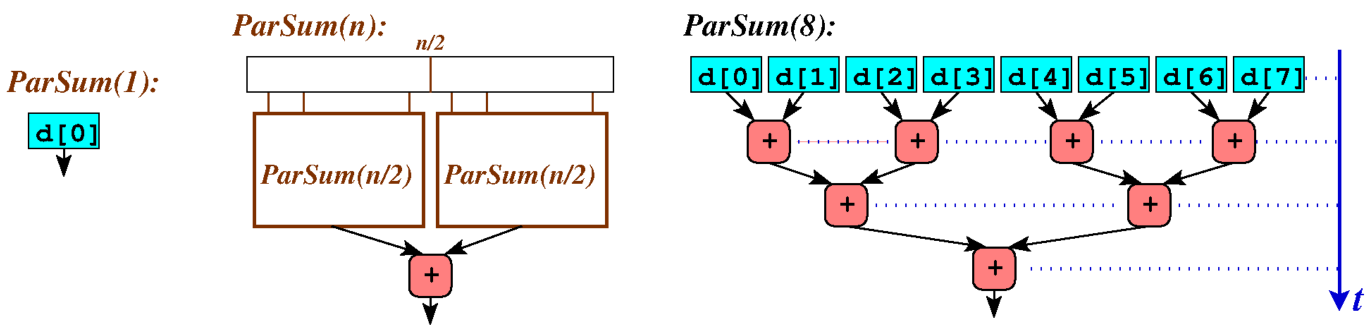
Design and Analysis of Parallel Algorithms

An Introduction

Summary of the Theory Part in TDDC78 and TDDD56



Christoph Kessler

Linköping University
Sweden

Spring 2020 — new Chapter 4 (May 2020)

Preface

This material is ©copyrighted by the author and intended for the internal use in my courses on parallel programming (TDDC78, TDDD56, ...) only. Please use it for your own studies only and do not share it on public archives.

The available chapter summarizes the design and analysis of parallel algorithms as presented in the lectures, and can be used as reading material together with the slides. We occasionally add additional material and background information as appropriate. Sections marked with an asterisk (*) include advanced material beyond the course contents.

Not every chapter/section is mandatory for every course. The following table gives some general directions at chapter level (Mandatory = lectured and exam-relevant; Optional = useful as further reading; Background = prerequisite knowledge, included for repetition). Exceptions from these chapter-level directions for specific sections will be given as footnotes in the text.

Chapter	TDDC78	TDDD56	TDDE31/ 732A54	TDDE35
1	Mandatory	Optional	Optional	Optional
2	Mandatory	Mandatory	Optional	Mandatory
3	Optional	Mandatory	—	Optional
4	Mandatory	Optional	—	Optional
A	Background	Background	—	Background

All feedback on the current version of this document is very welcome!

Linköping, 2020

Christoph Kessler

Contents

Chapter 1

Design of Parallel Programs

This chapter introduces parallel programming models in general and then describes the PCAM-method by Foster [26] for the design and engineering of parallel programs, using a generic task based programming model.

1.1 Parallel Programming Models

Parallel programming models are a way to abstract from hardware (e.g., processor types, interconnection networks) and technology (e.g., operating system, microarchitecture, instruction set and API syntax) and instead focus on the essence of programming for a whole class of parallel computer systems.

A *parallel programming model* specifies a set of basic operations (e.g., statements and expressions, memory accesses, or message passing constructs) together with the conditions under which they are applicable, how data can be stored, accessed and communicated, and how parallel activities can be synchronized (e.g., whether there is any "natural" synchronization (such as blocking receive operations or lock-step synchronous execution of shared memory accesses) provided to the programmer or not).

Hence, the programming model focuses on the qualitative aspect of programming (*what* can be expressed). Its foremost purpose is the portability of algorithmic ideas (pseudocode) across a larger family of parallel computer systems that are based on the same programming model but possibly differ in minor or parameterizable details (such as processor type, number of cores etc.). Programming models will later be complemented with a *cost model* (see Chapter 2) that associates a cost function with each primitive operation of the programming model.

Programming models can be *implemented* in various ways, typically in the form of programming languages, runtime systems/libraries, or combinations of these.

1.2 A Generic Task-Based Parallel Programming Model

A parallel computation consists of *tasks*, which execute *concurrently*. A task encapsulates a (part of a) *sequential program* and *local memory* that is only accessible to the task itself, not to other tasks.

Tasks are an abstract software concept that could be implemented in different ways. Tasks can be *mapped* to physical processors (i.e., assigned for execution) either by mapping them one-to-one by processes or threads and delegating the mapping to the operating system's CPU scheduler, or by a *task-based runtime system* that schedules at runtime multiple tasks in a time-sharing way over each worker-thread or -process; this many-to-one scenario is more flexible than the one-to-one approach as it allows programs to create many more tasks than there are processes or threads permitted in the target system. When designing parallel algorithms in general, we do not want to early constrain the degree of parallelism artificially, and thus do not prescribe any particular mapping of tasks to processes, threads or processors, but treat all tasks as if they each had their own dedicated execution resource so they could proceed in parallel (as far as synchronization constraints permit, which we discuss later). It is still possible to coarsen the task granularity later by merging several tasks into one by serializing their computations.

In addition to executing local operations from its program and accessing task-local memory, a task can at any time perform four special actions:

- send a message
- receive a message
- create a new task
- terminate.

This list describes a minimal task programming API; of course it can be extended in various ways, for instance with specialized variants of the send and receive operations.

Tasks already define a notion of *data locality*: Data items in a task's local memory can be considered "close" because they can be accessed through ordinary memory accesses (load, store), while other tasks' data can be considered "remote" because they are only accessible via interaction with these tasks (by sending or receiving messages to them).

Moreover, this task concept enforces *encapsulation* of "taskified" computations—all dependences on the context outside the task's data environment (i.e., task-local memory) have to be made explicit because the only way to access non-local data is via explicit message passing.

As long as tasks are only abstract units of work, we need not be specific about their execution and scheduling model yet. But when discussing the implementation of a task-based program, we also need to consider the execution model of the tasks, which is implied e.g. by the runtime system or operating system used on the parallel target machine. In the simplest case, tasks are "one-shot tasks" that are executed non-preemptively. *Non-preemptive execution* means that any task, once it started executing, will execute to completion and cannot be interrupted to switch context to any other task on the same processor. This imposes strict limitations on synchronization and communication, as the only point where a task could wait for others will be at its beginning. However, this model is appropriate e.g. for tasks that may run on accelerators that lack interrupt hardware and system software needed for preemption of processors. A more flexible execution model is *preemptive execution*, where a task can be interrupted, and the context of the executing processor temporarily switched to another task. Depending on the execution environment, this can either happen only at the synchronization points of the task (e.g., where waiting for an incoming message to arrive) or at any point during its execution. If preemptive execution is required, tasks become similar to threads or processes managed by an operating system.

1.2.1 Inter-Task Communication Models

Tasks can communicate by sending and receiving messages to each other.

Usually, a send operation is *non-blocking*, i.e., the sender needs not wait inside the send routine for the receiver to become ready to receive, and just fires off the message (possibly to a separate thread taking care of the communication itself) and continues with its next program statement.

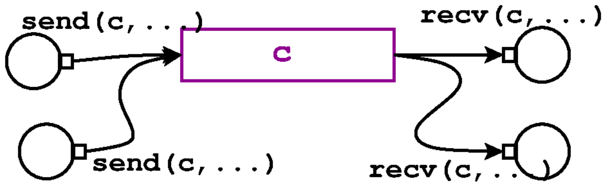


Figure 1.1: Channel-based communication between tasks.

This allows to overlap communication work and communication latency with subsequent work on the sender side.[1]

In contrast, the receive operation is often *blocking*, i.e., the receiver should wait for the message to arrive because the subsequent computations usually depend on the data being received.

It is mostly an API design issue how tasks can reference their communication partner in calls to send or receive operations. There are two main styles for that: channel-based and direct communication.

1.2.1.1 Communication via Named Channels and Ports

Here, messages are sent via named and FIFO-buffered *channels*, i.e., message queues. Senders and receivers are connected by referencing or subscribing to named channels (see Figure 1.1).

Tasks with channel-based communication are appropriate for modeling stream processing applications, where tasks are statically connected by steady streams of communication between data producer tasks and their corresponding data consumer tasks. The FIFO (first in, first out) buffering feature of streams is important for correct execution. In particular, it also allows pipelining of computations, i.e., the execution of data-producing and data-consuming tasks can overlap in time as long as they work on different data elements.

A widely used theoretical programming model for stream-based computation is *Kahn Process Networks* (KPN) [35]. A KPN is a directed acyclic graph of periodic tasks (i.e., the graph nodes) that communicate data between producer and consumer tasks via FIFO-buffered channels (i.e., the graph edges) with unlimited buffering capacity. One execution of a task is triggered at runtime by data arriving along all ingoing channels. Further such data-flow based programming models have been defined in the literature, often with stronger assumptions (such as Synchronous Data Flow, SDF) which then often can give stronger guarantees in analysis and implementation, e.g., by tight worst-case bounds on channel buffer sizes.

Channel-based message passing is less typical for high-performance computing than direct message passing. In contrast, it is more common in the domains of signal processing and big-data stream processing. Examples of channel-based programming models for tasks include Occam, Fortran M, TCP socket communication, and stream programming languages such as StreamIt, FastFlow, Apache Flink or Spark-Streaming.

1.2.1.2 Communication to Named Tasks

In HPC programming models, a more common scenario is that each task (as well as the concrete entity it is mapped to, such as thread, process or processor) is identified by a unique name, often

[1]It is however only possible if the message passing system has sufficient internal buffering capacity for all channels and messages. Otherwise, communication must happen in the form of a *rendevous*, i.e., the send operation and the corresponding receive operation must overlap in time so that data can be directly moved from sender memory to receiver memory without buffering. See also Chapter **??** on MPI.

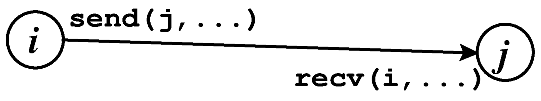


Figure 1.2: Message passing between directly named communicating tasks i and j.

an integer number. Hence, communication operations have an API such as "send message X to task 17" instead of "send message X on channel *ch*".

Naming tasks by unique integer numbers also makes it easier to reference and communicate with tasks that are created dynamically.[2]

A well-known HPC programming model that uses direct named addressing of processes in communication operations is the Message Passing Interface, MPI.

In principle, both channel-based and direct-named addressing of communication are equivalent. By defining a separate channel for each pair of existing task names, direct addressing can be emulated with a channel-based API. Likewise, channels can be implemented as a separate software layer atop a communication API that uses direct task addressing.

1.2.1.3 Shared Memory Programming

In shared memory programming, tasks need not send messages but share a common address space so that every task can read (global load) or write to (global store). Hence, all "communication" between tasks then can take place via shared data structures.

A global address space is convenient for the programmer because he/she needs not worry about distribution of data structures, i.e., which elements are local and which are nonlocal, and having to access the latter ones by message passing. With shared memory, there is no need for data distribution and no notion of "data ownership".

Because load and store instructions on shared memory are usually asynchronous, *conflicts* can occur if multiple tasks (may) concurrently access the same memory location and at least one of these accesses is a write access (a store instruction). In the case of conflicts, different relative orders of reads and writes can lead to different results of a computation. As the relative order of concurrent memory accesses even within the same program depends on many factors that are outside the program's control (such as relative speeds of processing units, the CPU scheduler behavior, interrupts due to external I/O events etc.), such conflicts must be avoided by the program enforcing a fixed execution order of its tasks with respect to these memory accesses, i.e., by making some tasks wait for others before they are allowed to proceed. For example, it is often necessary to synchronize producers and consumers of the (possibly) same data element(s) in shared memory. For synchronization, we can use techniques such as software-based or hardware supported mutual exclusion locks, semaphores, monitors, or non-blocking synchronization.

Shared memory parallel programming models that are commonly used in HPC are POSIX threads (pthreads) and OpenMP.

In general, a shared memory programming model makes parallel program development easier because the aspect of explicit data distribution can be ignored. In particular, there is no need to program for the explicit communication of data from data producers to data consumers.

[2]For instance, UNIX-based operating systems such as Linux maintain at any time a tree of processes, with links between parent processes and child processes they created. Creating a new process (via the operating system) increments a central counter to obtain the new process ID, such that a child process ID is always larger than its parent's ID. Knowing the process ID, messages and signals can be directed to the process.

On the downside, shared memory parallel programs can be hard to understand, and it can be hard to manage locality on systems where locality matters for efficiency reasons, such as NUMA (non-uniform memory access time) based shared memory systems.

1.2.2 Data-Parallel Programming

In data-parallel programming, parallelism stems from applying the *same operation* to *multiple elements* of the same data structure, such as "add 2 to every element of this array".

Depending on the size of the underlying data structure, data parallelism allows to extract massive parallelism for large data sets. It is especially suited for matrix computations (every matrix element update can become a task of its own) and image processing (every pixel computation can become a task of its own), for example. As each operation on a single data element now could define a task of its own, data parallelism can lead to extremely fine granularity of parallelism. Some real-world data-parallel programming models that are designed for fine-grained data parallelism, such as CUDA for general-purpose GPUs, indeed operate by default on individual data elements, although it is also possible to define one task as applying the operation to a group of data elements (e.g., in a sequential loop) rather than having singleton element tasks.

Data parallelism is one of the most portable types of parallelism, because it could be mapped efficiently to both SIMD and MIMD architectures, and to both shared and distributed memory systems.

Another big advantage of data-parallel computing comes from its restriction to a single thread of control flow. In other words, all processors executing a data-parallel program will always be operating at the same point in the program, namely, within the current data-parallel construct, whose work they jointly execute. This makes the code easier to understand and to debug, compared to unstructured MIMD code where different threads can, in principle, execute anywhere in the program code at the same time.

Even if data-parallel operations assume a shared address space (which might be given by the aggregate data structure they are applied on, such as an array), they can often be efficiently used also in a distributed memory scenario. For the latter scenario, the required communication operations can often be figured out automatically by the compiler and need not be written manually as send and receive calls any more, hence data-parallelism can make programming easier compared to general MIMD message passing programming. However, in the latter case, the most suitable partitioning and distribution of elements over local memories is often not clear, yet it has large impact on the resulting communication and thereby the overall performance. For that reason, programming environments targeting distributed memory systems usually leave it to the programmer to specify data locality constraints, or respectively, how data is to be partitioned and mapped.

Examples of HPC-relevant data-parallel programming models include CUDA (within the GPU kernels), data-parallel loops and array notation that are part of the Fortran standard since 1990/95, and the former *High-Performance Fortran (HPF)* targeting distributed-memory systems, but also data-parallel skeleton programming frameworks such as *SkePU*.

1.3 Foster's Methodology for Parallel Program Design

In this section, we will explain a systematic method for the design of parallel programs for computational problems, which was introduced by Foster [26]. An overview is given in Figure 1.3. The method is also called "PCAM"-method, given by the initial letters of the four main steps: (1) Partitioning, (2) Communication and synchronization, (3) Agglomeration, and (4) Mapping and scheduling.

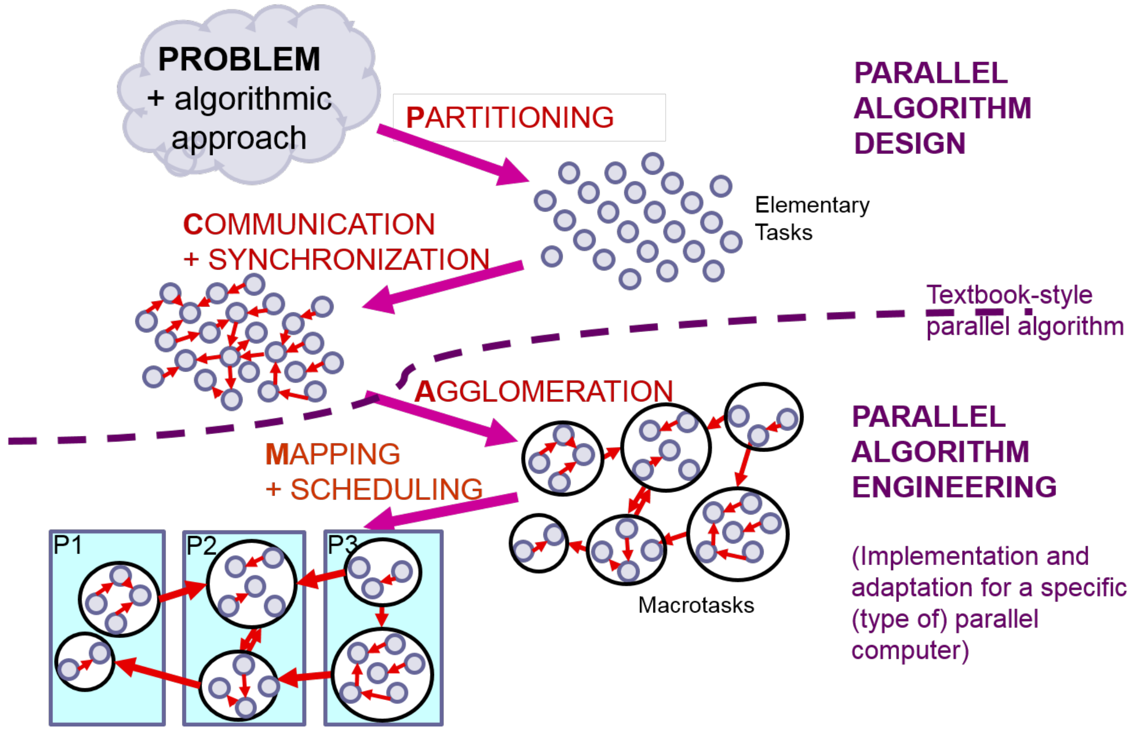

PROBLEM
+ algorithmic approach
PARTITIONING
PARALLEL ALGORITHM DESIGN
Elementary Tasks
COMMUNICATION + SYNCHRONIZATION
Textbook-style parallel algorithm
AGGLOMERATION
PARALLEL ALGORITHM ENGINEERING
MAPPING + SCHEDULING
P1
P2
P3
Macrotasks
(Implementation and adaptation for a specific (type of) parallel computer)

Figure 1.3: The PCAM Methodology for Parallel Program Design, adapted from Foster [26].

The method starts from an original non-parallel formulation of the given computational problem and an idea for a (hopefully, parallelizable) algorithmic solution. For example, the problem could be matrix-matrix multiplication $C = A \times B$ of $n \times n$ matrices A, B, C, and the algorithmic approach could be given by the standard textbook formula

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j} \quad \forall\ i,\ j = 0, ..., n-1.$$

The main four steps work as follows (details and further examples will be given in separate subsections below):

1. **Partitioning:**

   The first step, partitioning, identifies the main units of work in the algorithmic solution that could define *tasks* for a parallel formulation. The task granularity is a design aspect. For example, in the matrix-matrix multiply scenario, we could define each single memory access, each scalar multiplication and each scalar addition as a tiny task of its own. In practice it is more convenient to define larger units of work as tasks, where the work within a task is defined as a sequential subcomputation. For instance, the dot product for every pair (i, j) could define a task, resulting in n^2 tasks in total each performing $O(n)$ work in a sequential loop.

2. **Communication and synchronization:**

   In the second step, we identify dependences between the tasks, which might require synchronization and/or communication for correct execution, and for which we select suitable mechanisms and patterns.

In our matrix-matrix multiply example's variant with $O(n^3)$ single-operation tasks, there are obvious dependences of each individual multiplication task from the tasks loading its operands, and the summations to each dot product (i, j) determining $c_{i,j}$ depend on its partial products being calculated. In the variant starting from dot-product tasks, there are no dependences between the n^2 dot product computations, i.e., the different $c_{i,j}$ can be calculated in arbitrary order or in parallel.

The outcome of the first two steps is a textbook-style, very fine-grained parallel algorithm that could execute on an abstract parallel computer[3] that provides one processor (or at least one thread/process) for each task. Hence, all tasks could, in principle, execute concurrently, but conflicting tasks will still be executed mutually exclusively and dependent tasks will still be properly sequenced by the chosen communication/synchronization mechanisms.

This textbook-style, fine-grained parallel algorithm design is reusable in the sense that it can be used as a starting point for the more target-system specific steps that follow, and hence be reused when migrating to a different parallel execution platform.

3. **Agglomeration:**

   The agglomeration step tries to reduce the amount of residual inter-task communication by coarsening the granularity. It merges multiple tasks into *macro-tasks*, where the constituent tasks' computations within a macrotask are sequentialized. Each macrotask will be eventually be executed by one thread or process, i.e., on the same processor/core. Hence, synchronizations between tasks now belonging to the same macrotask will disappear due to their sequentialization. Likewise, inter-task communications within the same macrotask can now be resolved by fast local memory accesses, while communication between tasks belonging to different macrotasks will still result in inter-thread or inter-process communication, and thereby come at a certain cost. More aggressive agglomeration thus reduces the residual synchronization/communication more, but also reduces flexibility of the final step to achieve fully parallel and overlapping execution because it exposes fewer parallel macrotasks.

   In our matrix-matrix multiply example, one could, for instance, agglomerate the n^2 dot-product tasks to n matrix-vector product macrotasks, i.e., one macrotask per row of the C matrix.

4. **Mapping and scheduling:**

   Finally, the mapping step decides which macrotasks (basically, threads) will execute on which process*or* or core, where multiple macrotasks could of course time-share one processor/core. In the latter case, scheduling can determine in which order the macrotasks mapped to the same processor/core will be executed.

   In the matrix-matrix multiply example, we could map the n macrotasks remaining after agglomeration to p processors by assigning n/p matrix-vector product macrotasks to each processor.

After these four design steps are completed, the resulting design can be implemented in source code in some parallel programming language and/or library.

While we introduce the four steps one by one in the following subsections, there do exist dependences between them. In particular, the first two steps are, in general, interdependent and should thus be closely coupled. Likewise, agglomeration and mapping/scheduling are related as

[3]We will introduce such an abstract parallel computer model in Section 2.3.1.

they solve similar optimization problems; in fact, agglomeration could be considered a subproblem of mapping.

Design patterns are well-proven recipes for solving certain program design problems in a way that results in more flexible or extensible programs, especially in situations where a complex programming model allows for multiple possible ways of expressing a certain functionality. Well-known are, for example, the design patterns for object-oriented programming [27]. Designing parallel programs is of an at least similar degree of complexity, although here the performance aspect is at least as important as extensibility etc. While discussing the PCAM design method steps in more detail in the following sections, we will also, as a side-effect, identify a number of useful *parallel algorithmic design patterns*, which are well-proven combinations of partitioning and communication / synchronization techniques that apply to specific types of computational problems.

1.3.1 Partitioning

The partitioning step splits the computation (i.e., the work) performed by a program into pieces, i.e., tasks, which become the units of work that are considered further in the design process.

For message-passing models, not only the computation but also the data to be operated on is divided into pieces, and may be distributed over the tasks, thus creating the notion of *data ownership*.

In the partitioning step we focus on identifying opportunities for parallel execution. For now, we ignore the actual number of processors in the target computer, this will be a concern in later design steps (agglomeration and mapping).

There are two main methods for partitioning:

- **Domain decomposition**

  The data structures over which the computation is performed (often, large arrays) are partitioned into pieces. Often, but not always, these partitions are contiguous. The task partitioning is then usually implied by the data partitioning, assigning one task per partition of data to compute. One popular method is the so-called *owner-computes rule* for assignment statements, which says that the owner of the variable on the left-hand side of an assignment is in charge of executing the entire assignment statement, including evaluating its right-hand side expressions. In the matrix-matrix multiply example above, the n^2 single-element partitions of the C matrix would define the tasks, i.e., one dot product computation task for each corresponding $c_{i,j}$.

- **Functional decomposition**

  Functional decomposition splits the computation, but not the data. A common scenario is to have each function call or module execution define one task. This makes sense if the executions of at least some of these resulting tasks can be expected to overlap in time.

Often, both types of parallel decomposition can be combined in the same application. Usually module-based functional decomposition is then performed first. For modules operating on large data domains, their work can be further decomposed into smaller tasks by domain decomposition. This is often necessary because the amount of parallelism from module-based functional decomposition is typically quite limited, as it would define at most one task per module.

1.3.1.1 Partitioning by Domain Decomposition

The following computation works on a three-dimensional array (modeling a discretized three-dimensional domain, i.e., a regular 3D grid/mesh). It is a so-called *stencil computation* that updates

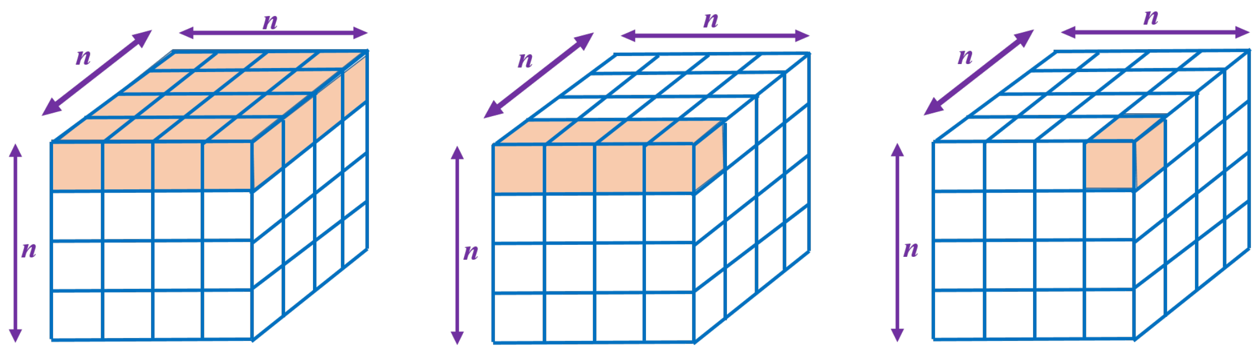


Figure 1.4: A three-dimensional domain (array) decomposed in 1, 2 or 3 dimensions. Data associated with a single task is shaded.

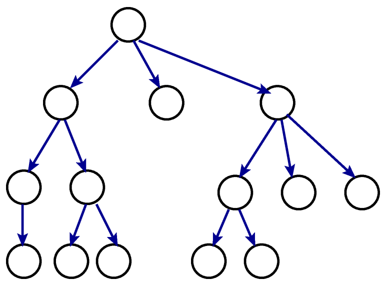


Figure 1.5: A tree of tasks constructed dynamically, starting from a single initial task (the root task). Edges show the task creation relation, i.e., connect each parent task to the child tasks it created.

an element as a function over its old value and the values of its nearest neighbor elements in the three dimensions:

```
for all i, j, k from 1 to n
   A[i][j][k] = f ( A[i-1][j][k], A[i][j-1][k], A[i][j][k-1], A[i][j][k],
                    A[i+1][j][k], A[i][j+1][k], A[i][j][k+1] );
```

More complex stencils covering a larger neighborhood can also occur. Typical occurrences of stencil computations are in iteratively solving space-discretized partial differential equations (PDEs) using the finite-differences method, and in signal or image convolutions where digital image filters are applied pixel-wise.

The domain of this computation may be decomposed in 1, 2, or 3 dimensions, see Figure 1.4. 3D-decomposition defines the largest number of tasks and thus offers the greatest flexibility.

1.3.1.2 Partitioning by Functional Decomposition

A common example for function call-based functional decomposition (see Figure 1.5 for illustration) is the parallel exploration of recursively defined search trees, e.g. in exhaustive design space exploration, in solving combinatorial optimization problems such as satisfiability of logical formulas by techniques such as branch-and-bound, in parallel traversal of search tree data structures or graphs, or in alpha-beta search for possible moves in game trees (N-queens, chess, Go etc.).

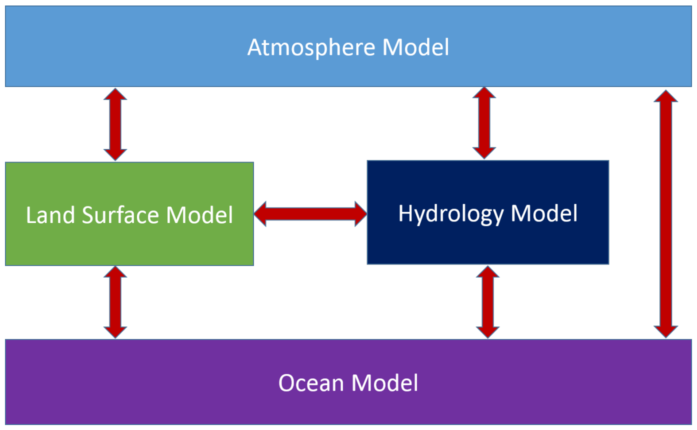


Figure 1.6: Partitioning a climate simulation application into different modules.

In such computations, a new task is created for each recursive call to the search function for the next level. A task will usually have to wait for its child tasks to complete before it can resume and complete itself. Hence, tasks located on the same path in the task creation tree can obviously not execute simultaneously. In contrast, tasks that are not located on the same path may execute in parallel.

Module-based functional decompositions actually distribute the application's code itself in the first hand, following the structuring of the overall simulation application into different submodules. This provides a coarse-grained decomposition into few tasks.

A common example for module-based functional decomposition are weather or climate simulations that use separate models and simulation methods for land areas, hydrology, oceans and atmosphere, see Figure 1.6. Each submodel simulation is defined in their own software modules, and the different concurrent submodel simulations are coupled together by data exchange to account for interaction of different submodels at model boundaries. Each submodel simulation is then typically further subdivided using domain decomposition techniques.

1.3.1.3 Summary: Partitioning Guidelines

Here we list a few rules of thumb for partitioning:

- *Partitioning should define many more tasks than there are processors.* This is because it is relatively easy to map many tasks to one processor where only a small parallel system is available, but starting out from a coarse-grain parallelized application and then having to find more parallelism later to scale up to a larger parallel system is more difficult. Also, having many more tasks than processors will give more flexibility for load balancing.

- *Partitioning should avoid redundant computation and storage* (unless necessary to gain parallelism, in some cases[4]). This is because replicating work and data storage puts the parallel implementation into disadvantage compared to a sequential solution and often results in a waste of parallel resources that are necessary to only break even with the sequential solution.[5]

[4]An example is the fully parallel mergesort algorithm (Section 3.4.3, which performs asymptotically more work than its sequential counterpart in order to make a scalable parallel algorithm possible.

[5]If there are unavoidable longer phases of little or no parallelism, the programmer should consider switching the idle processors to a low-power mode to save energy instead.

- *Partitioning should try to define tasks of similar size*, where this is possible (in particular in domain decomposition). This will alleviate load balancing, because assigning an equal number of tasks to processors then will imply an almost equal load balance, which in turn minimizes parallel execution time.

- *If possible, the number of tasks should scale with the problem size.* This will allow to use larger parallel systems on larger problems without redoing the entire design from scratch.

In many cases there will be more than one option for partitioning. The parallel program designer should then consider different alternatives for partitioning, estimate their impact on performance (with the implied communication requirements, expected load imbalance, partial work replication, memory footprint etc.) and choose the best one.

1.3.2 Communication and Synchronization

The second step in the PCAM design method considers the dependences and interactions between the tasks defined by the partitioning step, and determines appropriate communication and synchronization structures. In particular, synchronization and communication are necessary to coordinate task execution and to preserve data dependences across task boundaries.

Communication and synchronization are a special type of work that only exists in a parallel computing scenario, but not in sequential computing, because a single processor never needs to wait for itself nor send data to itself. In order to obtain an efficient parallel program for the same problem, communication and synchronization work should thus be minimized. This can often be achieved by paying special attention to data locality issues, but also by selecting cheaper communication patterns where applicable. For example, synchronous communication is not always necessary, and *asynchronous communication* often allows to overlap communication latencies with other, independent local work.

Local communication and synchronization patterns interact with nearest neighbor tasks only and are often faster and more flexible than *global communication and synchronization* where one task communicates or synchronizes with many others, or many with many.

Some problems have advanced algorithmic solutions that reduce the overall amount of arithmetic work to do, but lead to *unstructured communication and synchronization* patterns.

Another design option (if there is a choice at all) is to choose between *static vs. dynamic communication and synchronization.*

Even in the presence of strict dependences between tasks, the parallel design pattern *pipelining* can help to leverage parallelism and orchestrate overlapping execution where these dependent operations are to be performed on lots of data in sequence.

In the following, we will consider those communication and synchronization patterns in more detail and give some examples.

1.3.2.1 Local Communication

Stencil computations operate, usually in multiple iterative sweeps, on a (usually, multi-dimensional) data structure (e.g., an array) where, in each single sweep over the data structure, every element is updated as a weighted sum of the values of its adjacent (i.e., nearest neighbor) elements.

If using a domain decomposition in the partitioning step that maps the updating of a single or of a few neighbored elements to one task, this data dependence pattern implies a communication and synchronization pattern between tasks where every task only needs to interact with a few nearest neighbor tasks, i.e., a *local communication and synchronization pattern.*

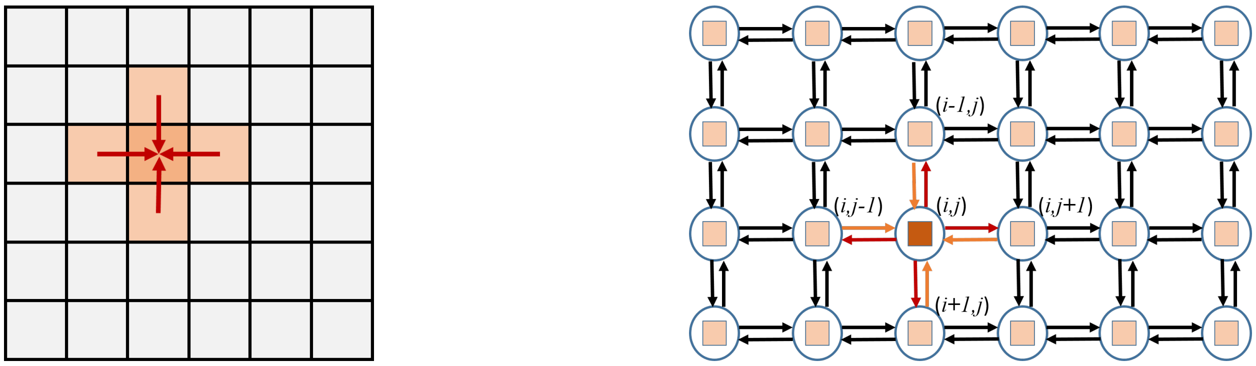


Figure 1.7: Left: A Jacobi-style five-point (2D) stencil operation on a 2D array updates *each* array element as a weighted sum of the *old* values of itself and its (up to) four direct neighbor elements (in north, east, south, and west direction). Special cases might apply for boundary elements of the 2D array. — Right: stencil update tasks after domain decomposition with one task per element to be updated. Every task communicates with (up to) 4 nearest neighbor tasks.

Figure 1.7 (left) shows the dependence pattern of a 2D finite difference computation (Jacobi) on a regular 2D domain (i.e., a 2D array). Figure 1.7 (right) shows the task-and-channel structure for this computation using domain decomposition with one task per element to be updated.

Stencil computations can be classified based on whether the stencil update operation accesses the old values of certain neighbor elements (i.e., the values computed in the previous sweep over all elements) or new values (i.e., values already updated in this same sweep) instead. These different update schemes result in different data dependence patterns and hence different kinds of precedence constraints between all the tasks. Two well-known update schemes in HPC are the *Jacobi update schema* and the *Gauss-Seidel update schema*, which are named after the corresponding iterative linear equation system solver types where these dependence patterns also occur.

1.3.2.1.1 Stencil Computation with Jacobi Update Schema The *Jacobi update schema* applies, in each sweep, for each element (i, j) the dependence pattern

$$New(i,j) \leftarrow f(\ Old(i-1,j),\ Old(i,j-1),\ Old(i,j),\ Old(i,j+1),\ Old(i+1,j)\ )$$

where *Old* holds the element values calculated in the preceding sweep, and *New* is a temporary array of the same size to which the updated element values are written in the current sweep. After the current sweep, we prepare for the next one by either copying back the element values from *New* to *Old*, or even faster, just swap the roles of (i.e., pointers to) *Old* and *New* for the next sweep.

