 writes keys with values 1, 21, and 81 in positions 1 to 3 of vector D, and processor P_1 writes keys with values 91, 61, and 71 in positions 4 to 6 of the same vector.
Before the next step, all the processors synchronize in another barrier.

One of the processors then exchanges the roles of vectors S and D (lines 18 and 19), as in the sequential algorithm. After that, all the processors need to synchronize with a barrier (the *single* directive in line 18 of the algorithm inserts a barrier after the corresponding piece of code by default) to synchronize the sorting of each digit. Otherwise, if a processor started sorting the next significant digit while another processor was sorting a previous digit, the shared matrix Gl and also vectors D and S might be inconsistent, leading to an incorrect solution.

Analysis

Table 16.1 shows the number of L_2 misses, the number of invalidations, and the execution time per processor for sorting 16 million key-pointer records for two different implementations: the plain implementation explained above and a memory conscious implementation, explained below.

The results are quite significant if we consider that a miss penalty on L_2 on the SGI Origin 2000 may be 75 processor cycles or more.[4] The direct cost of an invalidation is low. However, those invalidations may lead to L_2 misses, as happens in the plain implementation. In any case, both L_2 misses and invalidations are important, since we may not have a large number of invalidations but rather a large number of L_2 misses. With the memory conscious implementation, the number of L_2 misses and the number of invalidations have been dramatically reduced when compared to the plain implementation. This reduction yields a significant reduction of the execution time; the memory conscious implementation is about 8.9 times faster than the plain implementation.

Table 16.1. L_2 miss and invalidation average per processor and execution time for sorting 16M records with 8 processors with SF-Radix sort

SF-Radix sort	L_2 misses	Invalidations	Exec. Time (s)
Plain Impl.	9,828,950	9,725,467	51.29
Memory Conscious	1,534,000	251,744	5.76

The large number of misses and invalidations of the plain implementation are due to false sharing in accessing the global counter matrix during the count step (lines 6 to 10 of Alg. 2). The false sharing is caused by the use of C as a programming language. The C compiler places the elements of a row in consecutive positions in memory.[5] Therefore, two counters of bucket i for processors P_0 and P_1, placed in the same row, may be placed in the same cache line. Thus, processors P_0 and P_1 may concurrently access the same cache line during the count step, invalidating that cache line mutually.

A possible solution may be to swap column and row elements, allowing us reduce the false sharing between processors. This change means that $Gl[i][pid]$

[4] The number of cycles per miss can be obtained executing the command line *perfex -t* on the O2000 machine.

[5] Fortran places the elements of a column consecutively in memory. Hence, there would be no problem.

will become $Gl[pid][i]$ in the code of Alg. 2. Another solution (the memory conscious solution shown in Table 16.1) is to use a local counter vector per processor, combined with the use of the global counter matrix commented above. With those data structures we only have to modify steps 1 and 2 of Alg. 2, as follows: Now, each processor:

1. Initializes its local counters, and
2. Locally computes a histogram of the values of the keys with its local counters. Therefore, no false sharing occurs. Then, each processor intializes the global counter matrix with its local counters. We may have some false sharing updating the global counter matrix, but it is much less than before.

This memory conscious solution dramaticaly reduces the number of L_2 misses and invalidations. First, the number of accesses to the global counter matrix has been reduced from N, the number of records, to *nbuckets*, the number of buckets. Second, we have swapped columns and rows so that false sharing between processors is reduced when accessing global counter matrix.

Another possible drawback comes from the fact that, when a counter row ends in the middle of a cache line and the next counter row starts right after it, that cache line will also be the cause of false sharing misses. In order to overcome that drawback, it is possible to pad some useless bytes after the first row, so that the next row starts in a new cache line.

SM Algorithm Drawbacks

We have explained two versions of SF-Radix sort where the memory conscious implementation improves the plain one. However, we still have some potential performance problems with this algorithm.

The main problem is that data are moved from one processor's cache to another *every time we sort a digit.* For instance, the grey keys of vector D in Fig. 16.4 are problably in processor P_1's cache at the end of iteration 1; it has just written them. Then, after swapping the role of the vectors and synchronizing, iteration 2 starts and processor P_0 reads the white vector part of S (which played the role of D in the previous iteration). This means that processor P_0 will probably miss in the cache hierarchy and will have to fetch those keys from processor P_1's cache. That may happen for each digit.

Another potential problem is that processors have to synchronize several times with a barrier *every time they sort a digit.* The larger the number of digits, the bigger the problem may be. Barriers may impose a significant overhead. First, executing a barrier has some run-time overhead that grows quickly as the number of processors increases. Second, executing a barrier requires all processors to be idle while waiting for the slowest processor; this effect may result in poor processor utilization when there is load unbalance among processors [732].

16.3.3 Distributed Memory SF-Radix Sort

We explain parallel Alg. 3 and the procedure for sorting the same data set shown above in a distributed memory environment. Fig. 16.5 may be used as an aid for the explaination. As before, we explain the algorithm for iteration 1, which is for the least significant digit. Originally, the N keys are evenly distributed across the P processors, as can be seen in the example for two processors. The algorithm is as follows:

1. Each processor P_{pid} locally performs the three sequential steps of the counting algorithm on the least significant digit of b_0 bits using the local counter vector LC_{pid}. This is done in lines 3 to 18 of Alg. 3. During that sequence, a total of 2^{b_0} buckets may be formed in each processor. In the example, we are dealing with decimal digits, and the number of counters is 10.
2. All processors broadcast their local counters previously copied to a local counter matrix Gl (not shown in Fig. 16.5). The copy is done in lines 19 to 22. The broadcast is done with an MPI *allgather* style function. So, after this call, all the processors have all the counters in their local counter matrix Gl. The amount of data sent during that broadcast may be large. That is $\Theta(k \times P^2)$, where k is the size of the counters.
   However, what we are solving here is the parallel prefix problem [488]. The amount of data sent by a prefix sum is only $\Theta(k \times P)$, but that requires at most twice as many operations than broadcasting counters, Therefore, there is a tradeoff between the best algorithm to perform the sum and the amount of data to be sent. Here, k and P are small, so broadcasting is better than performing a prefix sum.
3. A set of consecutive buckets is assigned per processor. This can be computed locally with the help of the local counter matrix Gl received in the previous step. In order to obtain a perfect load balance, authors of [703] propose splitting those buckets that are on the frontier of two processors. Therefore, one part of the bucket goes to a processor, and the other part of the bucket to another processor. This is done by the *bucket_distribution* function in the algorithm (line 24).
4. Finally, and using the previous computation, the objective is to perform an all-to-all data communication. With this data communication, each processor obtains a set of consecutive global buckets. Each global bucket is formed with all the keys with the same value for the least significant digit. This is done in the algorithm with the *data_communication* function.

Next, the sorting of the second least significant digit with b_1 bits begins.

Analysis of the Data Communication

There are several possible implementations of the data communication (step 4 of the DM algorithm). This data communication problem has been exhaus-

Algorithm 3: Distributed memory SF-Radix sort

```
1: for $dig$ = least significant digit to most significant digit do
2:    initialize (LC, nbuckets)

3:    for $i = 0$ to $N - 1$ do
4:       $value \leftarrow get\_digit\_value(S[i], dig)$
5:       $LC[value] \leftarrow LC[value] + 1$
6:    end for

7:    $tmp \leftarrow LC[0]$
8:    $LC[0] \leftarrow 0$
9:    for $i = 1$ to $nbuckets - 1$ do
10:      $accum \leftarrow tmp + LC[i - 1]$
11:      $tmp \leftarrow LC[i]$
12:      $LC[i] \leftarrow accum$
13:   end for

14:   for $i = 0$ to $N - 1$ do
15:      $value \leftarrow get\_digit\_value(S[i], dig)$
16:      $D[LC[value]] \leftarrow S[i]$
17:      $LC[value] \leftarrow LC[value] + 1$
18:   end for

19:   $Gl[pid][0] \leftarrow LC[0]$
20:   for $i = 1$ to $nbuckets - 1$ do
21:      $Gl[pid][i] \leftarrow LC[i] - LC[i - 1]$
22:   end for

23:   allgather ($Gl$[pid][0], nbuckets, $Gl$)

24:   bucket_distribution($Gl$)
25:   data_communication(S, D, $Gl$)

26:   swap_addresses ( S, D )
27: end for
```

tively studied: the balanced all-to-all routing [657, 688, 689]. Here, we explain four possible non-complex implementations for didactic purposes, which we believe show interesting data communication aspects.

However, the performance differences between those implementations are not significant for SF-Radix sort. The performance differences observed in $C3$-Radix sort, which are more significant, are analyzed later in Sect. 16.4.3.

Bucket by Bucket Messages. This is a straight forward implementation, where the local buckets of a processor that have to be sent to/received from another processor are sent/received *one by one* synchronously. Therefore, a processor cannot continue doing anything else until a data communication has been completely finished. In addition, we have to pay an overhead cost for each message. Therefore, this data communication may be very costly if the number of messages is large.

We refer to this implementation as plain solution.

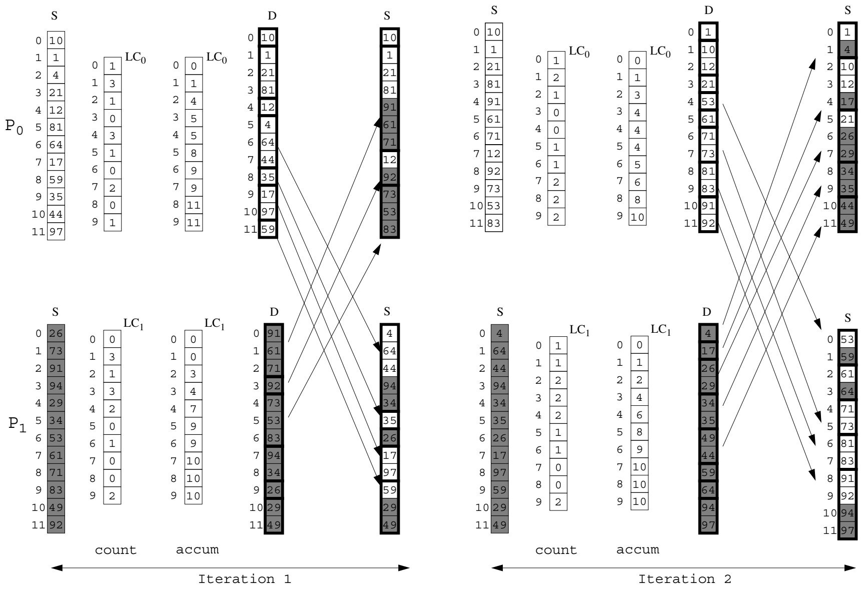


Fig. 16.5. Example for distributed SF-Radix sort for 2 decimal digit keys

Grouping Messages. In order to reduce the number of sends, we can group buckets that go to the same processor. Here, each processor sends only one message with all the local buckets that should go to one processor.

Then if the average number of buckets sent to each processor is k, we have reduced the number of messages to $1/k$ th. However, in some cases, we may not want to group all the buckets in a message, because this message may become too large. Large messages are difficult to send due to memory limits and network bandwidth. The larger the size of the message, the larger the number of resources that are blocked in a processor to send or recieve other messages. The work done in [610] analyzes the impact of this problem on the performance of the application. They propose sending all the data in phases instead of sending it all at once.

Note also that grouping messages has an extra cost. In the previous solution, processors receive buckets one by one and place them in the corresponding position of the destination vector. Now, once a message is received by a processor, this processor has to write the keys belonging to a bucket together with the keys beloging to the same bucket coming from other processors. Remember that the objective of this data communication is to create global buckets as we explained in the previous section. However, if the number of messages is high, the cost of re-placing buckets may be larger than the improvement achieved by reducing the overall overhead cost of the messages.

Therefore, one should take into account the interconnection network at hand (bandwidth, througput, etc.) and the overhead cost of sending and receiving messages in the implementation chosen.

Synchronous versus Asynchronous. The amount of time spent in communication may be reduced if we perform asynchronous send and/or receive operations. The reason is that we may overlap some communication with other communication and/or computation. For slow networks, the computation may be hidden behind the communication. However, for some other networks, overlapping computation may make the code larger and harder to read, achieving only slight improvements [691].

In any case, there are several combinations to perform communication operations in an asynchronous or synchronous way. Here, we send messages asynchronously and receive them synchronously. We perform asynchronously send operations so that processors do not have to wait until send operations finish completely in order to continue with the execution. Moreover, we perform synchronous receive operations because we need the data to continue sorting.

Collective Operations. Some communication operations like the all-to-all operation allow a group of processors to communicate with each other to implement a communication pattern composed of many elementary communication operations. These operations usually make the algorithm more readable and more efficient.

In our case, we can reduce the number of explicit messages to only one all-to-all communication operation, which is a collective operation [605]. We have implemented it with an MPI *alltoallv* function. With this function, each processor sends parts of its data to the rest of processors, including itself. Also, each processor receives a vector part from each of the processors, including itself [308].

Here, once data are received we also have to redistribute keys among buckets. This is an extra cost that must also be taken into account, since it may be significant in the overall performance.

Algorithm Drawbacks

The distributed memory SF-Radix sort has a serious problem related to the data communicated. The algorithm may keep moving one key from one processor to another at each data communication step. There are as many data communication steps as number of digits. Because of this, SF-Radix sort has very little data locality.

16.4 Memory Conscious Algorithms

As shown in Fig. 16.1 in the introduction, SF-Radix sort is not a good approach for solving our sorting problem. In this section, we analyze a better parallel sorting approach, $C3$-Radix sort.

We start by explaining the Cache Conscious Radix sort [433], the sequential algorithm on which $C3$-Radix sort is based. Then we analyze the shared and distributed memory versions of $C3$-Radix sort, and some results on the SGI O2000 are discussed. With this analysis, we want to show the possible influence of the computer architecture on the performance of a parallel algorithm, once the algorithm is conscious of the architecture of the target computer.

16.4.1 Cache Conscious Radix Sort

The Cache Conscious Radix sort, CC-Radix sort, is based on the same principle as radix sort. However, it starts sorting by the most significant digits. The objective of CC-Radix sort is to improve the locality of the key-pointer vectors S and D, and the counter vector C when sorting large data sets. This is done by optimizing the use of radix sort as explained in [433]. Other works, such as [12] and [47], use the same idea of sorting by the most significant digits with objectives other than exploiting data locality.

With the help of the example in Fig. 16.6 we explain CC-Radix sort. First, the algorithm performs the three steps of the counting algorithm on the most significant digit. With this process, called Reverse sorting in [433], keys are distributed in such a way that vector S may be partitioned into a total of $2^{b_{m-1}}$ buckets using the b_{m-1} most significant bits of the key (10 buckets in the example, because we are working with decimal digits).

Note that, after sorting by the most significant digit, keys belonging to different buckets are already sorted among these buckets. For instance, keys 1 and 4 (bucket 0) of the example are smaller than 10, 17 and 12 (bucket 1), and these latter are smaller than 29, 26 and 21 (bucket 2), and so on. Therefore, we only have to individually sort each bucket to obtain a completely sorted vector. Each bucket can be sorted by any sorting algorithm; we use the plain radix sort version mentioned above.

Actually, the idea behind this is a sample sort [236], where the sample keys have been statically chosen. In that sample sort, the sample is formed by sample keys with values 00, 10, 20, etc. However, in our case, the algorithm does not make the sample explicity.

16.4.2 Shared Memory $C3$-Radix Sort

The algorithm is very similar to Alg. 2, but now it has only one loop iteration of the code in lines 3 to 18, which is for the most significant digit. Then, each

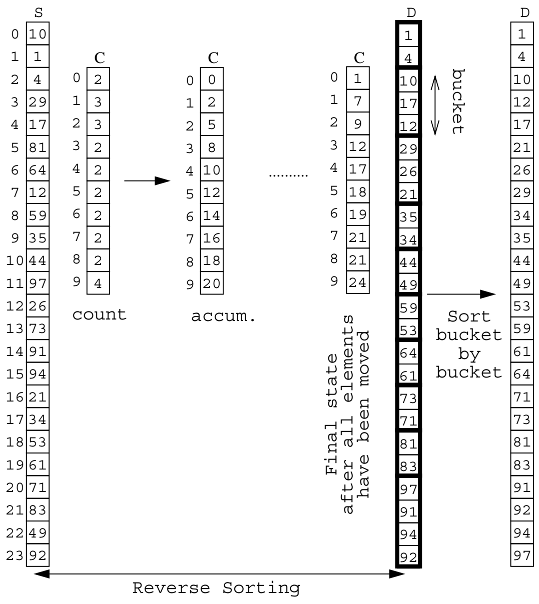


Fig. 16.6. Example for CC-Radix sort for 2 decimal digit keys

processor individually sorts bucket by bucket to obtain a completely sorted vector. Therefore, each processor performs six steps. The four first steps are the same as those explained for the shared memory SF-Radix sort, but this time they are applied only for the most significant digit. Thus, each processor:

1. Performs the initialization of its counters in the global counter matrix *Gl*.
2. Computes a histogram of the number of keys for each possible value of the most significant digit with its local keys.
3. Computes the local accumulation vector $LCacc_{pid}$ with the *partial_sum* call (using the global counters *Gl*).
4. Moves its local keys from vector S to vector D using local counters $LCacc_{pid}$.
5. Locally decides a distribution of the buckets formed in the previous step among processors in such a way that each processor has almost the same load. If it is not possible to obtain a good load balance among processors, all processors can perform steps 1 to 4 (a Reverse sorting step) again by the next most significant digit, b_{m-2}.
6. Individually sorts its corresponding buckets one by one using CC-Radix sort. Each bucket requires that only the $b - \sum_{i=1}^{rss} b_{m-i}$ least significant bits are sorted, where *rss* is the number of Reverse sorting steps.

Note that now, after the first five steps, each processor is always working with the same part of the vector, so that the number of movements from one processor's cache to another has been reduced to only one. Therefore, the shared memory $C3$-Radix sort can exploit the data locality much better than the shared memory SF-Radix sort. In addition, the number of synchronization points (three per digit to be sorted) has been reduced to those for the most significant digit. Therefore, the shared memory $C3$-Radix sort algorithm overcomes the drawbacks of the shared memory version of SF-Radix sort.

Analysis

Table 16.2 shows the number of L_2 misses, the number of invalidations, and the execution time for two versions of the shared memory $C3$-Radix sort algorithm. These two versions correspond to (i) the case where a global counter matrix Gl is used to compute the histogram (plain implementation), and (ii) the case where the histogram is computed using a local counter vector and then a global matrix Gl is updated (memory conscious implementation). On the one hand, we can see that a non memory conscious implementation may cause a significant loss of performance, even with a memory conscious algorithm like $C3$-Radix sort. The memory conscious implementation is about 8.5 times faster than the plain implementation.

On the other hand, if we compare the execution times and the number of L_2 misses and invalidations of the results in Table 16.2 to those in Table 16.1, we can see that $C3$-Radix sort is significantly better than SF-Radix sort. First, for the plain implementation, $C3$-Radix sort is 2.35 times faster than SF-Radix sort. The reason for this can be found in the number of L_2 misses and invalidations of both algorithms. The number of L_2 misses and invalidations of $C3$-Radix sort are only about 39% and 45% of the L_2 misses and invalidations of the SF-Radix sort algorithm, respectively.

As for the memory conscious implementation, the number of invalidations is similar to SF-Radix sort. However, the number of L_2 misses is half the number of L_2 misses of SF-Radix sort, and $C3$-Radix sort is 2.25 times faster than the memory conscious SF-Radix sort.

Table 16.2. Second level cache average number of misses and invalidations per processor, and execution time for sorting 16 million record with 8 processors with $C3$-Radix sort

$C3$-Radix sort	L_2 misses	Invalidations	Exec. Time (s)
Plain Impl.	3,875,000	4,338,310	21.69
Memory Conscious	833,559	311,237	2.55

16.4.3 Distributed Memory $C3$-Radix Sort

The algorithm is very similar to Alg. 3. As happens with the shared memory version, it performs only one iteration to sort the most significant digit (lines

2 to 23 in the code of Alg. 3), and then each processor individually and locally sorts a set of buckets. Each processor performs five steps. The first two steps are the same as the first two steps of the distributed memory SF-Radix sort. Therefore, each processor P_{pid}:

1. Performs the three sequential steps of the counting algorithm on the most significant digit.
2. Broadcasts its local counters.
3. Locally computes a bucket range distribution to get an even partition of the buckets among processors. If it is not possible to achieve a good load balance, each processor can start again at step 1 using the next most significant digit, that is, another Reverse sorting step using digit b_{m-2}. See [430] for more details.
4. Performs the only data communication step of this algorithm taking into account of the partitioning of the previous step. After this step, each processor has a set of global buckets in its local memory.
5. Locally sorts its global buckets. Each global bucket is sorted using CC-Radix. Only the the $b - \sum_{i=1}^{rss} b_{m-i}$ least significant bits of the keys of each bucket need to be sorted, where rss is the number of Reverse sorting steps.

$C3$-Radix sort overcomes the main drawback mentioned above for the distributed memory SF-Radix sort. $C3$-Radix sort performs only one step of communication, independently of the number of digits of the keys. Therefore, $C3$-Radix can exploit the data locality of the keys when sorting each bucket. Nevertheless, SF-Radix sort does not exploit the data locality because it performs as many communciation steps as number of digits.

Analysis

In this section, we analyze the four different implementations of the data communication step described in Sect. 16.3.3. We implement them on $C3$-Radix sort. Fig. 16.7 shows, from top to bottom, the execution time of the plain solution of $C3$-Radix sort (sending bucket by bucket), the solution that groups messages, the one that groups and asynchronously sends messages, and finally the implementation that performs all-to-all collective operations.

First, the number of sends (lines from one processor to another in the figure) of the bucket by bucket solution is significantly reduced by grouping messages (second figure from the top). Therefore, the execution time is also shorter than sending bucket by bucket. This is because we have reduced the overall overhead of the messages, keeping the same amount of data sent accross the network.

The third window from the top shows results for grouping and asynchronously sending messages. This implementation improves the two previous implementations. First, it reduces the overall overhead, and it also overlaps communication with computation and/or other communications. With the

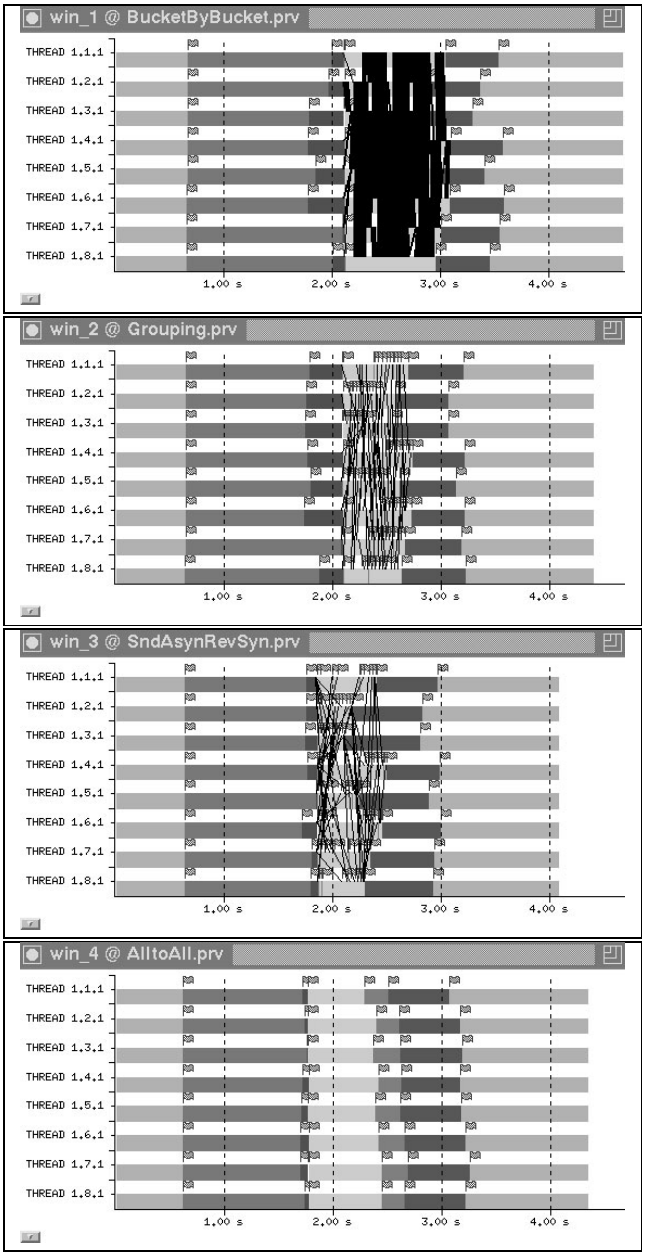


Fig. 16.7. Distributed memory execution times for different solutions of $C3$-Radix sort *(from top to bottom, bucket by bucket, grouping, grouping and asynchronously sending, and all-to-all collective operation solution)*

synchronous versions, every time a processor sends a message it cannot continue until the communication of the message is finished.

Finally, the last window shows the execution of the implementation using an all-to-all collective operation. This window does not show the messages explicitly because of the characteristics of the all-to-all message primitive. This operation gives a comparable performance to the implementation that groups messages (second window from top). The additional cost due to copies, commented in Sect. 16.3.3 is significant and makes this implementation worse than that of the third window. This additional cost can be seen in Fig. 16.7. It is the grey part (between marks) that completely falls between the *2.00 s and 3.00 s* vertical dotted lines in Fig. 16.7.