With domain decomposition into tasks each taking care of a single element (i, j) of *Old* and *New*, we need to make sure that the four neighbor elements' values are communicated to task (i, j) prior to evaluating expression f. This results in the communication schema shown in Figure 1.8.[6]

1.3.2.1.2 Stencil Computation with Gauß-Seidel Update Schema The *Gauß–Seidel update schema* applies, in each sweep, for each element (i, j) the dependence pattern

$$New(i,j) \leftarrow f(\ New(i-1,j),\ New(i,j-1),\ Old(i,j),\ Old(i,j+1),\ Old(i+1,j)\ )$$

which, in contrast to the Jacobi update schema, partially uses new values (computed in the same

[6]Note that the pseudocode uses, for better readability, a global addressing of array elements in arrays *Old* and *New*. This differs e.g. from MPI code that uses local addressing instead.

```
function ITERATIVE_JACOBI_STENCIL ( Old[][], New[][] )
{
  while (...) do
    for all tasks $(i, j)$ do in parallel
      send element $Old(i, j)$ to each neighbor
      receive elements $Old(i-1, j)$, $Old(i, j-1)$, $Old(i, j+1)$, $Old(i+1, j)$ from neighbors
      compute $New(i, j) \leftarrow f($ $Old(i-1, j)$, $Old(i, j-1)$, $Old(i, j)$, $Old(i, j+1)$, $Old(i+1, j)$ $)$
    (then copy back $Old(i, j) \leftarrow New(i, j)$ $\forall i, j$, or swap(Old, New), to prepare for next iteration)
}
```

Figure 1.8: Iterative 2D stencil sweeps over a 2D array (*Old*) using the Jacobi update schema. Pseudocode for all tasks (i, j) when parallelized by domain decomposition at finest possible granularity.

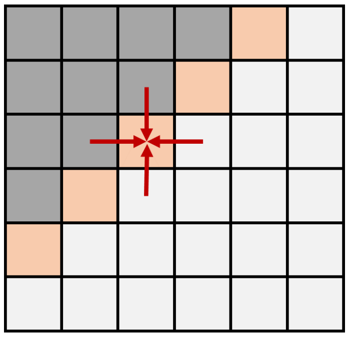


Figure 1.9: The dependence structure for element updates with the Gauß-Seidel update schema. Elements not located in the same row or same column could be updated in parallel, for instance, all elements on a diagonal.

sweep) instead of those computed in the previous sweep. This results in the dependence structure illustrated in Figure 1.9. Updates (i.e., evaluation of the stencil function f including reading *Old* and writing *New*) cannot overlap in time for elements (i, j) that are located in the same row or in the same column. However, there is still some parallelism; for instance, all elements on a diagonal could be updated in parallel without violating the precedence constraints.

Because it propagates update effects faster over the array, the Gauß-Seidel update schema generally leads to faster convergence speed than Jacobi, i.e., it needs fewer sweeps over the array. Also, the sequential computaion of Gauß-Seidel does not need a temporary array as long as processed by two properly nested forward loops that naturally preserve the element-update dependence structure.

On the other hand, the Gauss-Seidel update sweep offers a lower degree of parallelism than Jacobi: Updating element (i, j) cannot begin before the updates of elements $(i-1, j)$ and $(i, j-1$ are completed. With the Jacobi update schema, up to N^2 tasks could run in parallel for a $N \times N$ array, while with Gauß-Seidel at best all tasks on the same diagonal "wavefront", i.e., up to N tasks, are independent and could proceed in parallel, while the sequence of all $2N-1$ such diagonal wavefronts will have to be updated serially. Worse yet, the first and last diagonal in this sequence each contain only a single element to update ($(0, 0)$ and $(N-1, N-1)$ respectively), and the average number of independent updates per wavefront is only $N/2$. This imposes a serious limitation for

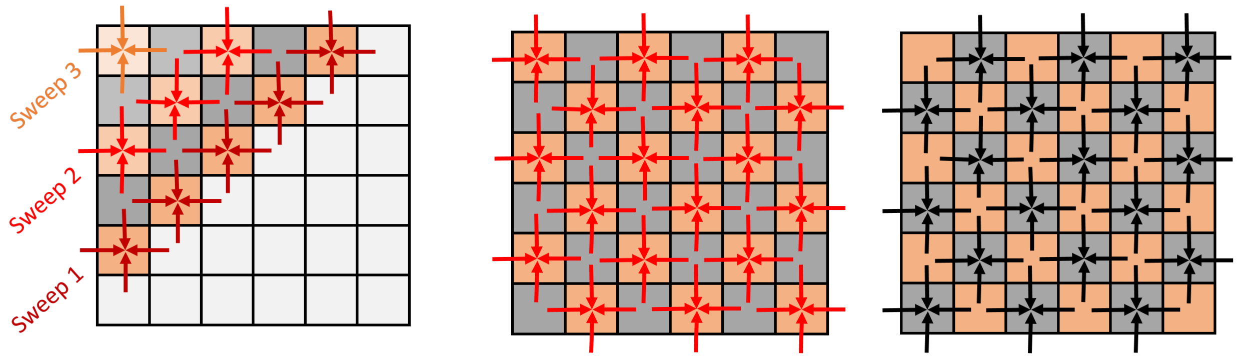


Figure 1.10: Left: Pipelined parallel execution of 3 subsequent Gauß-Seidel sweeps, starting from the top-left element $(0, 0)$. Middle and Right: Stencil computation with the red-black update schema, corresponding to the steady state of pipelining after the diagonal update wavefronts of the first N sweeps have entered the pipeline. The Red-black algorithm toggles in odd and even rounds between updating the red and the black elements respectively.

scaling up this type of computation to very large numbers of processing elements, e.g. on GPGPUs, many-core CPUs or large clusters.

1.3.2.1.3 Stencil Computation with Red-Black Update Schema

As just described, the degree of parallelism with the Gauß-Seidel update schema for a single sweep is very limited, which can be seen in Figure 1.9.

However, we can also see that, in principle, element updates from two *subsequent* sweeps (i.e., subsequent iterations of the outer while loop) could in fact overlap in time as long as they do not access the same memory locations, i.e., are at least two diagonals apart from each other. For instance, the update of element (i, j) in sweep t (writing (i, j) and reading $(i-1, j)$ and $(i, j-1)$) and the updates of $(i-1, j-1)$ or $(i, j-2)$ in sweep $t+1$ are not in conflict with each other and could proceed in parallel, as long as it is guaranteed that the values of $(i-1, j-1)$ and $(i, j-2)$ computed in sweep t have been read before when updating $(i-1, j)$ and $(i, j-1)$. A second array (New) is no longer necessary.

In other words, we can process several wavefronts from different sweeps simultaneously in a pipeline.

The resulting steady state of this pipeline is also known as *red-black update schema*. The name comes from a checkerboard-like coloring of the fields (i, j) for all $i, j \in \{0, ..., N-1\}$ using the two colors red and black so that two directly neighbored fields always have a different color (see Figure 1.10). Alternatingly, Gauss-Seidel update sweeps are executed over the red fields only and over the black fields only, i.e., only elements with the same color are updated together. As the read accesses and the write accesses always go to elements of different colors in each round, there will be no conflicts, i.e., the Gauss-Seidel dependence structure will be preserved.

The red-black update schema combines the advantages of Jacobi and Gauss-Seidel update schemas: The convergence speed and thus the overall computational work to perform is that of Gauss-Seidel, while the parallelism degree is (in the steady state) $N^2/2$, which is only 50% lower than with Jacobi updates and provides by far enough parallelism in practice.

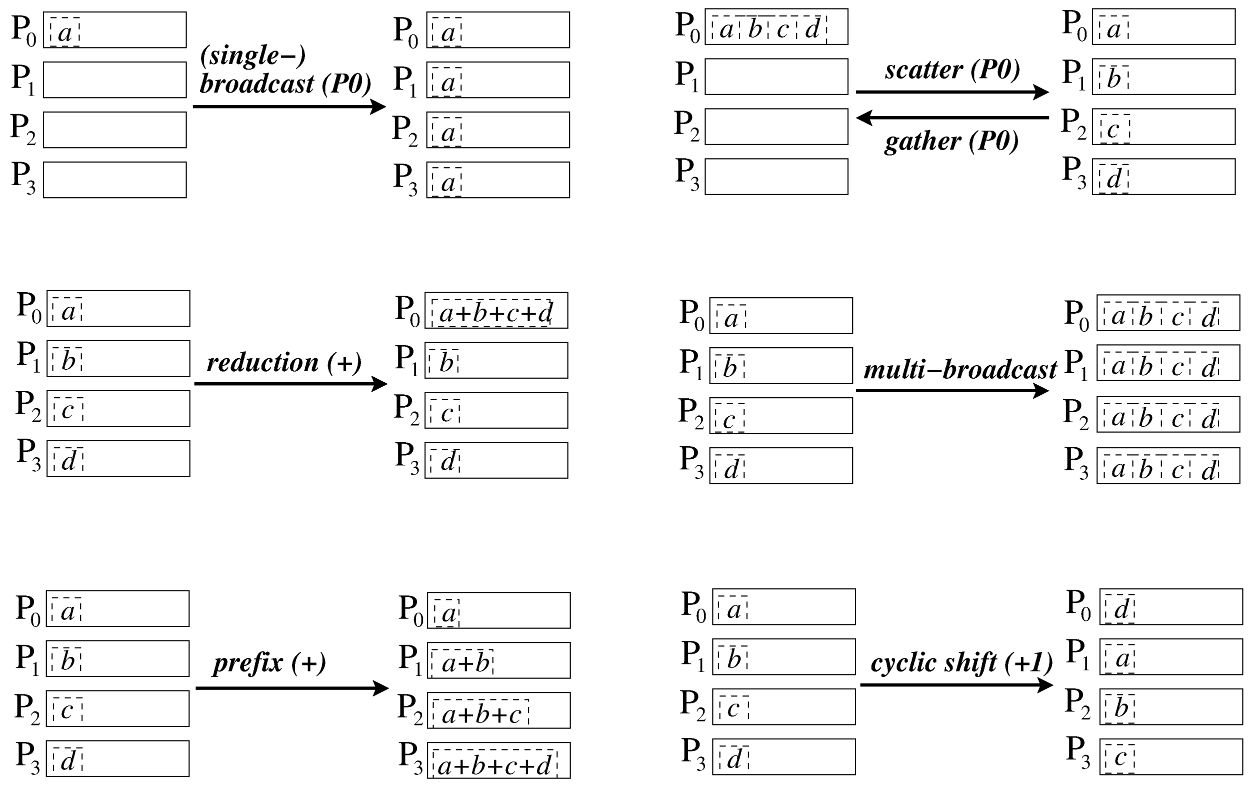


Figure 1.11: Some Common Global Communication Patterns.

1.3.2.2 Global Communication and Synchronization

In contrast to ordinary pair-wise communication between tasks using send and receive operations, *global communication operations* involve more than two communicating tasks. Examples of global communication include: one-to-many communication patterns such as *broadcast* and *scatter*, many-to-one communication patterns such as *reduction* and *gather*, and many-to-many communication patterns such as *alltoall*, see also Figure 1.11 for a survey. An example for global synchronization is *barrier synchronization*. Here we only consider one such global communication pattern (reduction) and one global synchronization pattern (barrier synchronization).

Reductions are a frequently occuring type of computation in scientific and engineering programs. It refers to the concept of accumulating many values, e.g. of array elements or stream elements, to a single scalar, often by summation, averaging, maximization or similar accumulation operation. In sequential, we would use a simple loop over the elements to be accumulated, together with a scalar accumulator variable.

In a parallel computing context, a naive solution for the reduction problem would be that of using a *central manager* task that "owns" the single accumulator variable. All other tasks will just send their contribution to the manager task and are done; the manager has to receive all these values and add them up sequentially. It is obvious that this (not so) parallel algorithm is not going to scale well, because the central manager task becomes a performance bottleneck. The key point is that the central-manager approach does not distribute the computation work and not half of the communication work either, leading to high load imbalance and sequentialization.

As an alternative to the central manager scenario, we use an approach that is based on the algorithmic design pattern *divide-and-conquer*.

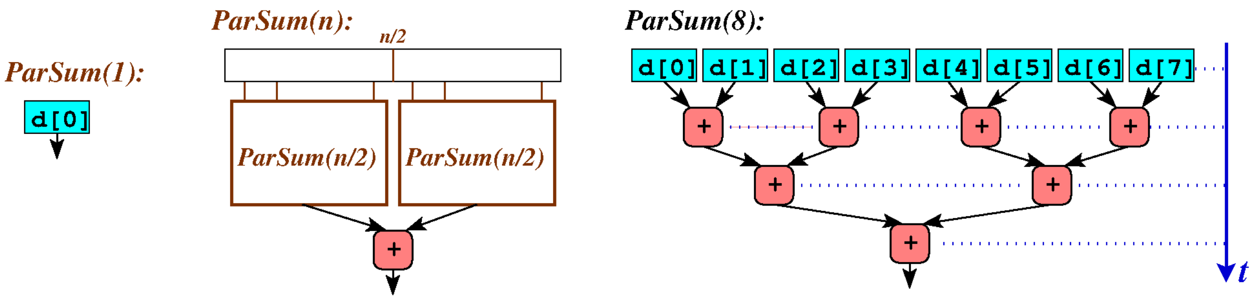


Figure 1.12: Left: Parallel sum computation using the parallel divide-and-conquer algorithmic design pattern. Right: The tasks (8 memory loads, 7 additions) resulting from the parallel algorithm with base case size 1 (element) applied to an array of 8 elements.

The *divide-and-conquer* design pattern aims for a recursive solution as follows: If the base case of recursion is reached, i.e., the given problem instance is small enough, it is solved directly, using some simple method applicable to very small problem instances. Otherwise, the given problem instance is divided into one or several *independent* subproblems that each are *smaller*[7] than the original one. These subproblems are solved recursively, and then their solutions are combined into a solution of the original problem instance.

For the most common type of reduction, the global sum problem, the base case size is 1 element (which only needs to be loaded from memory). The divide step in the general case exploits associativity of addition, which implies that

$$\sum_{i=0}^{n-1} x_i \;=\; \sum_{i=0}^{n/2-1} x_i \;+\; \sum_{i=n/2}^{n-1} x_i.$$

and the combining of subproblem solutions consists in adding up two partial sums.

As all subproblem instances created from a problem instance must be independent of each other, the divide-and-conquer pattern can be simply generalized to *parallel divide-and-conquer*, suggesting that the recursive solutions of the different subproblems form independent tasks that can execute in parallel. Synchronization is required especially at the termination of such a task to make sure that the combining step of the parent task will proceed only after all recursive tasks solving subproblem instances have terminated. See also Figure 1.12.

1.3.2.3 Global Synchronization

A *barrier synchronization* is a global synchronization construct that requires that all tasks, threads, processes or processors of a group have arrived at that point in the program before any of them is permitted to continue with the code following the barrier. In other words, there must be a point in time where all these tasks are waiting at the barrier, namely when the latest one has just reached it.

Figure 1.13 (left) shows the dependence graph of a barrier for a group of four tasks. Note that the barrier needs not be implemented exactly in this way (which would correspond to a message-based solution with an all-to-all communication pattern exchanging $p(p-1)$ single-word messages in total that acknowledge arrival at the barrier to all others in the group). Instead, the central-server pattern and/or parallel divide-and-conquer strategies as introduced above for reductions can be applied to reduce the total number of messages to $O(p)$, at the expense of more intermediate steps; see for example a simple two-step solution in Figure 1.13 (right) where a global gather of arrival

[7]This requirement is important to guarantee termination of the algorithm.

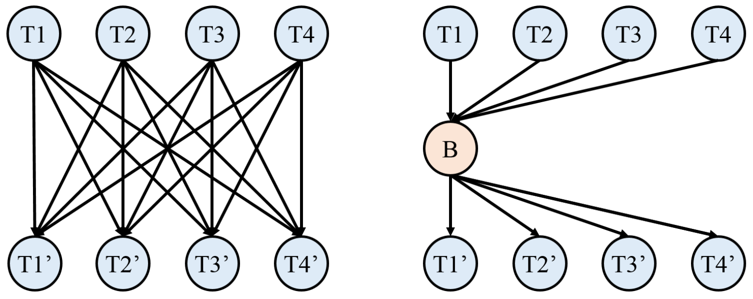


Figure 1.13: Barrier synchronization. Left: The dependence graph of a barrier synchronization point for 4 tasks; any task continuation T_i' cannot be started before each task T_i has arrived at the barrier point. — Right: One possible message-based implementation that uses an auxiliary central server task (B) and $O(p)$ messages in total.

acknowledgment messages is followed by a global broadcast of the "pass" message from the central server task B. We will later see that there exist more scalable communication algorithms based on parallel divide-and-conquer (i.e., trees) for both gathering or reducing over acknowledgment messages and for broadcasting.

In a shared memory environment, barriers can also be implemented using shared memory data structures and atomic accesses, for instance by a shared global integer counter that is initialized to 0. A task arriving at the barrier point atomically increments the counter and waits for the counter to reach the expected number p of group members. When that happens, all group members can continue past the barrier.

1.3.2.4 Unstructured and Dynamic Communication

In the stencil computation examples considered so far, we assumed that the multidimensional mesh modeling the data domain is regular, i.e., a multidimensional array where neighborhood in the domain coincides with neighborhood in array index expressions. This is fine as long as e.g. the modeled objects for a simulation look like a rectangular metal plate or a brick, i.e., have properties (such as surface normal vectors) that can be considered homogeneous over (large parts of) the entire domain. For irregular meshes this is no longer the case.

Irregular meshes result for example from adaptively modeling objects with curved volumes or surfaces for their behavior in simulations, such as simulating the air flow from a fan hitting a curved-surfaced cooling hood over a hot electronic component. The volumes or surfaces of such an object are tesselated into very many, very small elements that are also known as finite elements resp. finite volumes. These represent object areas (surface sections or volume segments) that are small enough so that their simulation-relevant properties (e.g., temperature or surface normal vector) can be assumed as locally homogeneous and constant without creating a significant discretization error. This leads to an adaptive tesselation: where, for instance, the surface curvature is stronger, the elements have to be made smaller than in more flat surface areas to keep the element-wide differences in surface normal vector within a very small bound. The adaptive tesselation of such objects into small elements is usually the result of using special (e.g., FEM) modeling tools, but can also change dynamically during a simulation where elements need to be refined and split if their observed discretization error gets too large.

An irregular mesh can consist of many millions of elements, where each element can have three or more neighbors. However, these will in general not be neighbors in memory. A common data

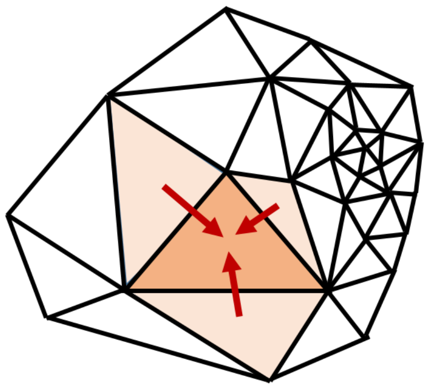


Figure 1.14: Jacobi-update step with a 3-point stencil over an irregular mesh of triangles modeling a curved surface.

```
while (...) {
  for (i=0; i<N; i++)
    new_temperature[i] = f( elem[i].temperature, ...,
                            elem[ elem[i].neighbor[0] ].temperature,
                            elem[ elem[i].neighbor[1] ].temperature,
                            elem[ elem[i].neighbor[2] ].temperature );
  for (i=0; i<N; i++)
     elem[i].temperature = new_temperature[i];  // copy back
}
```

Figure 1.15: Jacobi-update steps with 3-point stencil over an irregular mesh of triangles modeling a curved surface.

structure is a one-dimensional global array of structs, one struct per element, where each struct not only holds the element's properties such as temperature or normal vector coordinates but also a list of pointers to its direct neighbor elements in the domain (usually, the indexes of those elements' entries in the global array).

```
struct triangular_element {
  struct point3D *corner[3]; // pointers to triangle's corner points (coordinates)
  double nomalvector[3]; // normal vector direction coordinates
  double temperature;
  int neighbor[3]; // indices of adjacent triangles in elem array
} elem[N];
```

Now, a stencil computation updating element i (see Figure 1.14) needs to look up the indices of the neighbor elements of i first, and then look up their structs to access the needed values. For triangles, a simple stencil update computation could consider the 3 direct neighbors' temperature values, as shown in the pseudocode in Figure 1.15. This indirection allows for compact representation of the elements, but also leads to additional memory access costs. Moreover, because neighbor elements are, in general, no longer close to element i in the global array, we can expect more cache misses compared to a stencil computation on a regular domain.

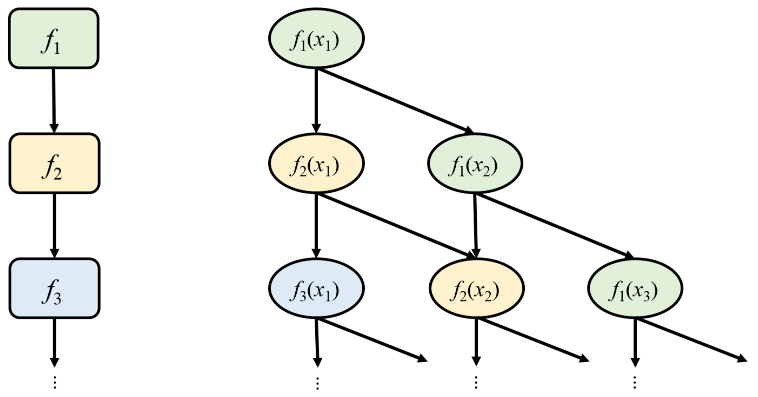


Figure 1.16: Left: The pipeline stage dependence graph shows the static, linear dependence structure between the pipeline stages f_1, f_2 etc. — Right: The pipeline task instance dependence graph shows the dependences between stage executions (i.e., task instances) for different input elements x_1, x_2 etc.

Partitioning an irregular mesh with possibly millions of elements into a fixed number of submeshes (ideally, contiguous, convex and of about equal computational workload) that could define one task each is not straightforward, and special tools such as METIS are available for that purpose.

For irregular meshes, the communication structure will usually be irregular and data dependent, and may even change dynamically during execution.

1.3.2.5 Pipelining

Pipelining is a parallel algorithmic design pattern that can be applied when processing a sequence of data elements, such as an array or stream of elements, where tasks do have dependences for the *same* element but where those tasks can overlap in time for *different* subsequent elements of the input stream.

More formally, we are given a sequence of k *dependent* computations (static tasks, stages) $\langle f_1, f_2, ..., f_k\rangle$ that is to be applied *elementwise* to the input data sequence $\langle x_1, ..., x_n\rangle$. See Figure 1.16 (left) for the static dependence graph among the different stages f_i. For any fixed input element x_j, we have to compute $f_i(x_j)$ before $f_{i+1}(x_j)$, because the latter will consume a partial result or state that has been produced by the former. Also, for pipeline stages f_i that carry a runtime state from one element computation to its next one, $f_i(x_j)$ must be computed before $f_i(x_{j+1})$. See Figure 1.16 (right) for the dependence graph among the different *stage task instances* $f_i(x_j)$. However, many stage task instances are independent of each other, for instance $f_i(x_j)$ and $f_{i-1}(x_{j+1})$.

The *pipelining principle* consists in overlapping, at any time, the execution of some or all k tasks f_i for up to k subsequent elements x_j. In other words, we define one subtask per possible combination (f_i, x_j), and these subtasks properly synchronized as requested above when being processed in the pipeline.

The pipelined computation executes in *rounds*, one round per input data element. In the first $k-1$ rounds, the pipeline is being filled, and j subtasks are ready to execute in round j for $1 \leq j \leq k-1$. This first phase of the pipeline execution is also referred to as the *prologue* of the pipeline computation, or *filling the pipeline.* From round k on (assuming that there are at least k input elements to process), exactly k subtasks are being processed in each round until the last input

Round 1:	compute	$f_1(x_1)$						
Round 2:	compute	$f_1(x_2)$	and	$f_2(x_1)$				
Round 3:	compute	$f_1(x_3)$	and	$f_2(x_2)$	and	$f_3(x_1)$		
...								
Round k:	compute	$f_1(x_k)$	and	$f_2(x_{k-1})$	and	...	and	$f_k(x_1)$
...								
Round $j \leq n$:	compute	$f_1(x_j)$	and	$f_2(x_{j-1})$	and	...	and	$f_k(x_{j-k+1})$
...								
Round n:	compute	$f_1(x_n)$	and	$f_2(x_{n-1})$	and	...	and	$f_k(x_{n-k})$
Round $n+1$:				$f_2(x_n)$	and	...	and	$f_k(x_{n-k+1})$
...								
Round $n+k$:								$f_k(x_n)$

Figure 1.17: Pipelined execution for a k-stage pipeline on an n-element input stream. The steady state of the pipeline is reached at round k, the epilogue begins after round n.

element enters f_1. This situation is also referred to as the *steady state* of the pipeline. Finally, the last $k-1$ rounds execute when all input elements have been read and the pipeline is being *drained*. In the $(k-j)$th-last round, for $j = 1, ..., k-1$, only j subtasks are ready to execute. This last phase is also referred to as the *epilogue* of the pipeline computation.

The pipelined execution is visualized in Figure 1.17.

The $n+k$ rounds are executed in sequential, and synchronization must make sure that subtasks of round $j+1$ are not started before all subtasks of round j are finished, for instance by bilateral producer-consumer synchronization between stage task instances or by simply placing a barrier synchronization between any two subsequent rounds.

Usually pipelines have a small constant number k of stages (tasks), hence we can assume that each stage could run on its own processor; it is of course also possible that multiple light-weight tasks time-share one processor for better load balancing. Moreover, heavy-weight stages might be further parallelized internally e.g. by domain decomposition if applicable. Given sufficiently many processors, the execution time for one round will be dominated by the slowest stage (task), and the overall execution time for the entire pipeline will be

$$O\left((n+k) \cdot \max_{1 \leq i \leq k} \text{time}(f_i)\right)$$

In the above scenario, we assumed a linear chain of dependences for the same input element, i.e., $f_1(x_i) \rightarrow f_2(x_i) \rightarrow ... \rightarrow f_k(x_i)$. However, a pipeline's task dependence graph needs not have the form of a linear chain but could be any directed acyclic graph[8]. Because any directed acyclic graph can be partitioned into levels, all tasks of a level could be conceptually merged into one macrotask, reducing the residual dependence structure to a linear chain of dependences again.

1.3.2.6 Summary: Communication and Synchronization Guidelines

In order to obtain scalable parallel algorithms, not only the computation but also the communication work needs be parallelized. Ideally, tasks should perform approximately equal numbers of communication operations, and many communication operations should be able to proceed concurrently.

[8]In some cases even feedback links back to earlier pipeline stages can occur in pipeline graphs. However, the task instance dependence graph is always acyclic.

If possible, tasks should communicate only with a small number of neighbors. This will later make it easier to map tasks to processors in a way that neighbored tasks often also end up on the same or neighbored cluster nodes. Depending on the interconnection network topology, nearest-neighbor communication is often faster (has shorter latency) than communication to more distant nodes.

The synchronization structure should be designed so that computations associated with different tasks can proceed concurrently.

Barrier synchronizations are simple to insert in a program, but can be overkill (i.e., are overly restrictive and can lead to programs that do not scale so well to large numbers of processors) in cases where only a few precedence constraints between specific tasks need to be preserved.

1.3.3 Agglomeration

After the partitioning and communication/synchronization steps we have created a design of a fine-grained parallel algorithm that could run on an idealized (abstract) parallel system with arbitrarily many processors (ideally, one per task) or maybe use some runtime system for time-sharing a fixed set of processors to run all tasks concurrently. However, the task granularity will in many cases be too fine to efficiently map on a fixed set of parallel processors—the overhead of creating, managing and scheduling very many and tiny tasks on a real-world parallel computer can easily outweigh any potential performance benefits from their parallel execution. In particular, a fine task granularity often also implies much communication with many short messages and/or much synchronization work that adds considerable inefficiencies compared to an equivalent sequential algorithm.

Agglomeration, also known as (task) *clustering* in the perspective of a task graph transformation, is a coarsening step that merges many light-weight tasks into more heavy-weight macro-tasks by sequentializing these tasks in some order that is compliant with their precedence constraints. The macro-tasks will become concrete work units that could be realized as threads or processes, for instance. Hence, agglomeration provides the transition from abstract tasks running on an abstract parallel machine towards concrete units of execution on a specific target architecture.

Agglomeration internalizes communication and / or synchronization between all tasks merged into the same macrotask. Hence, aggressive agglomeration will reduce communication/synchronization overhead as well as the overhead for scheduling many small tasks, but it will also reduce flexibility for mapping and scheduling and ultimately limit the degree of parallelism and thereby prevent scalability. In other words, there exists a trade-off between concurrency and communication, usually with a "sweet spot" somewhere that represents the optimal degree of agglomeration.

When merging tasks, it can sometimes be worthwile to *replicate* data and/or computation. While replication of work or memory footprint should, in general, be avoided in the interest of a well scalable parallel algorithm, considerate replication can in certain cases eliminate communication and/or synchronization and even prevent deadlocks.

In the following we consider some examples of agglomeration strategies.

1.3.3.1 Agglomeration of Tasks from Domain Decomposition

For domain-decomposed multidimensional data structures, the decomposition dimensionality could be reduced, e.g. from 3 to 2 dimensions (see Figure 1.18).

For domain-decomposed multidimensional data structures, adjacent tasks are combined, giving new 3D partitions (see Figure 1.19).

This allows to exploit *surface-to-volume* effects: A cube-shaped 3D partition with $k \times k \times k$ elements has a volume (i.e., amount of computational work) that grows cubically with the partition edge size k, while its surface (i.e., amount of communication work to exchange boundary elements

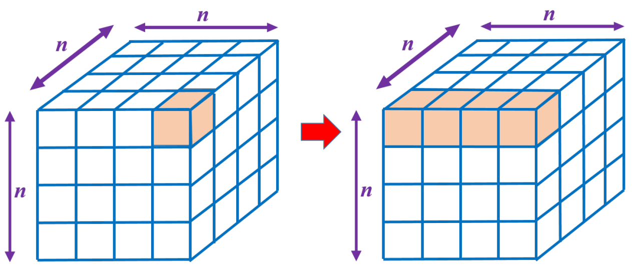


Figure 1.18: Agglomeration by reducing the dimensionality of partitioning from 3D to 2D for Jacobi-style finite-differences stencil computation on a regular 3D mesh (array). Adapted from Foster [26].

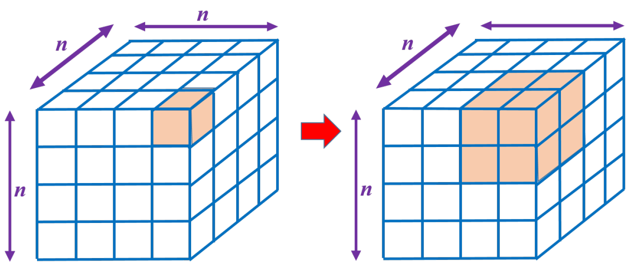


Figure 1.19: Agglomeration by 3D partitioning allows to leverage the surface-to-volume effect of communication to computation work. Adapted from Foster [26].

with neighbor partitions) only grows quadratically with partition edge size. Hence, the coarser the granularity, the better the ratio of computational work to residual communication (i.e., I/O operations or shared memory accesses).

1.3.3.2 Agglomeration of Tasks from Functional Decomposition

Exploration of subtrees in parallel divide-and-conquer recursion trees can be coalesced (sequentialized), by switching from parallel to sequential divide-and-conquer recursion. See Figure 1.20 (left).

The nodes in a computation with a tree-based data flow graph (e.g., reduction or search computation) can then be further combined along paths, because any two tasks on the same path in the data flow graph will be dependent and thus not execute simultaneously. See Figure 1.20 (right).

1.3.3.3 Agglomeration of Communication

After agglomeration of tasks' computational work by merging tasks, we can often bundle multiple residual communications from the same sender to the same receiver in one message. This reduces communication cost, because each message transfer incurs a constant time overhead (for software overheads, message packaging overhead and network transfer latency), and after the merging of several small messages into a longer one this time overhead is only incurred once.

An example is the exchange of boundary elements with neighbor macrotasks in 2D or 3D finite differences (Jacobi stencil computation) using 2D or 3D partitions respectively. Because all boundary elements of a macrotask's partition's edge or face connected to the boundary elements of

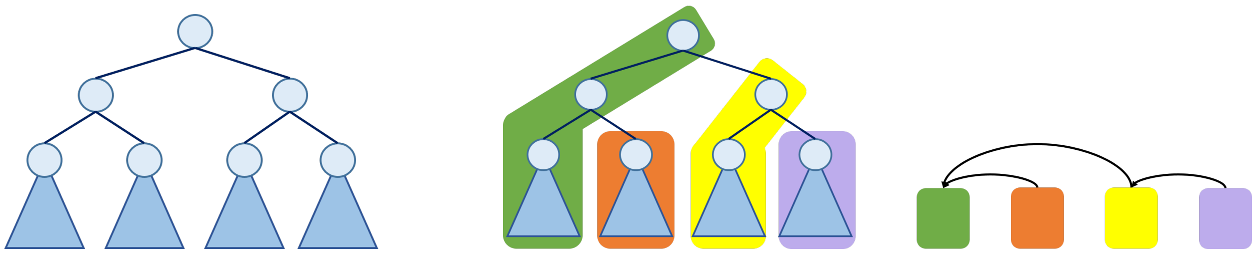


Figure 1.20: Left: Agglomeration of a parallel recursive search tree exploration or reduction computation can easily be achieved by switching from parallel to sequential execution of recursive calls for entire subtrees as soon as a certain criterion is met, such as a certain recursion depth. — Right: Further agglomeration of computations with tree-based dataflow graphs can be achieved by merging tasks along producer-consumer chains. The residual communication structure between macrotasks (colors) is shown to the right.

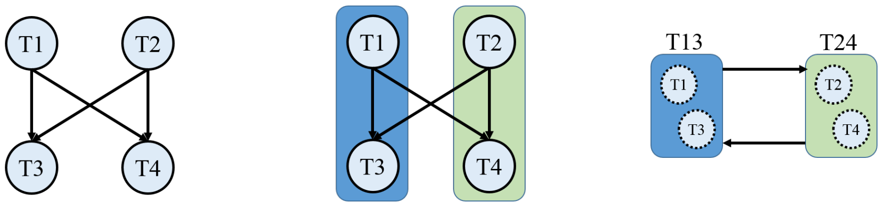


Figure 1.21: Agglomeration may cause artificial cycles that make a task graph unschedulable.

its neighbor macrotask's edge or face will be transfered from the same sender to the same receiver, they can be transfered in a single message.

1.3.3.4 Caution: Agglomeration May Cause Artificial Cyclic Dependences

A possible risk with agglomeration is that the acyclicity property of a task graph can be lost after merging some tasks. An example is shown in Figure 1.21. Starting from an acyclic task graph (left), merging $T1$ with $T3$ into a macrotask $T13$ and task $T2$ with $T4$ into a macrotask $T24$ (middle) leads to a residual task graph with a dependence cycle between the two macrotasks (right). This cycle of precedence constraints is artificial, as it did not exist in the original task graph. If the task execution model does not permit preemptive execution of tasks, then this cycle of precedence constraints makes the resulting task graph unschedulable (unless we are guaranteed to have 2 processors available that we can dedicate to running $T13$ and $T24$ in parallel, which is a constraint on the mapping and scheduling phase which might not be able to be guaranteed in all scenarios).

1.3.3.5 Summary: Agglomeration Guidelines

One should choose an agglomeration method that reduces communication by increasing locality.

If agglomeration replicates computation, we need to verify that the benefits really outweigh the costs. For that, we need a *cost model*, see Chapter 2.

Agglomeration has to be done with care in order not to introduce artificial dependence cycles by merging tasks.

If agglomeration replicates data, we need to verify that this does not compromise scalability.

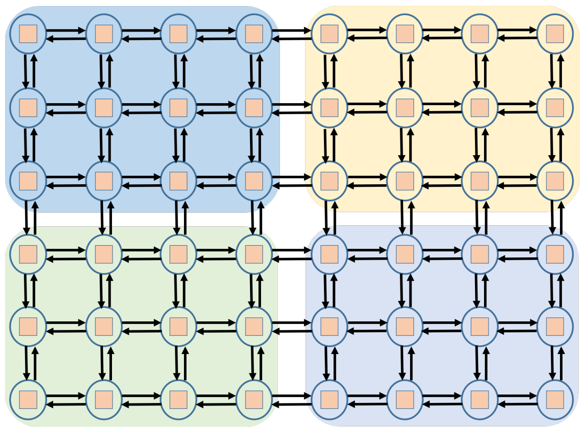


Figure 1.22: Mapping equal-sized contiguous 2D stencil-update task partitions to different processors (colors) for a regular 2D mesh.

We should aim at obtaining tasks with similar computation and communication costs. This recommendation becomes the more important the larger the macrotasks become, because the flexibility for later load balancing is reduced with increased agglomeration. Also, the number of tasks and degree of parallelism left after agglomeration should still scale with the problem size.

1.3.4 Mapping and Scheduling

Mapping refers to deciding for each task *where* (i.e., on which processor or compute node) it should execute.

Scheduling refers to deciding *where and when* each task should execute. Hence, mapping is a subproblem of scheduling. Mapping alone leaves some flexibility to a processor's or compute node's local scheduler with respect to how the tasks mapped to it can be ordered in time.

Usually, the goal of mapping and scheduling is to minimize the total execution time for the program, i.e., for all tasks. This is normally achieved by finding the right compromise solution between spreading the work evenly across the available processors, and placing tasks communicating frequently on the *same* processor to reduce overall communication time.

In a parallel system, the most loaded and thus longest-running processor determines the overall parallel execution time of a parallel program. Hence, load balancing is a big concern at this stage. In this section we will consider static vs. dynamic load balancing approaches as well as local vs. global ones.

1.3.4.1 Static Mapping for Regular Mesh Problems

In the simplest case, all (macro-)tasks could proceed in parallel and all perform approximately the same amount of work, both in computation and communication.

An example is the Jacobi-style finite differences stencil computation on a regular 2D mesh, see Figure 1.22. Mapping equally large partitions of (macro-)tasks to processors will also balance the workload among processors.

Beyond having the same size, each subset of stencil macrotasks mapped to a processor should form contiguous, ideally convex, partitions of the same dimensionality as the mesh itself, here 2D. This allows leveraging the surface-to-volume effect, keeping the amount of residual inter-processor

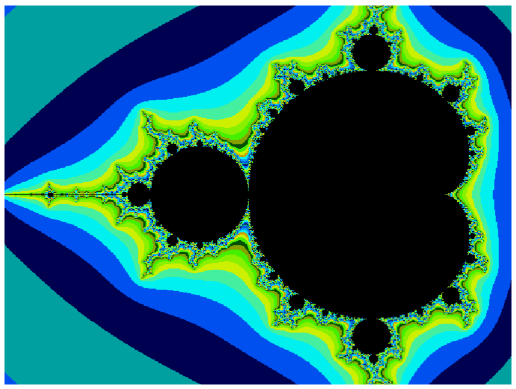


Figure 1.23: The Mandelbrot set (black pixels) plotted in the range $-2 \leq \Re(c) \leq 0.6$ and $-1 \leq \Im(c) \leq 1$. The color indicates the number of iterations (and hence, computational work) required to calculate the pixel value.

communication / synchronization links at a minimum in relation to the volume of computation (tasks) done by each processor.

Another positive side effect of forming equal-sized contiguous 2D partitions is that each processor only needs to communicate with four neighbor processors. This is advantageous e.g. if these processors are cores on a many-core architecture with a 2D mesh on-chip interconnection network, where communication with direct neighbors in the network is faster than with more distant processors on the same chip or even with processors located on a different chip.

1.3.4.2 Static Load Balancing

A simple example for a load balancing problem is computing the Mandelbrot Set.

The *Mandelbrot set* is defined as those points $c \in \mathbb{C}$ in the convex plane for which the sequence $z_{n+1} = z_n^2 + c$, started with $z_0 = (0, 0)$, does *not* converge to ∞ for $n \to \infty$. An interesting range for c is about $-2 \leq \Re(c) \leq 0.6$ and $-1 \leq \Im(c) \leq 1$ in the complex plane (see Figure 1.23)[9]; of course one could study any rectangular subrange of $\mathbb{C}$ to zoom in.

A 2D image visualizing the Mandelbrot set for a certain rectangular section of the complex plane is obtained as follows: For each pixel (x, y) we calculate the point c corresponding to its midpoint in the section of interest, and then, for n from 0 upwards, the sequence $z_n = z_{n-1}^2 + c$ until either convergence to ∞ is detected (i.e., $\mid z_n \mid$ exceeds some sufficiently large upper limit), or an upper limit $n_{\max}$ for the number n of sequence iterations is reached, where typical values for $n_{\max}$ are in the order of 100 or higher for decent approximation accuracy of the Mandelbrot set, also depending on the image resolution and the size of the section of $\mathbb{C}$ selected for plotting. The number n of iterations performed for c is used to select the color for pixel (x, y).

The Mandelbrot set computation is a so-called *embarassingly parallel* problem, because all pixels can be computed independently of each other. If, by 2D domain decomposition, each single pixel computation defines one task, there are no precedence constraints nor communication links between any tasks.

The work performed for each pixel (x, y) is linear in the number of iterations n to be performed for it, which is somewhere between 1 and $n_{\max}$ and we do not know it ahead of time. In Figure 1.23 the black pixels all required running $n_{\max}$ iterations, while the pixels colored in the "outer" colors

[9] $\Re(c)$ and $\Im(c)$ denote the real and imaginary part of $c \in \mathbb{C}$.

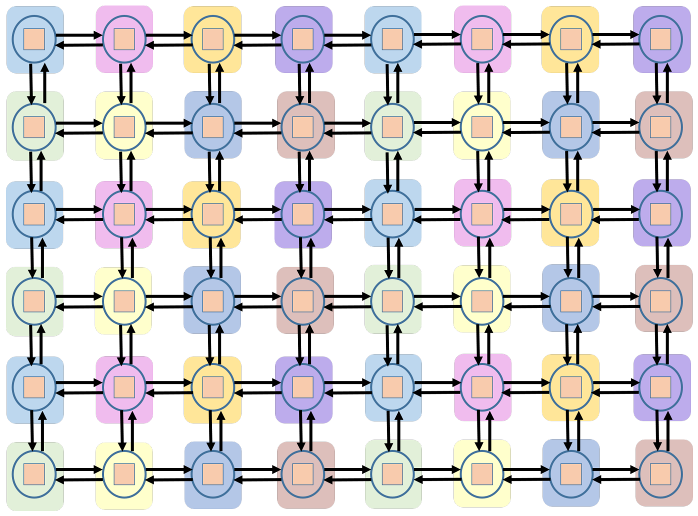


Figure 1.24: A cyclic mapping of (macro-)tasks to 8 processors (shown by the different colors) for static load balancing.

(shades of blue, yellow, green) only needed a few iterations each. If our image has R rows of pixels and we have a a certain number p of processors, we could naively perform a 1D partitioning of the image and map $\lceil R/p \rceil$ (or $\lfloor R/p \rfloor$) contiguous rows of pixels to each processor for calculation. However this would imply that those processors getting the outer rows would be quickly done as these mostly contain pixels for which few iterations are performed, while the processors getting inner rows have to do many time-consuming computations (black pixels). A 2D partitioning will not really improve the load imbalance problem.

Where communication with neighbor tasks is not a major concern, as is the case in the Mandelbrot example, we can actually deviate from the otherwise good idea of forming contiguous, convex partitions in the mapping step. This opens for a number of alternative task mappings:

- *Probabilistic methods* use pseudorandom assignments of tasks to processors. With high probability this will lead to a good load balance for our Mandelbrot example.

- *Cyclic methods* use a cyclic mapping scheme (preferably in as many dimensions as the given set of tasks), see Figure 1.24 for a 2D cyclic mapping. As now every processor gets pixels from all over the image, it is very likely that the overall load balance will be good.

  From Figure 1.24 it also becomes clear that probabilistic or cyclic mappings are not a good idea for stencil computations that frequently perform communication with nearest-neighbor tasks; a cyclic mapping would here actually maximize the amount of inter-processor communication.

In cases where work is not naturally load-balanced across the data domain or function call domain, the partitioning, agglomeration and mapping steps are strongly interdependent and should be considered together.

Where (macro-)task partitions should be kept contiguous due to significant inter-task communication, a number of *adaptive partitioning* techniques can be used where predictors for task workloads are available. For rectangular domains, there are, in particular, flat decompositions and hierarchical decompositions, which can both be further classified into uniform and non-uniform decompositions, see also the survey schema in Figure 1.25.

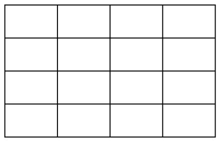
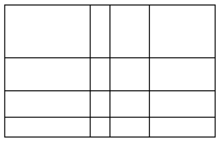
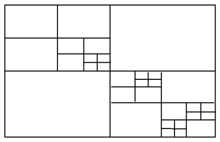
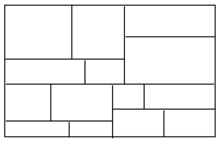
Decomposition type	Uniform	Non–uniform
Non–hierarchical	 *uniform grid / mesh*	 *non–uniform grid / mesh*
Hierarchical	 *quadtree / octree*	 *binary space partitioning*

Figure 1.25: Classification of domain decomposition types.

- Given p processors, a *flat uniform decomposition* partitions the overall set of tasks into p equally sized partitions; this can be done in one or multiple dimensions, see Figure 1.25 (upper left). This is the decomposition scheme that we used above for stencil computations over regular meshes.
- A *flat non-uniform decomposition* calculates the paritions so that the predicted workloads will be approximately equal across partitions, as far as possible; see Figure 1.25 (upper right).
- *Hierarchical decompositions* performs the partition process in a divide-and-conquer style, starting from the overall set of tasks and decomposing it recursively into (e.g., 2) subsets of roughly equal predicted work.
- *Binary space partitioning* splits each partition by an axis-parallel bisector line/plane along one dimension of the task space at a time into two subsets of roughly equal predicted work, and alternates between the dimensions in each hierarchy level. This is performed along a recursive subdivision tree of height (at best) $\lceil \log_2 p \rceil$ until (at least) p partitions exist, see Figure 1.25 (lower right).
- *Quadtrees* and *Octrees* are the recursive subdivision trees that are obtained by dividing a 2D or 3D task space uniformly into subpartitions along all dimensions simultaneously. In every subdivision step, a partition that is predicted to contain more than a given threshold of work will be further subdivided into 4 or 8 subpartitions, respectively. See Figure 1.25 (lower left) for a quadtree decomposition example.

Finally, there also exist *general methods for static task scheduling* to multiple processors, which can be used if the task workloads are irregular but statically known. In the literature there exists a wealth of static multiprocessor scheduling algorithms, both for independent tasks, for dependent tasks with an acyclic dependence graph, and for periodic tasks with acyclic or cyclic producer-consumer communication graph. For a survey, see e.g. [44]. We will briefly consider task scheduling in Section 2.5.5.

1.3.4.3 Dynamic Local Load Balancing

Dynamic load balancing does not need to know or predict the actual load requirements for the tasks ahead of time.

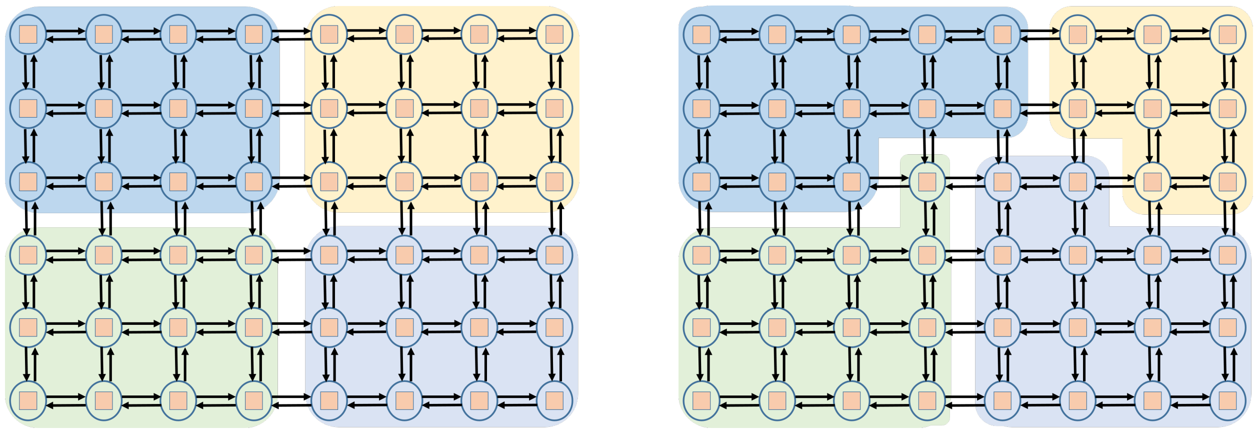


Figure 1.26: Local load balancing. Compared to the previous mapping of tasks to processors (left), some boundary tasks are migrated to less loaded neighbor tasks (right).

Dynamic *local* load balancing starts from some mapping, such as one with equally sized partitions, and regularly interrupts the iterative computation (e.g., every K iterations of the outer while loop in the Jacobi finite differences example above), calculates the current workload per processor, and then decides which tasks to migrate between processors to obtain a mapping expected to have a more balanced load distribution. Dynamic load balancing always comes at the expense of some runtime overhead, i.e., should not run too frequently so that its additional cost does not outweigh the future speedup gained from better load balance.

Local load balancing should aim at keeping contiguous, preferably convex partitions of tasks per processor in order to keep the communication work as low as possible. It should also try to migrate, in the first hand, tasks between neighbored processors because task migration cost involves a considerable communication volume too.

Figure 1.26 shows an example of dynamic local load balancing by adaptive domain decomposition. A reasonable strategy is that processors with too high workload migrate some tasks τ computing boundary elements to a less loaded processor owning a task communicating with τ.

1.3.4.4 Dynamic Global Load Balancing

Where the previously described, relatively simple load balancing techniques are not applicable or are not sufficiently effective in removing locally concentrated load imbalances, dynamic global load balancing techniques should be considered: these techniques based on *dynamic task scheduling* are more general and potentially more powerful because they keep a global view of all tasks and load balancing relevant decisions happen frequently, but they usually also come at an even higher runtime overhead than local dynamic load balancing.

As dynamic global task scheduling has a global scope and is thus more general and powerful than dynamic local load balancing, it is in particular used in the core of runtime systems for task-based parallel programming environments, such as StarPU, OmpSS and OpenMP4.x, XKaapi, SuperGlue, Cilk and TBB. Common to these runtime systems is an internal data structure (usually some kind of queue implementation, which can be shared or distributed in memory) that keeps the ready-to-execute tasks, and they also keep track of all tasks currently waiting for data or for other tasks (e.g., at synchronization points) and automatically promote them to the ready-to-execute data structure when the waiting is finished.

In the following, we consider a few techniques for realizing such ready-to-execute task queue

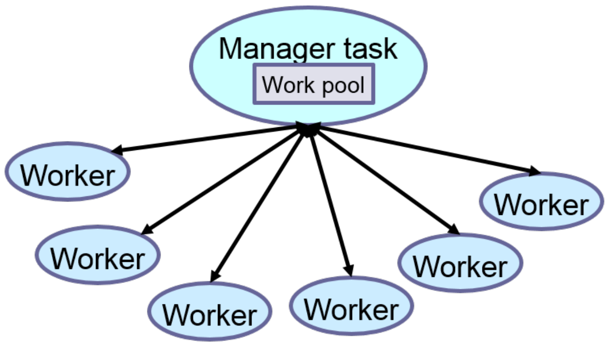


Figure 1.27: A global shared work pool (e.g., a shared queue holding task descriptors) used for dynamic load balancing, here modeled by a central work manager task. Either the central manager actively assigns (schedules) units of work to workers (modeled as worker tasks) aiming at balancing their expected workload, or workers getting idle request new work items from the central manager.

data structures.

1.3.4.4.1 Central Task Queue With the central manager approach, one manager task is dedicated to act as work pool server to all worker tasks. The data structure itself could be realized internally as a FIFO queue. The manager task accepts requests from workers for tasks and sends a task descriptor back, or NULL if its task queue is currently empty. See Figure 1.27.

The central manager can, unless being multithreaded itself, only serve one task at a time and thus easily become a performance bottleneck that limits scalability. If multithreaded, or if realized as a passive data structure without its own manager thread, then access to the queue data structure becomes a critical section and must be protected, which might lead to serialization and thereby to scalability problems again.

The FIFO strategy can of course be refined. For performance reasons, a locality aware strategy such as affinity-based scheduling is often preferable because it allows to keep the number of cache misses lower.

1.3.4.4.2 Hierarchical Task Queue Hierarchical task queues solve the manager bottleneck problem by employing multiple managers, one each in charge of a subgroup of workers, and a manager of these managers. Requests for new work will first go to the nearest manager, and only if the nearest manager has run out of tasks, it will request new work from the top manager.

1.3.4.4.3 Distributed Task Queue In a shared memory scenario, the central task FIFO queue can be organized by a round-robbin distribution of tasks into several FIFO subqueues so that simultaneous access by multiple threads can be supported as long as different subqueues are accessed.

1.3.4.4.4 Local Task Queues with Work-Stealing The work-stealing approach takes the optimistic approach that most of the time there are enough ready tasks locally available and idleness is thus rare. Each processor (usually, core) runs exactly one (pinned) worker thread which maintains its own local task pool, stored in shared memory.

This local task pool of each worker is often implemented as a doubly-ended queue (deque) where newly spawned tasks are pushed on top (as on a local stack of function calls) and the

```
taskqueue tasks[Nworkers];  // one per worker

void SPAWN ( task *t )
{
   tasks[myworker].push( t );
}

void SYNC()  // wait for last spawned subtask:
{
   (status, t) ← tasks[myworker].pop();
   if (status = STOLEN) {
      while (! t.done)
         // try to steal back ("leapfrogging"):
         STEAL_WORK( t.thief );  // wait for t
      return t.result;
   }
   else  // have to do it myself:
   return t.execute();
}

void STEAL_WORK( int victim )
{
   t = tasks[victim].STEAL();
   if (t != NULL) { // got some task:
      t.thief ← myworker;
      t.result ← t.execute();
      t.done ← TRUE;
   }
}

thread WORKER ( int id, task *root_task )
{
   if (id = 0)
      root_task.execute();
   else
      forever
         STEAL_WORK ( rand_victim() );
}
```

Figure 1.28: Pseudocode for each worker thread (executing routine WORKER) for a work stealing implementation in a shared memory environment. The special thread-local constant variable *myworker* contains the executing worker thread's unique ID between 0 and *Nworkers*−1. Adapted from [64].

worker itself pops its next task from the top (as on a local stack), while idle workers steal a task from a randomly chosen victim processors task queue bottom end. If the victim is also idle, we try another one, and so on. If all other workers are found to be idle too, the entire dynamically scheduled parallel computation is obviously finished and each worker can terminate. See Figure 1.28 for the pseudocode.[10]

Hence, as long as there are any tasks in the queue, it can be used like an ordinary local call stack at low overhead. Only if idle, there will be some major overhead for finding the victim and stealing (migrating) a task from there.

Depending on how the steal requests coming in from other processors are handled, some extra synchronization may be necessary to protect the deque data structure from corruption in the case of concurrent steal, pop and push operations.

It is also possible to apply the work stealing approach to a distributed memory message-passing scenario. Here, every node will keep its local task queue in its local memory and act as a global manager for steal requests to its own task queue. Steal requests will thus result in message passing[11] from the thief worker to the victim worker process and back.

1.3.4.4.5 Dynamic Scheduling of Independent Loop Iterations

Due to the simple, regular structure of for loops, no task queue with explicit task descriptors is necessary. Instead, in a shared memory environment, a single shared integer variable holding the first not yet scheduled loop iteration, initialized to the lower loop index bound, is sufficient, see Figure 1.29. Idle workers

[10]The details are relevant for TDDD56 only.

[11]Where available, one-sided communication can be used to simplify the implementation.

```
for ($i \leftarrow 0$; $i < N$; $i++$) {
    iteration($i$);
}
```

```
unsigned int $iter \leftarrow 0$;  // global shared counter
...
while ($iter < N$) {
    $i \leftarrow$ FETCH&ADD(&$iter$, 1);  // atomic
    iteration($i$);
}
```

Figure 1.29: Pseudocode of a sequential loop with independent iterations (left) and a parallel loop schedule using an atomic *fetch&add* instruction (right).

take the next iteration, or bunch of iterations, by performing an atomic fetch-and-add operation on this shared variable. Atomicity guarantees that all loop iterations are assigned exactly once for execution.

Different strategies for dynamic scheduling of independent loop iterations will be discussed in more detail in the forthcoming Chapter **??** on OpenMP programming.

Dynamic loop iteration scheduling would be directly applicable to the Mandelbrot example above, because the computations of different rows or different pixels, usually iterated over by for-loops, are completely independent of each other.

Dynamic scheduling of *dependent* loop iterations requires more advanced synchronization and will be handled in the forthcoming chapter on loop optimization and parallelization.

1.3.4.5 Summary: Mapping and Scheduling Guidelines

Static scheduling schemes are advisable if task execution times are statically known. Simple regular decompositions are sufficient where task execution times are expected to be equal across the entire task space.

Where task execution times are not known before execution time but expected to vary significantly, load balancing strategies need be taken into account. Simple static load balancing schemes such as probabilistic or cyclic mapping methods require a large enough number of tasks to ensure reasonable load balance, but come at little to no runtime overhead, unless communication becomes significant. For the latter case, local dynamic load balancing with task migration across partition boundaries should be considered.

The various dynamic load balancing strategies come at a certain runtime cost. Centralized schemes are simpler but a central manager may become a performance bottleneck; different alternatives exist.

Once a global dynamic scheduler is in place anyway, a task-based programming approach with fork-join style execution becomes a more natural choice in comparison to the SPMD style that is still predominant for high-performance computing today. For efficient global scheduling it is however of utmost importance that the scheduler is locality-aware, in order to reduce both cache misses and data communication cost.

1.4 Chapter Summary

In this chapter we have first given a survey of several parallel programming models (message passing, shared memory, data parallelism), all expressed as instances of the generic task-based parallel programming model introduced by Foster [26].

We then introduced Foster's generic method for the stepwise design of parallel programs in the scientific computing domain. The method consists of four main steps: Partitioning, Communication and Synchronization, Agglomeration, and Mapping and Scheduling. For each step, we have considered several different design choices and techniques that can be used for common situations in solving computational problems in the high-performance computing domain.

After the partitioning and communication/synchronization steps, we obtain an abstract, textbook-style, fine-grained parallel algorithm. This abstract parallel algorithm formulation contains the essence of the parallel solution design for the considered problem. It could then be reused across a wide range of parallel target architectures, by engineering it further in the last two steps, which are more specific to properties of a given parallel target architecture (such as number of processors, availability of shared memory, or cost of communication).

So far, we considered mostly the qualitative aspects of parallel program and algorithm design. For better design choices, we need to add the quantitative aspects too, i.e., cost models and systematic analysis of the impact of different design choices on parallel execution time, scalability, and memory requirements. Chapter 2 will consider cost models and quantitative analysis in great detail.

A number of parallel algorithmic design patterns have been encountered during this chapter. The most important ones are:

- Embarassingly parallel computations
- Data-parallel computations
- Stencil computations on regular and irregular meshes
- Parallel divide-and-conquer computations
- Reduction computations
- Central-manager resp. central-server
- Pipelining
- Domain decomposition (especially multi-dimensional ones)
- Local adaptive load balancing
- Dynamic loop iteration scheduling

1.5 Bibliographical Notes *

This chapter mainly presents and elaborates on the PCAM method for stepwise parallel program design, and follows in its structure and examples largely the corresponding chapter in Foster's seminal book [26], with the notable difference that synchronization has now been taken up explicitly in the second step and scheduling in the last one. Some new aspects and examples have been added, e.g. on barriers, pipelining and scheduling, and more emphasis has been put on different parallel algorithmic design patterns.

1.6 Exercises

1. For all the examples in this chapter, explain whether non-preemptive execution of tasks would be sufficient and, where not, why preemptive execution is required.

2. Explain how the Red-Black update schema is conceptually derived from pipelining the Gauß-Seidel schema, and write pseudocode for the parallel Red-Black update schema with one task per array element.

3. For the Jacobi stencil update sweeps on large *irregular* meshes/grids, we can expect worse cache hit rates than for regular meshes/grids. Why?

4. Formulate parallel pseudocode realizing barrier synchronization using a shared counter variable.

5. Compare the stage task instance dependence graph of Figure 1.16 with the pipeline schedule of Figure 1.17 using barrier synchronization between any two subsequent rounds. Which of the two gives more flexibility for parallel execution, and why? Which one is probably easier to analyze and debug?

6. Formulate parallel pseudocode for calculating the Mandelbrot set in the interval $[r_l : r_u] \times [i_l : i_u] \subset \mathbb{C}$ as a $R \times C$ pixel image. Try various domain decomposition strategies. Which mappings and load balancing schemes are most appropriate?

7. In the work-stealing pseudocode in Figure 1.28, why is it a good idea for a task waiting at a synchronization point for a stolen child task to steal back work from the thief ("leap-frogging")?

8. More exercises TBD ...

Chapter 2

Design and Analysis of Parallel Algorithms

In this chapter, we will consider the design of parallel algorithms from a quantitative perspective. We will introduce a number of fundamental parallel computation models / cost models, such as the PRAM model, the BSP model, the Delay model, the LogP model etc. We will show how to analyze parallel programs for different performance metrics of interest, such as parallel execution time, parallel work, parallel cost, parallel speed-up, parallel efficiency, and scalability. We will also analyze upper bounds on achievable speed-up given by Amdahl's Law and Gustafssons Law. Moreover we will introduce the work-time scheduling principle and Brent's Theorem. We will also consider the impact of data locality for cache-based architectures.

The analysis of parallel algorithms can be done either at design time, based on a parallel cost model, typically using *asymptotic analysis*, or by measurements after implementation on a concrete parallel machine, i.e., *empirical analysis*. While empirical analysis is certainly more accurate, fixing performance problems at this late stage might be very costly. We will thus focus on asymptotic analysis throughout most of this chapter.

2.1 Introduction

2.1.1 Computation Model = Programming Model + Cost Model

In sequential computing, the design of computers follows since the 1940s mostly the traditional *von-Neumann model*. The von-Neumann model assumes an abstract processor with registers and with program and data memory. The processor performs a step-by-step interpretation of the instructions of a (stored) program and executing for each one the requested operation on the arithmetic logical unit, on the program counter, and/or as a single-word read or write access to data memory. Even though processor architectures have evolved considerably since then, using techniques such as pipelining, SIMD execution, superscalar instruction-level parallel processing and out-of-order execution of instructions in order to increase the throughput for single-threaded programs, the von-Neumann style interface to the (assembler-level) programmer is still being maintained today.[1] Hence, the von-Neumann model constitutes a hardware-software interface that is widely agreed on

[1]Compilers for native programming languages such as C/C++ or Fortran map directly to sequential von-Neumann style code. All so-called high-level programming languages (interpreted, declarative, functional, constraint-based, domain-specific, etc.) finally result in code that is executed by a virtual machine or solver that in turn is written in a native programming language and thus executing in von-Neumann style.

and that can thus be considered a universal computation model for sequential computing, which is of great significance for the scalable and independent production of software.

Such a universal computation model does, unfortunately, not exist in the world of parallel computing, due to the inherent architectural diversity of parallel computer architectures. Different control structures (SIMD vs. MIMD) and memory organizations (shared vs. distributed memory and hybrid forms) lead to different hardware-software interfaces that require different types of computation models.

Parallel computation models are abstractions of classes of existing or idealized parallel computer architectures, including their system software and programming toolchains, that share characteristic features so that they can be addressed by the same set of programming constructs and by the same program design and analysis mechanisms. A parallel computation model should abstract from the concrete hardware and technology of its class of parallel architectures; it should specify the basic operations and any constraints of when these are applicable, and it should specify how data can be stored on its class of parallel architectures (e.g., shared memory or distributed memory). The abstraction from concrete parallel architectures and machines allows to describe parallel algorithms in a form that is reusable across the class of all parallel machines matching the computation model, and to analyze (parallel) algorithms e.g. for their expected parallel execution time or scalability behavior on any representative of this class even *before* their (time-consuming) implementation for a concrete parallel computer. In order to keep the number of different parallel computation models reasonably low, such models must apply to a broader class of parallel machines and thus focus only on the most characteristic architectural features with respect to programming interface and influence on execution time.

(Parallel) computation models have a qualitative side (the parallel programming model) and a quantitative side (the parallel cost model).

The *parallel programming model* provides the programmer with fundamental programming constructs and mechanisms how work can be spread over multiple execution units to overlap in time, how data can be stored and communicated between execution units (e.g., shared memory, message passing), and to what degree the underlying architecture is expected to support certain synchronization structures (e.g., barrier synchronization, transactional memory, or blocking communication) naturally, such that these need not be realized in software.

A *parallel cost model* defines a few key parameters such as the number of processing units or average communication bandwidth that are necessary to predict execution times, and a set of cost functions for the basic operations provided by the programming model. Moreover, it might define constraints on how the cost functions for multiple operations or program parts can be composed in order to predict costs for larger program units.

In order to be useful in practice, cost models should be able to explain available observations (e.g., measurements of execution times after implementing an algorithm) and predict future behavior (e.g., the expected execution time for a machine twice as large). In contrast, in order to be practical for the early analysis of execution time or scalability behavior (before implementation) and the choice among different algorithmic design alternatives, a cost model should be simple, i.e., have few parameters that lead to compact formulas for parallel execution time etc. that can be discussed and compared easily. For the same simplicity argument, a parallel cost model should abstract from architectural details considered less important for the problem at hand (such as concrete costs of rarely executed operations or even cache sizes or concrete network topologies) such that the analysis can be generalized and reused over a larger class of architectures. Another important simplification comes from using *scale analysis*, which, by using big-Oh notation for the asymptotic growth of functions (see Appendix A), focuses on the behavior for large problem sizes or machine sizes only, while completely ignoring special cases and different or even unexpected behavior at small sizes.

Early in the design process we prefer simple cost models (few parameters, short formulas) for easy comparison of design alternatives, while more elaborated cost models can be used later on to provide more accurate predictions as the software design and development proceed. Sufficiently detailed cost models can then be calibrated with empirical (measured) performance data to build predictors that could also be used to guide concrete optimizations, e.g., in run-time performance tuning of programs.

2.1.2 Parallel Execution Time

Parallel execution time denotes the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution.

The time T spent by a processor i when executing a parallel program from its start to its end is composed of three parts:

- Computation time, T^i_{comp}
- Communication/Synchronization time, T^i_{comm}
- Idle time, T^i_{idle}

with

$$T = T^i = T^i_{comp} + T^i_{comm} + T^i_{idle}$$

Communication time overhead is time spent in communication library routines excluding waiting time, for instance, the overhead for sending off messages or for receiving messages that have arrived. Synchronization time overhead (for shared-memory systems) is the overhead for synchronization, e.g. the time for acquiring an available lock variable and releasing it. However, all parallelism-related waiting (e.g., for an expected message to arrive, or for a lock variable to become free) contribute instead to idle time. In the world of sequential computing, these two time components do not exist, because a single processor executing everything never needs to wait for itself nor send itself a message, for example. Communication/synchronization time overhead and idle time add inefficiencies to a program execution, because only the computation time finally contributes to the result (the rest is spent on administration to enable proper parallel execution).

If we accumulate over all processing elements $T_{comm} = \sum_i T^i_{comm}$ and $T_{idle} = \sum_i T^i_{idle}$, and compare the accumulated parallel overheads $T_{comm} + T_{idle}$ to the accumulated *computational work* $T_{comp} = \sum_i T^i_{comm}$, we obtain a measure for how well the parallel program utilizes the resources of the parallel machine. The goal of designing efficient parallel algorithms must thus be to keep T_{comp} and T_{idle} as low as possible. For instance, idle time can often be reduced by proper load balancing and/or by reducing the degree of parallelism.

2.1.3 Towards Parallel Computation Models

The probably simplest (explicitly) parallel computation model is the so-called *PRAM* (*Parallel Random Access Machine*) [25]. It is a direct extension of the *RAM* (*Random Access Machine*) model for sequential computing, which combines the von-Neumann model with a simple cost model and which forms, since many decades, the formal basis for the time complexity analysis of sequential algorithms.

We therefore first recapitulate the RAM model in Section 2.2 before introducing the PRAM model in Section 2.3.1. More detailed parallel computation models that consider more features of real-world parallel computer architectures will follow in Section 2.3.

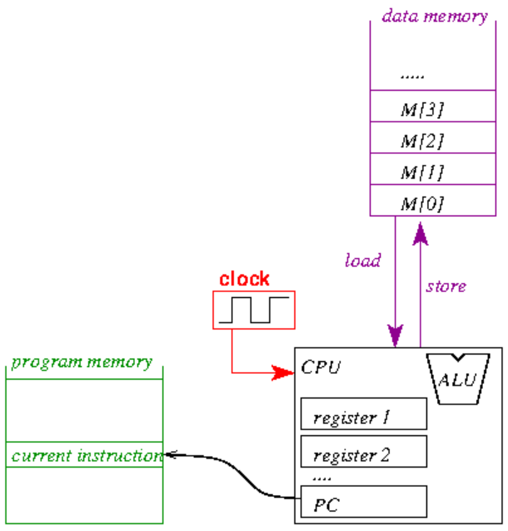


Figure 2.1: The RAM (Random Access Machine) model of an abstract sequential computer.

2.2 Recapitulation: The Random Access Machine (RAM) Model

The Random Access Machine (RAM), see also Figure 2.1 for an illustration, denotes a RISC-like cache-free single-processor register machine with an arbitrarily large data memory and a program memory. The processor consists of an arithmetic-logical unit, a program counter and an arbitrarily large number of registers; it can perform one instruction in each time step (clock cycle), either an arithmetic operation on registers, or a memory access (Load or Store), or a branch operation. Memory accesses have a latency of only one time step, i.e., the result of a Load instruction is available at the beginning of the next time step. This idealistic property of the RAM is not true on most modern processors, where an off-chip memory access can take 100 to 1000 times longer than an elementary arithmetic operation on registers. However, the simplicity of the RAM model with uniform cost one for each instruction execution, ignoring current cache contents etc., makes the analysis of algorithms and the relative comparison of different algorithms much easier: The execution time equals the *number* of executed instructions.

2.2.1 Example: Global Sum on a RAM

As an example for analyzing algorithms for the RAM model, we consider computing the global sum of N elements. Figure 2.2 shows the sequential pseudocode and a corresponding sequence of RAM pseudoinstructions.

As an expression in N, the execution time is

$$t = t_{load} + t_{store} + \sum_{i=2}^{N} (2t_{load} + t_{add} + t_{store} + t_{branch})$$

which, as the RAM model uniformly charges time $t_{load} = t_{store} = t_{add} = ... = 1$, becomes

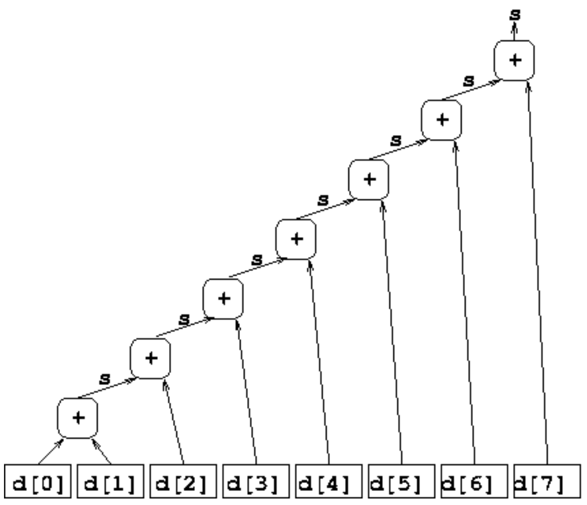
$$t = 5N - 3 \in \Theta(N)$$

Of course, variable `s` could have been allocated to a register throughout all the computation, eliminating one `load` and one `store` instruction in the loop body. With this optimization we could

```
function SeqSum ( int $d[0:N-1]$ )
{
  int $s \leftarrow d[0]$;
  for $i = 1, ..., N$ {
    $s \leftarrow s + d[i]$;
}
```

```
  load d[0] into register R1;
  store register R1 to s;
  setconst R2 to #1;  // i
Label L1:
  branch if R2 >= #N to L2;
  load s into R3;
  load d[R1] into R4;
  add R3, R4 into R5;
  store R5 to s;
  add R2, #1 into R2;
  branch to L1; // loop
Label L2:
```

Figure 2.2: SeqSum pseudocode and the corresponding RAM pseudoinstructions for computing the global sum of N elements in an array d on a RAM.