16.5 Conclusions

In this chapter, we consider the parallel sorting problem as a case study. To study this problem, we analyze two different algorithms, the Straight Forward Radix sort, SF-Radix sort, and the Communication and Cache Conscious Radix sort, $C3$-Radix sort. We also explain different implementation techniques to improve the memory hierarchy response, and how they can be applied to both algorithms. We analyze both the shared memory and the distributed memory implementations of those algorithms.

On one hand, this chapter shows the impact of the implementation details and the communication mechanisms on the performance of an algorithm.

For shared memory, the total execution time of different implementations of SF-Radix sort and $C3$-Radix sort is closely related to the number of L_2 misses and invalidations (those may lead to L_2 misses) of those implementations. We have seen that the larger the number of L_2 misses, the worse the performance of the implementation. For instance, the memory conscious implementation of the shared memory SF-Radix sort and $C3$-Radix sort are respectively 8.9 and 8.5 times faster, and it has a significantly smaller number of L_2 misses than the plain implementation of those algorithms. Some of the techniques explained in this chapter for reducing the number of true and false sharing misses are data placement, padding and data alignment.

For distributed memory, the large number of synchronous messages sent has an influence on the performance of $C3$-Radix sort. In this chapter, we have shown that reducing the number of messages may improve the performance of the implementation, because we reduce the fixed cost of overhead of the messages. Asynchronous messages may also improve the performance of the implementation if we can overlap communication with computation. For instance, the solution that groups and sends messages asynchronously is 1.25 times faster than the plain implementation of $C3$-Radix sort, which sends a large number of messages synchronously.

On the other hand, we show that $C3$-Radix sort, which is a memory conscious algorithm, performs much better than SF-Radix sort. Table 16.3 shows the total execution times of SF-Radix sort and $C3$-Radix sort for the two programming models (shared and distributed), and the improvements achieved by $C3$-Radix sort (how many times faster the algorithm is). These substantial

Table 16.3. Total execution times in seconds for the shared and distributed memory of SF-Radix sort and $C3$-Radix sort. The execution time is for sorting 16M records with 8 processors with SF-Radix sort. *(x)* indicates the speed up of $C3$-Radix sort compared to SF-Radix sort

Programming Model	SF-Radix sort	$C3$-Radix sort
Shared Memory	5.76	2.55 (2.25)
Distributed Memory	6.56	2.36 (2.78)

improvements (2.25 and 2.78 times faster) are caused by the reduction in the number of data communication steps to only one, and the better exploitation of data locality by $C3$-Radix sort.

Finally, we can observe that the distributed memory implementation performs better than the shared memory implementation of $C3$-Radix sort for the executions we have analyzed, as shown in Table 16.3. However, shared memory algorithms are much simpler than distributed memory algorithms. In any case, this relative performance may vary depending on the number of processors, the size of data to be sorted, the amount of data communicated, the computer architecture, the communication mechanisms used, etc.

This chapter concludes that the efficiency of the solutions for a given problem depends on the degree of memory consciousness of the algorithm, and on the algorithm implementation details chosen to solve the problem.

16.6 Acknowledgments

This work was supported by the Ministry of Education and Science of Spain under contract TIC-0511/98 and TIC2001-0995-C02-01, CIRI [6], CEPBA, "Direcció General de Recerca of the Generalitat de Catalunya" under grant 1998FI-00283-APTIND and by an IBM CAS Fellowship grant.

We wish to thank Jop Sibeyn, Peter Sanders and the anonymous reviewers for their valuable contribution to improve the quality of this paper. We would also like to thank Núria Nicolau-Balasch for her help in the bibliography search; CEPBA for letting us use Paraver, and specially to Jordi Caubet who helped us a lot in using the trace tools of Paraver. Finally, we wish to thank Xavi Serrano, Victor Mora, Alex Muntada, and Jose A. Rodríguez, system managers of our Lab., who helped us to round off this chapter by solving some technical problems.

[6] CEPBA-IBM Research Institute, `http://www.cepba.upc.es/ciri/`

Bibliography

[1] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[2] F. Abolhassan, J. Keller, and W. J. Paul. On the physical design of PRAMs. *COMPJ: The Computer Journal*, 36(8):756–762, 1993.

[3] A. Acharya, H. Zhu, and K. Shen. Adaptive algorithms for cache-efficient trie search. In *Proc. 1st Workshop on Algorithm Engineering and Experimentation*, volume 1619 of *LNCS*, pages 296–311, 1999.

[4] P. K. Agarwal, L. A. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions (extended abstract). In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–20, 1999.

[5] P. K. Agarwal, L. A. Arge, and J. Erickson. Indexing moving points (extended abstract). In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 175–186, 2000.

[6] P. K. Agarwal, L. A. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61(2):194–216, October 2000.

[7] P. K. Agarwal, L. A. Arge, T. M. Murali, K. Varadarajan, and J. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proc. 9th Symp. on Discrete Algorithms*, pages 117–126. ACM-SIAM, 1998.

[8] P. K. Agarwal, L. A. Arge, and J. Vahrenhold. Time responsive external data structures for moving points. In *Algorithms and Data Structures. 7th International Workshop, WADS 2001, Proceedings*, volume 2125 of *Lecture Notes in Computer Science*, pages 50–61, 2001.

[9] P. K. Agarwal, M. de Berg, S. Har-Peled, M. H. Overmars, M. Sharir, and J. Vahrenhold. Reporting intersecting pairs of convex polytopes in two and three dimensions. *Computational Geometry: Theory and Applications*, 23(2):195–207, September 2002.

[10] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6:407–422, 1991.

[11] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. M. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, 1999.

[12] R. Agarwal. A super scalar sort algorithm for RISC processors. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 240–246, 1996.

[13] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 305–313, 1987.

[14] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 173–185, 1988.

[15] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 204–216, 1987.

[16] A. Aggarwal and C. G. Plaxton. Optimal parallel sorting in multi-level storage. In *Proc. Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 659–668, 1994.

[17] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[18] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[19] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proc. of the ACM/IEEE Supercomputing Conference*, Dallas, Texas, USA, 2000.

[20] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[21] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, Algorithms and Applications.* Prentice Hall, 1993.

[22] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th Int'l. Conference on Very Large Databases*, pages 169–180, 2001.

[23] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMS on a modern processor: Where does time go? In *Proceedings of the 25th Int'l. Conference on Very Large Databases*, pages 54–65, 1999.

[24] S. Albers, A. Crauser, and K. Mehlhorn. Algorithms for very large data sets. unpublished course notes, 1997.

[25] L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal of Discrete Mathematics*, 9:129–150, 1996.

[26] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Separators of low cost. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002.

[27] J. Alemany and J. Thathachar. Random striping news on demand servers. Technical Report TR-97-02-02, University of Washington, Department of Computer Science and Engineering, 1997.

[28] V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, editors. *Computational Science - ICCS 2001, Part II*, volume 2074 of *LNCS*, 2001.

[29] J. F. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Franscico, CA, 1990.

[30] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2001.

[31] V. Allis. A knowledge-based approach to connect-four. The game is solved: White wins. Master's thesis, Vrije Univeriteit, The Netherlands, 1988.

[32] V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Department of CS, Limburg, Maastrich, 1994.

[33] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–608, 1990.

[34] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[35] S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 476–482. ACM Press, 2001.

[36] M. Altieri, C. Becker, and S. Turek. On the realistic performance of linear algebra components in iterative solvers. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing, Proc. of the Int. FORTWIHR Conference on HPSEC*, volume 8 of *LNCSE*, pages 3–12. Springer, 1998.

[37] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[38] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, Sept. 1990. in ACM SIGARCH 90(3).

[39] N. M. Amato and E. A. Ramos. On computing Voronoi diagrams by divide-prune-and-conquer. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 166–175, 1996.

[40] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[41] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[42] B. S. Andersen, J. A. Gunnels, F. Gustavson, and J. Waśniewski. A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. In *Proc. of the 6th Int. Conference on Applied Parallel Computing*, volume 2367 of *LNCS*, pages 287–296, Espoo, Finland, 2002. Springer.

[43] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3rd edition, 1999. `http://www.netlib.org/lapack/lug`.

[44] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, 1997.

[45] T. Anderson, M. Dahlin, J. Neefe, D. Paterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[46] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[47] A. Andersson and S. Nilsson. A new efficient radix sort. In *FOCS: Symposium on Foundations of Computer Science (FOCS)*, pages 714–721. IEEE, 1994.

[48] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 335–342. ACM Press, 2000.

[49] V. V. Anshelevich. The game of Hex: An automatic theorem proving approach to game programming. In *National Conference on Artificial Intelligence (AAAI)*, pages 189–194, 2000.

[50] P. M. Aoki. Generalizing "search" in generalized search trees. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 380–389, 1998.

[51] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing*, pages 2–20. Carleton University Press, 1997.

[52] L. A. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms (extended abstract). In *Algorithms and Data Structures. 4th International Workshop, WADS '95*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345, Berlin, 1995. Springer.

[53] L. A. Arge. The I/O-complexity of ordered binary decision diagram manipulation. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 1004, pages 82–91, 1995.

[54] L. A. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, 1996.

[55] L. A. Arge. External memory data structures. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 1–29. Springer-Verlag, 2001.

[56] L. A. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer, 2002.

[57] L. A. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickeremesinghe. *TPIE User Manual and Reference*, 082902 edition, Aug. 2002. Draft, online at `http://www.cs.duke.edu/TPIE/`.

[58] L. A. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[59] L. A. Arge and P. Bro Miltersen. On showing lower bounds for external-memory computational geometry problems. In *External Memory Algorithmss*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 139–159. American Mathematical Society, Providence, RI, 1999.

[60] L. A. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proc. 8th Scand. Workshop on Algorithmic Theory*, volume 1851 of *LNCS*, pages 433–447. Springer, 2000.

[61] L. A. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *Proceedings of the 29th Annual Symposium on Theory of Computing*, pages 540–548. ACM, 1997.

[62] L. A. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proc. 7th Intern. Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *LNCS*, pages 471–482. Springer, 2001.

[63] L. A. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Advances in Database Technology – EDBT 2000. 7th International Conference on Extending Database Technology*, volume 1777 of *Lecture Notes in Computer Science*, pages 413–429, Berlin, 2000. Springer.

[64] L. A. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB'98: Proceedings of the 24th International Conference on Very Large Data Bases*, pages 570–581, 1998.

[65] L. A. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems (extended abstract). In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 685–694, 1998.

[66] L. A. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proceedings of the 10th European Symposium on Algorithms (ESA '02)*, pages 88–100, 2002.

[67] L. A. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range searching (extended abstract). In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 346–357, 1999.

[68] L. A. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. 2nd Workshop on Algorithm Engeneering and Experiments (ALENEX)*, 2000.

[69] L. A. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location (extended abstract). In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry*, pages 191–200, 2000.

[70] L. A. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Algorithms – ESA ´95. 3rd European Symposium*, volume 979 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1995.

[71] L. A. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *37th Annual Symposium on Foundations of Computer Science*, pages 560–569, 1996.

[72] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[73] D. L. at al. The directory-based cache coherence protocol for the DASH multiprocessors. In *Proceedings of the 17 Int'l. Symposium on Computer Architecture*, pages 148–159. IEEE, 1990.

[74] M. J. Atallah and J.-J. Tsay. On the parallel decomposability of geometric problems. *Algorithmica*, 8:209–231, 1992.

[75] F. M. auf der Heide and R. Wanka. Parallel bridging models and their impact on algorithm design. In Alexandrov et al. [28], pages 628–637.

[76] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[77] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[78] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[79] M. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1990.

[80] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[81] S. Baden and S. Fink. A Programming Methodology for Dual-Tier Multicomputers. *IEEE Transactions on Software Engineering*, 26(3):212–226, 2000.

[82] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[83] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Journal of Information Systems*, 21(6):497–514, 1996.

[84] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proceedings of the 6th International Conference on Information and Knowledge Management*, pages 1–8. ACM, 1997.

[85] R. Baeza-Yates and B. Ribeiro-Neto, editors. *Modern Information Retrieval.* Addison-Wesley, 1999.

[86] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 94035-1000, December 1995.

[87] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[88] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 211–219, 1995.

[89] M. Baldamus, K. Schneider, M. Wenz, and R. Ziller. Can american checkers be solved by means of symbolic model checking? *Electronic Notes in Theoretical Computer Science*, 43, 2002.

[90] L. M. Baptist and T. H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with distributed memory and parallel storage. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 242–250, June 1999.

[91] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Int'l. Symposium on Computer Architecture*, pages 3–14. IEEE, 1998.