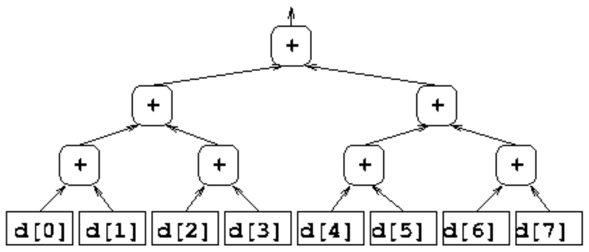


Figure 2.3: The data flow graphs of two algorithms for array sum computation. Left: the data flow graph corresponding to the RAM algorithm SeqSum of Section 2.2. Right: the data flow graph corresponding to the PRAM divide-and-conquer based algorithm ParSum of Section 2.3.1.4.

do it in

$$t = 3N - 3 \in \Theta(N)$$

i.e., still in time linear in N. The asymptotic notation abstracts from the concrete constant coefficient values, and hence the expression $\Theta(N)$ remains valid also if such optimizations are applied or if e.g. load or store instructions are charged a higher (but still constant) cost.

A *data flow graph* is an abstract representation of a computation's operations (vertices) and their precedence constraints due to data flow (directed edges) in the form of a *directed acyclic graph* (DAG). An edge (u, v) implies that the value produced (written to a register or memory location) by operation u is consumed (read) by operation v. The data flow graph is yet another computation model that abstracts from the actual computation technology, such as the register machine in the RAM model; for instance, the same data flow graph could even be realized completely or partly in hardware as an arithmetic circuit. For that reason, the data flow graph model is also known as the *arithmetic circuit model* or *DAG model* in the literature.

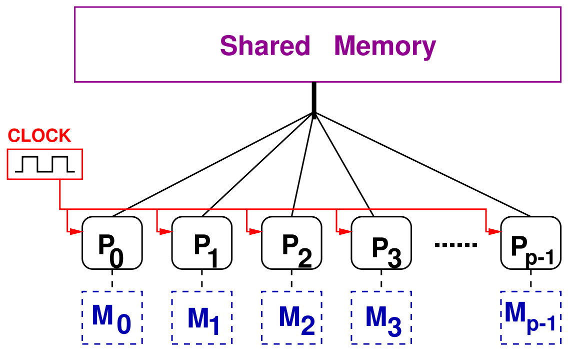


Figure 2.4: Schematic diagram of a PRAM.

Figure 2.3 (left) shows the data flow graph of our example RAM algorithm for array sum. The linear chain of dependences from one loop iteration to the next one, caused by the partial sum value being handed over in accumulator variable `s`, makes the problem look inherently sequential, i.e., not a good match for a PRAM. However, we will soon see that it can be parallelized quite well, if a different algorithmic approach is chosen.

2.3 Some Parallel Computation Models

2.3.1 The Parallel Random Access Machine (PRAM) Model

2.3.1.1 The PRAM Model

The *Parallel Random Access Machine* (PRAM) denotes an abstract shared-memory MIMD-parallel computer with an arbitrary number of p processors (p is an algorithm design parameter and the only parameter used in the PRAM cost model, which is one of the main reasons for its simplicity).

Figure 2.4 shows a visualization of the PRAM memory structure. The shared memory is assumed to be sequentially consistent[2]. PRAM processors might have private memory sections in the shared memory, e.g. for storing thread-private variables or procedure call frames. The PRAM has no caches, all memory accesses go directly to shared memory, and all memory accesses take one unit of time (one clock cycle).

Each PRAM processor executes one instruction (arithmetic operation, memory access or branch) in each clock cycle. All processors are fed the same global clock signal, and the PRAM offers natural (hardware-guaranteed) synchronization after *every* executed clock cycle, so that all PRAM processors execute instructions in lock-step mode.

2.3.1.2 Discussion: Simplifications Made by the PRAM Model

The PRAM model has often been criticized for being unrealistic due to its oversimplification. The largest criticism concerns the uniform and single-cycle memory access time, regardless of which and how many other PRAM processors access the same or other memory locations at the very same time. On all real-world machines the memory access bandwidth (i.e., the number of memory

[2]In fact, an even stronger property holds, namely, *strict consistency*. As the PRAM processors proceed in lock-step with a common clock signal, all their memory accesses will (for all but the Arbitrary CRCW variant) take effect on memory in the *same* relative order for repeated program runs. Such deterministic execution is not generally guaranteed by sequential consistency.

accesses from processors to shared memory that can be handled per unit of time) is always bounded by a finite value, which can be higher or lower depending on the used technology and architectural techniques.

Worse yet, on real-world machines the memory access latencies usually can vary a lot, depending e.g. on the current cache contents, which adds difficulties e.g. for the static analysis of programs for real-time constraints; to some degree this aspect also applies to the RAM model already. In other words, the PRAM relieves the early algorithm designer to worry about data locality, which is of high significance for performance in practice. And if the unit-time memory access latency assumption does not hold any more, then lock-step synchronization is no longer realistic either, unless we would set an extremely low clock speed to make all operations equally slow.

The natural-synchronous execution feature is in stark contrast to all asynchronous parallel computation models and in particular to all thread-based parallel computations on real-world parallel architectures. In those cases we can not make any assumption at all about the relative speed of different threads, for example because interrupts of individual cores may happen at any time and because operations such as memory accesses or branches might take very different time depending on cache contents and its coherence state, on TLB contents etc.). On such systems, we need to protect potential access conflicts (e.g., with respect to the relative ordering of parallel read and write accesses to shared variables) by explicit synchronization constructs such as mutual exclusion or barriers. Instead, the PRAM guarantees to progress by exactly one step on each processor at every clock cycle. This strong property allows for deterministic parallel computation and removes the need for many (though, in general, not all) explicit synchronizations, leading to simpler (pseudo-)code.

Finally, any real-world parallel machine has a finite number of processors, while PRAM algorithms can assume arbitrarily large numbers p of PRAM processors, so that p could adapt and scale e.g. with the size of the problem instance to be computed. This requires later either an "engineering" of the PRAM algorithm to make it work with any fixed number of processors, or using a task-based runtime system that multiplexes an arbitrary number of virtual PRAM processors (threads) to a fixed number of processors, at the expense of runtime overhead for task management. In other words, the assumption of an unbounded number of PRAM processors relieves the early algorithm designer from the problem and possibly overhead of *mapping and scheduling.*

Summarizing, the most severe simplifications of the PRAM model are:

- unbounded number of processors;
- all operations take 1 unit of time so cycle-by-cycle lock-step MIMD parallel execution is possible;
- unbounded memory access bandwidth and unit-time memory access.

So why do we even consider the PRAM model? Just because of its simplicity. It allows us to study the fundamental, inherent parallelism in an algorithmic solution. The focus on fine-grained parallelism and unlimited resources provides us with an upper bound of the speedup that we possibly could expect on ideal hardware. It also allows us to quickly assess the asymptotic execution time, work, speedup etc. as simple formulas, so we can quite easily compare different algorithmic designs relatively to each other, and discard algorithms that are inferior already at this stage:

An algorithm that does not scale under the PRAM model
will not scale under any other, more realistic, parallel computation model.

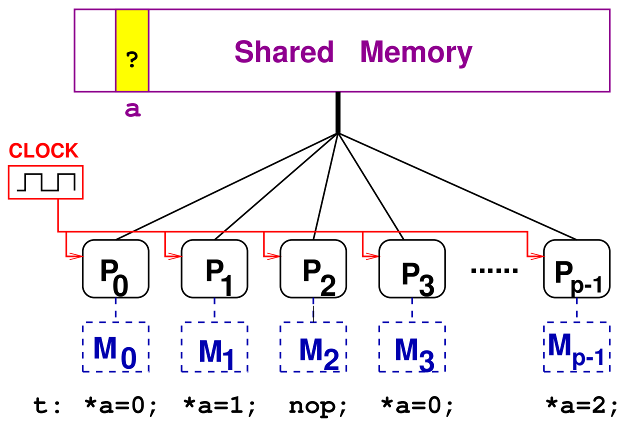


Figure 2.5: Example of simultaneous (concurrent) access by multiple PRAM processors to the same memory location a.

Therefore we use the PRAM model as a first (and only first) step in the design and analysis of parallel algorithms. In subsequent steps we will complement it by increasingly more realistic models such as BSP, LogP etc. to be described later.

2.3.1.3 PRAM Model Variants for Memory Access Conflict Resolution

The PRAM allows up to p shared memory accesses to happen simultaneously in the same clock cycle. We need to define what happens in the case of access conflicts, i.e., if several of these simultaneous accesses (load, store, or even both) go to the same memory location. See Figure 2.5 for a simple example with simultaneous ("concurrent") write accesses to the same shared memory location.

A number of variants of the PRAM model have been introduced in the literature that differ by the constraints on simultaneous memory location accesses and the methods to resolve such conflicts in "hardware". Generally we can say that PRAM variants with strong resolution mechanisms are more powerful in that they can solve certain problems asymptotically faster than others, while the variants with more constraints and weaker resolution mechanisms are (somewhat) closer to real-world parallel computers. Common to all variants is that simultaneous read *and* write accesses to the same location are not allowed and need be prevented by proper algorithm design. At any clock cycle, a memory location is thus either the target of only load instructions or only of store instructions, but not of both.

The *Exclusive Read, Exclusive Write (EREW) PRAM* allows simultaneous accesses to the same location (also referred to as "concurrent access" in the PRAM literature) only to *different* locations in the same cycle.

The *Concurrent Read, Exclusive Write (CREW) PRAM* allows either simultaneous reading by multiple PRAM processors from the same location *or* exclusive writing by one PRAM processor to it.

The *Concurrent Read, Concurrent Write (CRCW) PRAM* allows either simultaneous reading from *or* simultaneous writing to the same location. For the conflict resolution, the following sub-variants have been defined:

- *Weak CRCW PRAM*: For concurrent writes, only a pre-defined constant value, such as 0, can be written.

- *Common CRCW PRAM*: Concurrent writes need to write the *same* value.

- *Arbitrary CRCW PRAM*: Concurrent writes commit in an arbitrary (implementation defined) order, all but the last value written are lost. This feature is sometimes combined with randomization to allow a random PRAM processor to "win", i.e., commit last, among all such writers, with uniform probability distribution.

- *Priority CRCW PRAM*: The writer with the highest rank (thread resp. processor ID) wins and writes, the other values are lost.

- *Combining CRCW PRAM*: No value to be written is lost. All values written to the same location in the same clock cycle are automatically accumulated in that location (in a single clock cycle!) using some predefined binary associative and commutative function, such as addition ($\rightarrow$ global sum), maximum ($\rightarrow$ global maximum), minimum, etc.

There is no need for ERCW in the literature, so it does not exist.

In the above description, the PRAM variants are indeed ordered by increasing power and flexibility: Every algorithm that can be executed by a certain PRAM variant in a certain asymptotic parallel time can also be executed by all stronger models, with at least the same time complexity or better. The Combining CRCW PRAM is certainly the strongest model. In fact, it could add up to p numbers in a single clock cycle, just by having every processor concurrently write its value there. The weakest PRAM variant is the EREW PRAM model. Hence, when designing PRAM algorithms it is important to state the model variant assumed for it. Algorithm designers try to specify their algorithm for the weakest PRAM variant that still supports a certain asymptotic complexity, it will then also work at least as good for all stronger variants.

2.3.1.4 Example: Global Sum on a EREW-PRAM

As an example for PRAM algorithm design and analysis, we consider the problem of computing the global sum $\sum_{i=0}^{n-1} x_i$ of n numbers $x_0, x_1, ..., x_{n-1}$ stored in an array. For simplification of the following discussion, we assume for now that the problem size n is a power of 2. We will see that, on an EREW PRAM with n processors, the global sum $\sum_{i=0}^{n-1} x_i$ can be computed in $\lceil \log_2 n \rceil$ time steps.

When discussing the data flow graph of the sequential sum algorithm SeqSum above, it became already clear that SeqSum cannot be directly converted into a parallel algorithm by, e.g., just parallelizing a loop, because of the cross-iteration data dependences. Hence, we must design a new parallel algorithm, ParSum, from scratch.

For the design of ParSum we use the well-known algorithmic design pattern *Divide-and-Conquer*. Divide-and-Conquer solves a large problem instance by dividing it into smaller *independent* subproblems, solving these either recursively or, if already small enough, directly with a baseline algorithm, and finally combines the subproblem solutions into a solution for the original problem.

Applying this to ParSum, we exploit the *associativity* of addition[3], in particular, that

[3]Addition is not necessarily associative if considering floatingpoint numbers with a representation of limited precision. In that case, the result might differ between SeqSum and ParSum, usually only in an insignificant way, due to different roundoff errors as partial sums are combined differently in the two algorithms.

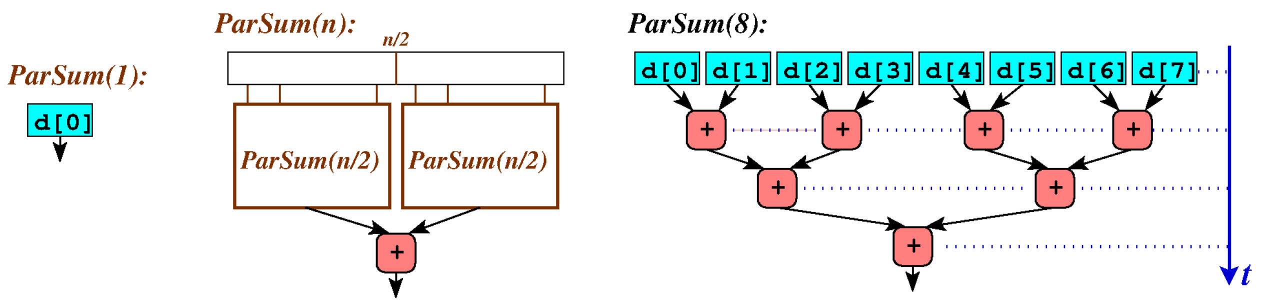


Figure 2.6: The ParSum algorithm idea: Applying the parallel divide-and-conquer pattern to the global sum problem.

$$\sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n/2-1} x_i + \sum_{i=n/2}^{n-1} x_i$$

This allows us to divide the problem in half, then solve two independent problems of size $n/2$ recursively with a size-one subproblem as base case (solved by an ordinary load of an array element), and combine the two subsolutions (i.e., partial sums) by simply adding them up.

While this actually does not change the overall amount of computational work to do, the structure of the computation (such as the relative order of operations) has changed in comparison to SeqSum.

A first way to gain parallelism is by exploiting the fact that the subproblem instances of a problem instance have to be independent for divide-and-conquer computations, so they can be computed in parallel on different processors. This leads to the algorithmic design pattern variant *Parallel Divide-and-Conquer*. Figure 2.6 (left) shows the data-flow graph template for the general case, and Figure 2.6 (right) shows the unfolded data flow graph for the case $n = 8$.

For the analysis of ParSum, let $T(n)$ denote the parallel time for ParSum given a problem of size n. $T(n)$ includes time components for the following phases:

- Divide phase: trivial (one pointer computation suffices to determine the middle of the array section corresponding to the problem instance), this takes time $O(1)$;
- Recursive calls: parallel time $T(n/2)$ (the workload is balanced if both subproblems have equal size $n/2$, so their maximum is just $T(n/2)$);
- The base case: basically an ordinary load operation, time $O(1)$;
- Combine phase: an addition of two partial sums, which takes time $O(1)$.

Plus some constant-time overhead for the function call and checking for the base case.

This yields the following recurrence equation system describing $T(n)$:

$$\begin{aligned} T(n) &= T(n/2) + O(1) \quad \text{for } n > 1 \\ T(1) &= O(1) \end{aligned}$$

This recurrence equation system has the solution

$$T(n) \in \Theta(\log n)$$

```
// parsum, recursive formulation in Fork:

sync int ParSum( sh int *d, sh int n)
{
  // $ is the thread's rank in [0...n-1]
  sh int s1, s2, s; // group-wide shared variables
  if (n==1)
     return d[0]; // base case of recursion
  if ($ < n/2)    // split processor group into 2 parallel subgroups:
     s1 = ParSum( d, n/2 );
  else                      s2 = ParSum( d+n/2, n-n/2 );
  // subgroups merge here again, implicit barrier synchronization
  seq
     s = s1 + s2; // by thread 0 only
  return s;
}

int main()
{
 ... // create n threads and enter PRAM mode
 sh int s;  // variable for global sum;
 ... // read n, d
 s = ParSum( d, n );
 ... // output s
}
```

Figure 2.7: Recursive formulation of the ParSum algorithm in the PRAM programming language Fork [37].

which we could either prove by induction or look up from the generic recurrence problem solution given by the so-called master theorem (see e.g. the textbook by Cormen *et al.* [17, Ch. 4]).

The ParSum algorithm idea can be formulated in various ways in concrete parallel programming languages, and we will show a few examples in the following, but all formulations will finally result in the same dataflow graph (Figure 2.10), i.e., the same fundamental operations (i.e., work) are performed with the same dependence structure in the form of a balanced binary tree.

In the PRAM programming language *Fork* [37], a recursive formulation closely following the design above would look as displayed in Figure 2.7. It assumes a group of n threads, each running on its own PRAM processor, where the built-in variable `$` denotes the thread's rank in $\{0, ..., n-1\}$. Initially, a group consisting of all threads simultaneously invokes `ParSum` for the overall problem instance. In the base case, groups consist of one thread only, which performs the memory access to "its" element of `d`. In the recursive case, the group is, like the work load, split in half, where the first subgroup takes care of the first recursive call and the second subgroup of the other one. When both subgroups have finished (which they should do approximately simultaneously as their workload and sizes were balanced) there is an implicit barrier synchronization to finalize the two-sided `if` statement with its group splitting, after which the parent group resumes and its first thread (rank 0) adds up the partial results.

Alternatively, the ParSum algorithm could be formulated in the task-based language *Cilk*, as

```
// ParSum, recursive formulation in Cilk:

cilk int ParSum ( int *d, int from, int to )
{
   int mid, sumleft, sumright;
   if (from == to)
      return d[from];    // base case
   else {
      mid = (from + to) / 2;
      sumleft = spawn ParSum ( d, from, mid );
      sumright = ParSum( d, mid+1, to );
      sync;
      return sumleft + sumright;
   }
}

main()
{
  ...
  ParSum ( data, 0, n-1 );
  ...
}
```

Figure 2.8: Recursive formulation of the ParSum algorithm in the task-based programming language Cilk [48].

shown in Figure 2.8. A call to a Cilk task with parameters `d`, `from` and `to` will calculate the sum of elements in the (shared) array `d` between positions `from` and `to`.

Note that only the first recursive call is spawned off as a separate task that could be executed by a different worker thread. Of course it would likewise be possible to also spawn a task for the second recursive call. However, then the parent task would then have nothing more to do and just wait at `sync` for the two child tasks to terminate (its worker would pause it and put it into a waiting queue until the child tasks have terminated so the parent task becomes ready again and can resume as soon as this or other worker thread is ready to take it. The "inlining" of this second task is an optimization that tries to reduce the overall number of tasks and thereby the overall task managment and scheduling overhead.

Figure 2.9 shows how a recursive formulation of ParSum could be formulated in the task-based programming model of OpenMP (version 3 and later).

The same resulting data flow graph in Figure 2.10 can also be achieved by formulating the ParSum algorithm in a non-recursive, i.e. iterative, way. In the PRAM programming language Fork [37], this could be expressed for a group of n threads (each running on its own PRAM processor) as shown in Figure 2.11. The set of active threads is controlled by the `while` loop, which doubles the distance `d` between the still active threads (fulfilling the `if` condition). This iterative algorithmic technique of in-place updating with repeatedly doubling the access distance is also known as *recursive doubling* and is thus closely related with parallel divide and-conquer. We will later see further applications of the recursive doubling technique.

```
// ParSum, recursive formulation in OpenMP3+:

int parsum ( int *d, int from, int to )
{
   int mid, sumleft, sumright;
   if (from == to)
      return d[from];    // base case
   else {
      mid = (from + to) / 2;
#pragma omp task shared(sumleft)
      sumleft = parsum ( d, from, mid );
#pragma omp task shared(sumright)
      sumright = parsum( d, mid+1, to );
#pragma omp taskwait
      return sumleft + sumright;
   }
}

main()
{
#pragma omp parallel
 {
#pragma omp single
   parsum ( data, 0, n-1 );
 }
}
```

Figure 2.9: Recursive formulation of the ParSum algorithm using the task-oriented constructs of OpenMP 3.0 and later.

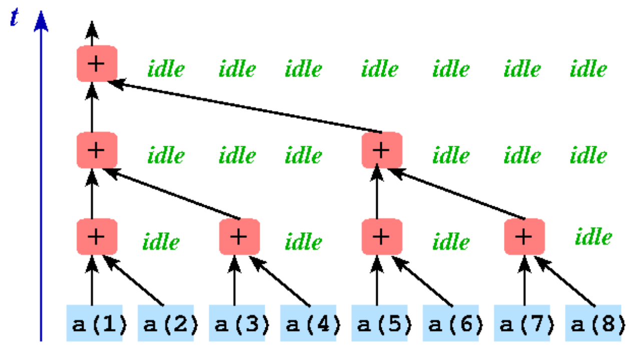


Figure 2.10: The data flow graph resulting from the ParSum algorithm applied to $p = n = 8$ PRAM processors.

```
int ParSum_iterative ( sh int a[], sh int n )
{
 int d, dd;
 int ID = $; // $ is the thread rank in 0...n-1
 d = 1;
 while (d<n) {
    dd = d;
    d = d*2;
    if (ID%d==0) // d divides ID:
       a[ID] = a[ID] + a[ID+dd];
 } // implicit barrier synchronization between iterations
}
```

Figure 2.11: Iterative formulation of ParSum in the PRAM language Fork [37], and the resulting data flow graph.

2.3.1.5 Example: Global Sum on a Combining-CRCW-PRAM

On a Combining CRCW PRAM with addition as the combining operation, the global sum problem can be solved in a constant number of time steps using n processors: Each processor i reads its data element `a[`i`]` and writes it simultaneously with the others to the shared memory location of the sum accumulator variable:

```
sh int s;
pr int i = ID; // processor rank ID in 0...n-1
s = a[i];      // concurrent write, combining by addition
```

While the memory access interface for a Combining CRCW PRAM would be very expensive to realize in hardware, we take up this example as an illustration of the fact that different PRAM models can lead to different asymptotic time complexities for the same problem (here, logarithmic time for EREW PRAM vs. constant time for Combining CRCW PRAM). Hence, it is always important when describing a parallel algorithm to state what PRAM model variant (i.e., requirements on hardware support for memory accesses) is assumed. For more examples of such separation results between different PRAM variants we refer to [37] Section 2.5.

2.3.2 Delay model

The Delay model charges a cost for the communication in an otherwise idealized multicomputer: Point-to-point communication of a block of n data words costs time

$$t_{msg}(n) = t_s + nt_w$$

see also Figure 2.12.

Here, the *communication startup time* t_s includes:

- the software overhead on the sender side (e.g., kernel entry for an I/O call, copying parameters or even the entire data block to a system buffer, and initiating the sending);
- the actual network latency (accumulated message routing delays and signal propagation delays, which is usually proportional to the network distance of sender and receiver), counted

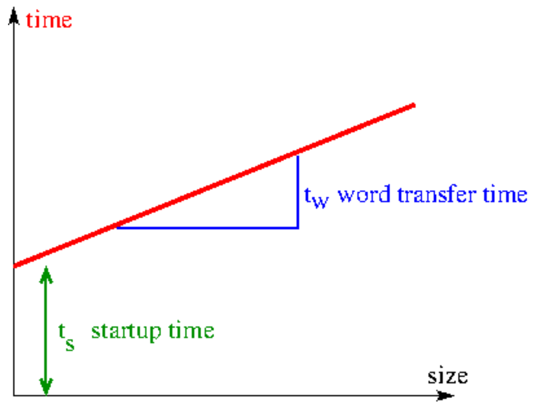


Figure 2.12: The linear cost function for point-to-point message transfer, as assumed in the Delay model

from when the first byte of the message leaves the sender node until the first byte arrives at the receiver node;

- the software overhead on the receiver side (e.g., kernel entry for serving an I/O interrupt, copying the data block to the memory of the receiving process, and initiating the sending.

The communication startup time t_s thus accounts for all hardware and software overheads and can be interpreted as the time $t_{msg}(0)$ for communicating a zero-byte message.

The *word transfer time* t_w is the reciprocal of the end-to-end (e.g., at MPI level) communication bandwidth and denotes the pure transfer time per word sent, not including the communication startup time. It depends on the network bandwidth and in most cases also on the node architecture, e.g. on the speed of the I/O interface and the local copying of data.

The values of startup time and word transfer time depend strongly on the interconnection network architecture and on the system software layers on top. In interconnection networks for clusters, the word transfer time is for modern high-speed networks about one order of magnitude below one nanosecond[4], while the startup time can, even if high-speed networks are used, be in the order of a microsecond[5] or even higher, i.e., the startup time matches the data transfer time for thousands of bytes.

Note that the Delay model still assumes that the network is not overloaded; for instance, it assumes that there are no delays due to conflicts at routing due to multiple messages wanting to travel along the same network link at the very same time.

2.3.2.1 Example: Communication Cost in the Parallel Sum Algorithm

Consider the data flow in ParSum between data-producing and data-consuming operations in Figure 2.10. If executing the algorithm on a message passing architecture, cross-processor dependences will result in communication, as shown in Figure 2.21 (right). The critical path now involves $\log_2 n$ cross-processor dependences. Using the Delay model, the time complexity of the n-processor PRAM algorithm ParSum is augmented by the communication cost

$$t_{comm}(n, n) = \log_2 n \cdot (t_s + t_w)$$

[4]On NSC Triolith (Infiniband network), the MPI point-to-point communication bandwidth is 7 GB/s.
[5]On NSC Triolith (Infiniband network), the startup time at MPI level is 1μs.

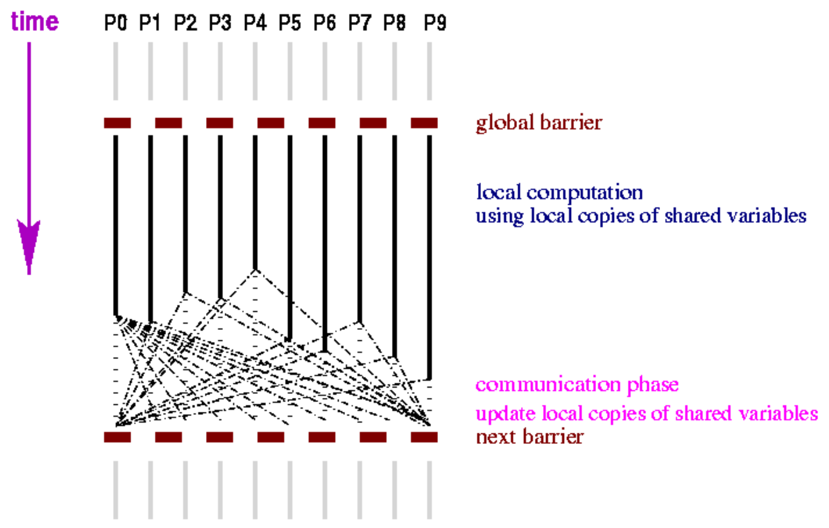

time
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9
global barrier
local computation
using local copies of shared variables
communication phase
update local copies of shared variables
next barrier

Figure 2.13: A BSP superstep executed by a BSP machine with 10 processors.

to account for the accumulated delay of these $\log_2 n$ single-word messages.

2.3.3 Bulk-Synchronous Parallel (BSP) Model

The *bulk-synchronous parallel* (BSP) computation model was introduced in 1990 by Valiant [61] to bridge the abstraction gap between the convenient but quite unrealistic PRAM model on one side and the real-world parallel computers on the other side.

A *BSP computer* is an abstract message passing architecture characterized by four cost model parameters, (p, L, g, s). As with the PRAM model, p denotes the number of processors. L is the overhead time for barrier-synchronizing all processors. g is the normalized network bandwidth for single-word messages, i.e., a message containing 1 word sent from a processor at time t reaches the receiver at time $t+g$. Finally, s denotes the processor's speed, i.e., arithmetic operations performed per unit of time. The s parameter is often abstracted away by normalizing the other parameters in s so that the unit of time becomes 1, as with the PRAM model. In general, L and g are not constants but functions that depend on p to allow for modeling scalable BSP machines.

A BSP computer is a MIMD-parallel computer with distributed memory. Program execution follows the SPMD principle, i.e., a fixed set of processors execute one process each from the beginning of the parallel program execution, each invoking one copy of the program starting at `main()`.

For an easier analysis of algorithms for BSP computers, the BSP model introduces constraints on the structure of computations, i.e., BSP programs are less flexible than PRAM programs; however that is often not a problem for (MPI-)parallel computations in science and engineering which in many cases match this structure.

Specifically, BSP programs are organized in so-called *supersteps*. A superstep starts with a (conceptual) global *barrier synchronization*, hence all BSP processors will start execution of the same superstep together. It follows a *local computation phase* where only locally available data can be accessed on each BSP processor. Once done with it, each BSP processor enters a *global communication phase*, where it first sends off single-word sized messages to zero or more other BSP processors using a `BSP_Send` construct similar to `MPI_Send`, and then receives such messages from zero or more other BSP processors using a `BSP_Recv` construct (similar to `MPI_Recv`). (There also exist variants of the one-sided communication primitives `Get` and `Put` in implementations of the BSP model, such as BSPlib). The superstep ends with the barrier synchronization that marks the beginning of the next superstep. Once that barrier is passed, we can be sure that all pending messages of the communication phase of the preceding superstep are finished and all communicated data has been written into its destination buffers. As the barrier synchronization point guarantees the arrival of all messages of the preceding superstep, the BSP communication primitives could be realized as non-blocking ones, with joint synchronizations at the barrier point.

For the current superstep, let w_i denote the (worst-case) work done by BSP processor i in the local computation phase of that superstep. Then,

$$w = \max_{1 \leq i \leq p} w_i$$

denotes the maximum computation workload of any BSP processor in this superstep, i.e., the performance bottleneck in the computation phase.

The pattern and worst-case volume of a superstep's global communication phase is abstracted by the so-called h-relation. For the current superstep, let h_i denote the (worst-case) total number of words sent and received by BSP processor i in the superstep[6]. Then,

$$h = \max_{1 \leq i \leq p} h_i$$

denotes the maximum communication workload of any BSP processor in this superstep, i.e., the performance bottleneck in the communication phase.

Overall, the worst-case parallel execution time of a specific superstep *step* is estimated in the BSP cost model as

$$t_{step} = w_{step} + h_{step}\ g + L$$

where w_{step} denotes the worst-case work w for *step* and h_{step} denotes the worst-case communication volume h for *step*.

A BSP program is a sequence (trace) of supersteps executed, all separated by (logical) barriers. Hence, analyzing the cost of BSP programs is straightforward: by summing up over the t_{step} costs of each superstep execution.

2.3.3.1 Example: Global Maximum Computation (Non-Optimal Algorithm)

As a simple example, we consider the problem of computing the maximum of n numbers $A[0, ..., n-1]$ on a BSP machine with cost model parameters (p, L, g, s).

For simplicity, we use here a non-optimal algorithm that follows the central-server pattern (here, the central server role is taken by BSP processor 0). Figure 2.14 (left) shows the pseudocode, which consists of two subsequent supersteps. In the computation phase of the first superstep each BSP processor calculates the local maximum within its partition into its local variable m. In the communication phase, BSP processor 0 (acting as the central server here) collects messages with the local maxima from all other processors and writes them into a temporary array of partial maxima m_i, $i = 1, ..., p-1$. When passing the barrier synchronization after step 1, we know that all m_i values are properly written. In the second superstep, only processor 0 is active and maximizes over the m_i, including its own local contribution m.

[6]Note that, differently from the Delay model, the BSP model does not model the effect of different sizes of messages, i.e., the impact of message startup time. In other words, sending one large message of K words from BSP processor i to j is charged the same communication time as for K single-word messages. This is a simplification made in order to reduce the number of communication relevant parameters in the BSP cost model to basically one, namely, g.

```
// $A[0..n-1]$ distributed block-wise across $p$ processors
step {
  // local computation phase:
  $m \leftarrow -\infty$;
  for all $A[i]$ in my local partition of $A$ {
    $m \leftarrow \max(m, A[i])$;
  // communication phase:
  if myPID $\neq 0$
    BSP_send ( $m$, 0 );
  else    // on $P_0$:
    for each $i \in \{1, ..., p-1\}$
      BSP_recv ( $m_i$, $i$ );
}
step {
  if myPID $= 0$
    for each $i \in \{1, ..., p-1\}$
      $m \leftarrow \max(m, m_i)$;
}
```

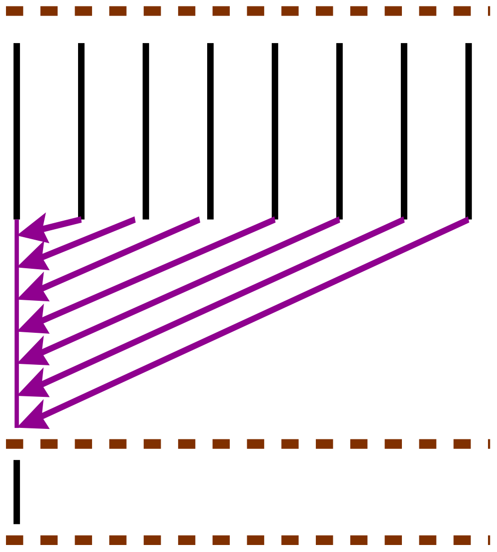


Figure 2.14: Left: A simple non-optimal BSP algorithm for global maximum computation. Right: Communication flow of the algorithm with 8 BSP processors. Time flows downwards, local computations are shown in black and message flow is shown by purple arrows. Barrier synchronizations are shown by dashed horizontal lines.

Figure 2.14 (right) shows the resulting communication flow of the algorithm executed on a BSP machine with 8 processors. It is easy to see that processor 0, here acting as the central server, becomes the communication bottleneck and a problem for scaling to large p. This can also be verified using the BSP cost model:

In the first superstep, each BSP processor performs work $\Theta(n/p)$. The communication phase has a worst-case maximum volume of $h = p - 1$ (received values by processor 0). Superstep 2 only consists of local computation work of $\Theta(p)$, as it locally maximizes over p values. Adding the cost components for both supersteps, we obtain

$$t(n,p) \;=\; w + hg + L \;\in\; \Theta\left(\frac{n}{p} + pg + L\right)$$

For $p \in o(n/p)$, i.e., $p \in o(\sqrt{n})$, the first term (computational work) is (asymptotically) dominating over the cost of communication and synchronization. As p grows larger ($p \in \Omega(\sqrt{n})$), the second term, which is linear in p, starts to dominate. Communication time breaks even (asymptotically) with computation time at $p \in \Theta(\sqrt{n})$, hence we will find a speedup saturation there and slowdown if increasing p further (see also Sect. 2.5.7).

2.3.4 LogP model

The *LogP* model [18] is a cost model for two-sided message passing (e.g., using MPI) programs, assuming that small constant-size messages (i.e., a few bytes) are communicated. (A generalization for longer messages will follow later). It generalizes over the Delay model (Sect. 2.3.2) by a more detailed modeling of the three main factors impacting message delays: the node-level software overhead (parameter o), the network latency (parameter L), and the communication bandwidth

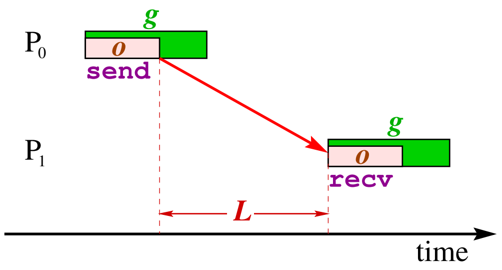


Figure 2.15: The communication cost model parameters L, o and g of the LogP model. In this example, we have $o < g$.

(parameter g). The fourth cost model parameter in LogP is, of course, the number of processors[7] (P), here, nodes in a message passing system.

Figure 2.15 illustrates the four cost model parameters L, o, g and P.

The *latency* parameter L models the network delay of a single-word message from the time of the `send` routine being done with its work to the earliest point in time when the `recv` routine could start receiving the message upon arrival.

The overhead o models the software overheads of `send` and `recv` routines (including called subroutines and system calls). The LogP model assumes that these overheads are symmetric, i.e., the total software overhead is $2o$.

The *gap* parameter g models the inverse of the bandwidth, i.e., the minimum time after which each node can inject or fetch the next message from its network interface so that messages from / to all nodes can be transported without overloading the network. The network capacity per processor is L/g messages to or from each processor.

As with the BSP model, the LogP cost model parameters (L, o, g) are typically expressed as multiples of the CPU cycle time, so that there is no need for a clock speed parameter.

The LogP model still abstracts from the network topology: the parameter L and g only model the worst-case or average-case timing behavior over all possible pairs of communicating nodes, assuming every node communicates with some other node at the same time, but the model makes no difference yet with respect to how ”close” these nodes are to each other in terms of network links.

The transmission time for a small message in the LogP model is

$$2 \cdot o + L$$

as long as the network capacity is not exceeded, i.e., as $g \leq o$. If $g > o$, it is the network capacity and not the software overhead that forms the communication bottleneck. In that case, we need to pay attention to the timing of messages. An example is given below.

2.3.4.1 Example: Broadcasting a Value

As a simple example we consider the problem of broadcasting a single-word value from one node (P0) to all other nodes on a 2-dimensional hypercube. An example for $P = 4$ is shown in Figure 2.16, where broadcasting is done by $P0$ first sending to $P1$ then $P2$, and $P2$ forwarding to $P3$, which results in an overall execution time (makespan) of $18\mu s$ for the given LogP cost parameter values $o = 2\mu s,\;\; g = 3\mu s,\;\; L = 5\mu s$.

[7]The use of the term ”processors” for ”nodes” in the LogP literature is because LogP was introduced at a time (1993) where nodes usually had just one single processor.

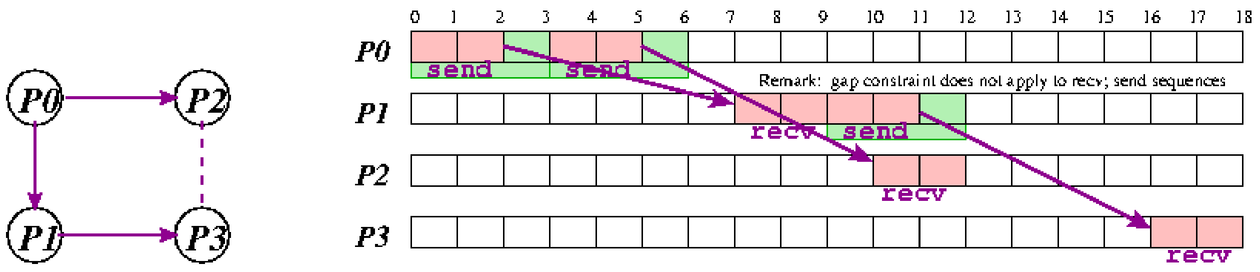


Figure 2.16: Broadcasting a single-word value on a 4-node hypercube (left) and the schedule resulting for LogP example parameters $P = 4$, $o = 2\mu s$, $g = 3\mu s$, $L = 5\mu s$ (right).

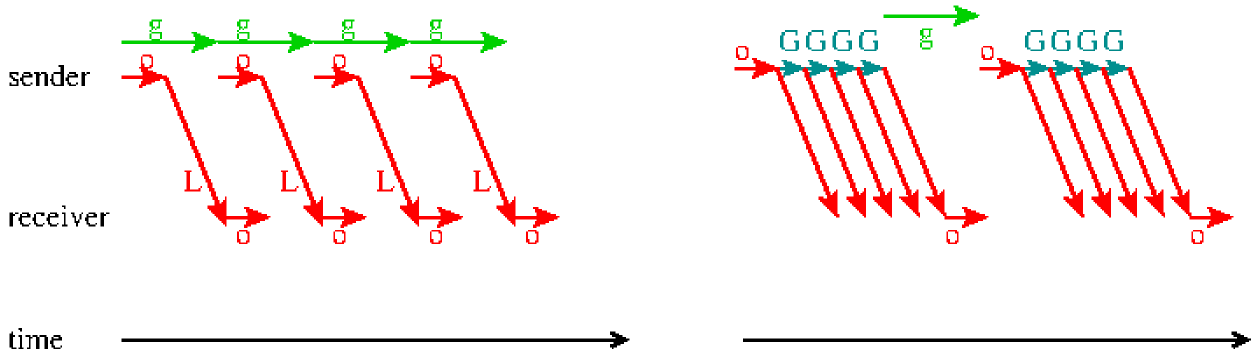


Figure 2.17: Left: Modeling a n-word message using the LogP model. Right: Using the LogGP model.

2.3.4.2 LogGP Model

For simplicity, the LogP model assumes uniform-sized messages so that the cost model parameters o, g and L are normalized to that size. While this simplifies analysis, the LogP model becomes inaccurate where messages of different sizes need to be analyzed. Simply charging a delay of $max(g, 2o+L)$ for each word of the message, i.e., $t_n = (n-1)g+2o+L$, will lead to an overestimation of the real cost (see Figure 2.17) because real-world parallel architectures are optimized for block data transfers.

The LogGP model [3] extends the LogP model by introducing a fifth parameter G, the *gap per word*, to model block data communication. As the effect of the single-word message overhead is included in g but not in G, we generally have $G < g$.

For the communication of an n-word-block, the LogGP-model charges time

$$t'_n = o + (n-1)G + L + o = (n-1)G + 2o + L$$

as illustrated in Figure 2.17 (right), which leads to more realistic predictions of communication delays of longer messages.

2.4 Simple Analysis of Cache Impact

The sequential and parallel cost models considered up to now ignored, for simplicity, the impact of data locality on the cost of memory accesses. As this impact is very significant in practice and affects the constant coefficients in the asymptotic cost estimations, explicit modeling of the memory hierarchy is required in more elaborate performance modeling, both for sequential and

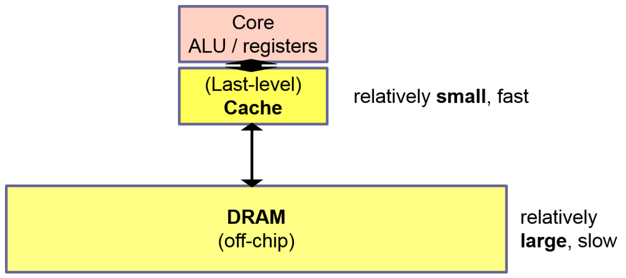


Figure 2.18: Abstraction of a RAM with memory hierarchy, here with one cache memory level.

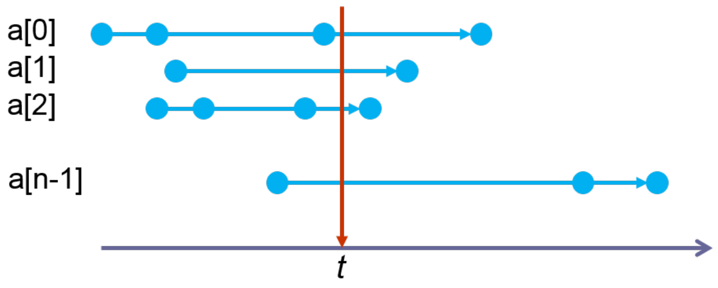


Figure 2.19: Illustration of the working set $WS_A(t)$ of the execution of algorithm A at time t.

parallel computations. In the following we focus on the impact of caches (in particular, of the *last-level cache*, i.e., the largest on-chip cache closest to off-chip memory) on the performance of sequential code. See also Figure 2.18.

We call a program variable (e.g., an array element) *live* between its first and its last access in an algorithms execution.

As following up the life spans of all program variables can be a very complex task, we will focus here on the larger data structures of an algorithm (e.g., larger arrays) and mostly ignore scalar variables and small arrays for simplicity.

The *working set* (also known as *working space*) of algorithm A at time t is defined as

$$WS_A(t) = \{v : \text{variable } v \text{ live at time } t\}$$

This definition refers to the time points in the execution trace of algorithm A. As this trace usually depends on the program's input or at least its size, we assume, for now, a fixed input.

The *worst-case working set size* of A is defined as

$$WSS_A = \max_t \ | WS_A(t) |$$

Accordingly, the *average-case working set size* of algorithm A is defined as

$$WSS_A = \text{avg}_t \ | WS_A(t) |$$

As a rule of thumb, we use:

Algorithm A has good cache locality if WSS_A does not exceed 90% of the cache size.

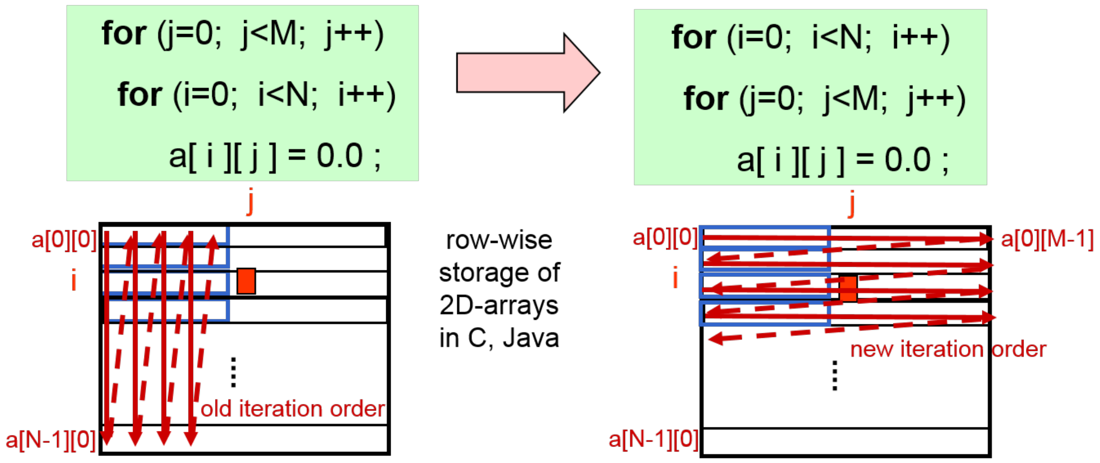


Figure 2.20: The impact of loop interchange on the working set size and on data access locality on a cache-based architecture.

This rule of thumb assumes a fully associative cache with a cache line size of one word (e.g., one array element) and perfect implementation of the LRU (Last Recently Used) cache line replacement strategy. (This is a simplification, because in practice, cache line sizes contain multiple data words and LRU is only approximately implemented in processors because an exact LRU implementation would be too expensive.) In particular, the impact of the cache line size is not modeled yet.

The value of 90% of the cache size is chosen in order to keep a 10% reserve for various "small" data items such as current instructions, loop variables, or stack frame contents.

In practice, issues such as the data layout of data structures in memory and the cache line size do matter. As an example, we consider the Loop Interchange scenario of Figure 2.20. A two-dimensional array `a[][]` is initialized with zeroes by two nested loops. On the left hand side, the j loop is outermost and the i loop is innermost. The loop nest accesses the elements $a[i][j]$ in column-wise traversal, which does not match well with the array's storage layout (assuming the row-wise storage layout of 2D arrays in languages such as C and Java): Accessing the first element $a[0][0]$ results first in a cold cache miss and brings in the corresponding memory block in a cache line. The same happens for all other elements $a[i][0]$ accessed right afterwards. Hence, the "real" WSS is not just N, but N times the cache line size. If fewer than N cache lines can be accommodated simultaneously in the cache, the first block of a will already be evicted again from the cache when we access $a[0][1]$, resulting in a capacity miss. Hence, *every* access to a will be a cache miss!

As here the relative order of all MN write accesses does not matter for correctness, it is valid to interchange the two loop headers (right hand side). And now the array layout matches perfectly with the traversal order, every block of a is brought only once into cache and accessed completely. The "real" working set size of this code after loop interchange is just the size of one cache line.

Our rule of thumb, possibly refined for considering the cache line size, allows for more realistic performance prediction for simple regular algorithms. However, it is hard to statically analyze the WSS for more complex, irregular algorithms. In such cases, empirical analysis can be an alternative, e.g. by inspecting the processor's hardware performance counters to obtain the number of cache misses per executed instructions.

2.5 Analysis of Parallel Algorithms

2.5.1 Parallel Time, Work, Cost

In the same way as execution time in the complexity analysis of sequential algorithms, the metrics parallel time, parallel work and parallel cost that we introduce in the following are usually worst-case measures, unless explicitly stated otherwise.

We consider an explicitly parallel (e.g., PRAM) algorithm. The algorithm can use as many processors as possible. However, we know that, for any given input of fixed size, the amount of parallel work at any time during the parallel algorithm execution is limited, in the extreme case to 1 (for example, during inherently sequential parts of the computation). Let $p_i(n)$ denote the maximum number of (PRAM) processors (i.e., the current degree of parallelism) that the algorithm could use in time step i of its execution. Then, $p_A(n) \ = \max_i p_i(n)$ is the maximum number of processors used at any time step of the execution of A, i.e., the maximum degree of parallelism. Adding processors beyond $p_A(n)$ will not give any speedup because these excess processors would not have any work to do at any time.

The *parallel time* $t_A(n)$ of an algorithm A given any input of size n denotes the maximum number of parallel time steps participating in the execution of A on any input of size n, where in each (parallel) time step i, for $1 \leq i \leq t_A(n)$, the algorithm uses $p_i(n)$ processors, i.e. as many processors as needed to execute step i in constant time.

The *parallel work*

$$w_A(n) = \sum_{i=1}^{t_A(n)} p_i(n)$$

performed by algorithm A given input of size n denotes the maximum number of instructions performed by any processor required under the same circumstances. Parallel work can be found out by counting executed constant-time operations (i.e., arithmetic instructions, branches, memory accesses and communication/synchronization operations) across all participating processors. However, any waiting (idle time) for other processors does *not* belong to the parallel work.

The *parallel cost*

$$c_A(n) \ = \ t_A(n) \cdot p_A(n)$$

is defined as the product of parallel time and the number of participating processing elements.

Because some time steps i, for $1 \leq i \leq t_A(n)$, may not be able to use all $p_A(n)$ processors, it is easy to see that $w_A(n) \leq c_A(n)$.

If considering the data flow graph of the computation performed by a parallel algorithm (where only the performed instructions, i.e., those defining the work, are represented as nodes), the parallel time $t_A(n)$ corresponds to the length of the longest path from any source node to any sink node of the data flow graph. This path (or these paths, as there might be multiple paths of longest length) of the data flow graph is also called the *critical path.* We also find the terms *depth* of A in the literature (if interpreting the data flow graph as an arithmetic-logical circuit).

2.5.2 Parallel Work Optimality and Parallel Cost Optimality

Consider a parallel algorithm A and a sequential algorithm *seq* for the same computational problem (such as array sum in the example following below).

A parallel algorithm A is asymptotically *work-optimal* iff $w_A(p,n) = O(t_{seq}(n))$.

A parallel algorithm A is asymptotically *cost-optimal* iff $c_A(p,n) = O(t_{seq}(n))$.

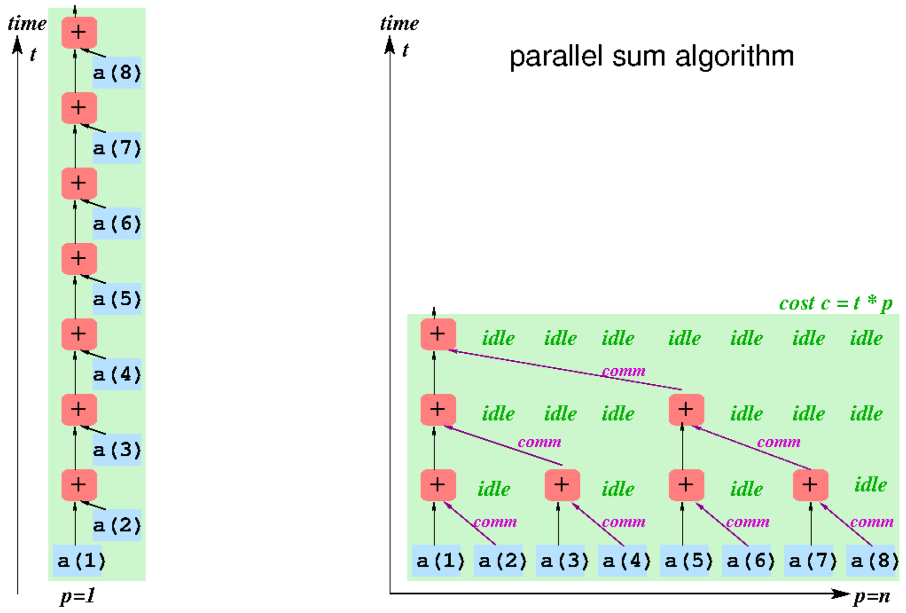


Figure 2.21: Analyzing parallel time, work, and cost for the array sum problem. Left: sequential algorithm (SeqSum), Right: divide-and-conquer based parallel algorithm. The time axis is directed upwards in this figure, the processor axis to the right. Load operations are shaded in blue, additions in red, and the cost rectangle in green.

2.5.3 Example: Array Sum

As an example, let us reconsider the array sum problem, for which we have already seen a sequential algorithm in Section 2.2 and a PRAM algorithm in Section 2.3.1.

Given the *problem size* n (here, the number of array elements to sum up) and a PRAM with p processors (note that the RAM for sequential execution is just a PRAM with $p = 1$ processor), we discuss in the following the parallel sum algorithm's time $t(p, n)$, parallel work $w(p, n)$ and cost $c(p, n) = t(p, n) \cdot p$. (For better readability, we omit the parallel algorithm's name subscripts in t, w and c.)

If $p = 1$, there is no parallel divide-and-conquer step and the parallel algorithm coincides with the sequential algorithm. The sequential algorithm SeqSum, whose execution is displayed in Figure 2.21 (left), performs in total $n - 1$ additions, n loads, and $O(n)$ other instructions, i.e.,

$$t(1, n) = t_{SeqSum}(n) = O(n)$$

A sequential algorithm's work (performed instructions) is always identical to its execution time because it never needs to communicate with/synchronize with itself nor wait for itself:

$$w(1, n) = t_{SeqSum}(n) = O(n)$$

Hence, the cost for the special case $p = 1$ is

$$c(1, n) = t(1, n) \cdot 1 = O(n).$$

For the parallel algorithm, the maximum degree of parallelism $p_A(n)$ is defined by step 1 of the algorithm where $p_1(n) = n$ independent load instructions could, in principle, run in parallel on n

PRAM processors (recall that the PRAM model does not put any limits on memory bandwidth). For subsequent time steps, the degree of parallelism halves in each level of the recursion tree. After $\Theta(\log n)$ time steps only one processor is active for calculating the final global sum. Hence, the parallel time is

$$t(n,n) = O(\log n)$$

The parallel algorithm performs the same amount of work as the sequential algorithm (i.e., n Loads, $n-1$ additions etc., see also Figure 2.21 (right)), the only difference being that some of them now overlap in time. Hence,

$$w(n,n) = O(n)$$

i.e., the parallel sum algorithm is also asymptotically work-optimal.

The parallel cost is the time-processor product, thus

$$c(n,n) = t(n,n) \cdot n = O(n \log n)$$

Here, we notice a significant, asymptotically growing discrepancy between parallel work and cost, namely, by a factor of $\log n$. Hence, $c(n,n) \in \omega(w(n,n))$, i.e., $c(n,n)/w(n,n) \longrightarrow \infty$ for $n \to \infty$, which implies that the parallel sum algorithm is *not* cost-optimal. In other words, the parallel sum algorithm does not utilize its n processors efficiently, and this gets only worse with growing n, which can also be seen in Figure 2.21 (right) where the green cost rectangle will be more and more sparsely populated with work as n increases.

2.5.4 Example Continued: Trading Parallelism for Cost-Effectiveness

We will now see how we can make the parallel sum algorithm cost-optimal. The main idea is to decrease the degree of parallelism significantly (i.e., by a more than constant factor) in order to make the cost rectangle much "narrower": Instead of $p = n$ processors, let us now use only $p = n/\log_2 n$ processors. This implies, of course, that each of those processors now has to do the work of up to $\lceil \log_2 n \rceil$ processors of ParSum.[8] We organize this new, "narrowed" parallel algorithm in two phases:

In the first phase, each processor computes sequentially the global sum of "its" $\log n$ local elements, each using SeqSum. This takes time $O(\log n)$ and produces an array of $n/\log_2 n$ partial sums.

In the second phase, they together compute the global sum of these $n/\log n$ partial sums using the ParSum algorithm.

The narrowed algorithm needs time $O(\log n)$ for Phase 1 (local summation) and $O(\log n)$ for Phase 2 (global summation). Overall, the parallel time for both phases remains $O(\log n)$.

Its parallel cost now becomes $n/\log n \cdot O(\log n) = O(n)$, that is, *linear*!

In the graphical interpretation (see Figure 2.21 (right)), we have, in comparison to ParSum, compressed the processor axis by a factor of $\log_2 n$, but only gained an increase in time by a small constant factor, because of the high sparsity of operations in most time steps of the ParSum algorithm.

This narrowing technique is an application of the so-called *work-time rescheduling* principle, and its effect on improving cost efficiency is described by *Brent's theorem*.

[8]For simplicity of presentation, we assume from now on that $\log_2 n$ divides n so that we can skip the rounding issue, which is here anyway insignificant from an asymptotic perspective.

2.5.5 Some Simple Task Scheduling Techniques *

Scheduling task graphs for parallel systems is a very intensively researched topic since about 50 years. Depending on the task graph properties and the machine model assumed, almost all variants of the scheduling problem are NP hard, and there exist, beyond exact algorithms that might be impractical for large problem instances, numerous algorithmic techniques to heuristically or approximatively solve a scheduling problem, either *statically* (the entire task graph is known beforehand) or *dynamically* (on-line, i.e., initially only the earliest tasks need be known and the graph can grow at runtime as tasks are being executed). Here we shortly describe a few of these families of scheduling algorithms.

2.5.5.1 Greedy Scheduling

Greedy scheduling algorithms begin with an initially empty schedule and pick the tasks from the task graph in topological or in some heuristically chosen order, one by another. A task is placed in the partial schedule either *as soon as possible* (ASAP), i.e., as early as data dependences from predecessor tasks allow and a suitable resource/processor is free, or *as late as possible* (ALAP), i.e., as late as data dependences to successor tasks allow and a suitable resource/processor is free. If there is no free slot to place the task, the schedule is stretched by inserting a time step to accommodate the new task.

2.5.5.2 List Scheduling

A popular variant of greedy scheduling is *list scheduling*, which proceeds in topological order and schedules data-ready tasks as soon as possible. Due to topological sorting, tasks are always appended at the currend "end" of the schedule as all their predecessors have already been placed earlier, hence it is not necessary to stretch the schedule anywhere except at the end. If multiple tasks are ready at the same time (all their predecessors have been placed), a tie is broken among them due to some priority criterion, such as the longest remaining path towards any sink of the dependence graph.

We call a task τ *data-ready* on processor i at time t if all predecessors τ_1, ..., τ_q of τ, scheduled at times t_1,...,t_q on processors $p[\tau_1]$,...,$p[\tau_q]$, have finished their execution and (on message passing architectures) the time for communicating its result from $p[\tau_j]$ to i has also elapsed. Hence the earliest time $ASAP(\tau, i)$ for executing τ on i is

$$ASAP(\tau, i) = \max_{j=1,\dots,q} (t_q + t_{comp}(\tau_j) + t_{comm}(\tau_j, p[\tau_j], i))$$

where the communication delay $t_{comm}(\tau_j, i', i)$ for the result of τ_j to be sent from its processor i' to destination processor i is derived from a cost model for point-to-point message passing, such as the Delay model, for $i \neq i'$, and is 0 for $i = i'$.

The task τ can then be placed at time t_i, which is either $ASAP(\tau, i)$ if i is available from that time for at least $t_{comp}(\tau)$ time units, or any later time $> ASAP(\tau, i)$ from where this condition holds. Greedy list scheduling would then pick a processor i from $i = 0, ..., p-1$ that minimizes t_i. The chosen processor i's schedule is updated to include τ starting from the calculated time t_i.

2.5.5.3 Critical-Path Scheduling

Critical-path scheduling is a variant of greedy scheduling that determines the critical path(s) in the task graph as far as not scheduled yet and, at any time, picks the next task to be placed in the schedule from the current critical path, i.e., in decreasing order of time criticality. As with greedy scheduling, tasks are inserted e.g. as soon as possible where dependences and resource availability allow, and new time steps are inserted if no applicable free slot is available in the current schedule.

2.5.5.4 Layer-wise Scheduling

Layer-wise scheduling algorithms decompose the task graph into layers of independent tasks. Layers are then processed in topological order: All tasks from one layer are scheduled (e.g., using list scheduling) before proceeding to the next layer.

2.5.5.5 Optimal Scheduling

The previously discussed static scheduling techniques (greedy, critical-path, layer-wise) are heuristics and cannot guarantee (time-)optimality of the resulting schedule. Some scheduling techniques are able to give guarantees about the worst-case makespan of the resulting schedule, such as being not more than twice as long as the optimal schedule.

Where optimality is desired, generic techniques for combinatorial optimization, such as Linear programming (LP), Integer linear programming (ILP), Constraint programming (CP), Genetic programming (GP), Dynamic programming (DP) etc. can often be used. In particular, task scheduling problems with dependences can usually be expressed in a rather straightforward way using LP, ILP or CP techniques.

2.5.5.6 Clustering

Clustering is an agglomeration technique for task graphs. It refers to preprocessing the task graph by repeatedly merging contiguous subgraphs of multiple tasks in the task graph into a single unit for scheduling (macrotask). All computation within one macrotask is sequentialized, in some topological order of the original tasks in the cluster.

This allows to coarsen the task granularity and to reduce the complexity of the scheduling problem, at the expense of limiting the scheduler's flexibility, and thus possibly miss optimal solutions.

2.5.5.7 Dynamic Scheduling

So far we have discussed static scheduling techniques. In many cases, not all tasks and/or their execution times or dependences are known before program execution. Sometimes not only the size but also the structure or other scheduling-relevant parameters of the task graph depend on input data or other run-time context. For such cases, dynamic scheduling can be used.

Dynamic scheduling techniques maintain a task pool data structure that keeps track of already known tasks and dependences, and schedules data-ready tasks to workers one by one, based on greedy strategies as in List scheduling. The overall run-time overhead of dynamic scheduling for task creation and management can be limited by clustering techniques that reduce the number of tasks to schedule.

2.5.6 Work-Time Rescheduling and Brent's Theorem

Figure 2.22 shows the principle of work-time rescheduling and illustrates the proof of Brent's Theorem.[9]

The given PRAM algorithm A with execution trace shown to the left uses an arbitrary maximum number p_A of (simultaneously active) PRAM processors, here up to 8, and completes in $t_A(n)$ time

[9]This section is optional for TDDC78.

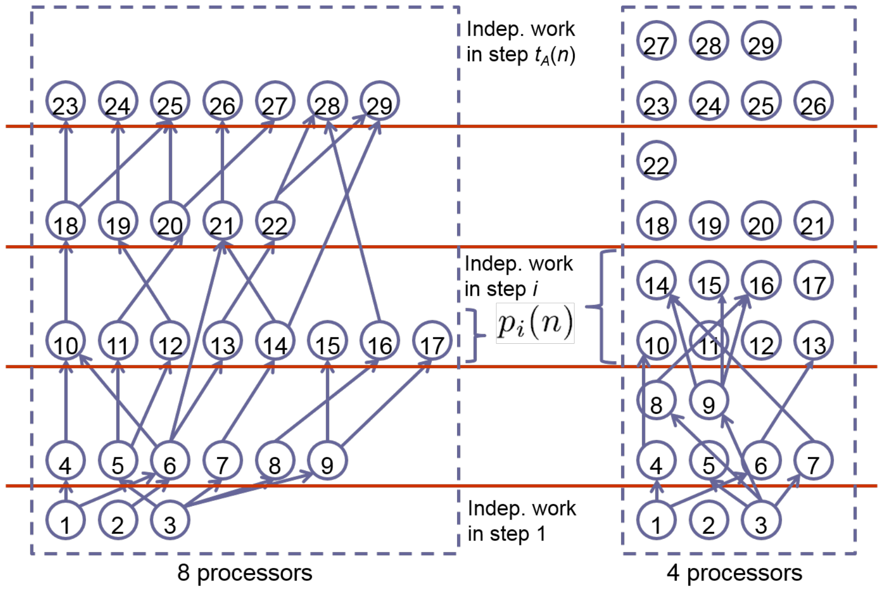


Figure 2.22: Principle of work-time rescheduling. Time flows upwards, and the layers are numbered by PRAM time steps from 1 (bottom) to $t_A(n)$ (top).

steps for a problem of size n. The $p_i(n)$ independent unit-time operations of the ith time step of A are rescheduled to a fixed number p (here, $p = 4$) of processors, resulting in a new parallel algorithm A'. As each of those now need to do the work of multiple PRAM processors of A, the time might grow somewhat, which we can bound from above:

Theorem 2.1 *(**Brent's Theorem** [11])*
A PRAM algorithm A performing work $w_A(n)$ in $t_A(n)$ time steps using an arbitrary number of PRAM processors can be implemented to execute on a p-processor PRAM with parallel execution time

$$O\left(t_A(n) + \frac{w_A(n)}{p}\right).$$

Proof: Let $p_i(n)$ be the number of processors that are active in time step i of the execution of A for $1 \le i \le t_A(n)$. By rescheduling[10], step i can be performed in $O\left(\lceil p_i(n)/p \rceil\right)$ steps with p processors. Hence the total time with p processors is

$$\begin{aligned}\sum_{i=1}^{t_A(n)} O\left(\left\lceil \frac{p_i(n)}{p}\right\rceil\right) &\le \sum_{i=1}^{t_A(n)} c_i \left\lceil \frac{p_i(n)}{p}\right\rceil \\ &\le \left(\max_{1\le i\le t_A(n)} c_i\right) \sum_{i=1}^{t_A(n)} \left(\frac{p_i(n)}{p} + 1\right) \\ &= O\left(t_A(n) + \frac{w_A(n)}{p}\right)\end{aligned}$$

[10]This is actually a slight simplification. To be precise, we need to make the assumption that the target architecture is able to execute the $p_i(n)$ rescheduled operations on p processors in time $O(p_i(n))$, i.e., with only a constant factor of overhead. PRAMs that fulfill this criterion are called *self-simulating*. See [37] for details.

for constants c_i, $1 \leq i \leq t_A(n)$, since $\sum_{i=1}^{t_A(n)} p_i(n) = w_A(n)$. $\square$

Note that, as A could use arbitrarily many PRAM processors, $t_A(n)$ denotes the length of the critical path (longest chain of dependent operations) through the program. Hence, one conclusion from Brent's Theorem is that the critical path is a lower bound of the parallel execution time for any concrete number of processors.

Beyond the critical path, the amount of work impacts parallel execution time: The second term, $w_A(n)/p$, is actually as good as we could hope for, it implies perfect load balancing. Unfortunately, the proof of Brent's theorem is non-constructive for the general case: It assumes that it is obvious to find out the active PRAM processors in each time step, which is not generally true. Apart from a few rather obvious cases, it does not give us a concrete recipe for engineering the algorithm into a form that uses a fixed number of processors for just that work. Likewise, an automated cycle-by-cycle rescheduling at runtime by a task-based runtime system incurs prohibitively high runtime overheads for extremely fine-grained (here, unit-time) tasks in practice so this is not a reasonable solution either. To some degree, highly hardware-multithreaded architectures such as GPUs take that cycle-by-cycle work rescheduling approach, although even these have limitations (e.g., on GPUs the smallest possible granularity of work is determined by the size of a warp) and eventually also have an upper limit for the amount of hardware threads to use.

2.5.7 Parallel Speedup and Efficiency

2.5.7.1 Parallel Speedup

(Parallel) speedup is the factor by which execution time is reduced due to using p processors, compared to one processor.

Speedup can either be determined empirically, by time measurements for a particular program run (input), or asymptotically by worst-case analysis for large input size n.

Consider a computational problem $\mathcal{P}$ (e.g., sorting n numbers or multiplying two $n \times n$ matrices), and a parallel algorithm A for calculating $\mathcal{P}$. By executing A on the target machine (or on a computational model) we can obtain the parallel time

$$T(p) = \text{time to execute parallel algorithm } A \text{ on } p \text{ processors}$$

and the time

$$T(1) = \text{time to execute parallel algorithm } A \text{ on 1 processor}$$

as a sequential baseline time to compare with.

Moreover, let T_s denote the time to execute the best serial algorithm A_s known for $\mathcal{P}$. Running A_s on one processor of the parallel machine yields another base-line time to compare with.

The *absolute speedup* is defined as

$$S_{abs} = \frac{T_s}{T(p)}$$

The *relative speedup* is defined as

$$S_{rel} = \frac{T(1)}{T(p)}$$

Given that, by assumption, $T(1) \geq T_s$, we see that

$$S_{abs} \leq S_{rel}$$

Where does the possible discrepancy between $T(1)$ and T_s come from, given that both are executed on the same one processor of the target system? In some cases where A_s has the very

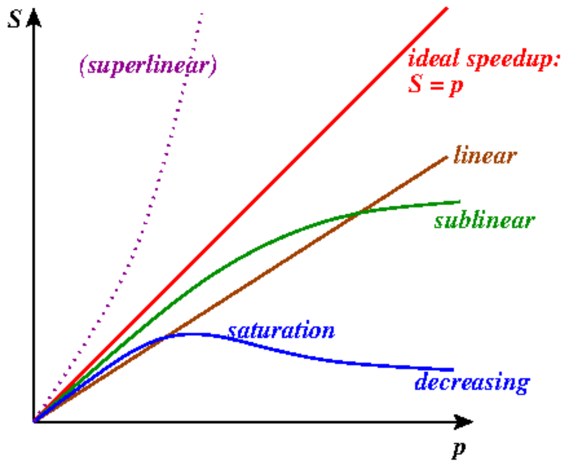


Figure 2.23: Speedup diagrams for various types of parallel algorithms.

same structure as A, the difference will be typically small and caused by overheads of constructs built into the parallel algorithm, such as task creation and scheduling over a single worker or mutual exclusion synchronization overhead, which are not necessary in an algorithm designed from the outset for a sequential machine or computation model. Another possible reason can be that the parallel algorithm is not work-optimal, i.e., performs asymptotically more work than the best sequential algorithm for the same problem.

Because the numbers often look better, relative speedup is the more popular one to display in papers on parallelization. In fact, it is often not explicitly stated what kind of speedup is shown in speedup diagrams; in those cases it is almost always relative speedup.

Speedup diagrams show the speedup (usually, relative speedup, see above) as a function $S(p)$ (resp., $S(n,p)$) of the number p of processors used. Figure 2.23 shows a few characteristic speedup curves for several types of parallel algorithms with different scalability behavior.

Some problems are *embarassingly parallel* or *trivially parallel*, such as parallel independent parameter sweeps over sequential simulations, or *data parallel problems* such as matrix-matrix product, LU decomposition, or ray tracing. Such problems often have very low amount of communication/synchronization, sometimes only at the beginning and the end of the computation, but a large amount of computational work to do that can easily be partitioned into independent tasks. As long as load balancing is not of a concern, such problems usually scale very well and show speedup close to the ideal (unit) speedup curve $S(p) = p$. Where load cannot be balanced well, we will instead see sublinear speedup growth that eventually flattens out.

Other problems might likewise be *work-bound* with algorithms that can use a large number of processors and scale linearly, but use a parallel algorithm that introduces overheads e.g. for dynamic task management that likewise grow with the number of processors used, hence the speedup curve will still be linear (linear $S \in \Theta(p)$) but with a slope lower than 1. Such linear-speedup algorithms are still asymptotically work-optimal, like those with ideal speedup.

Parallel algorithms with tree-like task graphs, e.g. reductions such as the ParSum algorithm described earlier, typically scale well for $p \ll n$ but will exhibit sublinear speedup growth as granularity becomes very fine, as in $p \in \Theta(n)$, thus $S(p) \in o(p)$. For example, the ParSum algorithm's speedup function for $p = n$ is $S(n) \in \Theta(n/\log n)$ as seen earlier.

As p grows very large and task granularity becomes extremely fine, every parallel algorithm will eventually deviate from ideal/linear speedup, as the communication/synchronization share of the work begins to dominate the computational work (i.e., the algorithm becomes *communication-bound*). A typical behavior is that, for some value of p, a saturation point is reached where speedup does not grow any more, regardless how many additional processors are used. Worse yet, adding more processors usually makes the parallel computation slower as the communication/synchronization work (growing with p) starts to dominate computational work (decreasing with p) beyond this point.