[92] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[93] R. D. Barve and J. S. Vitter. A simple and efficient parallel disk merge sort. In *Proc. 11th Ann. ACM Symposium on Parallel Algorithms and Architectures*, pages 232–241, 1999.

[94] F. Bassetti, K. Davis, and D. Quinlan. Temporal locality optimizations for stencil operations within parallel object-oriented scientific frameworks on cache-based architectures. In *Proc. of the Int. Conference on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, Nevada, USA, 1998.

[95] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17(2):342–380, 1994.

[96] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[97] R. Bayer and G. Unterauer. Prefix B-Trees. *ACM Trans. on Database Systems*, 2(1):11–26, 1977.

[98] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304. ACM Press, 1999.

[99] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[100] L. Becker, A. Giesen, K. H. Hinrichs, and J. Vahrenhold. Algorithms for performing polygonal map overlay and spatial join on massive data sets. In *Advances in Spatial Databases. 6th International Symposium, SSD '99*, volume 1651 of *Lecture Notes in Computer Science*, pages 270–285, Berlin, 1999. Springer.

[101] L. Becker, K. H. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 190–197, 1993.

[102] N. Beckmann and H.-P. Kriegel. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD*

International Conference on Management of Data, volume 19.2 of *SIGMOD Record*, pages 322–331, June 1990.

[103] G. Bell and J. Gray. High performance computing: Crays, clusters, and centers. what next? Technical Report MSR-TR-2001-76, Microsoft Research, Microsoft Corporation, August 2001.

[104] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 29–38, 2002.

[105] S. Benkner. VFC: The vienna fortran compiler. *Scientific Programming*, 7(1):67–81, 1999.

[106] S. Benkner and V. Sipkova. Language and compiler support for hybrid-parallel programming on SMP clusters. In *Proceedings of the 4th International Symposium on High Performance Computing*, volume 2327 of *LNCS*. Springer, 2002.

[107] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. GenBank. *Nucleic Acids Research*, 30(1):17–20, 2002.

[108] J. L. Bentley. Algorithms for Klee's rectangle problems. Unpublished Notes, 1977.

[109] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[110] J. L. Bentley and J. B. Saxe. Decomposable searching problems: I. static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.

[111] M. Beran. Decomposable bulk synchronous parallel computers. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99: Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 349–359, 1999.

[112] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the Sixteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 78–86, 1997.

[113] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27.2 of *SIGMOD Record*, pages 142–153, June 1998.

[114] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index strucuture for high-dimensional data. In *22nd International Conference on Very Large Data Bases*, pages 28–39, 1996.

[115] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 745–754. ACM Press, 2000.

[116] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning ways (book)*. Academic Press, vol.I and II, 850 pages, 1982.

[117] R. Berrendorf and B. Mohr. PCL — the performance counter library: A common interface to access hardware performance counters on microprocessors. Technical report, Research Center Juelich GmbH, Juelich, Germany, 2000. `http://www.fz-juelich.de/zam/PCL`.

[118] S. Berson, R. Muntz, S. Ghandeharizadeh, and X. Ju. Staggered striping in multimedia information systems. *ACM SIGMOD International Conference on Management of Data*, 23(2):79–90, 1994.

[119] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search symbolic model checking = efficient conformant planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 467–472, 2001.

[120] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking.

In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 473–478, 2001.

[121] S. N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational Geometry*, 19(2):175–195, 1998.

[122] J. Bevmyr. *Data Parallel Implementation of Prolog.* PhD thesis, Uppsala University, 1996.

[123] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Database Theory – ICDT '99. 7th International Conference, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235, Berlin, 1999. Springer.

[124] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[125] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: A case study on portability of cache performance. In *Proc. 5th International Workshop on Algorithm Engineering*, volume 2141, pages 26–38, 2001.

[126] G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. Pucci. On the effectiveness of D-BSP as a bridging model of parallel computation. In Alexandrov et al. [28], pages 579–588.

[127] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. of the Int. Conference on Supercomputing*, Vienna, Austria, 1997.

[128] Y. Birk. Random RAIDs with selective exploitation of redundancy for high performance video servers. *Readings in multimedia computing and networking*, pages 671–681, 2001.

[129] G. Blankenagel and R. H. Güting. XP-Trees: External priority search trees. Informatikbericht 92, FernUniversität Gesamthochschule Hagen, May 1990.

[130] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 245–254. IEEE Computer Society, 1994.

[131] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[132] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1636–1642, 1995.

[133] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973.

[134] A. Blumer, J. A. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, 1985.

[135] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[136] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402, 1996.

[137] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces—index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[138] C. Böhm and H.-P. Kriegel. Determining the convex hull in large multidimensional databases. In *Data Warehousing and Knowledge Discovery. Third International Conference, DaWaK 2001*, volume 2114 of *Lecture Notes in Computer Science*, pages 294–306, 2001.

[139] J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study: Heap construction. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[140] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–72, 1993.

[141] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *The VLDB Journal*, pages 54–65, 1999.

[142] B. Bonet and H. Geffner. Planning with incomplete information as heurstic search in belief space. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 52–61, 2000.

[143] O. Boruvka. Über ein Minimalproblem. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.

[144] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 140–146, 1999.

[145] P. J. Braam. The Coda distributed file system. *Linux Journal*, 1998.

[146] J. H. Breasted. *The Edwin Smith Surgical Papyrus*, volume 1–2. The Oriental Institute of the University of Chicago, 1930 (Reissued 1991).

[147] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial.* SIAM, second edition, 2000.

[148] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 574–584, 1995.

[149] T. Brinkhoff. *Der Spatial Join in Geo-Datenbanksystemen.* PhD thesis, Ludwig-Maximilians-Universität München, 1994. (in German).

[150] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23.2 of *SIGMOD Record*, pages 197–206, June 1994.

[151] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22.2 of *SIGMOD Record*, pages 237–246, June 1993.

[152] A. Brinkmann, K. A. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual Symposium on Parallel Algorithms and Architectures*, pages 119–128. ACM Press, 2000.

[153] A. Brinkmann, K. A. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, pages 53–62. ACM Press, 2002.

[154] G. S. Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, pages 52–58, 1996.

[155] G. S. Brodal and R. Fagerberg. Funnel heap — a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002.

[156] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[157] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory (SWAT '98)*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer-Verlag, 1998.

[158] A. Z. Broder, R. Kumar, F. Manghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th International World-Wide Web Conference*, 2000. `http://www9.org`.

[159] A. Z. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *20th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 3, pages 1454–1463. IEEE Comput. Soc. Press, 2001.

[160] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, New York, USA, 1998. SIAM.

[161] C. Browne. *Hex Strategy: Making the right connections.* A K Peters, 2000.

[162] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[163] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):142–170, 1992.

[164] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proc. 11th Ann. Symposium on Discrete Algorithms*, pages 859–860. ACM-SIAM, 2000.

[165] W. Burgard. Personal communication, 2002.

[166] D. C. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report CS TR-95-1261, Computer Science Department, University of Wisconsin, Madison, Wisconsin, USA, 1995.

[167] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, pages 165–204, 1994.

[168] P. B. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Algorithms and Data Structures. 4th International Workshop, WADS '95*, volume 955 of *Lecture Notes in Computer Science*, pages 381–392, Berlin, 1995. Springer.

[169] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, January 1995.

[170] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application controlled file caching. In *Proceedings of 1st USENIX Operating Systems Design and Implementation*, pages 165–177, 1994.

[171] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of the Supercomputing Conference.* IEEE/ACM, 2000.

[172] F. Cappello and O. Richard. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques PACT*. IEEE/IFIP, October 1999.

[173] F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Proceeding of the 6th High Performance Computer Architecture Conference*. IEEE, January 2000.

[174] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proceedings of the Sixth Canadian Conference on Computational Geometry*, pages 263–268, 1994.

[175] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM Journal on Computing*, 30(2):561–575, June 2000.

[176] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of 1st USENIX Operating Systems Design and Implementation*, pages 1–14, 1999.

[177] E. Charniak and D. McDermott. *Introduction to artificial intelligence.* Addison-Wesley, 1985.

[178] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proc. 6th International Symposium on High-Performance Computer Architecture*, pages 195–205, 2000.

[179] E. Chávez and G. Navarro. A metric index for approximate string matching. In S. Rajsbaum, editor, *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, volume 2286 of *LNCS*, pages 181–195. Springer, 2002.

[180] B. M. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[181] B. M. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, January 1992.

[182] B. M. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[183] B. M. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.

[184] B. M. Chazelle, L. J. Guibas, and D.-T. Lee. The power of geometric duality. *BIT: Computer Science*, 25(1):76–90, 1985.

[185] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[186] P. M. Chen and D. A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual Symposium on Computer Architecture, Computer Architecture News*, volume 18(2), pages 322–331. ACM Press, 1990.

[187] T.-F. Chen and J.-L. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[188] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5):972–999, October 1992.

[189] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Parallel and Distributed Information Systems (PDIS)*, pages 31–42, 1996.

[190] K. L. Cheung and A. W. Fu. Enhanced nearest neighbour search on the R-tree. *SIGMOD Record*, 27(3):16–21, 1998.

[191] K. Chew and A. Silberschatz. On the avoidance of the double paging anomaly in virtual memory systems. Technical Report CS-TR-92-20, University of Texas at Austin, 1992.

[192] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the*

Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 139–149, 1995.

[193] A. A. Chien. Computing platforms. In Foster and Kesselman [309], chapter 17.

[194] T. Chilimbi, M. Hill, and J. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.

[195] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[196] T. M. Chilimbi, M. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[197] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graphs problems. *Communications of the ACM*, 25(9):659–665, 1982.

[198] F. Chong, R. Barua, F. Dahlgren, J. Kubiatowicz, and A. Agarwal. The sensitivity of communication mechanisms to bandwidth and latency. In *Proceedings of the Int'l. Conference on High-Performance Computer Architecture*, pages 37–46. IEEE, 1998.

[199] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th Int'l. Conference on Very Large Databases*, pages 127–141, 1985.

[200] E. Chow and D. Hysom. Assessing performance of hybrid mpi/openmp programs on smp clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001.

[201] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegy–Mellon University, 1976.

[202] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, California, USA, 1995.

[203] A. Cimatti and M. Roveri. Conformant planning via model checking. In *European Conference on Planning (ECP)*, pages 21–33, 1999.

[204] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, pages 875–881, 1998.

[205] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proceedings of the 43rd Annual Symposium on Foundations of Computer Science*, pages 219–227. IEEE, 2002.

[206] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the 7th Annual Symposium on Discrete Algorithms*, pages 383–391. ACM–SIAM, 1996.

[207] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[208] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 268–279. ACM, 1985.

[209] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9:251–280, 1990.

[210] P. F. Corbett, D. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmaricich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. R. Morgen, and A. Zlotek. Parallel file systems for IBM SP computers. *IBM Systems Journal*, 34(2):222–248, 1995.

[211] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[212] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical report, Department of Computer Science, Dartmouth College, 1994.

[213] T. H. Cormen and M. T. Goodrich. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys*, 28(4es):208–208, Dec. 1996. Position Statement.

[214] T. H. Cormen and M. Hirschl. Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation. Technical Report PCS-TR96-293, Dartmouth College, Computer Science, Sept. 1996.

[215] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[216] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[217] D. Cornell and P. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Trans. Software Engineering*, 16(2):248–258, 1990.

[218] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. Technical report, Data Engineering Lab, Department of Informatics, Aristotle University of Thessaloniki, Greece, 1999.

[219] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29.2 of *SIGMOD Record*, pages 189–200, June 2000.

[220] T. Cortes, S. Girona, and L. Labarta. PACA: A distributed file system cache for parallel machines. performance under UNIX-like workload. Technical Report UPC-DAC-RR-95/20, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[221] T. Cortes and J. Labarta. A case for heterogenenous disk arrays. In *Proceedings of the International Conference on Cluster Computing*, pages 319–325. IEEE Computer Society Press, 2000.

[222] T. Cortes and J. Labarta. Extending heterogeneity to RAID level 5. In *Proceedings of the USENIX 2001*, pages 119–132. USENIX Association, 2001.

[223] T. P. P. Council. www.tpc.org.

[224] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[225] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/Output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, 1995.

[226] A. Crauser. *External Memory Algorithms and Data Structures in Theory and Practice.* PhD thesis, MPI-Informatik, Universität des Saarlandes, 2001.

[227] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[228] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[229] A. Crauser and K. Mehlhorn. *LEDA-SM, A Platform for Secondary Memory Computation.* Max-Planck-Institut für Informatik, Saarbrücken, Germany, Mar. 1999.

[230] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[231] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[232] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[233] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[234] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc, 1st edition, 1999.

[235] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of ACM International Conference on Management of Data(SIGMOD)*, pages 257–266, 1993.