2.5.7.2 Parallel Efficiency

Parallel efficiency describes the effectiveness with which an algorithm exploits the available resources, i.e., the parallel processors. It is defined as speedup divided by the number of processors used. Accordingly we have two different notions of parallel efficiency:

In most cases, parallel efficiency is described by *relative efficiency*

$$E_{rel} = \frac{S_{rel}}{p} = \frac{T(1)}{p \cdot T(p)} = \frac{T(1)}{cost(p)}$$

If instead taking the best sequential algorithm as a baseline, we obtain the *absolute efficiency*

$$E_{abs} = \frac{S_{abs}}{p}$$

Absolute efficiency thus reflects the ratio between execution time on a single processor and total execution time over p processors. If splitting the latter into its components of work and overhead (communication time and idle time), we can see one reason for where non-perfect efficiency might come from:

$$E = \frac{T(1)}{\underbrace{T_{comp}}_{= \, work} + \underbrace{T_{comm} + T_{idle}}_{= \, overhead}}$$

Even a work-optimal parallel algorithm will thus suffer from communication overhead and idle time and never achieve full 100% efficiency.

2.5.7.3 Speedup anomalies

A *speedup anomaly* occurs if an (implementation of) an algorithm running on p processors may execute faster than expected.

We have to distinguish between one-time effects of "abnormal" acceleration by parallel processing that may occur for special numbers of processors only, and asymptotic speedup that will occur for *all* $p \geq p_0$ from some p_0, i.e., for $p \to \infty$. Although often wrongly used for all kinds of speedup anomaly, the term *superlinear speedup* only refers to this latter case of a speedup anomaly where the (asymptotic) speedup function grows faster than linear, i.e., is in $\omega(p)$ (see Figure 2.23).

Possible causes for speedup anomalies include the following:

- **Cache effects** By using more processors while keeping the problem size fixed, we also increase the overall amount of cache memory available while the working set size per processor might decrease due to the reduction in per-processor work. If the problem at hand is memory-access bound, the number of cache capacity misses will decrease as a larger and larger share of the working set fits into cache memory. For some (possibly, large) number of processors, the per-processor work might get small enough so that entire working set might fit into cache memory, and beyond that point there will be no further speedup anomaly observable (at least not due to cache effects).

- **Search anomalies** Parallel algorithms change the relative order of instruction executions compared to sequential execution, as far as permitted by the synchronization constructs that preserve dependences. Some parallel search algorithms might benefit from a changed order in that the item searched for happens to be discovered much earlier. An example is given below.

If considering absolute speedup, we can prove that such asymptotic behavior is impossible.

Theorem 2.2 *There is no absolute superlinear speedup for arbitrarily large p.*

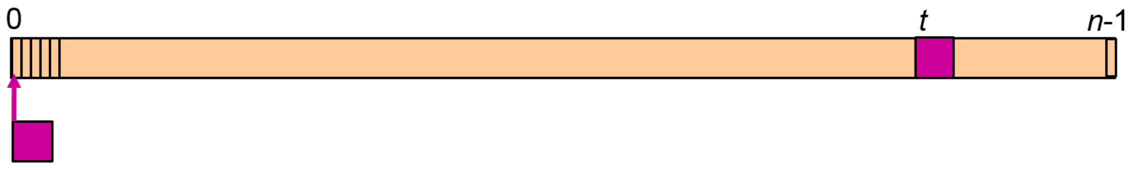
Proof idea: by contradiction. Assume that, for the computational problem $\mathcal{P}$ at hand, the asymptotically fastest sequential algorithm S has time complexity T_S. Further assume that we designed a parallel algorithm A with worst-case parallel time complexity T_A. If the absolute speedup $S_{abs}(p) = T_S/T_A(p)$ is in $\omega(p)$, then $T_S \in \omega(p \cdot T_A(p))$, i.e., T_S is asymptotically growing faster than $p \cdot T_A(p)$ by some nonconstant term in p. But then we could apply work-time rescheduling to design a new parallel algorithm A' that performs all work of A on *one* processor in a round-robin instruction-by-instruction simulation. If we assume (as with Brent's Theorem) that such simulation can be done with constant time overhead, the new algorithm A' has time complexity $T_{A'} \in \Theta(p \cdot T_A)$, which must be asymptotically faster than T_S due to the above assumption that $T_S \in \omega(p \cdot T_A)$. However, this contradicts the assumption that S be the fastest sequential algorithm for $\mathcal{P}$. $\square$

2.5.7.4 Search Anomaly Example: Simple String Search

For illustrating algorithmically induced speedup anomalies, we consider a simple problem, inspired by bioinformatics, of searching for a constant-sized substring (e.g., a specific genome subsequence) in a large input string (e.g., a genome sequence).

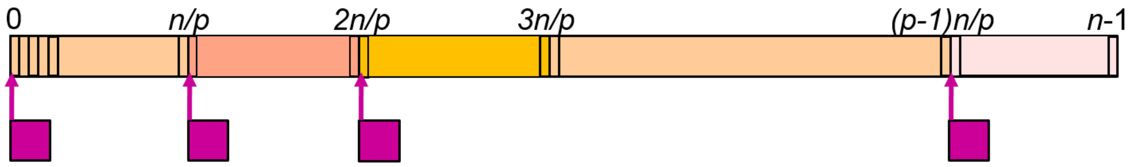
Formally, we are given a large unknown string of length $n + m - 1$ and a pattern of constant length $m \ll n$. The problem is to search for *any* (not necessarily the first one) occurrence of the pattern in the string, if it occurs, and otherwise report that the pattern does not occur. In total, there are n possible positions in the string for a match.

We consider a very simple sequential algorithm performing a linear search. (There exist better algorithms, e.g. based on preprocessing the pattern, but we want to keep it simple here). The algorithm steps through all positions 0, 1, ..., compares in each position the up to m characters from that position with the pattern characters, and stops as soon as it has found a full match. If no match is found after all n possible positions have been tried, it returns that the pattern does not occur.



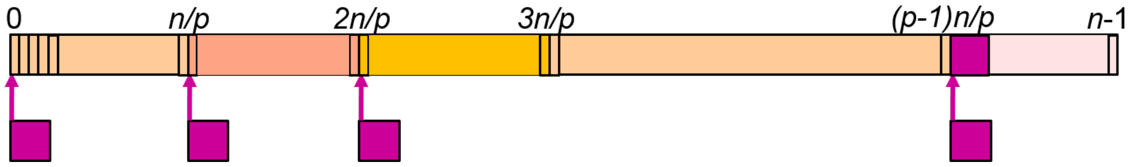
If the pattern is found at its first occurrence at position t in the string, the algorithm used $\Theta(t)$ time as m is constant. Hence, the algorithm spends time $O(1)$ in the best case and $O(n)$ in the worst case (and also in the average case).

Let us consider a parallel algorithm for simple string search. The set of positions to check for a match in the input string is split in blocks of size n/p, each processor searches linearly through its block. As soon as any processor finds a match, it informs the others and all can stop searching immediately.



Case 1: The pattern is not found in the string.

Then the parallel time will be n/p steps, and the speedup is $n/(n/p) = p$, perfect.



Case 2: The pattern is found in the *first* position scanned by the *last* processor, i.e, $n - n/p$, and there is no other occurrence at positions before $n - n/p$.

The parallel algoritm will then terminate after 1 (one) time step as the first match succeeds and all can finish immediately. The sequential algorithm will here run for $n - n/p$ steps, hence the observed speedup is $n - n/p$, i.e., superlinear. This sounds too good to be true! What is the problem?

The problem is that this is not the worst case but the best case for the parallel algorithm, while it is (almost) the worst case for the sequential one. We could have achieved the same effect with the sequential algorithm,too, if we would alter its string traversal order to imitate the time behavior of the parallel algorithm, namely, checking the string positions in the order 0, n/p, $2n/p$, ... $n - n/p$, 1, $n/p + 1$, $2n/p + 1$, ..., $n - 1$. In that case, we see no superlinear speedup any more.

2.5.8 Amdahl's Law

We consider the execution trace of any (parallel or parallelizable) algorithm A (i.e., its performed work) executed on one processor. Amdahl's Law (see below) classifies the executed instructions in the trace, somewhat simplifying, into only two categories:

- *sequential part* A^s which is inherently sequential (no parallelism at all);
- *parallelizable part* A^p, which we assume can be sped up perfectly by using an arbitrarily large number p of processors.

The total work can be written as $w_A(n) = w_{A^s}(n) + w_{A^p}(n)$, and the parallel time using p processors will, by assumption, be $T(p) = T_{A^s} + \frac{T_{A^p}}{p}$.

Theorem 2.3 ***(Amdahl's Law*** [4])
If the sequential part of A is a fixed *fraction of the total work irrespective of the problem size n, that is, if there is a constant β with*

$$\beta = \frac{w_{A^s}(n)}{w_A(n)} \leq 1$$

then the relative speedup of A with p processors is limited by

$$\frac{p}{\beta p + (1 - \beta)} < 1/\beta.$$

Amdahl's Law is visualized in Figure 2.24. The diagram in Figure 2.25 shows the speedup as predicted by Amdahl's Law for various values of β.

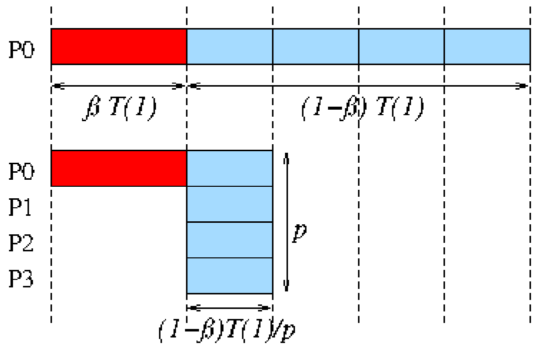


Figure 2.24: Illustration of Amdahl's Law

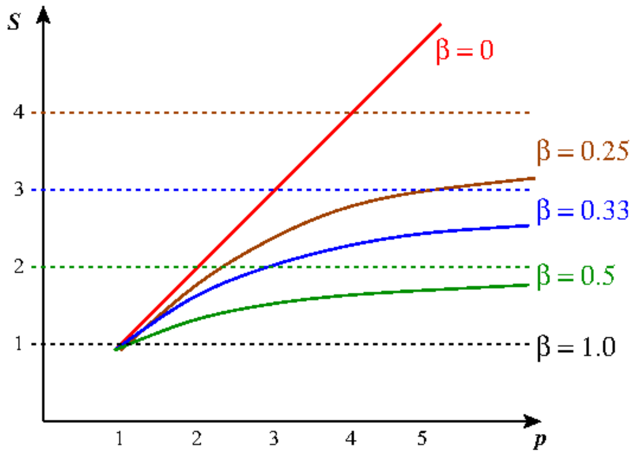


Figure 2.25: Upper speedup bounds as predicted by Amdahl's Law, for different fractions β of sequential work.

Proof of Amdahl's Law:
By definition, the relative speedup is

$$S_{rel} = \frac{T(1)}{T(p)} = \frac{T(1)}{T_{A^s} + T_{A^p}(p)}$$

We assume perfect parallelizability of the parallel part A^p, that is, $T_{A^p}(p) = (1-\beta)T(p) = (1-\beta)T(1)/p$:

$$S_{rel} = \frac{T(1)}{\beta T(1) + (1-\beta)T(1)/p)} = \frac{p}{\beta p + 1 - \beta} \leq 1/\beta$$

which completes the proof. □

Amdahl's Law is about the limits of accelerating a computation by using parallel processing. However, parallelization is only one way of accelerating programs. Other program optimizations include: SIMDization, instruction scheduling, data locality improvements, using special hardware accelerators, etc. Amdahl's Law can easily be generalized to be applicable to optimizations in general:

Theorem 2.4 ***(Amdahl's Law, generalized formulation)***
If we speed up a fraction $(1-\beta)$ of a computation by a factor p, the overall speedup is limited from above by $\frac{p}{\beta p + (1-\beta)}$, which is $< \frac{1}{\beta}$.

From this generalized statement, we can derive the following guidelines:

- Optimize for the common case.

  If $1-\beta$ is small, the optimization has little effect.

- Ignored optimization opportunities (also) limit the speedup as they contribute to the non-optimizable share β.

  Even if we could speed up the rest arbitrarily ($p \longrightarrow \infty$), speedup is still bound by $\frac{1}{\beta}$.

2.5.9 Weak Scaling and Isoefficiency

So far, our speedup notion is based on *strong scaling*, i.e., the problem size was kept fixed when scaling up the number of processors.

Weak scaling involves scaling up the problem size simultaneously as more processors are being used. This alternative view of scalability is motivated by the trend among many HPC users, especially from the scientific computing domains, that with the availability of larger machines solving larger problems is given preference over solving problems of the same size faster.[11] In that case, the goal of parallelization is not speedup but "sizeup".

2.5.9.1 Isoefficiency *

Isoefficiency is a metric that considers by *how much* the problem size n (which impacts the parallel work) must scale up with p to keep the parallel efficiency E constant.

For a given parallel algorithm A, the *isoefficiency function* for A is a function of p that expresses the *increase in problem size* required for A to retain a given parallel efficiency E.

[11] The motivation is that such larger problems that may previously have been out of reach and now could be solved within reasonable time on a larger machine might generate entirely new scientific insights.

If the isoefficiency-function for A is linear or sublinear, then we consider A to be well scalable.

Otherwise, if the isoefficiency-function is superlinear, algorithm A needs a disproportionally large increase in n to keep the same parallel efficiency.

2.5.9.2 Gustafssons Law

Recall that Amdahl's law assumes that the sequential work A^s is a *constant fraction* β of the total work. So far we assumed that the overall work w_A is fixed. However, the work usually depends on the problem size, n. Hence, when scaling up n, the work $w_A(n)$ will then scale up as well. Amdahl's Law would then pessimistically assume that even $w_{A^s}(n) = \beta w_A(n)$ would scale up with growing n.

Gustafsson's Law is an adaptation of Amdahl's Law for weak scaling, where we assume that the parallelizable part of the work scales linearly in n while the sequential part of the work remains constant in n:

Theorem 2.5 ***(Gustafssons Law*** [30]*)*
Assuming that the sequential work is constant *(independent of n), given by a sequential fraction α in an* unscaled *(e.g., size $n = 1$ (thus $p = 1$)) problem such that $T_{A^s} = \alpha T_1(1)$, $T_{A^p} = (1 - \alpha)T_1(1)$, and that $w_{A^p}(n)$ scales linearly in n, the* scaled speedup *for $n > 1$ is predicted by*

$$S^s_{rel}(n) \;=\; \frac{T_n(1)}{T_n(n)} \;=\; \alpha + (1 - \alpha)n \;=\; n - (n - 1)\alpha.$$

The sequential part is assumed to be replicated over all processors.

Figure 2.26 shows an illustration of Gustafsson's Law.

Proof of Gustafssons Law:
The scaled speedup for $p = n > 1$ is

$$S^s_{rel}(n) \;=\; \frac{T_n(1)}{T_n(n)} \;=\; \frac{T_{A^s} + w_{A^p}(n)}{T_{A^s} + T_{A^p}}$$

Assuming perfect parallelizability of A^p up to $p = n$ processors, we obtain

$$S^s_{rel}(n) \;=\; \frac{\alpha + (1 - \alpha)n}{1} \;=\; n - (n - 1)\alpha. \qquad \square$$

Compared to Amdahl's Law, Gustafsson's Law yields better speedup predictions for data-parallel algorithms where the parallelizable work scales nicely with the problem size while the sequential part is usually small and independent of the problem size.

2.6 Fundamental Parallel Algorithms

Beyond the already discussed reduction (e.g., parallel sum) problem, we will now study two further fundamental algorithmic problems that are often used as building blocks in parallel algorithms for more complex problems. For example, we will see in Chapter 3 that the parallel prefix sums problem can be a useful subroutine in certain parallel sorting algorithms.

These two problems also yield further examples for the application of the previously introduced parallel algorithmic patterns *parallel divide-and-conquer* and *recursive doubling*, and they are good targets for further application of the tools that we learned so far for the analysis of parallel time, work and cost.

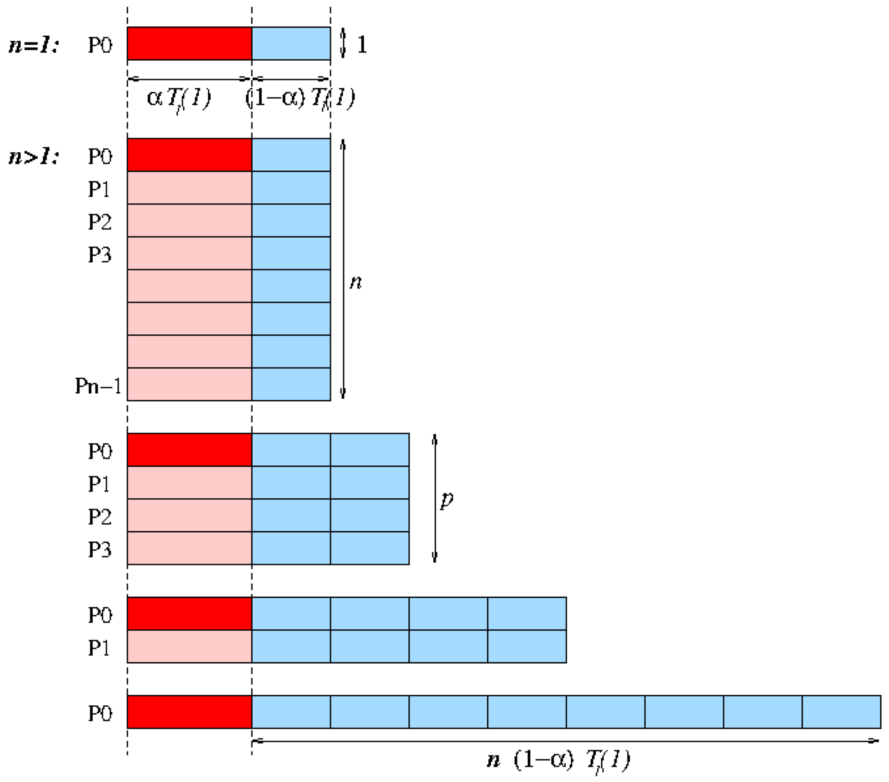


Figure 2.26: Illustration of Gustafsson's Law

2.6.1 Data Parallelism

Data parallelism is parallelism derived from independent applications of the same operation over all (or many) elements in large sets of data, such as arrays or lists. An example could be to increment all elements of an array by 1, but the operations could be much more complex functions than that, and thus data parallelism is more general than SIMD parallelism. All operation applications on different elements are independent and could thus be partitioned into independent tasks of arbitrarily fine granularity—in the extreme case, the granularity would be just one element per task, or one element per "virtual processor" as we did earlier with the PRAM model. Data parallelism can scale to thousands and even millions of processors, given that enough data elements are available and these tasks can be mapped to processing elements without significant runtime overheads. For example, Graphics processing units (GPUs) are specifically designed for processing rather fine-grained data parallelism on very large numbers of hardware threads

2.6.2 Data-Parallel Sum Computation

The data-parallel sum algorithm in Figure 2.27 is an iterative variant of the divide-and-conquer based ParSum algorithm of Sect. 2.3.1.4, which works in-place on the input array, i.e., stores the partial sums in elements of the input array that are no longer needed.

Comparing to the previous task-based formulations in Sect. 2.3.1.4, we have thus formulated ParSum as a data-parallel algorithm by replacing task-based recursion by iteration and by having the algorithm work in-place on a global array data structure,

It is interesting to note that the distance between accessed array elements doubles in each

```
real $a$ : array[0..$N-1$];
int stride;

stride $\leftarrow$ 1;
while stride $< N$ do
  forall $i$ : [0..$N-1$] in parallel do
    if $(i+1)$ mod (stride*2) $= 0$
      then
        $a[i] \leftarrow a[i-\text{stride}] + a[i]$;
  stride = stride * 2;
// finally, sum in $a[N-1]$
```

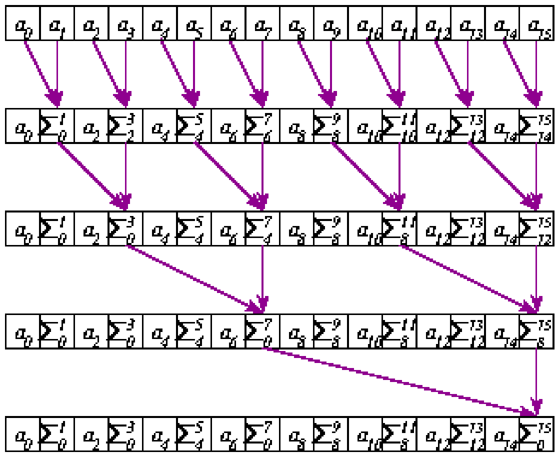


Figure 2.27: Left: In-place, iterative variant of the ParSum algorithm. Right: resulting data flow for $P = N = 16$.

iteration of the `while` loop, i.e., the scope of partial results doubles in every iterative step. Hence, after $\lceil log_2 N \rceil$ iterations, the global sum of all elements is computed. This algorithmic design pattern, which is closely related to parallel divide-and-conquer, is also known as *recursive doubling*.

2.6.3 Parallel Prefix-Sums Algorithms

2.6.3.1 The Prefix-Sums Problem

We are given a domain S (e.g., the integers) with a binary associative operator $\oplus$ on S, and an input sequence of n items $x_0, \ldots, x_{n-1} \in S$

The prefix sums problem consists in computing the sequence y of *prefix sums* defined by

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \le i < n$$

Prefix sums is an important building block of many parallel algorithms, such as some parallel sorting algorithms (Chapter 3).

Typical operations used for the $\oplus$ operator include integer addition, maximum, bitwise AND, and bitwise OR.[12]

A real-world analogy to the prefix sums problem is calculating the daily balances of a bank account. Initially the account has balance 0. Applying daily changes x_i (deposits or withdrawals x_0, x_1, ...) will result in the daily balances $y_i = \sum_{j=0}^{i} x_j$: x_0, $x_0 + x_1$, $x_0 + x_1 + x_2$, ...

The prefix sums problem as described above is also known as the *inclusive prefix sums* problem (the summation is up to and including the ith element). In contrast, the *exclusive prefix sums* problem calculates $y_i = \sum_{j=0}^{i-1} x_j$, so here y_0 is always 0.

Another real-world example, where however only positive elements are applicable, is calculating the page numbers for the table of contents in a book composed from separate chapters. If x_i denotes the number of pages for Chapter i, then the inclusive prefix sum $y_i = x_1 + ... + x_i$, with $y_0 = 0$, can be used to calculate the first page $y_{i-1} + 1$ of Chapter i.

[12]For the use in a parallel prefix sums algorithm, the $\oplus$ operation must at least be *associative* as the order of combining the partial results might differ when switching from a sequential to a parallel algorithm (cf. Figure 2.3). Where the implementation of the data flow of a parallel algorithm also allows that partial results may be combined in non-specified order (e.g., in arrival order), the $\oplus$ operator must also be *commutative*.

```
void seq_prefix( int x[], int n, int y[] )
{
  int i;
  int ps;  // i'th prefix sum
  if (n>0) ps = y[0] = x[0];
  for (i=1; i<n; i++) {
    ps += x[i];
    y[i] = ps;
  }
}
```

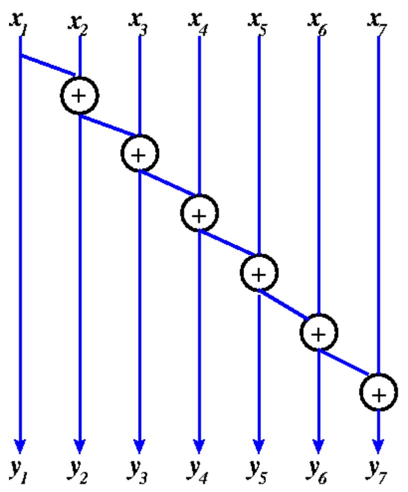


Figure 2.28: Left: Sequential prefix sums algorithm. Right: The dataflow graph of sequential prefix sums, for $n = 7$.

2.6.3.2 Sequential Prefix Sums Computation

Figure 2.28 (left) shows C code for the sequential prefix sums algorithm.

Figure 2.28 (right) shows the data flow graph of `seq_prefix`. It forms a linear chain of dependences along which the values are carried over from one iteration to the next one in variable `ps`.

Even if we would try to run this data flow graph somehow in parallel on n virtual processors as far as possible with the given dependence structure, we would not see any parallel speedup: The parallel time would remain $\Theta(n)$, for doing only work $\Theta(n)$, and the cost would even be $\Theta(n^2)$. Hence, this algorithm seems to be inherently sequential. Does this also hold for the prefix sums problem in general—or maybe not?

2.6.3.3 Naive Parallel Prefix Sums

As a preparatory exercise, we begin with a naive parallel algorithm for the prefix sums problem by simply applying the definition and identifying independence:

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \leq i < n$$

If we assign one processor for computing each y_i, the resulting parallel algorithm has parallel time $\Theta(n)$, and work and cost in $\Theta(n^2)$, hence, no improvement over the sequential algorithm. Worse yet, this algorithm is not work-optimal.

But we observe that this inefficient parallel algorithm performs a lot of redundant computation, namely, multiple computation of common subexpressions. This shows that there could indeed be some potential for reducing the work without decreasing parallelism too much.

2.6.3.4 Upper-Lower Parallel Prefix Sums Algorithm

We apply the algorithmic technique *parallel divide&conquer*, which already had served us well for the reduction problem in Section 2.3.1.4.

We consider the simplest variant, called UPPER/LOWER PARALLEL PREFIX. Figure 2.29 (left) shows data-flow graph template for the recursive formulation. The prefix sums for an array of size

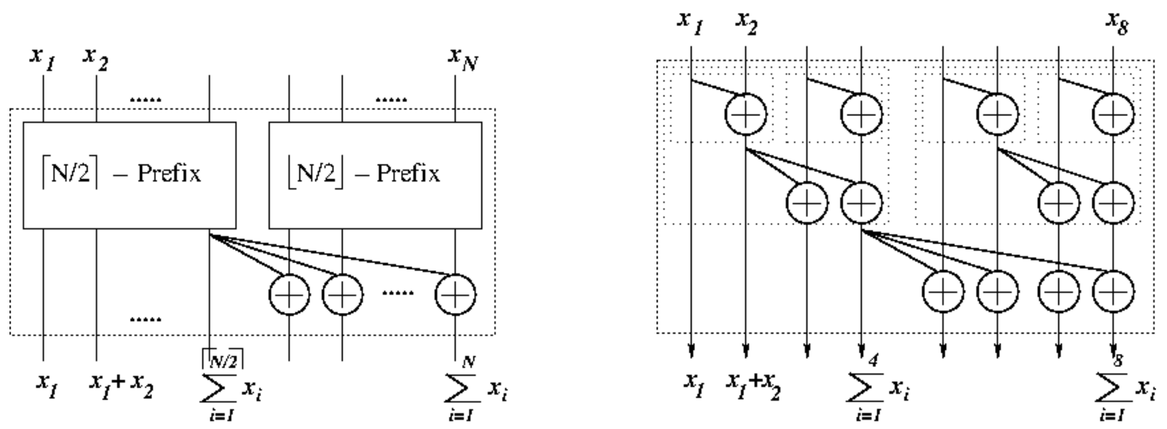

x_1 x_2 x_N
$\lceil N/2 \rceil$ – Prefix
$\lfloor N/2 \rfloor$ – Prefix
x_1 $x_1 + x_2$ $\sum_{i=1}^{\lceil N/2 \rceil} x_i$ $\sum_{i=1}^{N} x_i$
x_1 x_2 x_8
x_1 x_1+x_2 $\sum_{i=1}^{4} x_i$ $\sum_{i=1}^{8} x_i$

Figure 2.29: Left: The data-flow graph template for the recursive case in UPPER-LOWER PARALLEL PREFIX; time flows from top to bottom, each PRAM processor takes care of one input element. Right: The data-flow graph resulting from UPPER-LOWER PARALLEL PREFIX for $N = 8$.

$N > 1$ is computed by computing two prefix sums in parallel, the prefix sums for the lower half of the array (x_0, ..., $x_{\lceil n/2 \rceil - 2}$) and the prefix sums for the upper half ($x_{\lceil n/2 \rceil - 1}$, ...,$x_{n-1}$). These two recursive applications of the algorithm are independent of each other and can thus run in parallel, as shown in Figure 2.29. As the obtained prefix sums for the upper half of the array still do not take the values of the lower half into account, they need to be corrected by adding $\sum_{j=0}^{\lceil n/2 \rceil - 2} x_j$ to each of those $\lfloor n/2 \rfloor$ prefix sums of the upper half, which can all be done in parallel (by a data-parallel addition). The base case of the recursion is a size-one problem, which is basically a load of an array element's value from memory.

Figure 2.29 (right) shows the unfolded data-flow graph resulting from applying UPPER/LOWER PARALLEL PREFIX for $N = 8$.

The parallel time of the upper-lower parallel prefix sums algorithm is in $\Theta(\log n)$ steps (why?). Its work is dominated by the $n/2\ \log_2 n$ additions (why?), thus in $\Theta(n \log n)$. The parallel cost is likewise in $\Theta(n \log n)$.

While this is already a huge improvement in time over the sequential algorithm, this algorithm is not work-optimal. Moreover, it will not work for the EREW PRAM model, because it needs concurrent read (namely, wherever we tap into the partial sum $y_{\lceil n/2 \rceil - 2}$ computed by one PRAM processor ($\lceil n/2 \rceil - 2$) from the $\lfloor n/2 \rfloor$ processors with higher indices working on the same subproblem instance).

2.6.3.5 Recursive-Doubling Based Parallel Prefix Sums Algorithm

A simple EREW (exclusive read, exclusive write) prefix sums algorithm can be implemented using the recursive-doubling technique, which we already know from the dataparallel sum algorithm of Section 2.6.2.

Figure 2.30 shows an iterative formulation in data-parallel pseudocode.

The algorithm uses $p = n$ EREW PRAM processors that together performs $\log_2 n$ iterations, each doing constant work per iteration, hence the parallel time is in $\Theta(\log n)$ and the work is in $\Theta(n \log n)$. We see that the RECURSIVE-DOUBLING PARALLEL PREFIX SUMS algorithm is not work-optimal (why?).

```
real $a$ : array[0..$N-1$];
int stride;
stride $\leftarrow$ 1;
while stride $< N$ do
   forall $i$ : [0..$N-1$] in parallel do
      if $i \geq$stride then
         $a[i] \leftarrow a[i-$stride$] + a[i]$;
   stride := stride * 2;
(* finally, sum in $a[N-1]$ *)
```

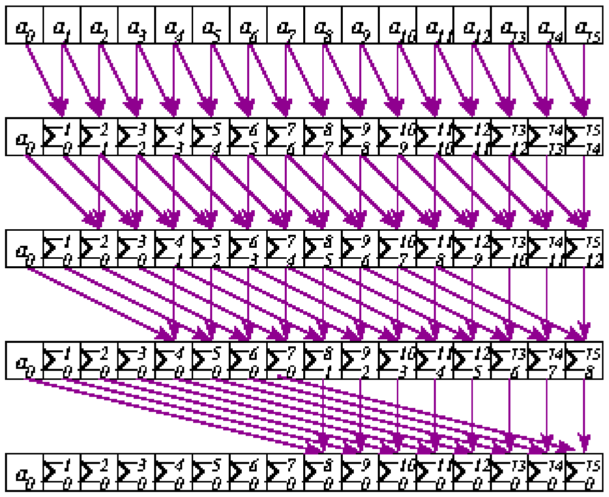
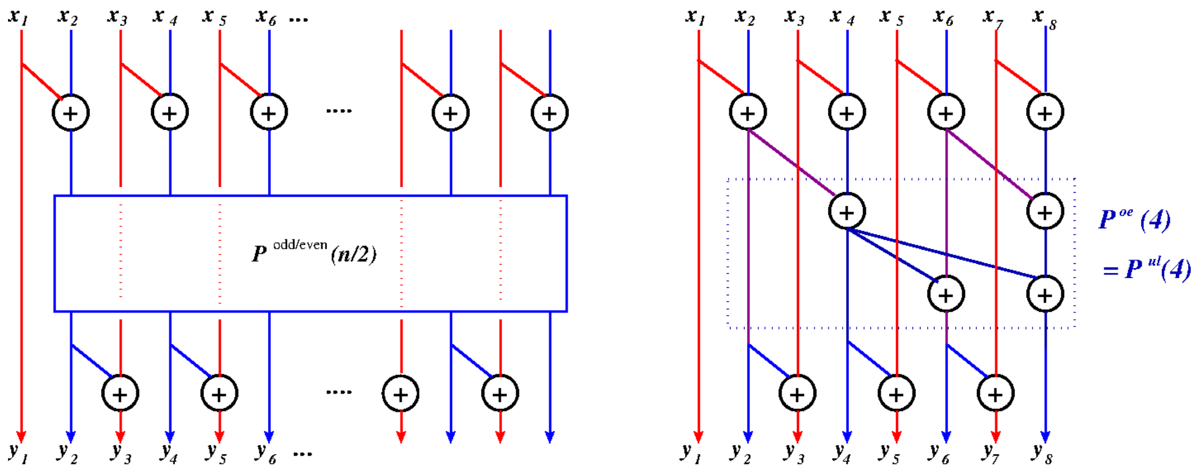


Figure 2.30: Left: Recursive-doubling based parallel prefix sums algorithm. Right: Data flow for $p = n = 16$.



x_1 x_2 x_3 x_4 x_5 x_6 ...
$P^{\text{odd/even}}(n/2)$
y_1 y_2 y_3 y_4 y_5 y_6 ...
x_7 x_8
$P^{oe}(4)$
$= P^{ul}(4)$
y_7 y_8

Figure 2.31: Odd-Even Parallel Prefix Sums algorithm

2.6.3.6 Odd-Even Parallel Prefix Sums Algorithm

Figure 2.31 shows the data-flow graph template for the recursive part of the Odd-Even Parallel Prefix Sums algorithm.

The Odd-Even Parallel Prefix Sums algorithm works also for EREW PRAMs (no result is used more than twice). Its parallel time is $2\log_2 n - 2$ time steps, i.e., in $\Theta(\log n)$, the work is $2n - \log n - 1 \in \Theta(n)$.

The factor 2 in parallel work complexity comes from $O(n)$ additions per recursion step—actually $n-1$ additions per recursion step as the postprocessing additions for the odd elements do actually one addition less $(n/2-1)$ than the preprocessing additions for the even elements:

The recurrence equation system (with the base case at $n = 2$) for the parallel work is:

$$W_{oddeven}(n) = W_{oddeven}(n/2) + n - 1 \quad (2.1)$$
$$W_{oddeven}(2) = 1 \quad (2.2)$$

As n is a power of 2, say 2^k for some integer $k > 0$, we can conclude:

$$W_{oddeven}(n) = \sum_{i=1}^{k}(2^i - 1) \tag{2.3}$$
$$= 2^{k+1} - k - 1 = 2n - \log n - 1. \tag{2.4}$$

The cost is in $\Theta(n \log n)$. Comparing to the work, we observe that the algorithm is not cost-optimal. By using Brent's theorem, a linear-cost algorithm can finally be obtained: The LADNER-FISCHER PARALLEL PREFIX SUMS algorithm [45] is cost-optimal, with time in $\Theta(\log n)$ and cost in $\Theta(n)$ if using $\Theta(n/\log n)$ virtual processors only.

2.6.4 List Ranking

2.6.4.1 The Parallel List Data Type

Linked lists are a fundamental data structure in sequential computing. They consist of, usually, heap-allocated list elements linked by `next` pointers, hence, the elements of a linked list need not be stored consecutively and not in order in heap memory.

```
struct list_item {
   struct data_item *data;
   struct list_item *next;
}
```

Unfortunately a linked list is poorly suited as a data type for (data-)parallel computing, because it lacks constant-time random access to the list elements: As the list head is the only entry point, each processor would have to chase a number of `next`-pointers throughout the list to find its element to work on already incurs a worst-case time that is linear in the length of the list (namely, the time to reach to the the last element of the list.

Hence, for dataparallel algorithms on lists, we define a new data type: the *parallel list*. A parallel list is an (unordered) *array* of list items, stored in shared memory, such that, at the finest level of granularity of parallelism, each (PRAM) processor is responsible for just one element, which it can access by array indexing in constant time.

We could interpret a parallel list as a sub-heap of finite size (n elements) dedicated to storing list items of one type only; list items can still be stored out of order, can belong to multiple lists of the same type, and some of the elements in the list item array might not be linked at all (i.e., are unused).

2.6.4.2 Parallel End-of-List Finding by Pointer Jumping

We begin with a simple problem: for each element in a parallel list, find the end of its linked list. The result will, for each list element, be a pointer to the last element in the `next`-linked list. Note that a parallel list might actually contain the elements of multiple entangled `next`-linked lists, and knowing an elements's list end item also indicates to which of possibly multiple lists it belongs to.

This problem can be solved in sequential in linear time (How?). Leveraging the random-access feature of the parallel list, we should be able to find the end of the list with n processors faster than in linear time.

As the main algorithmic technique, we again use recursive doubling, now not applied to doubling distances between index positions in an array but between elements linked by `next`-pointer chains.

```
function POINTER_JUMPING_FIND_LISTEND (
                struct list_item[] parlist, int N )
{
  forall k in [1..N] in parallel do
    // each processor k works on its element e:
    struct list_item *e  ←  &(parlist[k]);
    e →chum  ←  e →next;
    while e →chum ≠ null
            and e →chum→chum ≠ null
    do  // synchronous accesses:
      e →chum  ←  e →chum→chum;
    od
  od
}
```

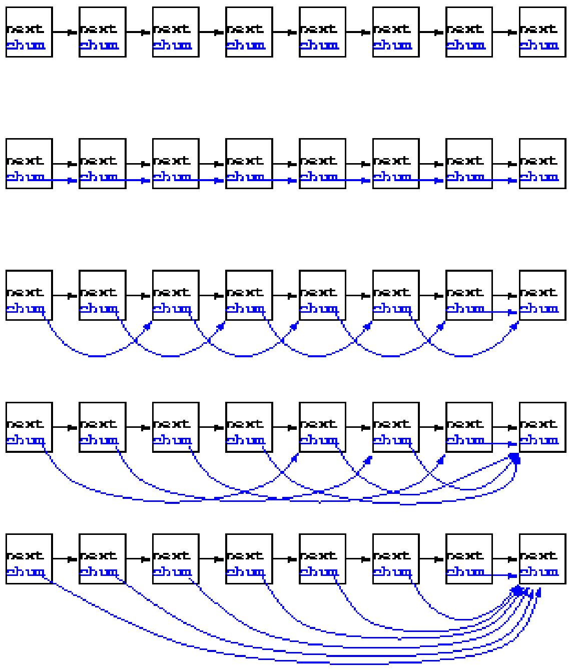


Figure 2.32: Left: The parallel pointer jumping algorithm for finding the end of each element's list. Right: Application of the algorithm to a parallel list with $n = 8$ elements on an 8-processor PRAM. Time flows from top to bottom.

This variant of the recursive doubling technique is known as *parallel pointer jumping* and was introduced by Wyllie [67].

Figure 2.32 (left) shows the pseudocode of a POINTER-JUMPING algorithm for finding, for each element, the end of its list. The algorithm assumes a (EREW) PRAM with $p = n$ PRAM processors, where each processor $k \in \{0, ..., n-1\}$ is responsible for updating one list element in each iteration. The PRAM lock-step synchronous execution guarantees that, for all processors still participating in iterating over the `while` loop iterations, that all updates have taken place before the loop exit condition is checked and possibly the next iteration is started. The loop terminates when all processors have finished the `while` loop.

The right hand side of Figure 2.32 shows an example how the pointer-jumping algorithm proceeds in its $\lceil \log_2 n \rceil$ iterations. Each list element holds, beyond its `next` pointer, an auxiliary pointer `chum` that is initialized to the same value as the `next` pointer, i.e., `chum[`k`]` initially points to the next list element. In each iteration, `chum[k]` is updated (simultaneously on all PRAM processors) to point to `chum[chum[`k`]]`, i.e., splices out its direct successor from its `chum` list, see Figure 2.33 where the `chum` pointers are shown in blue color. The processor taking care of that successor element does the same operation at the same time, and so on. The effect of this technique is that, after each iteration, the number of `chum` lists is doubled and the length of the longest chum chain is halved. Consequently, this procedure will reach a stable state after $\lceil \log_2 N \rceil$ iterations, where all `chum[`k`]` point to the last list element.

We can conclude that the algorithm has parallel execution time in $\Theta(\log N)$. It performs work $\Theta(n \log n)$, i.e., it is *not* work-optimal (why not?).

```
function POINTER_JUMPING_LIST_RANKING (
                struct list_item[] parlist, int N )
{
  forall k in [1..N] in parallel do
    // each processor k works on its element e:
    struct list_item *e ← &(parlist[k]);
    e →rank ← 1;
    e →chum ← e →next;
    while e →chum ≠ null
          and e →chum→chum ≠ null
    do  // synchronous accesses:
      e →rank ← e →rank + e →chum→rank;
      e →chum ← e →chum→chum;
    od
  od
}
```

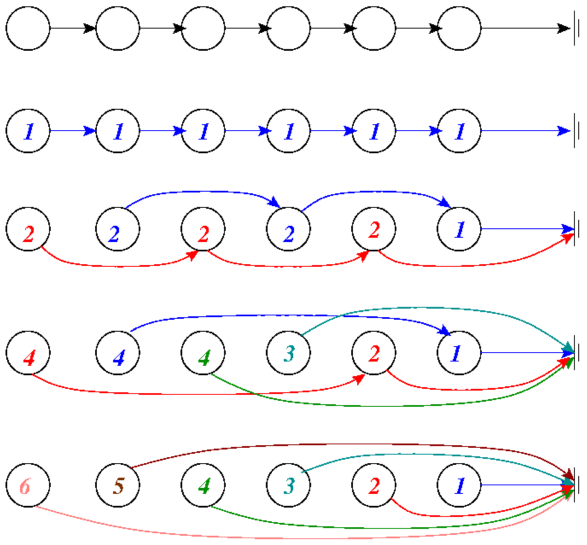


Figure 2.33: Left: Wyllie's pointer-jumping parallel list ranking algorithm. — Right: Application of the algorithm to a parallel list with $n = 8$ elements on an 8-processor PRAM. Time flows from top to bottom. For better presentation, items are shown in list order. Topmost row: Initial situation. Second row: after initialization of the `rank` fields. Third to fifth row: `rank` and `chum` values after the first, second and third iteration of the `while` loop, respectively.

2.6.4.3 Parallel List Ranking by Pointer Jumping

We now extend the problem from just finding the end of the list to computing the *rank* of each element in its list, i.e., its distance (including itself) to the end of its list. This problem is also known as *list ranking*.

We extend the POINTER-JUMPING algorithm as follows: Each list element is extended by an integer counter called `rank` that is initialized (in parallel) to 1. In each step of the `while` loop, each PRAM processor adds to its own current `rank` value the current `rank` value of its `chum`, which it splices out of its `chum` list. The pseudocode can be found in Figure 2.33 (left).

Figure 2.33 (right) illustrates for a parallel list with $n = 6$ elements how the parallel LIST RANKING algorithm works with $p = n$ PRAM processors.

The time and work complexity of parallel LIST RANKING remains the same as for the previous POINTER JUMPING algorithm: We note that, as before, every step doubles the number of `chum` lists and halves their lengths, hence, we are done after $\lceil \log_2 n \rceil$ steps.

The pointer-jumping based list ranking algorithm can be further extended e.g. for computing prefix sums on a list in parallel. We leave this as an exercise to the reader.

2.7 Chapter Summary

We introduced a number of parallel computation models. A parallel computation model consists of a parallel programming model and a cost model. The programming model provides abstractions for the main constructs controlling parallel program execution, data sharing and synchronization. The cost model provides the quantitative basis for the design and analysis of parallel algorithms under the computation model.

Early in the design process for parallel algorithms, we prefer simple performance models, such as PRAM, Delay, and BSP. This helps to avoid expensive design errors that lead to poor scalability. In particular, an algorithm that does not scale properly under the idealistic PRAM model will not scale on any more realistic cost model either.

At later stages we will work with refined performance models (BSP, LogP, LogGP, working space) that account for more properties of real-world parallel computer systems.

By conducting simple experiments on the target system and performing a regression analysis, we can derive concrete values of the cost model parameters, such as average per-node communication bandwidth or latency, hence our parallel time formulas might even serve as rough estimations for concrete execution times. We can then, during implementation and testing, compare the observed performance to the predictions by the model. If a huge difference is observed, this might be a hint for possible implementation errors or inefficiencies.

In this chapter we focused on asymptotic analysis, which ignores constant factors and lower-order terms in cost formulas. We introduced methods for the analysis of parallel algorithms for various cost models, using the global sum as a running example.

We introduced the metrics (parallel) time, work, cost, speedup, and efficiency. We defined time-optimality, work-optimality and cost-optimality, and have seen several examples for both work- and cost-optimal and -non-optimal parallel algorithms.

In some cases, we could transform an algorithm that is work-optimal but not cost-optimal into a cost-optimal algorithm by reducing the degree of parallelism and rescheduling the work to fewer processors. Brent's theorem tells us that such rescheduling is, in principle, always possible and the resulting parallel time is only bounded by the critical path length and the overall amount of work to be performed. Moreover, such transformation may require a possibly nontrivial effort on the side of the programmer and/or incur runtime overhead when using a task scheduling runtime system.

We have learned about speedup anomalies and their technical and/or algorithmic reasons, examples, and asymptotic concerns. Amdahl's Law and Gustafsson's Law establish upper bounds on speedup, depending on the amount of sequential work done by a parallel algorithm.

When designing an algorithmic solution for a problem to be solved in parallel, a number of parallel algorithmic design patterns have proven to be useful. In this chapter we encountered the following ones:

- Data parallelism
- Parallel divide-and-conquer
- Recursive doubling
- Pointer jumping

We introduced different parallel algorithms for solving fundamental algorithmic problems, prefix sums and list ranking. We will later see applications of parallel prefix sums as a subroutine in parallel sorting algorithms.

2.8 Bibliographical Notes *

The PRAM model was introduced by Fortune and Wyllie in 1978 [25].

Theoretical foundations for emulating PRAMs on non-PRAM (e.g., distributed-memory) parallel hardware were introduced by Mehlhorn and Vishkin [50] and Ranade [52, 53]. A complete hardware and software stack for the PRAM realization *SBPRAM* [1, 51] is described in the book by

Keller, Kessler and Träff [37], which is accompanied by an open-source PRAM programming framework for the PRAM programming language *Fork* [37, 42, 41] including a compiler, system tools and PRAM simulator.[13] Other realizations of the PRAM or relaxed PRAM model in hardware are described e.g. by Forsell [23, 24] and Vishkin [66, 65].

A classical reference for PRAM algorithms is the book by JaJa [32]. PRAM algorithms such as parallel prefix sums and parallel sorting are also covered in the books by Cormen et al. [17], Jordan and Alaghband [34] and Grama *et al.* [29]. For more details on the PRAM model and algorithms we refer to the comprehensive PRAM literature.

The BSP model was introduced by Valiant in 1990 [61] and slightly refined by McColl in 1993 [49]. More recently, Valiant added *Multi-BSP* [62], a generalization of the BSP model for hierarchical (clustered) parallel systems, such as clusters of nodes each containing multiple multicore CPUs, such that each hierarchy level can have its own g and L parameters for communication between submachines at that level.

Implementations of the BSP model include the BSP programming libraries *BSPlib* [31] and *PUB* [10], and the language *NestStep* [40]. These implementations from the 1990s mainly targeted clusters, usually atop MPI. More recently, BSPlib implementations for standard multicore CPUs have been added [68], and implementations for heterogeneous multicore systems have also been proposed [33]. Some BSP libraries like BSPlib also offer unbuffered one-sided communication (`hp_put` etc.), in addition to buffered one-sided communication (put etc.). The former does not imply strict consistency but often yields higher performance where applicable from a correctness point of view.

A good source of information about BSP and its implementations is also the report *Questions and Answers about BSP* by Skillicorn et al. [57]. An extension of the BSP model for nested parallelism by nestable supersteps was proposed by Skillicorn [56] and an extension for distributed shared memory by Kessler [40, 38]. A variant of BSP is the *CGM* (*coarse grained multicomputer*) model proposed by Dehne [19], which has the same programming model as BSP but an extended cost model that also accounts for aggregated messages in coarse-grained parallel environments.

Many BSP algorithms have been described in the literature, especially during the 1990s. The book by Bisseling [7], shows how to write programs from the scientific computing domain for BSP using BSPlib. In fact, many distributed-memory message-passing programs (written e.g. in MPI) basically *are* BSP programs, as they implicitly adhere to a restricted structure of synchronization and communication/memory accesses, even if not expressing this structure explicitly in terms of BSP-specific constructs.

The LogP model was introduced in 1993 by Culler *et al.* [18], its refinement LogGP for longer messages in [3]. A message passing cost model for point-to-point communication in clusters that is more accurate than LogGP was proposed by Seinstra and Koelma [54], a generalization for clustered message-passing architectures by Campbell [12].

For determining time-optimal broadcast trees in LogP, see e.g. Karp *et al.* [36].

List scheduling was introduced by Graham [28]. For a survey of task scheduling methods for parallel systems we refer e.g. to the book by Sinnen [55].

The use of parallel prefix sums as a core routine in other parallel algorithms is described by Blelloch [9].

[13] Available for Solaris and Linux. Description and download: http://www.ida.liu.se/~chrke55/fork

2.9 Exercises

Here we provide some questions for repetition, reflection and further exploration. As usual, there are deliberately no answers to the questions included.

1. To which of the following parallel computer architectures is the PRAM model the closest match?
   (a) standard multicore CPU with 8 cores
   (b) general-purpose GPU
   (c) cluster architecture

2. Write a recursive formulation of the ParSum algorithm in pthreads.

3. Write an iterative formulation of the ParSum algorithm in OpenMP (not using the task construct).

4. Give high-level CREW and EREW PRAM algorithms for copying the value of memory location $M[1]$ to memory locations $M[2], ..., M[n+1]$.

5. On a RAM the maximum element in an array of n real numbers can be found in $O(n)$ time. We assume for simplicity that all n elements are pairwise different.

   (a) Give an EREW PRAM algorithm that finds the maximum element[14] in time $\Theta(\log n)$. How many processors do you need at least to achieve this time bound?
   What is the work and the cost of this algorithm? Is this algorithm cost-effective with n processors? With $n/\log n$ processors?

   (b) Give an algorithm for a Common CRCW PRAM with n^2 processors that computes the maximum element in constant time.
   (*Hint:* Arrange the processors conceptually as a $n \times n$ grid to compare all n^2 comparisons of pairs of elements simultaneously. An element that is smaller in such a comparison cannot be the maximum. Use the concurrent write feature to update the entries in an auxiliary boolean array m of size n appropriately, such that finally holds $m[i] = 1$ iff array element i is the maximum element. Given m, the maximum location i can be determined in parallel using n processors.)
   What is the work and the cost of this algorithm? Is this algorithm cost-effective?

6. Give a $O(\log n)$ time EREW PRAM algorithm for computing parallel prefix sums on a parallel list. (Hint: Use the pointer doubling technique.)

7. A program contains a function m that cannot be parallelized and that accounts for 40% of the execution time. What is the speedup limit for this program on a p-processor machine?

8. A program contains a function m' that accounts for x % of the execution time. What should be the speedup of m' in order to speed up the overall program by a factor of 2?

9. A program contains a function m'' that can be parallelized to achieve speedup 3. What fraction of the overall execution time must m'' account for in order to double the overall speedup of the program?

[14]Further reading on the maximum problem: Any CREW PRAM algorithm for the maximum of n elements takes $\Omega(\log n)$ time, see Cook/Dwork/Reischuk *SIAM J. Comput.* 1986. There exist CRCW PRAM algorithms for n processors that take $O(\log \log n)$ time, see Valiant *SIAM J. Comput.* 1975, Shiloach/Vishkin *J. Algorithms* 1981.

10. Given a message passing parallel computer whose interconnection network is a d-dimensional hypercube. Design a deterministic parallel algorithm (using send and receive operations) for broadcasting a block of data from a single source node s to all other nodes. Using either the Delay model or the LogP model, what is the worst-case execution time of your algorithm?

11. The BSP model requires a conceptual barrier synchronization to properly separate executions of subsequent supersteps. Assume now that blocking receive operations are used in the communication phase. For what kind of communication patterns would the communication phase already imply a barrier synchronization so that no extra barrier operation is necessary?

12. Show that the cost of a cost-optimal parallel algorithm A is of the same order of magnitude as the work of the optimal sequential algorithm S: $c_A(n) = \Theta(t_S(n))$.

13. Formulate iterative versions of ParSum, Recursive-Doubling based Parallel Prefix Sums and Wyllie's list ranking algorithm as BSP algorithms and derive their parallel time. How many supersteps do you need in each case?

14. Compare Brent's theorem with Amdahl's Law. What are the similarities? Is maybe one a special case of the other one?

15. Explain: Could the problem of required concurrent-read in the Upper-Lower Parallel Prefix Sums algorithm be fixed for execution on an EREW PRAM by implementing and calling a broadcast subroutine for EREW PRAM?

    Sketch a good EREW broadcast algorithm and discuss its implications for the overall parallel time complexity of the Upper-Lower parallel Prefix Sums algorithm.

16. Explain why the recursive-doubling parallel prefix sums algorithm is not work-optimal.

17. Explain why Wyllie's pointer-jumping algorithm for finding the end of the list is not work-optimal.

18. How would you modify Wyllies pointer-jumping list ranking algorithm to calculate *exclusive* list ranks, i.e., element distances from the list end excluding the current item?

19. Compare the structure and properties of the recursive-doubling parallel prefix sums algorithm with those of Wyllie's pointer-jumping algorithm for parallel list ranking. What are the similarities?

20. Assume that the `data` field of each parallel list item contains an integer value. Extend Wyllie's pointer-jumping algorithm for parallel list ranking to computing prefix sums over the `data` values in a parallel list.

21. More exercises TBD ...

Chapter 3

Parallel Sorting Algorithms

Sorting[1] is one of the most important subroutines in computing. Accordingly, it has been intensively researched in sequential algorithmics since the early days of computing. Well-known sequential sorting algorithms include bubble sort, mergesort, heapsort, quicksort, radix sort and many more.

Sorting often works on large data sets, which sometimes are so large that they cannot be kept and sorted in main memory in one pass and require external sorting algorithms that perform multiple passes over the data and store it temporarily in secondary storage.

Sorting is challenging for modern computer architectures in that it involves relatively little computational work, but mainly control (conditional branches), memory accesses, and data movement.

The design of scalable parallel sorting algorithms is not straightforward, as opposed to many numerical kernel problems where the existing loop-based control structure of a sequential algorithm often can be transformed into a (data-) parallel computation. The design of parallel sorting algorithms has been heavily investigated already since the earliest days of parallel computing, namely, since the late 1960s, and many parallel sorting algorithms are known today.

In this chapter, we will consider four representatives:

- Parallel Quicksort including variants
- Parallel Samplesort
- Bitonic Sort
- Parallel Mergesort

Parallel Quicksort, Bitonic Sort and Parallel Mergesort are all based on the parallel divide-and-conquer algorithmic design pattern. They are presented here for shared-memory and analyzed for the PRAM model. The parallel samplesort algorithm also supports the distributed memory paradigm.

3.1 Parallel Quicksort

3.1.1 Sequential Quicksort Revisited

Quicksort is one of the most prominent incarnations of the algorithmic design pattern *Divide-and-Conquer*. The pseudocode is shown in Figure 3.1.

[1]This chapter is relevant for TDDD56 only.

```
SeqQuicksort ( int $a[l:r]$ )
{ // sort $n = r - l + 1$ elements in $a$
  if $l \geq r$ // base case $n = r - l + 1 \geq 1$
    return; // nothing to sort
  // divide phase:
  choose a pivot $a[i]$ (e.g. randomly) for some $i$ in $l...r$;
  // determine $a_{low} = \{a[j] \in a[l:r]\text{: } a[j] \leq a[i]\}$ and $a_{high} = \{a[j] \in a[l:r]\text{: } a[j] \geq a[i]\}$:
  $k \leftarrow$ SeqPartition ( $a[l:r]$, $a[i]$ ); // partition $a$ in-place w.r.t. pivot $a[i]$
  // now the pivot is at position $k$.
  // recursive calls:
  SeqQuicksort ( $a[l:k-1]$ ); // sort $a_{low}$
  SeqQuicksort ( $a[k+1:r]$ ); // sort $a_{high}$
  // combine phase: trivial (no concatenation necessary due to in-place partitioning)
}
```

Figure 3.1: Sequential Quicksort algorithm. The invocation at the top level of the recursion tree should be SeqQuicksort($a[0:n-1]$).

It is well known that sequential quicksort has time complexity $O(n \log n)$ in the best case and on average, and $O(n^2)$ in the worst-case. The best case occurs if we happen to pick a pivot element such that partitioning splits the sequence perfectly in half in each recursion step.

3.1.1.1 Remarks on Pivot Choice

The worst case time in Quicksort occurs if we happen to pick the minimum or maximum element as the pivot element $a[i]$ in each step so the problem only gets smaller by one per recursion step. Hence, pivot choice strategies that perform well on average are important.

Just picking always the first or last element in the subsequence might lead to the worst-case scenario e.g. for pre-sorted input data.

If the pivot element is chosen randomly, quicksort is still not unlikely to generate unbalanced subproblems, but the average case complexity will be in $O(n \log n)$, and the constant factors in this big-Oh term are usually lower for Quicksort than for most other sorting algorithms.

A better (but also slightly more expensive) strategy is to draw a random sample e.g. of size $O(\sqrt{n})$ and choose the pivot element as the median of these. This improves the average size balance of a_{low} to a_{high}.

3.1.1.2 Remarks on Partitioning

Figure 3.2 (left) shows the pseudocode for in-place partitioning (reordering) of an array of n elements. The principle is illustrated in Figure 3.2 (right).

The in-place partitioning avoids allocating separate temporary arrays for a_{low} and a_{high}. Access through pointers/indices suffices, and concatenation is implicit by having the subarrays already in the correct relative order, so that no additional copying is required. Moreover, the algorithm is cache-friendly (why?).

```
int SeqPartition ( int $a[l : r]$, $pivpos$ )
{
  if $(r - l = 0)$   // $n = 1$, nothing to partition
    return 0;
  $pivot \leftarrow a[pivpos]$;  // copy pivot element for comparisons
  int $left = l$, $right = r + 1$;
  while $(left < right)$ {
    $left$++;
    while ( $left \leq r$ and $a[left] < pivot$ )  $left$++;
    $right$++;
    while ( $right \geq l$ and $a[right] > pivot$ )  $right$--;
    if $(left \leq r)$
      exchange $a[left \leftrightarrow a[right]$;
  }
  if $(left \leq r)$  // undo extra exchange:
    exchange $a[left] \leftrightarrow a[right]$ ;
  exchange $a[right] \leftrightarrow a[l]$;  // move pivot to its proper position $right$
  return $right - 1$;
}
```

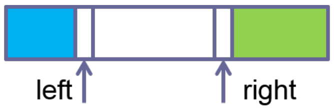


Figure 3.2: Sequential in-place partitioning

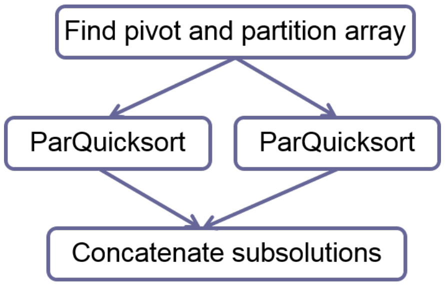


Figure 3.3: Simple Parallel Quicksort: dependence structure for the recursive case

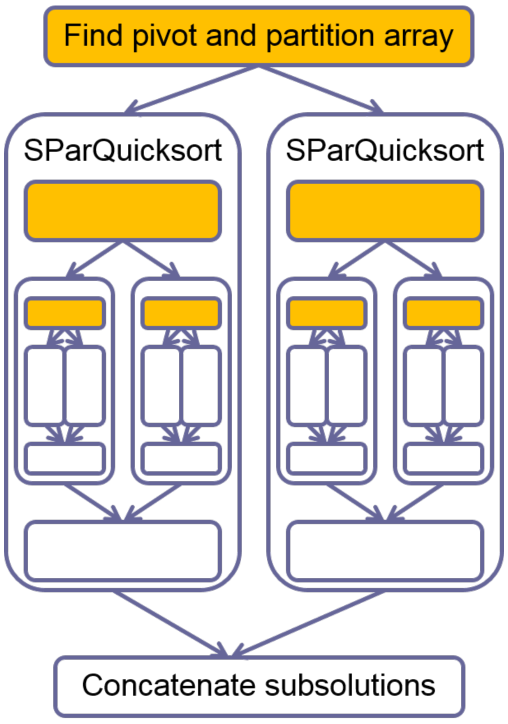


Figure 3.4: Simple Parallel Quicksort, unfolded three times

3.1.2 Simple Parallel Quicksort

Divide-and-conquer algorithms reduce a given problem instance of size n to $q \geq 1$ subproblem instances of size $< n$. A fundamental property of divide-and-conquer is that these q subproblem instances (of size $< n$) are independent of each other. In the context of parallel processing, this means that for $q > 1$ the recursive solutions of these q subproblems can proceed in parallel, i.e., the q recursive calls define independent parallel tasks. This directly leads to the Simple Parallel Quicksort algorithm.

For the analysis of Simple Parallel Quicksort, let $T(n)$ denote its parallel execution time on a problem of size n on a n-processor PRAM. Obviously, $T(1) = O(1)$. For the recursive case, we obtain the recurrence equation

$$T(n) \;\geq\; T(n/2) + T_{partition}(n) + T_{concat}(n) + O(1) \;=\; T(n/2) + O(n)$$

In the best case,

$$T(n) \;=\; T(n/2) + O(n) \;=\; O(n) + O(n/2) + O(n/4) + ... + O(1) \;=\; O(n)$$

The work performed is $O(n \log n)$ in the best case (Why?).

We notice that the sequential partitioning (with time linear in the input size) does all the heavy work in Quicksort. It becomes the performance bottleneck for Simple Parallel Quicksort. In particular, the first call to partitioning for the entire n element array is executed by one processor only (Hint: remember Amdahl's Law ...).

In general, we can conclude that for parallel divide-and-conquer algorithms with nontrivial divide and combine operations the achievable speedup is generally quite limited if the divide and the combine operations are not parallelized.

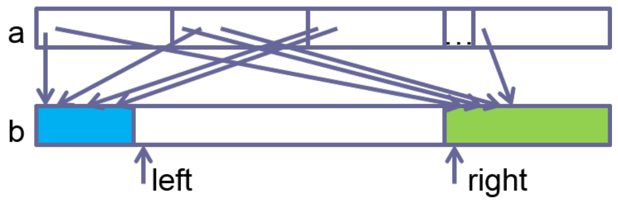


Figure 3.5: Parallel partitioning.

Considering its asymptotic behavior for time, work and cost for large numbers of processors (such as $p \in \Theta(n)$), Simple Parallel Quicksort is very far from optimal. In particular, the speedup is then limited by $O(\log n)$ (why?).

For small processor counts such as $p \in O(\log n)$ the asymptotic behavior is better but performance suffers from the early, heavy-weight partition calls done by one or very few processors only.

Unfolding the parallel divide-and-conquer recursion completely would unfold a recursion tree consisting of $\Omega(n)$ tasks (i.e., calls to Simple Parallel Quicksort), where the leaves represent the base case of the recursion and the tasks close to the leaves are all fairly light-weight. If using dynamic task creation and dynamic scheduling to map these to a fixed number p of processors, the aggregated overhead cost would be very high.

We can introduce a task granularity control mechanism that avoids too small tasks by suppressing creating and spawning new tasks for recursive calls where the problem size would be below a certain threshold such as n/p. In that case, the recursive call(s) would be executed sequentially by the same processor, i.e. the computation degenerates to ordinary sequential quicksort.

With this modification, the aggregated overhead is much lower but the parallel time is still $\Theta(n)$ in the best (and average) case: First, it is still dominated by the first sequential call to partition, which already uses time $\Theta(n)$. In the best case, the topmost, parallel part of the recursion tree has a depth of $\log_2 n - \log_2(n/p) = \log_2 p$ recursion steps until the problem sizes reach the threshold n/p, hence the overall parallel time for that part is dominated by the critical path of partition calls taking time $n + n/2 + n/4 + ... + n/p = \Theta(n)$ in the best-case scenario. In the "bottom" part of the recursion tree below that threshold, processing all non-spawned, sequential sort calls for problems of size $< n/p$ will take parallel time at least $O((n/p)\log(n/p))$ using p processors (why?).

3.1.3 Fully Parallel Quicksort

How can we parallelize the work-intensive partitioning step and obtain a Fully Parallel Quicksort?

While many PRAM processors could certainly perform comparisons and data moves in parallel, this requires proper and scalable synchronization. For example, for making sure that each element is taken care of by exactly one processor.

One approach could be to keep the degree-2 divide step in parallel divide-and-conquer and use shared counters *left* and *right* accessed by atomic fetch&increment operations on the shared counters in partitioning (see Figure 3.5). Of course, this requires that the hardware provides scalable (and constant-time) fetch&increment operations. If concurrent atomic fetch&increment operations are just serialized by the memory access interface in some order, as done in standard multicore processors today, there will be no gain, at least not in the asymptotic sense.

If scalable fetch&increment instructions are supported, we parallelize the main **while** loop in

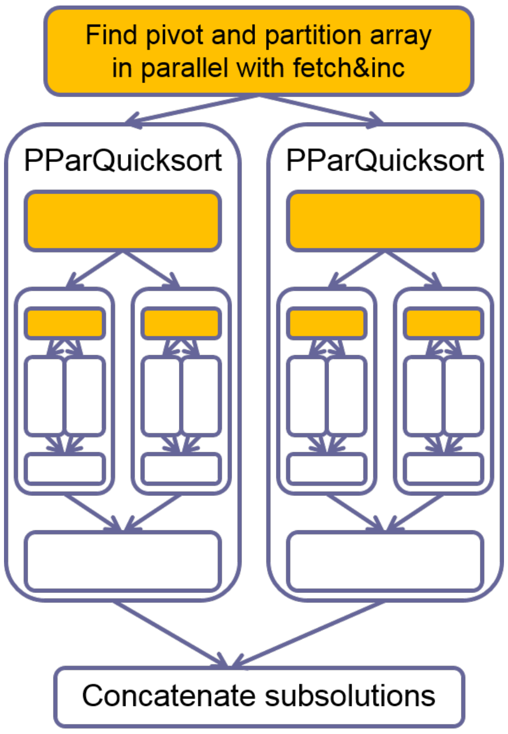


Figure 3.6: FULLY PARALLEL QUICKSORT, unfolded three times.

SEQPARTITION as follows. Each processor is responsible for the elements of a slice of array a of size n/p. We do no longer do in-place partitioning as before, but copy the elements from a to their new position in an auxiliary array, $b[0 : n-1]$. Finally, we copy back the array b in parallel to array a. Alternatively, one could pre-allocate the shadow array $b[0 : n-1]$ already before sorting and toggle between a and b as the target for partitioning, depending on the recursion depth in parallel Quicksort. The pseudocode is shown in Figure 3.7.

Let us analyze the resulting FULLY PARALLEL QUICKSORT with Fetch&Increment:

If up to p *fetch&inc* instructions can execute in parallel in constant time, parallel partitioning of n elements takes time $O(n/p)$ on p processors (why?), and it can use up to $p < n$ processors.

With that, the parallel time of FULLYPARALLELQUICKSORT becomes:

$$\begin{aligned} T(n) &\geq T(n/2) + T_{ParPartition}(n) + T_{concat}(n) + O(1) \\ &= T(n/2) + O(n/p) \\ &= O(n/p) + O(n/p) + ... + O(n/p) \\ &= O((n \log n)/p) \end{aligned}$$

The average-case parallel work of FULLYPARALLELQUICKSORT with fetch&increment is $O(n \log n)$.

The above time complexity holds only if the hardware implementation does allow to parallelize all concurrent *fetch&inc* operations. Up to $p = n$ concurrent calls to *fetch&inc* can occur per level, which is not practical for larger n, and even in the PRAM world this is only possible with a COMBINING CRCW PRAM, i.e., with an extremely strong (and thus, extremely expensive) shared memory interface.

As long as there is no such powerful hardware available, we could try to implement highly parallel *fetch&inc* in software instead.

```
int ParPartition( int $a[l:r]$, int $pivpos$ )
{ // pivot position $pivpos \in \{l, ..., r\}$
  int $n \leftarrow r - l + 1$; // problem size
  if $n \leq 1$ return 0; // base case

  $pivot \leftarrow a[pivpos]$;
  shared int $b[0:n-1]$; // temporary shared array
  shared int $left \leftarrow 0$, $right \leftarrow n-1$; // shared counters

  forall $i$ in $l...r$ do in parallel
    if ( $a[i] \leq pivot$ )
      $b[fetch\&inc(left, +1)] \leftarrow a[i]$;
    else
      $b[fetch\&inc(right, -1)] \leftarrow a[i]$;

  // and copy back $b$ to $a[l:r]$ in parallel for in-place partitioning:
  forall $i$ in $l...r$ do in parallel
    $a[i] \leftarrow b[i-l]$;
  return $left - 1$;
}
```

Figure 3.7: Parallel partitioning using *fetch&inc*

.

A straightforward solution, a software implementation of *fetch&inc* protected by a mutex lock, would serialize all *fetch&inc* operations anyway.

But we know that parallel *fetch&inc* operations can also be calculated using parallel prefix sums, for which linear-work parallel algorithms with logarithmic time complexity are known that can scale up to n processors. Hence, a factor $\log p$ must be added to the time complexity:

$$T(n,p) \;\geq\; O((n/p)\log n \log p)$$

and, for $p = n$, we obtain the expected parallel time $\Theta((\log n)^2)$.

3.2 Parallel Sample Sort

The parallel Quicksort variants suffered from the problem of efficiently dividing a large problem instance sufficiently fast into at least p independent units of work. Moreover, Quicksort suffers by design from a quadratic worst-case work complexity, translating into bad load balance in the parallel variants; finding good pivots with very high probability is thus essential for efficiency.

Parallel Sample Sort is a sorting method that is based on p-way partitioning rather than 2-way partitioning as in Quicksort. If the p-way partitioning can be done efficiently in parallel even for larger p, the sorting algorithm needs not be recursive any more.

We first choose $p-1$ pivots (splitters) $s_1 ... s_{p-1}$ from array a, sort them (sequentially), and add the two artificial pivots $s_0 = -\infty$ and $s_p = \infty$. See also Figure 3.8 for illustration.

Now, each processor $i \in \{0, ..., p-1\}$ creates p empty local lists $L_{i,j}$ for $j = 0, ..., p-1$ and partitions n/p elements of input array a into its corresponding p local lists according to the pivots:

- $a[k]$ into list $L_{i,j}$ iff $s_j < a[k] < s_{j+1}$, for $k = i(n/p)...(i+1)(n/p) - 1$.

Each processor j then sorts its list L_j sequentially. Finally, each processor j gathers all partial lists $L_{i,j}$ into list L_j.

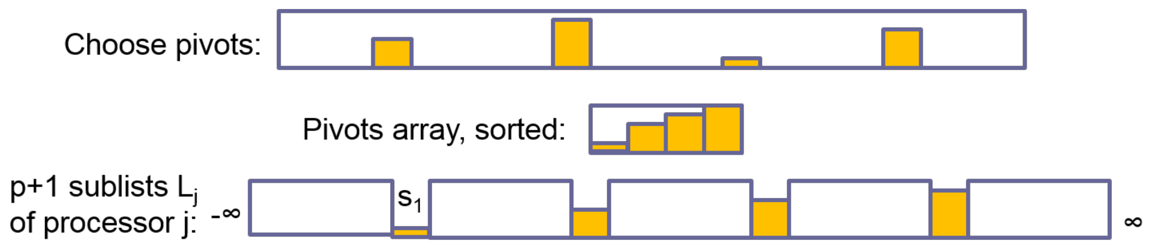


Figure 3.8: Parallel Sample Sort visualization

3.2.1 Example

In this tiny example, we assume $p = 3$ processors and an input array of size $n = 15$:

9, 3, 17, 4, 5, 20, 19, 11, 1, 8, 7, 2, 15, 14, 6

Let us assume that the random sample of size $\lceil\sqrt{15}\rceil = 4$ pivot candidates is

17, 14, 7, 6

We sort these on one processor and choose $p - 1$ pivots, here every $\sqrt{n}/p \approx 2$ entries:

7, 17

Now the processors partition in parallel: each processor goes through its slice of $n/p = 5$ input elements, and adds each element to one of p local lists.