[236] D. S. D. DeWitt, J. Naughton. Parallel sorting on shared nothing architecture using probabilistic splitting. In *Proceedings of the 1st Int'l. Conference on Parallel and Distributed Info Systems*, pages 280–291, 1992.

[237] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *In Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.

[238] A. Darte. On the complexity of loop fusion. In *Proc. of the Int. Conference on Parallel Architectures and Compilation Techniques*, pages 149–157, Newport Beach, California, USA, 1999.

[239] M. de Berg, M. J. van Kreveld, M. H. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, Berlin, second edition, 2000.

[240] De La Torre, P. and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 - Parallel Processing, vol. II*, volume 1124 of *LNCS*, pages 352–360, 1996.

[241] E. DeBenedictis and J. M. De Rosario. nCUBE parallel I/O software. In *Proceedings of 11th International Phoenix Conference on Computers and Communication*, 1992.

[242] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *SPAA '97*, pages 106–115, 1997.

[243] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. 13th International Parallel Processing Symposium (IPPS'99)*, pages 14–20, Puerto Rico, 1999.

[244] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, pages 1–31, Sept. 2002. Electronic version of the journal, DOI: 10.1007/s00224-002-1066-2.

[245] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse-grained multicomputers. In *Proc. of the 9th Symp. on Computational Geometry*, pages 298–307. ACM, 1993.

[246] F. Dehne, S. Mardegan, A. Pietracaprina, and G. Prencipe. Distribution sweeping on clustered machines with hierarchical memories. In *Proc. of 2002 IPDPS Conference*, 2002.

[247] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of 7th International Parallel Processing Symposium Workshop on Input/Output in Parallel Computer Systems*, 1993.

[248] E. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems, HPCS*, pages 153–163, 1997.

[249] D. C. Dennet. Minds, machines, and evolution. In C. Hookway, editor, *Cognitive Wheels: The Frame Problem of AI*, pages 129–151. Cambridge University Press, 1984.

[250] D. deWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebreaker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 1–8. ACM, 1984.

[251] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *13th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, 1996.

[252] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[253] J. F. Dillenburg and P. C. Nelson. Perimeter search (research note). *Artificial Intelligence*, 65(1):165–178, 1994.

[254] W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems. *ACM Trans. Comput. Syst.*, 34:145–189, 2001.

[255] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[256] C. C. Douglas. Caching in with multigrid algorithms: Problems in two dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[257] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[258] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[259] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems*, pages 75–80, Schloss Elmau, Germany, 2001. IEEE Computer Society Press.

[260] J. Eckerle. *Memory-Limited Heuristic Search (German)*. PhD thesis, University of Freiburg, 1998. DISKI, Infix.

[261] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency: Comparison with existing algorithms. *ACM Transactions on Graphics*, 3(2):86–109, April 1984.

[262] S. Edelkamp. Suffix tree automata in state space search. In *German Conference on Artificial Intelligence (KI)*, pages 381–385, 1997.

[263] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.

[264] S. Edelkamp. Prediction of regular search tree growth by spectral analysis. In *German Conference on Artificial Intelligence (KI)*, pages 154–168, 2001.

[265] S. Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Model Checking*, pages 40–48, 2002.

[266] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 274–283, 2002.

[267] S. Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research*, 2003. Submitted. (Preliminary version available from PUK-Workshop 2002).

[268] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning (ECP)*, pages 135–147, 1999.

[269] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 2003.

[270] S. Edelkamp and P. Leven. Directed automated theorem proving. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 145–159, 2002.

[271] S. Edelkamp and U. Meyer. Theory and practice of time-space trade-offs in memory limited search. In *German Conference on Artificial Intelligence (KI)*, pages 169–184, 2001.

[272] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.

[273] S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.

[274] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *International Journal on Computer Mathematics*, 13:209–219, 1983.

[275] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1987.

[276] D. R. Edelson and I. Pohl. A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85–102. USENIX Association, 1991.

[277] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, 1984.

[278] S. J. Eggers and T. E. Jereniassen. Eliminating false sharing. In *Proceedings of the Intl. Conference on Parallel Processing*, pages 377–381, 1991.

[279] N. Eiron, M. Rodeh, and I. Steinwarts. Matrix multiplication: A case study of enhanced data cache utilization. *ACM Journal of Experimental Algorithmics*, 4, 1999.

[280] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113, 1988.

[281] J. W. Estes. *The Medical Skills of Ancient Egypt.* Canton: Science History Publications, 1989.

[282] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proc. of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 303–314, 1995.

[283] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[284] R. Fadel, K. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220:345–362, 1999.

[285] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[286] C. Faloustos, R. Ng, and T. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. on Computers*, pages 546–560, 1995.

[287] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 247–252, 1989.

[288] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[289] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[290] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[291] A. Felner. Finding optimal solutions to the graph-partitioning problem with heuristic search. In *Symposium on the Foundations of Artificial Intelligence*, 2001.

[292] Z. Feng and E. Hansen. Symbolic heuristic search for factored markov decision processes. In *National Conference on Artificial Intelligence (AAAI)*, pages 455–460, 2002.

[293] J. Fenlason and R. Stallman. *GNU gprof*. Free Software Foundation, Inc., Boston, Massachusetts, USA, 1998. `http://www.gnu.org`.

[294] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Workshop on Embedded Systems (EMSOFT)*, number 2211 in LNCS, pages 469–485. Springer, 2001.

[295] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the 7th Annual Symposium on Discrete Algorithms*, pages 373–382. ACM–SIAM, 1996.

[296] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[297] P. Ferragina, R. Grossi, and M. Montangero. Note on updating suffix tree labels. *Theoretical Computer Science*, 201(1–2):249–262, 1998.

[298] P. Ferragina and F. Luccio. Dynamic dictionary matching in external memory. *Information and Computation*, 146(2):85–99, 1998.

[299] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[300] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. *Information Sciences*, 135(1–2):13–28, 2001.

[301] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In U. Banerjee, editor, *Proc. of the Fourth Int. Workshop on Languages and Compilers for Parallel Computing*, LNCS. Springer, 1991.

[302] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 762 of *LNCS*, pages 416–425. Springer, 1993.

[303] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[304] U. A. Finke and K. H. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 119–126, New York, 1995. ACM Press.

[305] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20(4):345–369, 1983.

[306] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7, 1964.

[307] R. W. Floyd and R. L. Rivest. Algorithm 489: The algorithm SELECT—for finding the ith smallest of n elements [M1]. *Communications of the ACM*, 18(3):173, March 1975.

[308] M. Forum. http://www.mpi-forum.org/.

[309] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Pub., July 1998.

[310] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153–174, April 1984.

[311] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Artificial Intelligence Research*, 9:367–421, 1998.

[312] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK, 2001.

[313] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 83–91, 2002.

[314] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[315] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[316] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, July 1997.

[317] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[318] M. Frigo. A fast fourier transform compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, pages 169–180, Atlanta, GA, 1999.

[319] M. Frigo. Portable high-performance programs. Technical Report MIT/LCS/TR-785, MIT, 1999.

[320] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proc. of the Int. Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, Washington, USA, 1998.

[321] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[322] A. Fukunaga, G. Tabideau, S. Chien, and C. Yan. ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *International Symposium on AI, Robotics and Automation in Space*, 1997.

[323] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[324] D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[325] M. Gardner. *The Mathematical Games of Sam Loyd*. Dover Publications, 1959.

[326] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, Department of Computer Science, ETH Zurich, 1993.

[327] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs with action costs. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 13–22, 2002.

[328] S. Ghandeharizadeh and C. Shahabi. Management of physical replicas in parallel multimedia information systems. *LNCS, Foundations of Data Organisation and Algorithms 1993*, 730:51–68, 1993.

[329] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. of the Int. Conference on Supercomputing*, pages 317–324, Vienna, Austria, 1997.

[330] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, 1997.

[331] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* PhD thesis, University of California at Berkeley, Dec. 1991.

[332] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, 1997.

[333] M. Ginsberg. Step toward an expert-level bridge-playing program. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 584–589, 1999.

[334] F. Giunchiglia and P. Traverso. Planning as model checking. In *European Conference on Planning (ECP)*, pages 1–19, 1999.

[335] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[336] C. M. Gold. Problems with handling spatial data – the Voronoi approach. *CISM Journal ACSGC*, 45(1):65–80, 1991.

[337] B. Golden and T. Magnanti. Transportation planning: Network models and their implementation. *Studies in Operations Management*, pages 365–518, 1978.

[338] J. Goldstein and R. Ramakrishnan. Contrast plots and P-Sphere trees: Space vs. time in nearest neighbor searches. In *VLDB 2000: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 429–440, 2000.

[339] G. H. Golub and C. F. Van Loan. *Matrix Computations.* John Hopkins University Press, third edition, 1998.

[340] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, 1992.

[341] G. H. Gonnet and P.-Å. Larson. External hashing with limited internal storage. *Journal of the Association for Computing Machinery*, 35(1):161–184, 1988.

[342] J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, 1997.

[343] M. T. Goodrich. Constructing the convex hull of a partially sorted set of points. *Computational Geometry: Theory and Applications*, 2(5):267–278, March 1993.

[344] M. T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995.

[345] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[346] S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications (extended abstract). In *Algorithms – ESA 2000. 8th Annual European Symposium*, volume 1879 of *Lecture Notes in Computer Science*, pages 220–231, Berlin, 2000. Springer.

[347] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[348] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft SQL server. In *Proceedings of the 24th Int'l. Conference on Very Large Databases*, pages 86–97, 1998.

[349] G. Graefe and P. Larson. B-tree indexes and CPU caches. In *Int'l. Conference on Data Engineering*, pages 349–358. IEEE, 2001.

[350] T. G. Graf. *Plane-sweep construction of proximity graphs.* PhD thesis, Fachbereich Mathematik, Westfälische Wilhelms-Universität Münster, Germany, 1994.

[351] T. G. Graf and K. H. Hinrichs. Algorithms for proximity problems on colored point sets. In *Proceedings of the Fifth Canadian Conference on Computational Geometry*, pages 420–425, 1993.

[352] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[353] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1989.

[354] A. Grama, V. Kumar, S. Ranka, and V. Singh. Architecture independent analysis of parallel programs. In Alexandrov et al. [28], pages 599–608.

[355] E. D. Granston and H. A. G. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 7th international conference on Supercomputing*, pages 11–20. ACM Press, 1993.

[356] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[357] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of USENIX Summer 1994 Technical Conference*, pages 197–207, 1994.

[358] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. High performance parallel implicit CFD. *Parallel Computing*, 27(4):337–362, 2001.

[359] R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. Rapporto di Ricerca CS-96-14, Università "Ca' Foscari" di Venezia, Italy, 1996.

[360] R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 87–106. American Mathematical Society, Providence, RI, 1999.

[361] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual Symposium on Theory of Computing*, pages 397–406. ACM, 2000.

[362] O. Günther. Efficient computation of spatial joins. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 50–59, 1993.

[363] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *1990 International Conference on Parallel Processing*, volume I, pages 312–321, St. Charles, Ill., 1990.

[364] P. Gupta, R. Janardan, and M. Smid. Efficient algorithms for counting and reporting pairwise intersections between convex polygons. *Information Processing Letters*, 69(1):7–13, January 1999.

[365] C. Gurret and P. Rigaux. The Sort/Sweep algorithm: A new method for R-tree based spatial joins. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management*, pages 153–165, 2000.

[366] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[367] S. Gutmann. *Robust navigation of autonomous mobile systems (German).* PhD thesis, University of Freiburg, 1999. ISBN 3-89838-241-9.

[368] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD '84 Proceedings of Annual Meeting*, volume 14.2 of *SIGMOD Record*, pages 47–57, June 1984.

[369] K. Haase. Framer: A persistent portable representation library. In *European Conference on Artificial Intelligence (ECAI)*, pages 732–738. Wiley, 1994.

[370] W. Hackbusch. *Multigrid Methods and Applications.* Springer, 1985.

[371] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Sciences.* Springer, 1993.

[372] T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, 1990.

[373] T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[374] T. Hagerup. Improved shortest paths on the word RAM. In *27th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 61–72. Springer, 2000.

[375] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990.

[376] M. Hall Jr. *Combinatorial Theory.* Blaisdell Publishing Company, 1967.

[377] H. Han, G. Rivera, and C. Tseng. Software support for improving locality in scientific codes, 2000.

[378] J. Handy. *The Cache Memory Book.* Academic Press, 1998.

[379] P. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. Lebeck, and E. Parker. The combinatorics of cache misses during matrix multiplication. *Journal of Computer and Systems Sciences*, 63:80–126, 2001.

[380] E. A. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 83–98, 2002.

[381] E. A. Hansen and S. Zilberstein. Monitoring and control of anytime algorithms: a dynamic programming approach. *Artificial Intelligence*, 126:139–158, 2001.

[382] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[383] J. Harper, D. Kerbyson, and G. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, 1999.

[384] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.

[385] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. Technical Report TR99-06, The Department of Computer Science, University of Arizona, 1999.

[386] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4-6):175–181, 2000.

[387] E. A. Heinz. *Scalable Search in Computer Chess.* Vierweg, 2000.

[388] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM*, 49(1):35–55, January 2002.

[389] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[390] L. Hellerstein, G. Gibson, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[391] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 303–312, 2002.

[392] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, second edition, 1996.

[393] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proceedings of the Second ACM Workshop on Advances in Geographic Information Systems*, pages 136–143, 1994.

[394] D. Henrich, C. Wurll, and H. Wörn. Multi-directional search with goal switching for robot path planning. In *Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE98)*, volume 2, pages 75–84, 1998.

[395] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002.

[396] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38:1612–1630, 1989.

[397] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? In *VLDB 2000: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 506–515, 2000.

[398] K. H. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26(5):255–261, January 1988.

[399] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1, 2001. `http://developer.intel.com/technology/itj/q12001/articles/art_2.htm`.

[400] D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[401] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27.2 of *SIGMOD Record*, pages 237–248, June 1998.

[402] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 606–618, 1995.

[403] J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Artificial Intelligence Research*, 14:253–302, 2001.

[404] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27(9), pages 23–35. ACM Press, 1992.

[405] R. Holte and I. Hernadvölgyi. A space-time tradeoff for memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI)*, pages 704–709, 1999.

[406] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)*, pages 326–333, 1981.

[407] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[408] P. V. C. Hough. Method and means for recognizing complex patterns. U. S. Patent No. 3069654, 1962.

[409] J. H. Howard. An overview of the Andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 23–26, 1988.

[410] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific American*, 4:44–50, 1990.

[411] W. Hsu, A. Smith, and H. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[412] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation*, pages 266–271, 1993.

[413] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. In *Proceedings of the 2nd Merged Symposium International Parallel and Distributed Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP)*. IEEE, 1999.

[414] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, pages 396–405, 1997.

[415] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A high performance portable file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, 1995.

[416] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[417] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. In *German Conference on Artificial Intelligence (KI)*, pages 229–243, 2001.

[418] F. Hülsemann, P. Kipfer, U. Rüde, and G. Greiner. *gridlib:* flexible and efficient grid management for simulation and visualization. In *Proc. of the Int. Conference on Computational Science, Part III*, volume 2331 of *LNCS*, pages 652–661, Amsterdam, The Netherlands, 2002. Springer.

[419] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proceedings of the 5th ACM-SIAM Computing and Combinatorics Conference*, volume 1627 of *LNCS*, pages 51–60. Springer, July 1999.

[420] IBM Corporation, http://www-124.ibm.com/developerworks/oss/jfs/. *JFS website.*

[421] C. Icking, R. Klein, and T. A. Ottmann. Priority search trees in secondary memory. In *Graph-Theoretic Concepts in Computer Science. International Workshop WG '87, Proceedings*, volume 314 of *Lecture Notes in Computer Science*, pages 84–93, Berlin, 1988. Springer.

[422] W. B. L. III and R. B. Ross. An overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, 1999.

[423] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Third IEEE International Conference on Cluster Computing*, pages 37–44, Oct. 2001.

[424] P. Jackson. *Introduction to Expert Systems.* Addison Wesley Longman, 1999.

[425] G. Jacobson. Space efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554. IEEE, 1989.

[426] H. V. Jagadish. On indexing line segments. In *16th International Conference on Very Large Data Bases*, pages 614–625, 1990.

[427] D. James. Sci (scalable coherent interface). In M. Dubois and S. S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors.* Kluwer, 1989.

[428] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[429] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, California, USA, 1995.

[430] D. Jiménez-González, J. Larriba-Pey, and J. Navarro. Communication Conscious Radix Sort. In *Proceedings of the Int'l. Conference on Supercomputing*, pages 76–82. ACM, 1999.

[431] D. Jiménez-González, J. Navarro, and J. Larriba-Pey. Fast parallel in-memory 64 bit sorting. In *Proceedings of the Int'l. Conference on Supercomputing*, pages 114–122. ACM, 2000.

[432] D. Jiménez-González, J. Navarro, and J. Larriba-Pey. The effect of local sort on parallel sorting algorithms. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 360–367. IEEE, 2002.

[433] D. Jiménez-González, J. Navarro, and J. Larriba-Pey. CC-radix: a cache conscious sorting based on radix sort. In *11th Euromicro Workshop on Parallel, Distributed and Network-based Processing.* IEEE, 2003 (to appear).

[434] D. Jiménez-González, J. Navarro, and J.-L. Larriba-Pey. Analysis of a cache conscious sorting based on radix sort. Technical Report UPC-DAC-2000-2, Universitat Politecnica de Catalunya, 2000.

[435] O. G. Johnson. Three-dimensional wave equation computation on vector computers. In *Special issue — Supercomputers — Their Impact on Science and Technology computational physics*, volume 72(1), pages 90–95. IEEE Computer Society Press, 1984.

[436] T. Johnson and D. Shasha. The performance of current b-tree algorithms. *ACM Trans. on Database Systems*, 18(1):51–101, 1993.

[437] T. Johnson and D. Shasha. 2q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th Int'l. Conference on Very Large Databases*, pages 439–450, 1994.

[438] W. E. Johnston. Rationale and strategy for a 21st century scientific computing architecture: The case for using commercial symmetric multiprocessors as supercomputers. *International Journal of High Speed Computing*, June 1998.

[439] R. Jones. *Garbage Collection.* John Wiley & Sons, 1996.

[440] R. Joseph, D. Brooks, and M. Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. In *Proc. Workshop on Complexity Effective Design, (held in conjunction with ISCA-28)*, 2001.

[441] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, University of Alberta, 1999.

[442] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, 1994.

[443] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612, June 1996.

[444] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Database Theory – ICDT '99. 7th Interna-*

tional Conference, Proceedings, volume 1540 of *Lecture Notes in Computer Science*, pages 257–276, Berlin, 1999. Springer.

[445] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[446] D. R. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 1997. ACM Press.

[447] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in LNCS, pages 195–209. Springer, 2000. Extended version to appear in Journal of Discrete Algorithms.

[448] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155. Carleton University Press, 1996.

[449] P. D. Karp and S. M. Paley. Knowledge representation in the large. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 751–758. Morgan Kaufmann, 1990.

[450] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Annual Symposium on Theory of Computing*, pages 125–136. ACM, 1972.

[451] N. Katoh and K. Iwano. Finding k farthest pairs and k closest/farthest bichromatic pairs for points in the plane. *International Journal of Computational Geometry and Applications*, 5(1&2):37–51, May/June 1995.

[452] H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1194–1201, 1996.

[453] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behaviour to reduce cache leakage power. In *Proc 28th International Symposium on Computer Architecture*, pages pp. 240–251, 2001.

[454] J. Kelly. *Artificial Intelligence: A Modern Myth.* Ellis Horwood, 1993.

[455] A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proceedings of the 25th Int'l. Conference on Very Large Databases*, pages 30–41, 1999.

[456] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.

[457] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, February 1983.

[458] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, February 1986.

[459] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):66–74, 1983.

[460] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison Wesley, 2nd edition, 1998.

[461] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, Nevada, USA, 1997.

[462] J. Koehler. Elevator control as planning problem. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 331–338, 2000.

[463] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 261–272, 1999.

[464] J. Komlós. Linear verification for spanning trees. In *Proc. 25th Ann. Symposium on Fondations of Computer Science*, pages 201–206. IEEE, 1984.

[465] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[466] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[467] R. E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *National Converence on Artificial Intelligence (AAAI)*, pages 266–272, 1998.

[468] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1–2):9–22, 2001.

[469] R. E. Korf, M. Reid, and S. Edelkamp. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1–2):199–218, 2001.

[470] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *National Conference on Artificial Intelligence (AAAI)*, pages 1202–1207, 1996.

[471] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.

[472] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29.2 of *SIGMOD Record*, pages 201–212, June 2000.

[473] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the Twenty-second International Conference on Very Large Data Bases*, pages 215–226, 1996.

[474] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations*, 2(1):1–15, 2000.

[475] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 61–74, 1994.

[476] D. Kotz and C. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, 1993.

[477] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26.2 of *SIGMOD Record*, pages 324–335, June 1997.

[478] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance optimization of 3D multigrid on hierarchical memory architectures. In *Proc. of the 6th Int. Conference on Applied Parallel Computing*, volume 2367 of *LNCS*, pages 307–316, Espoo, Finland, 2002. Springer.

[479] M. Kowarschik, C. Weiß, and U. Rüde. Data layout optimizations for variable coefficient multigrid. In *Proc. of the Int. Conference on Computational Science, Part III*, volume 2331 of *LNCS*, pages 642–651, Amsterdam, The Netherlands, 2002. Springer.

[480] H.-P. Kriegel, T. Brinkhoff, and R. Schneider. The combination of spatial access methods and computational geometry in geographic database systems. In *Advances in Spatial Databases. 2nd Symposium, SSD '91*, volume 525 of *Lecture Notes in Computer Science*, pages 5–22, Berlin, 1991. Springer.

[481] O. Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, 1994.

[482] T. M. Kroeger and D. D. E. Long. Predicting future file-system actions from prior events. In *Proceedings of USENIX Annual Technical Conference*, pages 319–328, 1996.

[483] P. Kumar and E. Ramos. I/O-efficient construction of Voronoi diagrams. Unpublished Manuscript, 2002.

[484] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings Publ. Company, 1994.

[485] V. Kumar and E. J. Schwabe. Improved Algorithms and Data Structures for Solving Graph Problems in External Memory. In *Proc. 8th Symp. on Parallel and Distrib. Processing*, pages 169–177. IEEE, 1996.

[486] S. Kurtz. Approximate string searching under weighted edit distance. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 156–170. Carleton University Press, 1996.

[487] S. Kurtz. Reducing the space requirement of suffix trees. *Software — Practice and Experience*, 29(13):1149–1171, 1999.

[488] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[489] R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 613–622, 1999.

[490] R. E. Ladner, R. Fortna, and B. H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics — From Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*, pages 78–92. Springer-Verlag, 2002.

[491] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. In *Advances in Computer Chess VII*, pages 135–162. University of Limburg, Maastricht, Netherlands, 1994.

[492] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[493] A. LaMarca and R. E. Ladner. Influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[494] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[495] T. Lang, C. Wood, and E. Fernández. Database buffer paging in virtual storage systems. *ACM Trans. on Database Systems*, 2(4):339–351, 1977.

[496] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 472–483. ACM, 1998.

[497] P.-Å. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[498] J. Laudon and D. Lenoski. System overview of the sgi origin 200/2000 product line, 1997.

[499] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27:15–26, 1994.

[500] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX Annual Technical Conference*, 1997.

[501] D. Lenoski and W.-D. Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann, 1995.

[502] A. Levy and D. S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118:1–14, 2000.

[503] H. Levy, T. Messinger, and R. Morris. The cache assignment problem and its application to database buffer management. *IEEE Software Engineering*, 22(11):827–838, 1996.

[504] D. M. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of Electrical Engeneering and Computer Science, Massachusetts Institute of Technology, 1998.

[505] H. Li and K. C. Sevick. NUMACROS: Data parallel programming on numa multiprocessors. In *Proc. of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, San Diego, CA, 1993.

[506] Z. Li and K. Ross. Fast joins using join indices. *VLDB Journal: Very Large Data Bases*, 8(1):1–24, 1999.

[507] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[508] W. Litwin. Linear hashing: A new tool for files and tables addressing. In *International Conference On Very Large Data Bases (VLDB '80)*, pages 212–223. IEEE Comput. Soc. Press, 1980.

[509] A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *Model Checking Software (SPIN)*, pages 112–127, 2002.

[510] J. Lo, L. Barroso, S. Eggers, and K. Gharachorloo. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Int'l. Symposium on Computer Architecture*, pages 39–50. IEEE, 1998.

[511] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23.2 of *SIGMOD Record*, pages 209–220, June 1994.

[512] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25.2 of *SIGMOD Record*, pages 247–258, June 1996.

[513] M. Löbbing and I. Wegener. The number of knight's tours equals 33,439,123,484,294 — counting with binary decision diagrams. *The Electronic Journal of Combinatorics*, 3(1), 1996.

[514] D. Lomet. The evolution of effective B-tree: Page organization and techniques: A personal account. *SIGMOD Record*, 30(3), 2001.

[515] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 196–205, 2000.

[516] U. Lorenz. Controlled conspriracy-2 search. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1770 of *LNCS*, pages 466–478, 2000.

[517] D. Loshin. *Efficient Memory Programming.* McGraw-Hill, 1998.

[518] H. Lötzbeyer and U. Rüde. Patch-adaptive multilevel iteration. *BIT*, 37(3):739–758, 1997.

[519] M. Loukides. *System Performance Tuning.* O'Reilly & Associates, Sebastopol, CA, 1990.

[520] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[521] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. 10th Intern. Symp. on Algorithms and Computations*, volume 1741 of *LNCS*, pages 307–316. Springer, 1999.

[522] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. 12th Ann. Symp. on Discrete Algorithms*, pages 89–90. ACM–SIAM, 2001.

[523] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. 13th Ann. Symp. on Discrete Algorithms*, pages 372–381. ACM–SIAM, 2002.

[524] A. Maheswari, J. Vahrenhold, and N. Zeh. On reverse nearest neighbor queries. In S. Wismath, editor, *Proceedings of the 14th Canadian Conference on Computational Geometry*, pages 128–132, 2002.

[525] H. G. Mairson. The program complexity of searching a table. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS '83)*, pages 40–47. IEEE Comput. Soc. Press, 1983.

[526] S. M. Majercik and M. L. Littmann. Using caching to solve larger probabilistic planning problems. In *National Conference on Artificial Intelligence (AAAI)*, pages 954–959, 2000.

[527] N. Mamoulis and D. Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1), 2003. to appear.

[528] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[529] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

[530] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and *st*-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[531] T. A. Marsland. *Ecyclopedia of Artificial Intelligence*, chapter Computer Chess and Search, pages 224–241. Wiley & Sons, 1992.

[532] T. A. Marsland and J. Schaeffer, editors. *Chess, Computers, and Cognition.* Springer, 1990.

[533] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[534] Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. In *Proc. 11th Ann. Symposium on Discrete Algorithms*, pages 839–848. ACM–SIAM, 2000.

[535] J. Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.

[536] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, 2(3):169–186, 1992.

[537] D. A. McAllester. Conspiricy-number-search for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[538] D. A. McAllester. Automatic recognition of tractability in inference relation. *Journal of the ACM*, 40(2):284–303, 1993.

[539] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[540] E. M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[541] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1984.

[542] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.

[543] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.

[544] C. Meinel and C. Stangier. Hierarchical image computation with dynamic conjunction scheduling. In *Conference on Computer Design (ICCD)*, pages 354–359, 2001.

[545] H. Mendelson. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, SE-8(6):611–619, 1982.

[546] J. Merlin, D. Miles, and V. Schuster. The portland group distributed OMP: Extensions to OpenMP for SMP-clusters. In *Proceedings of 2nd European Workshop on OpenMP (EWOMP)*, Edinburgh, UK, September 2000. Edinburgh Parallel Computing Centre. Electronic Proceedings, `http://www.epcc.ed.ac.uk/ewomp2000/`.

[547] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.

[548] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997.

[549] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. TOP500 supercomputer sites, `http://www.top500.org`, 2002.

[550] U. Meyer. External Memory BFS on Undirected Graphs with Bounded Degree. In *Proc. 12th Ann. Symposium on Discrete Algorithms*, pages 87–88. ACM–SIAM, 2001.

[551] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report A.I. MEMO 58 Rvsd. and MAC-M-129, Cambridge, Massachusetts, 1963.

[552] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[553] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[554] E. F. Moore. The shortest path through a maze. In *Proc. Intern. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[555] R. W. Moore. Enabling petabyte computing. `http://www.nap.edu/html/whitepapers/ch-48.html`, 2000.

[556] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society*, 15:99–117, 1994.

[557] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[558] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Computer Systems Laboratory, Stanford University, 1994.

[559] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of 2nd USENIX Operating Systems Design and Implementation*, pages 3–17, 1996.

[560] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73. ACM Press, 1992.

[561] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, San Francisco, California, USA, 1997.

[562] M. Müller. *Computer Go as a sum of local games.* PhD thesis, ETH Zürich, 1995.

[563] M. Müller. Partial order bounding: A new approach to game tree search. *Artificial Intelligence*, 129(1–2):279–311, 2001.

[564] M. Müller. Proof set search. Technical Report TR01-09, University of Alberta, 2001.

[565] M. Müller and T. Tegos. Experiments in computer amazons. *More Games of No Chance, Cambridge University Press*, 42:243–257, 2002.

[566] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[567] K. Munagala and A. Ranade. I/O-Complexity of Graph Algorithms. In *Proc. 10th Ann. Symposium on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.

[568] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender:. Complexity of finite-horizon markov decision process problems. *Journal of the ACM*, 4:681–720, 2000.

[569] I. Murdock and J. H. Hartman. Swarm: A log-structured storage system for linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, pages 1–10. USENIX Association, 2000.

[570] S. Näher and K. Mehlhorn. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[571] Namesys, http://www.reiserfs.org/. *ReiserFS Whitepaper.*

[572] C. Navarro, A. Ramirez, J. Larriba-Pey, and M. Valero. On the performance of fetch engines running dss workloads. In *Proceedings of the EUROPAR Conference*, pages 591–595. Springer Verlag, 2000.

[573] G. Navarro. A partial deterministic automaton for approximate string matching. In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing*, pages 95–111. Carleton University Press, 1997.

[574] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[575] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proceedings of the 11th Data Compression Conference*, pages 459–468. IEEE, 2001.

[576] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In M. Crochemore and M. Paterson, editors, *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, number 1645 in LNCS, pages 14–36. Springer, 1999.

[577] J. J. Navarro, E. Garcia-Diego, J.-L. Larriba-Pey, and T. Juan. Block algorithms for sparse matrix computations on high performance workstations. In *Proc. of the Int. Conference on Supercomputing*, pages 301–308, Philadelphia, Pennsylvania, USA, 1996.

[578] B. Nebel. Personal communication, 2002.

[579] J. v. Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945. `http://www.histech.rwth-aachen.de/www/quellen/vnedvac.pdf`.

[580] A. Newell, V. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings International Conference on Information Processing (ICIP '59)*, pages 256–264. Butterworth, 1960.

[581] N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4-5):447–476, 1997.

[582] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. File access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[583] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[584] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, October 1982.

[585] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[586] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th Ann. ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[587] M. H. Nodine and J. S. Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, 1995.

[588] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: a RISC machine sort. In *Proceedings of the 1994 SIGMOD International Conference on Management of Data*, pages 233–242. ACM Press, 1994.

[589] M. T. O'Keefe. Shared file systems and Fibre Channel. In *Proceedings of the 6th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, 1998.

[590] R. Oldfield and D. Kotz. A parallel file system for computational grids. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201, 2001.

[591] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K replacement algorithm for database disk buffering. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 297–306. ACM, 1993.

[592] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[593] OpenMP Forum. OpenMP C and C++ application program interface, version 2.0. `http://www.openmp.org`.

[594] J. A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19.2 of *SIGMOD Record*, pages 343–352, June 1990.

[595] O. Organization. http://www.openmp.org/.

[596] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, second edition, 1994.

[597] A. Östlin and R. Pagh. Rehashing rehashed. Manuscript, 2002.

[598] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, 1985.

[599] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, Berlin, 1983.

[600] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[601] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th Int'l. Conference on Data Engineering*, pages 567–574. IEEE, 2001.

[602] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of Symposium on Frontiers of Massively Parallel Computation*, 1995.

[603] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using R-trees. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 44–55, 1999.

[604] P. P. V. Paraver and A. tool. http://www.cepba.upc.es/paraver/.

[605] S. Parthasarathy, M. J. Zaki, and W. Li. Custom memory placement for parallel data mining. Technical Report TR653, Rochester University, 1997.

[606] J. M. Patel and D. J. DeWitt. Partition based spatial–merge join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25.2 of *SIGMOD Record*, pages 259–270, June 1996.

[607] D. A. Patterson, G. A. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference*, pages 109–116. ACM Press, 1988.

[608] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, 1994.

[609] E. Pattison-Gordon. Thenetsys: A semantic network system. Technical Report DSG-93-02, Decision Systems Group, Brigham and Womens Hospital, Boston, 1993.

[610] A. Pinar and B. Hendrickson. Interprocessor communication with memory constraints. In *Proceedings of the Twelfth Annual Symposium on Parallel Algorithms and Architectures*, pages 39–45. ACM Press, 2000.

[611] E. L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.

[612] PostgreSQL. http://www.postgresql.org/.

[613] F. P. Preparata. A new approach to planar point location. *SIAM Journal on Computing*, 10(3):473–482, August 1981.

[614] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction.* Springer, Berlin, second edition, 1988.

[615] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, Nov. 1957.

[616] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[617] M. Raab and A. Steger. "Balls into Bins" – A simple and tight analysis. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170. LNCS, 1998.

[618] N. Rahman, R. Cole, and R. Raman. Optimising predecessor data structures for internal memory. In *Proc. 5th International Workshop on Algorithm Engineering*, volume 2141 of *LNCS*, pages 67–78, 2001.

[619] N. Rahman and R. Raman. Analysing the cache behaviour of non-uniform distribution sorting algorithms. In *Proc. 8th Annual European Symposium on Algorithms*, volume 1879 of *LNCS*, pages 380–391, 2000.

[620] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. *ACM Journal of Experimental Algorithmics*, 6, 2001. Preliminary version in *Proc. 2nd Workshop on Algorithm Engineering and Experiments*, 2000.

[621] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *ACM Journal of Experimental Algorithmics*, 5, 2001.

[622] V. Ramachandran. Parallel algorithm design with coarse-grained synchronization. In Alexandrov et al. [28], pages 619–627.

[623] M. V. Ramakrishna and W. R. Tout. Dynamic external hashing with guaranteed single access retrieval. In *Foundations of Data Organization and Al-*

gorithms: 3rd International Conference (FODO '89), volume 367 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 1989.

[624] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching (extended abstract). In *Proceedings of the Thirteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 25–35, 1994.

[625] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *Int'l. Symposium on Computer Architecture*, pages 155–164. IEEE, 2001.

[626] A. Ramirez, J. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Optimization of instruction fetch for decision support workloads. In *Proceedings of the Int'l. Conference on Parallel Processing*, pages 238–245, 1999.

[627] A. Ramirez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proceedings of the Int'l. Conference on Supercomputing*, pages 119–126. ACM, 1999.

[628] A. Rapoport. *Two-Person Game Theory.* Dover, 1966.

[629] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods (FM)*, pages 195–211, 1999.

[630] J. H. Reif and S. Sen. Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems. *SIAM Journal on Computing*, 21(2):466–485, June 1992. Erratum (missing part of Appendix A): *SIAM Journal on Computing* 23(2):447–448(1994).

[631] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[632] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[633] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. of the ACM/IEEE Supercomputing Conference*, Dallas, Texas, USA, 2000.

[634] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 134–142. ACM, 1990.

[635] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 176–187, 1995.

[636] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems & Applications*, 4(3):34–43, 1995.

[637] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[638] K. Ross and J. Rao. Making B+-trees cache conscious in main memory. In *Proceedings of the SIGMOD Int'l. Conference on the Management of Data*, pages 475–486. ACM, 2000.

[639] D. Rotem. Spatial join indices. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 500–509, 1991.

[640] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24.2 of *SIGMOD Record*, pages 71–79, June 1995.

[641] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In G. Ganger, editor, *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 188–201. ACM Press, 2001.

[642] U. Rüde. Fully adaptive multigrid methods. *SIAM Journal on Numerical Analysis*, 30(1):230–248, 1993.

[643] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, Mar. 1994.

[644] T. A. Runkler. *Information Mining (German).* Vierweg, 2000.

[645] S. Russell. Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI)*, pages 1–5. Wiley, 1992.

[646] G. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the 8th Int'l. Conference on Very Large Databases*, pages 257–262, 1982.

[647] J.-R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry.* Elsevier, Amsterdam, 2000.

[648] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In D. T. Lee and S.-H. Teng, editors, *Proceedings of the 11th Annual International Symposium on Algorithms And Computation*, volume 1969 of *LNCS*, pages 410–421. Springer, 2000.

[649] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual Symposium on Discrete Algorithms*, pages 225–232. ACM–SIAM, 2002.

[650] K. Salem and H. Garcia-Molina. Disk Striping. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 336–342. ACM Press, 1986.

[651] A. Samuel. Some studies in machine learning using the game of checkers-recent progress. *IBM Journal of Research and Development*, pages 210–229, 1959.

[652] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX Conference*, 1985.

[653] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67:305–309, 1998.

[654] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[655] P. Sanders. Reconciling simplicity and realism in parallel disk models. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms*, pages 67–76. ACM Press, 2001.

[656] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the Eleventh Annual Symposium on Discrete Algorithms*, pages 849–858. ACM Press, 2000.

[657] P. Sanders and R. Solis-Oba. How helpers hasten h-relations. In *8th European Symposium on Algorithms, LNCS 1879*, pages 392–402, 2000.

[658] J. Santos and R. Muntz. Using heterogeneous disks on a multimedia storage system with random data allocation. Technical Report 980011, University of California, Los Angeles, Computer Science Department, 1998.

[659] J. R. Santos and R. R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configuration. In *Proceedings of the*

Sixth International Conference on Multimedia, pages 303–308. ACM Press, 1998.

[660] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the international conference on Measurements and modeling of computer systems*, pages 44–55. ACM Press, 2000.

[661] N. I. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[662] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of the 1st European Workshop on OpenMP (EWOMP)*, pages 32–39. Lund University, Lund, Sweden, 1999. Electronic proceedings, `http://www.it.lth.se/ewomp99/`.

[663] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of 8th ACM Symposium on Operating System Principles*, pages 96–108, 1981.

[664] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281, August 1995.

[665] J. Schaeffer. Conspiracy numbers. *Advances in Computer Chess 5*, pages 199–217, 1989.

[666] J. Schaeffer. The games computer (and people) play. *Advances in Computers 50*, pages 189–266, 2000.

[667] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2–3):273–290, 1991.

[668] H.-J. Schek, R. Weber, M. Mlivoncic, D. Paschoud, J. Bosshard, and R. Grob. CHARIOT — content-based image retrieval. Accessible via URL `http://simulant.ethz.ch/Chariot/` (accessed 11 Jun. 2002).

[669] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000. SIGPLAN Notices 35(11).

[670] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of 1st USENIX Confererence on File and Storage Technologies*, pages 231–244, 2002.

[671] R. Schneider and H.-P. Kriegel. The TR*-tree: A new representation of polygonal objects supporting spatial queries and operations. In H. Bieri and H. Noltemeier, editors, *Computational Geometry - Methods, Algorithms and Applications. International Workshop on Computational Geometry CG '91, Proceedings*, volume 553 of *Lecture Notes in Computer Science*, pages 249–263, Berlin, 1991. Springer.

[672] P. D. A. Schofield. Complete solution of the eight puzzle. In *Machine Intelligence 2*, pages 125–133. Elsevier, 1967.

[673] S. Schroedl. *Negation as Failure in Explanation-Based Generalization.* PhD thesis, Computer Science Department Freiburg, 1998. Infix, 181.

[674] E. J. Schwabe and I. M. Sutherland. Flexible usage of parity storage space in disk arrays. In *8th Annual Symposium on Parallel Algorithms and Architectures*, pages 99–108. ACM Press, 1996.

[675] E. J. Schwabe and I. M. Sutherland. Flexible usage of redundancy in disk arrays. *Theory of Computing Systems*, 32(5):561–587, 1999.

[676] E. J. Schwabe, I. M. Sutherland, and B. K. Holmer. Evaluating approximately balanced parity-declustered data layouts for disk arrays. In *Proceedings of the*

Fourth Workshop on Input/Output in Parallel and Distributed Systems, pages 41–54. ACM Press, 1996.

[677] M. Schwartz and T. E. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications*, pages 539–552, 1980.

[678] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[679] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991.

[680] R. Seidel and U. Adamy. On the exact worst case query complexity of planar point location. *Journal of Algorithms*, 37(1):189–217, October 2000.

[681] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27.2 of *SIGMOD Record*, pages 154–165, June 1998.

[682] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. In *Proc. of the Int. Conference on Computational Science, Part I*, volume 2073 of *LNCS*, pages 107–116, San Francisco, California, USA, 2001. Springer.

[683] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the thirteenth International Conference on Very Large Data Bases*, pages 507–518, 1987.

[684] A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 297–302, 1989.

[685] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *11th ACM Symposium of Discrete Algorithms*, pages 829–838, 2000.

[686] SGI Corporation, http://oss.sgi.com/projects/xfs/. *XFS: A high-performance journaling file system.*

[687] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th Int'l. Conference on Very Large Databases*, pages 510–521, 1994.

[688] J. F. Sibeyn. Routing with finite speeds of memory and network. In *Proc. 22nd Symposium on the Mathematical Foundations of Computer Science, LNCS 1295*, pages 488–497. Springer-Verlag, 1997.

[689] J. F. Sibeyn. Faster gossiping on butterflies. In *Proc. 28th International Colloquium on Automata, Languages and Programming, LNCS 2076*, pages 785–796. Springer-Verlag, 2001.

[690] J. F. Sibeyn. One-by-one cleaning for practical list ranking. *Algorithmica*, 32:342–363, 2002.

[691] J. F. Sibeyn, F. Guillaume, and G. Seidel. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56:156–180, 1999.

[692] J. F. Sibeyn and M. Kaufmann. BSP-like external memory computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *LNCS*, pages 229–240, 1997.

[693] D. P. Siewiorek and R. W. Swarz. *Reliable Computer Systems: Design and Evaluation.* Digital Press, Bedford, Massachusetts, USA, 1992.

[694] A. Silberschatz, H. Korth, and S. Sudarsham. *Database System Concepts.* McGraw-Hill Higher Education, fourth edition, 2001.

[695] H. Simitici and D. A. Reed. A comparison of logical and physical parallel I/O patterns. *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications)*, 12(3):364–380, 1998.

[696] H. A. Simon and A. Newell. Heuristic problem solving: The next advance in operation research. *Operations Research*, 6, 1958.

[697] D. J. Slate. Interior-node score bounds in a brute-force chess programm. *ICCA Journal*, 7(4):184–192, 1984.

[698] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[699] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[700] M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 20, pages 877–935. Elsevier, Amsterdam, 2000.

[701] E. Smirni and D. A. Reed. Workload characterization of I/O intensive parallel applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 169–180, 1997.

[702] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[703] A. Sohn and Y. Kodama. Load balanced parallel radix sort. In *Proceedings of the 12th international conference on Supercomputing*, pages 305–312. ACM Press, 1998.

[704] J.-W. Song, K.-Y. Whang, Y.-K. Lee, M.-J. Lee, and S.-W. Kim. Spatial join processing using corner transformation. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):688–695, July/August 1999.

[705] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, Georgia, USA, 1999.

[706] K. Stoffel, M. Taylor, and J. Hendler. Integrating knowledge and data-base technologies. Technical report, University of Maryland, 1996.

[707] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[708] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[709] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.

[710] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proc. of the ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 24–35, Santa Clara, California, USA, 1993.

[711] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.

[712] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. In *Proceedings of the International Parallel and Distributed Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 1178–1192. IEEE, 1999.

[713] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[714] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 4(14):862–874, 1985. ULTRACOMPUTER NOTE #51, TR #69 May 1983, Courant Institute of Mathematical Sciences, Computer Science Dept., New York University.

[715] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.

[716] L. Technologies. High density holographic data storage. `http://www.bell-labs.com/org/physicalsciences/projects/hdhds/1.html`, 2000.

[717] O. Temam. Investigating optimal local memory performance. In *Proc. ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 218–227, San Diego, California, USA, 1998.

[718] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

[719] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proc. of the ACM/IEEE Supercomputing Conference*, Portland, Oregon, USA, 1993.

[720] G. Tesauro. Temporal difference learning and TD-Gammon. *Communication of the ACM*, 38(3):58–68, 1995.

[721] R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *Proceedings of 8th International Parallel Processing Symposium Workshop on Input/Output in Parallel Computer Systems*, 1994.

[722] R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proc. of SC98: High Performance Networking and Computing.* IEEE, Nov. 1998.

[723] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM, May 1999.

[724] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[725] F. A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID: A disk array management system for video files. In *Proceedings of the First ACM International Conference on Multimedia*, pages 393–400. ACM Press, 1993.

[726] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[727] J. Torrellas, M. Lam, and J. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proc. of the Int. Conference on Parallel Processing*, volume 2, pages 266–270, Pennsylvania, USA, 1990.

[728] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[729] E. Torrie, C.-W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 204–213, Limassol, Cyprus, 1995.

[730] P. Trancoso, J. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of dss commercial workloads in shared memory multiprocessors. In *Proceedings of the Int'l. Conference on High-Performance Computer Architecture*, pages 250–260. IEEE, 1997.

[731] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid.* Academic Press, 2001.

[732] C.-W. Tseng. Communication analysis for shared and distributed memory machines. In *Proc. of the Workshop on Compiler Optimizations on Distributed Memory Systems*, 1995.

[733] A. Tung, Y. Tay, and H. Lu. Broom: Buffer replacement using online optimization by mining. In *CIKM*, pages 185–192, 1998.

[734] A. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[735] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[736] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[737] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellligence*, 3:331–360, 1991.

[738] UltraSPARC. *User's Manual.* Sun Microsystems, 1997.

[739] U. Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, NJ, 1996.

[740] J. Vahrenhold and K. H. Hinrichs. Planar point location for large data sets: To seek or not to seek. *ACM Journal of Experimental Algorithmics*, 7(Article 8), 2002. 21 Pages.

[741] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[742] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[743] H. J. van den Herik. Strategy in chess endgames. *Computer Game-Playing: Theory and Practice*, pages 87–105, 1983.

[744] A. J. van der Steen and J. Dongarra. Overview of recent supercomputers. `http://www.top500.org/ORSC/2002/`, July 2002.

[745] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS '75)*, pages 75–84. IEEE Comput. Soc. Press, 1975.

[746] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[747] S. P. Vanderwiel and D. J. Lilja. Data prefetching mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[748] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.

[749] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[750] D. E. Vengroff and J. S. Vitter. Efficient 3-D range searching in external memory. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 192–201, 1996.

[751] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. In *Proc. Scandinavian Workshop on Algorithm Theory*, volume 1851 of *LNCS*, pages 419–432, 2000.

[752] H. M. Vin, S. S. Rao, and P. Goyal. Optimizing the placement of multimedia objects on disk arrays. In *International Conference on Multimedia Computing and Systems*, pages 158–166. IEEE Computer Society, 1995.

[753] J. S. Vitter. Online data structures in external memory. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 1999.

[754] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[755] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August/September 1994.

[756] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 131–141. IEEE Comput. Soc. Press, 1999.

[757] G. I. Webb. Efficient search for association rules. In *Knowledge Discovery and Data Mining*, pages 99–107, 2000.

[758] R. Weber and S. Blott. An approximation-based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no. 9141), October

1997. Also accessible via URL `http://mercator.inf.ethz.ch/paper/HTR24.pdf` (accessed 11 Jun. 2002).

[759] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*, pages 194–205, 1998.

[760] G. Weikum. Pros and cons of operating system transactions for database systems. In *Proceedings of 1986 Fall Joint Computer Conference*, pages 1219–1225, 1986.

[761] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[762] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids.* PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, 2001.

[763] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proc. of the ACM/IEEE Supercomputing Conference*, Portland, Oregon, USA, 1999.

[764] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of the ACM/IEEE Supercomputing Conference*, Orlando, Florida, USA, 1998.

[765] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[766] W. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[767] D. S. Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, volume 1900 of *LNCS*, pages 774–784, August 2000.

[768] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, 1991.

[769] M. J. Wolfe. *High-Perfomance Compilers for Parallel Computing.* Addison–Wesley, Redwood City, California, USA, 1996.

[770] S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[771] L. Xiao, X. Zhang, and S. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal on Experimental Algorithmics*, 5, 2000.

[772] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. Hallaron. Space- and time-efficient BDD construction via working set control. In *Asia and South Pacific Design Automation*, pages 423–432, 1998.

[773] Q. Yi, V. Advi, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–181, Vancouver, Canada, June 2000. ACM.

[774] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *27th International Conference on Very Large Data Bases*, pages 421–430, 2001.

[775] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, School of Computer Science, Carleton University, 2002.

[776] H. Zhang and M. Martonosi. A mathematical cache miss analysis for pointer data structures. In *Proc. 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[777] W. Zhang. Depth-first branch-and-bound versus local search. In *National Conference on Artificial Intelligence (AAAI)*, pages 930–935, 2000.

[778] W. Zhang and P. Larson. A memory-adaptive sort (masort) for database systems. In *Proceedings of the CASCON Conference*. IBM, 1996.

[779] Z. Zhang and X. Zhang. Cache-optimal methods for bit-reversals. In *Proc. Supercomputing'99*, 1999.

[780] V. V. Zhirnov and D. J. C. Herr. New frontiers: Self-assembly and nanoelectronics. *IEEE Computer*, 34(1):34–43, 2001.

[781] G. M. Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics.* Springer, New York, second edition, 1998.

[782] G. Zimbrão and J. M. de Souza. A raster approximation for the processing of spatial joins. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*, pages 558–569, 1998.

[783] R. Zimmermann and S. Ghandeharizadeh. HERA: Heterogeneous extension of raid. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 4, pages 2159–2165. CSREA Press, 2000.

Index