Processor 0:	lists	$L_{0,0} = (3, 4, 5)$,	$L_{0,1} = (9)$,	$L_{0,2} = (17)$
Processor 1:	lists	$L_{1,0} = (1)$,	$L_{1,1} = (11, 8)$,	$L_{1,2} = (20, 19)$
Processor 2:	lists	$L_{2,0} = (7, 2, 6)$,	$L_{2,1} = (15, 14)$,	$L_{2,2} = ()$

Then each processor concatenates the sublists $L_{*,j}$ matching its ID j to obtain L_j and sorts them:

Processor 0 sorts: $L_0 = (3,\ 4,\ 5,\ 1,\ 7,\ 2,\ 6) \ \rightarrow\ (1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7)$
Processor 1 sorts: $L_1 = (9,\ 11,\ 8,\ 15,\ 14) \ \rightarrow\ (8,\ 9,\ 11,\ 14,\ 15)$
Processor 2 sorts: $L_2 = (17,\ 20,\ 19) \ \rightarrow\ (17,\ 19,\ 20)$

3.2.2 Analysis

Sequential sorting of the $p - 1$ pivots takes time $O(p \log p)$.

Parallel p-way-partitioning can be done in time $O((n/p) \log p)$ because the right list $L_{i,j}$ can be determined by binary search in the sorted pivots array.

Assuming an average case length of every list L_j in $O(n/p)$, the simultaneous sequential sorting of p lists L_j can be done in time $O((n/p) \log(n/p))$.

In total, the parallel time will thus be

$$O((n/p + p) \log(n/p + p))$$

which is time-optimal for $p \in O(n/p)$, i.e., for $p \in O(\sqrt{n})$.

3.2.3 Discussion

The advantages of Parallel Samplesort compared to Fully Parallel Quicksort are the following:

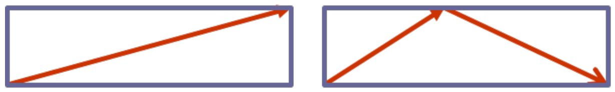


Figure 3.9: Visualizations of a monotonic sequence (left) and a bitonic sequence (right).

- no recursive calls: the problem instance is immediately decomposed into small pieces of size (approximately) n/p;
- Parallel Samplesort can also be used on message-passing machines (which requires one one-to-all (broadcast) communication for the pivot elements and parallel all-to-one (multi-gather) communications for the sublists $L_{i,j}$ to processor j).

In contrast, the disadvantages are rather minor:

- Up to $O(\sqrt{n})$ processors can be used effectively;
- the partitioning and sorting is not in-place, the lists $L_{i,j}$ need separate array(s);
- there might be minor load imbalances, however the probability for large imbalance is very low, due to randomly selecting and sorting $\sqrt{n}$ splitter candidates.

3.3 Bitonic Sort

From mathematics we know *monotonic* sequences. A monotonic sequence is ordered, either monotonically increasing or decreasing, see Figure 3.9 (left) for an illustration.

We can generalize this concept to *bitonic sequences*, which are composed either from a monotonically increasing subsequence followed by a monotonically decreasing subsequence, or vice versa. In other words, bitonic sequences have one "peak" that separates the two subsequences with different orders. See Figure 3.9 (right) for an illustration. The peak can be at any position in the sequence. In particular, each monotonic sequence is also a bitonic sequence (but not vice versa).

Formally, a sequence of numbers $a = (a_0, ..., a_{n-1})$ is called *bitonic* if either there is a k in $\{0, .., n-1\}$ such that $a_0 = ... = a_k = ... = a_{n-1}$ or the sequence can be rotated to that form.

As an example, $a = (4, 1, 2, 7, 6, 5)$ is a bitonic sequence (it could be rotated left to $(1, 2, 7, 6, 5, 4)$. The peak position is at $k = 3$, i.e., $a_3 = 7$ is the largest element, the subsequence $(1, 2, 7)$ is the ascending part, and $(6, 5, 4)$ is the descending part

In contrast, $(4, 1, 7, 6, 5, 2)$ is not a bitonic sequence as it cannot be rotated to have a unique peak.

3.3.1 Batcher's Lemma and Min-Max Partitioning

Theorem 3.1 (***Batcher's Lemma***) *[5]:*

If a is a bitonic sequence, then the sequences

$$a' = \min(a_1, a_{n/2+1}), ..., \min(a_{n/2}, a_n) \text{ and}$$

$$a'' = \max(a_1, a_{n/2+1}), ..., \max(a_{n/2}, a_n)$$

are both bitonic and $\max(a') = \min(a'')$.

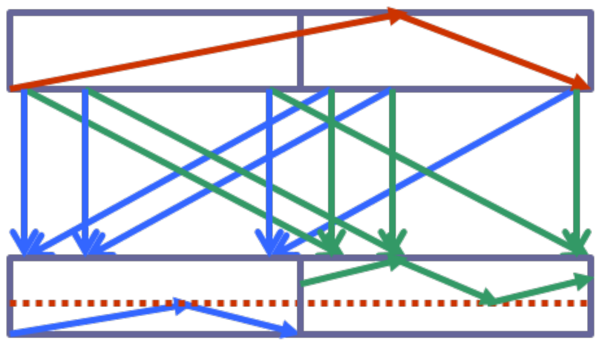


Figure 3.10: Batcher's parallel min-max partitioning step in Bitonic Merge.

```
BitonicMerge ( int $a[0:n-1]$, int $order$)
{ // $a$ must be bitonic
    compute $a'$, $a''$ according to Batcher's Lemma if $order$ is ascending
    (do it with min and max exchanged if $order$ is descending)
    return concat( BitonicMerge( $a'$, $order$ ), BitonicMerge( $a''$, $order$));
} // now sorted in order
```

Figure 3.11: Bitonic Merge algorithm. The parameter *order* (either ascending or descending) will be needed later for bitonic sorting.

The proof is not shown here, see e.g. Cormen *et al.* [17].

As an example, we apply Batcher's lemma to the sequence $a = (4, 1, 2, 7, 6, 5)$ from above. Then we obtain:

$a' = (\min(4,7), \min(1,6), \min(2,5)) = (4, 1, 2)$, which is bitonic.

$a'' = (\max(4,7), \max(1,6), \max(2,5)) = (7, 6, 5)$, which is bitonic.

3.3.2 Bitonic Merge

Batcher's Lemma implies that we can use the min-max computation as a divide step to split a bitonic sequence into two of half the size that can be processed independently as they are properly separated (like a_{left} and a_{right} in Simple Parallel Quicksort).

This property is exploited in a parallel divide-and-conquer based parallel algorithm called Bitonic Merge. Figure 3.11 shows the Bitonic Merge algorithm and Figure 3.12 illustrates its parallel execution for three unfolded parallel recursion steps.

BitonicMerge follows the parallel divide-and-conquer algorithmic design pattern.

A bitonic sequence can be bitonic-merged in parallel time $\Theta(\log n)$ with n processors ($n/2$ doing min, $n/2$ doing max):

The base case of the (parallel) recursion is n=2, which is trivial to merge from two length-1 subsequences.

The parallel time of BitonicMerge is:

$$\begin{aligned} T_{BitonicMerge}(1) &= O(1) \\ T_{BitonicMerge}(n) &= T_{BitonicMerge}(n/2) + T_{MinMax}(n) + T_{concat}(n) + O(1) \end{aligned}$$

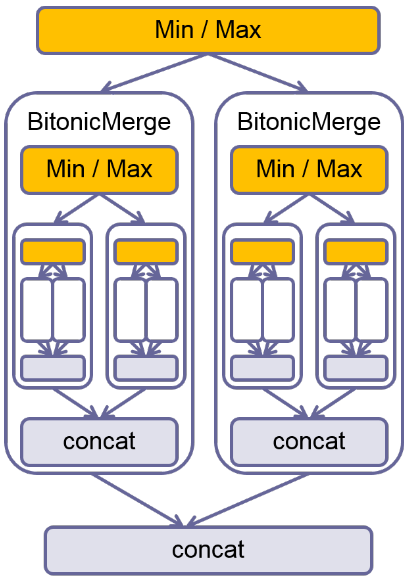


Figure 3.12: Bitonic Merge, unfolded three times.

Hence,

$$\begin{aligned} T_{BitonicMerge}(n) &= T_{BitonicMerge}(n/2) + O(1) \\ &= O(1) + O(1) + ... + O(1) \\ &= O(\log n) \end{aligned}$$

The work performed by BITONICMERGE is:

$$\begin{aligned} W_{BitonicMerge}(1) &= O(1) \\ W_{BitonicMerge}(n) &= 2W_{BitonicMerge}(n/2) + O(n) \\ &= ... \\ &= O(n \log n) \end{aligned}$$

3.3.3 Bitonic Sort

We have seen that BITONICMERGE brings any bitonic sequence into sorted form. But how do we derive, from unsorted input data, a bitonic sequence to get started?

The idea of bitonic sorting is to use BITONICMERGE as the combine step in yet another parallel divide-and-conquer algorithm: We sort a nontrivial problem instance (an arbitrary sequence) by splitting it in half and recursively sorting its halves in ascending and descending order respectively. These resulting subsequences are (by induction assumption) monotonic, we concatenate these to a bitonic sequence and flip the *order* where necessary, then we can run BITONICMERGE to bring the entire sequence into sorted form. The pseudocode and an illustration of the overall structure of the BITONICSORT algorithm in Figure 3.13.

The parallel time $T(n)$ of BITONIC SORT is

```
BitonicSort ( int $a[n]$, int $order$ )
{  // $a$ is an arbitrary sequence;
   // $order \in$ {ascending, descending}
   BitonicSort( $a[1{:}n/2]$, ascending ); // on procs. $0,\dots,n/2-1$
   ||      // in parallel with:
   BitonicSort( $a[n/2+1{:}n]$, descending ); // on procs. $n/2,\dots,n-1$
   // a is now bitonic
   BitonicMerge ( $a$, $order$ );
}
```

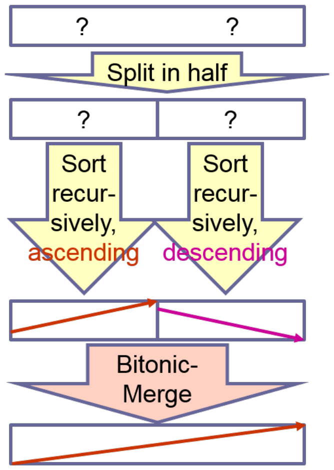


Figure 3.13: Left: Bitonic Sort pseudocode. Right: Overall structure of the recursive case.

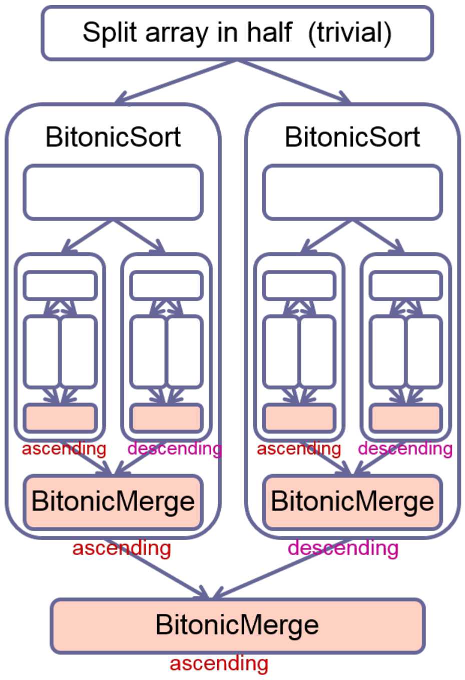


Figure 3.14: Bitonic Sort, unfolded three times.

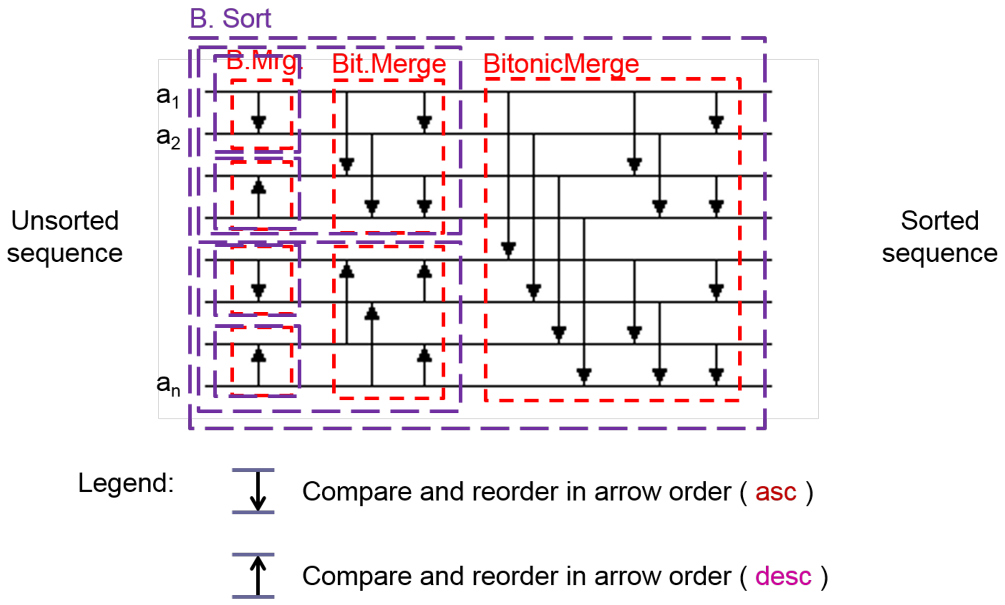

B. Sort
B.Mrg.
Bit.Merge
BitonicMerge
a1
a2
an
Unsorted sequence
Sorted sequence
Legend:
Compare and reorder in arrow order (asc)
Compare and reorder in arrow order (desc)

Figure 3.15: Bitonic Sorting unfolded for $p = n = 8$, showing the nested parallel divide-and-conquer structure.

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/2) + T_{split}(n) + T_{BitonicMerge}(n) + O(1) \\ &= T(n/2) + O(\log n) \\ &= O(\log n) + O(\log n/2) + ... + O(1) \\ &= O(\log^2 n) \end{aligned}$$

Hence, Bitonic Sort is not time-optimal; we know that a parallel sorting network with time $O(\log n)$ exists [2], though that one has a prohibitively high constant factor and is thus only of theoretical interest. For Bitonic Sort the constant factor is very small.

The parallel work $W(n)$ of Bitonic Sort is

$$\begin{aligned} W(1) &= O(1) \\ W(n) &= 2W(n/2) + O(n \log n) \\ &= ... \\ &= O(n \log^2 n) \end{aligned}$$

3.3.4 Remarks on Bitonic Sort

Bitonic Sort is a *sorting network*; it has been designed as a massively parallel hardware algorithm (comparator network) from the beginning.

This is possible because Bitonic Sort is an *oblivious algorithm*, i.e., its control flow only depends on input size, not on input contents: Given the input size, there is no difference in its

```
SeqMerge ( int $a[k]$, int $b[k]$, int $c[2k]$ )
{
  int $ap \leftarrow 0$,  $bp \leftarrow 0$,  $cp \leftarrow 0$;
  while ( $cp < 2k$ ) { // assume $a[k] = b[k] = \infty$
    if ($a[ap] < b[bp]$)  $c[cp{+}{+}] \leftarrow a[ap{+}{+}]$;
    else $c[cp{+}{+}] \leftarrow b[bp{+}{+}]$;
  }
}
```

Figure 3.16: Sequential merging of two ordered sequences a, b into sequence c.

```
SeqMergesort ( int $a[0:n-1]$, int $c[0:n-1]$ )
{ // input array in $a$, output array in $c$
  if ($n = 1$) return;
  // divide and conquer:
  SeqMergesort ( $a[0{:}n/2-1]$, $c[0{:}n/2-1]$ );  // $n/2$ elements
  SeqMergesort ( $a[n/2{:}n-1]$, $c[n/2{:}n-1]$ );  // $n-n/2$ elements
  // now the subarrays in $c$ are sorted.
  SeqMerge ( $c[0:n/2-1]$, $c[n/2:n-1]$, $a[0:n-1]$ );
}
```

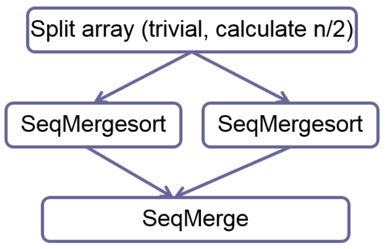


Figure 3.17: Left: Sequential Mergesort algorithm. Right: Dependence structure of the recursive case.

asymptotic execution time in the best, average and worst case. In contrast, (parallel) Quicksort and Samplesort are non-oblivious algorithms.

Bitonic Sort has stable, well predictable performance, hence it is well suited for realtime computing.

For using $p < n$ processors, we can again add granularity control: The parallel recursion in BitonicSort is stopped at problem size n/p, all smaller problem instances are sorted sequentially instead (for instance, using Sequential Mergesort).

3.4 Parallel Mergesort

3.4.1 Sequential Mergesort revisited

Mergesort is a well-known sequential algorithm and a typical representative of divide-and-conquer algorithms. It recursively cuts an unsorted sequence in two halves, sorts each one recursively, and merges the two sorted subsequences into one. The base case is, if not stated otherwise, reached for 1-element sequences, which are trivially sorted.

Merging takes two sorted blocks of length k and combines them into one sorted block of length $2k$.

The well-known sequential merging algorithm is shown in Figure 3.16. Its time complexity is $O(k)$. The algorithm could also be formulated for in-place merging (copying back).

The execution time of SeqMergesort is $\Theta(n \log n)$. (Why?)

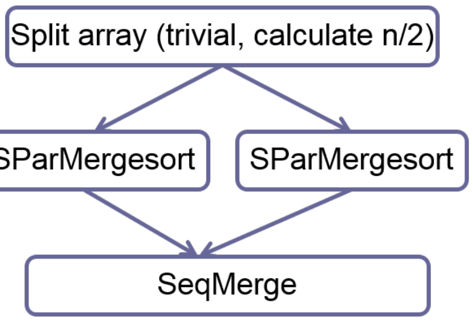


Figure 3.18: Simple Parallel Mergesort, dependence structure of the recursive case.

```
SParMergesort ( int $a[0:n-1]$ )
{
  if ($n = 1$) return;  // nothing to sort
  if (depth_limit_for_recursive_parallel_decomposition_reached()) {
    SeqMergesort( $a[0:n-1]$ ); // switch to sequential
    return;
  // parallel recursion:
  SParMergesort( $a[0:n/2-1]$, $c[0:n/2-1]$ );  // on procs. $0,...,n/2-1$
  ||      // in parallel with:
  SParMergesort( $a[n/2:n-1]$, $c[n/2,n-1]$ );  // on procs. $n/2,...,n-1$
  // now the two subarrays are sorted
  seq  // on processor 0 only:
    SeqMerge( $a[0:n/2-1]$, $a[n/2:n-1]$, $c[0:n-1]$ );
}
```

Figure 3.19: Simple Parallel Mergesort algorithm.

3.4.2 Simple Parallel Mergesort

Mergesort, being based on the divide-and-conquer pattern, exposes independent subproblem instances in the form of its two recursive calls. On a parallel computer, we could run these independent calls as parallel tasks, on different resources (cores) if available. This results in the Simple Parallel Mergesort algorithm.

Unfortunately there is not much parallelism near the root, which is just where merging becomes much more work-intensive. The last merge in the root of the recursion tree takes linear time and becomes the performance bottleneck in Simple Parallel Mergesort. As with parallel Quicksort variants, we can limit the recursive parallel decomposition to a maximum recursion depth (resp., minimum problem size), beyond which we switch to SeqMergesort in order to control the number of tasks generated.

The parallel time of Simple Parallel Mergesort is

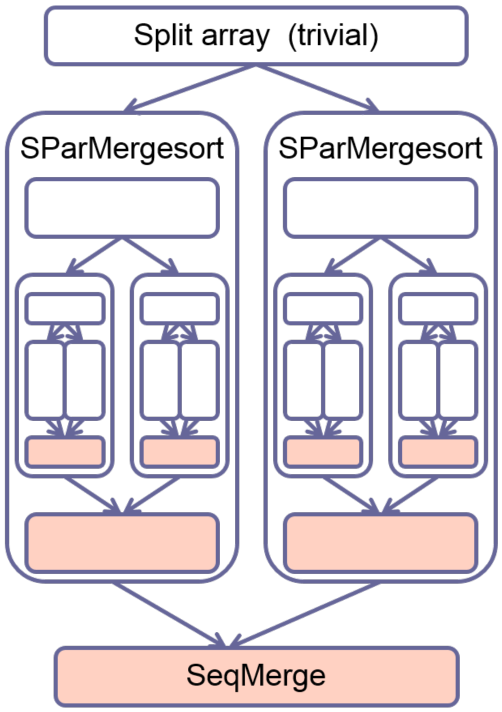


Figure 3.20: Simple Parallel Mergesort, unfolded three times.

$$
\begin{aligned}
T(1) &= O(1) \\
T(n) &= T(n/2) + T_{split}(n) + T_{SeqMerge}(n) + O(1) \\
&= T(n/2) + \Theta(n) \\
&= \Theta(n) + \Theta(n/2) + \Theta(n/4) + ... + O(1) \\
&= O(n)
\end{aligned}
$$

The parallel work is in $\Theta(n \log n)$.

The structure of Simple Parallel Mergesort is symmetric to Simple Parallel Quicksort. Here, all the heavy work is done in the SeqMerge() calls, which is the counterpart of SeqPartition in Simple Parallel Quicksort.

This limits speedup to $O(\log n)$.

3.4.3 Fully Parallel Mergesort

3.4.3.1 How to Merge in Parallel?

For each element of the two arrays to be merged, we calculate its final position in the merged array by cross-ranking. The *rank* of any value x with respect to an ordered array a is defined by

$$rank(x, (a_0, ..., a_{n-1})) \;=\; \#\{\text{elements } a_i < x\}$$

We can compute the rank by a sequential binary search in a, see Figure 3.21 for illustration. We know that binary search on a sorted sequence of length n takes time $O(\log n)$.

The pseudocode for parallel merging is given in Figure 3.22.

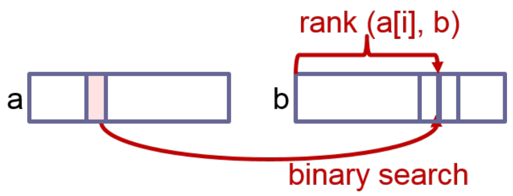


Figure 3.21: Finding the rank of $a[i]$ in b by binary search.

```
PARMERGE ( int $a[n_1]$, int $b[n_2]$, int $c[n_1 + n_2]$ )
{
  // Merge sequences $a$ and $b$ in parallel using cross-ranking
  // Simplifying assumption: All elements in both a and b are pairwise different
  for all $i \in \{0, ..., n_1 - 1\}$ in parallel
    $rank\_a\_in\_b[i]$ = COMPUTE_RANK( $a[i]$, $b$, $n_2$ );
  for all $i \in \{0, ..., n_2 - 1\}$ in parallel
    $rank\_b\_in\_a[i]$ = COMPUTE_RANK( $b[i]$, $a$, $n_1$ );
  for all $i \in \{0, ..., n_1 - 1\}$ in parallel
    $c[i + rank\_a\_in\_b[i]] \leftarrow a[i]$;
  for all $i \in \{0, ..., n_2 - 1\}$ in parallel
    $c[i + rank\_b\_in\_a[i]] \leftarrow b[i]$;
}
```

Figure 3.22: Parallel Merging by cross-ranking.

As the time for one binary search in an ordered sequence of n elements is $O(\log n)$ and all searches can proceed in parallel if done by their own PRAM processor, the parallel time for ParMerge is $O(\log n)$, too. The work of ParMerge is $O(n \log n)$.

3.4.3.2 Example for Parallel Merging

As an example for parallel merging, consider the two sorted arrays $a = (2,3,7,9)$ and $b = (1,4,5,8)$. As common in C-based programming languages, array indices start at 0. We can use up to 8 processors, where each processor computes the rank of one element of a and b w.r.t. the other array by binary search. Cross-ranking yields:

$$\begin{aligned} rank\_a\_in\_b &= (1,1,3,4) \\ rank\_b\_in\_a &= (0,2,2,3) \end{aligned}$$

Using the rank arrays, the processors can determine the destination indices and copy in parallel as follows:

Processor 0:	moves $a[0]$ to $c[0+1] = c[1]$
Processor 1:	moves $a[1]$ to $c[1+1] = c[2]$
Processor 2:	moves $a[2]$ to $c[2+3] = c[5]$
Processor 3:	moves $a[3]$ to $c[3+4] = c[7]$
Processor 4:	moves $b[0]$ to $c[0+0] = c[0]$
Processor 5:	moves $b[1]$ to $c[1+2] = c[3]$
Processor 6:	moves $b[2]$ to $c[2+2] = c[4]$
Processor 7:	moves $b[3]$ to $c[3+3] = c[6]$

After copying, we have

$$c = (1,2,3,4,5,7,8,9)$$

3.4.3.3 Analysis of Fully Parallel Mergesort

By replacing SeqMerge with ParMerge, we obtain the Fully Parallel Mergesort algorithm, which is visualized in Figure 3.23.

The parallel time of Fully Parallel Mergesort is

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/2) + T_{split}(n) + T_{ParMerge}(n) + O(1) \\ &= T(n/2) + O(\log n) \\ &= O(\log n) + O(\log n/2) + ... + O(1) \\ &= O(\log^2 n) \end{aligned}$$

The work of Fully Parallel Mergesort is

$$\begin{aligned} W(1) &= O(1) \\ W(n) &= 2W(n/2) + \Theta(n \log n) \\ &= \Theta(n \log^2 n) \end{aligned}$$

Hence, the asymptotic time and work complexities of Fully Parallel Mergesort are on-par with Bitonic Sort.

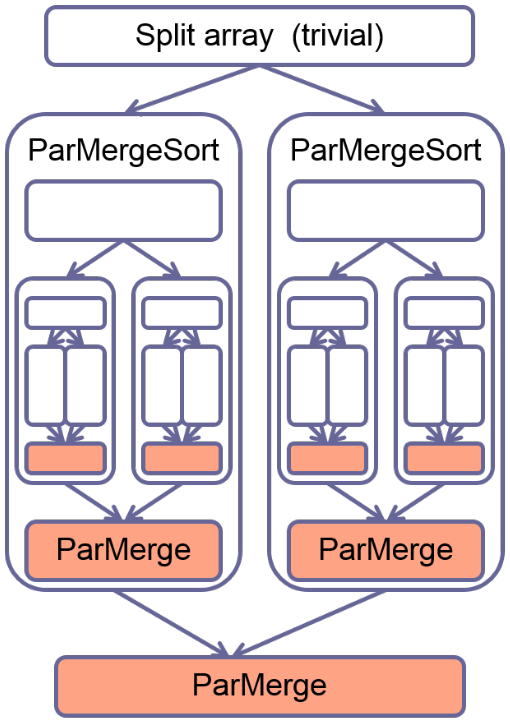


Figure 3.23: Fully Parallel Mergesort, unfolded three times.

Fully Parallel Mergesort is, like Bitonic Sort, an oblivious algorithm, hence it could be used for constructing a sorting network.

Task granularity control for a given fixed number p of processors can be introduced like in the other parallel divide-and-conquer based sorting algorithms. The resulting parallel time on p processors is

$$T(n,p) = \Theta\left(\frac{n}{p}\log n + (\log n)^2\right).$$

3.5 Chapter Summary

We considered a few common parallel sorting algorithms:

- Simple Parallel Quicksort, where partitioning remains sequential;
- Fully Parallel Quicksort using parallel Fetch&Increment or parallel prefix sums for parallel partitioning;
- Parallel samplesort, which is not recursive and uses p-way partitioning;
- Bitonic Sort, a parallel sorting network;
- Simple Parallel Mergesort, another sorting network with similar properties as Bitonic Sort;
- Fully Parallel Mergesort, where the parallel merging is achieved using binary search.

Parallel Quicksort, Bitonic Sort, and Parallel Mergesort are based on the Parallel Divide-and-Conqer algorithmic design pattern; the latter two use it actually twice, at outer level to parallelize sorting and at inner level to parallelize merging. Hence, we have identified a new algorithmic design pattern:

- Nested Parallel Divide-and-Conquer

Many more parallel sorting algorithms exist: e.g. parallel rank sort, parallel radix sort, ...

Algorithm engineering necessary to adapt textbook algorithms to perform efficiently on real parallel systems

Example: use SIMD hardware, run on GPUs, on distributed memory, on special network topologies,

3.6 Bibliographical Notes *

For a review of sequential sorting algorithms we refer to standard textbooks on algorithms, such as Cormen et al. [17].

Tsigas and Zhang [60] present a variant of parallel Quicksort for shared-memory multiprocessors. GPU-specific algorithms for parallel quicksort and parallel samplesort have been contributed e.g. by Cederman and Tsigas [14] and by Leischner, Osipov and Sanders [47], respectively.

Träff [59] shows that the overall amount of data movement in (fully) parallel quicksort on clusters can be reduced from (expected) $O(n \log n)$ to $O(n)$, by replacing the traditional pairwise element exchange in partitioning by exchanging pivots only and postponing the element redistributions to the leaves of the recursion tree.

There do exist time-optimal ($O(\log n)$) parallel sorting algorithms for n processors, but these are very complex: The AKS-network [2] and Cole's pipelined parallel mergesort [15].

3.7 Exercises

1. Self-Assessment: Fill in the table!

Parallel Sorting Algorithm	Max #Procs Used	Parallel Time	Parallel Work	Parallel Cost	Restrictions
SIMPLE PARALLEL QUICKSORT					
FULLY PARALLEL QUICKSORT, using fetch&incr.					
FULLY PAR. QUICKSORT with par. prefix sums					
PARALLEL SAMPLESORT					
BITONIC SORT					
SIMPLE PARALLEL MERGESORT					
FULLY PARALLEL MERGESORT					

2. Explain why sequential in-place partitioning is cache-friendly.

3. Actually, FULLY PARALLEL QUICKSORT using certain algorithms for the parallel prefix sums in parallel partitioning is also an instance of the Nested parallel divide-and-conquer pattern. Explain!

4. Elaborate on the granularity control in BITONIC SORT for $p < n$ and derive the resulting parallel execution time $T(n, p)$.

5. Give an iterative formulation of parallel merge sort (use a while loop instead of recursion)

6. Elaborate on the granularity control in Fully Parallel Mergesort. Show that the parallel time complexity $T(n,p) = \Theta\left(\frac{n}{p}\log n + (\log n)^2\right)$ can be achieved for given $p < n$ processors.

7. On a Combining CRCW PRAM it is actually possible to sort n pairwise different numbers in *constant time* using n^2 processors (***Hint***: Each of the n^2 processors $p_{i,j}$ is responsible for comparing two distinct elements $a[i]$ and $a[j]$).

   (a) Sketch the algorithm, and discuss its work, cost, and whether it is work-optimal/cost-optimal or not.

   (b) Explain where the Combining CRCW property is used, and how the algorithm would need to be adapted for weaker PRAM variants.

   (c) Can you extend the algorithm to make it work also if the elements are not pairwise different?

   (d) This algorithm is also known as Ranksort. Why?

Chapter Acknowledgements

We thank Jörg Keller from FernUniversität Hagen, Germany, for the parallel samplesort example and for an early version of the slide set on parallel sorting, out of which grew our slide material for the parallel sorting lecture in the course TDDD56 at Linköping University, which finally became the basis for this chapter.

Chapter 4

Parallel Linear Algebra Computations

4.1 Sequential Linear Algebra Computations

4.1.1 BLAS

BLAS (*Basic Linear Algebra Subroutines* is a de-facto standard library for (single-threaded execution of) linear algebra kernels that perform dataparallel operations on dense arrays (1D "vector", 2D "matrix") and that are frequently used operations in high-performance computing applications. It was introduced since 1979 as a sequential numerical library with interfaces for Fortran and C. Larger HPC software library packages such as LAPACK and LINPACK are built atop BLAS.

Good BLAS implementations are internally heavily optimized for (sequential) data locality, use of SIMD instructions, low loop overhead, and fast addressing.

There exist vendor-specific BLAS implementations (e.g. Intel MKL, AMD ACML, Cray libsci, Nvidia CUBLAS, etc.) and open-source implementations (e.g. the sequential reference BLAS (netlib), GNU GSL BLAS, ATLAS, GotoBLAS, OpenBLAS, etc.).

BLAS is organized in three levels, where the higher levels can be implemented using (calling) functionality implemented in lower levels:

- **Level-1 BLAS** [46]:

  Level-1 BLAS operations are vector-vector operations, such as adding or multiplying two vectors, dot product, vector sum, vector scale, vector copy, etc. A straightforward sequential implementation would use just one loop.

- **Level 2 BLAS** [22]:

  Level 2 BLAS provides fundamental vector-matrix operations, where a straightforward sequential implementations would use 2 nested loops. For example, generic dense matrix-vector multiplication.

- **Level 3 BLAS** [21]:

  Level 3 BLAS functions implement matrix-matrix operations where a straightforward sequential implementations would use 3 nested loops. For example, generic versions of dense LU decomposition and dense matrix-matrix multiplication.

In the following subsections we will consider each level with an example operation, and also elaborate on some features of BLAS.

Table 4.1: BLAS 1 Functions

`_ASUM`	Array sum (ℓ-1 norm of a vector): $s \leftarrow \lvert x\rvert = \sum_i x_i$
`_AXPY`	Elementwise add a scaled vector: $y \leftarrow \alpha x + y$
`_COPY`	Deep-copy a vector: $y \leftarrow x$
`_DOT`	Dot product of two vectors: $s \leftarrow x \cdot y = x^T y = \sum_i x_i \cdot y_i$
`I_AMAX`	Index of largest (absolute-value) vector element: first i such that $\forall k: \ \lvert x_i\rvert \geq \lvert x_k\rvert$
`_NRM2`	Euclidean (ℓ-2) norm of a vector: $s \leftarrow \lvert\lvert x\rvert\rvert_2$
`_ROTG`, `_ROT`	Generate / apply plane rotation
`_ROTMG`, `_ROTM`	Generate / apply modified plane rotation
`_SCAL`	Scale a vector: $y \leftarrow \alpha x$
`_SWAP`	Swap two vectors: $y \leftrightarrow x$

4.1.2 Level-1 BLAS

A summary of the Level-1 BLAS functions is given in Table 4.1, where operands x and y denote vectors, s a scalar variable, and α a scalar value. The placeholder symbol _ in the generic operation names must be filled with a type letter (see Section 4.1.3) to obtain a full function name.

As an example, let us consider the `_AXPY` operation, which elementwise multiplies a scalar α with a vector x and adds ("*p*lus", thus the name) elementwise to a vector y, to which the result is stored back. A straighforward sequential implementation is the following:

```
for(i=0; i<n; i++)
   y[i] = a[i] * x + y[i];
```

4.1.3 Data Types and Naming Convention

BLAS operations are designed for the HPC programming languages Fortran and C; they are usually the most frequently executed code parts of scientific applications, and often occur inside the implementations of other library functions. Hence, convenience for application-level programmers is considered less critical for the BLAS level, while BLAS implementations and invocations must be efficient; for example, the overhead of runtime type polymorphism (as known from some object-oriented languages such as Java) cannot be tolerated here. For this reason, the BLAS API is not polymorphic in the operands' element data type. Instead, the operands' element data type must be given explicitly, here in the form of a prefix letter to the operation name:

- `S` = single precision floatingpoint real,
- `D` = double precision floatingpoint real,
- `C` = complex single precision floatingpoint,
- `Z` = complex double precision floatingpoint.

Hence, the fully qualified names of BLAS-1 functions are, for example, `DCOPY` (double-precision floatingpoint vector copy), `SAXPY` (single-precision floatingpoint AXPY), `DASUM` (double-precision vector sum), `DDOT` (double-precision dot product), `CDOTC` (complex single-precision dot product) etc.

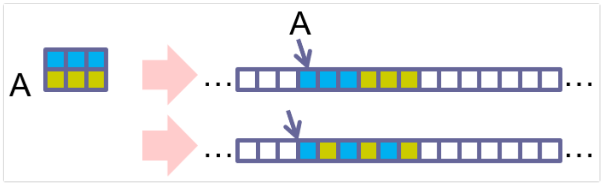


Figure 4.1: A two-dimensional array (matrix) (left) with its memory layout in C (top, row-wise) and Fortran (bottom, column-wise).

4.1.4 Vectors, Matrices and Special Matrix Types

The address space of a process in computer main memory is one-dimensional, hence multidimensional data is generally represented in memory in flattened (linearized to 1D) form. This flattening is dependent on the programming language used: In C (as well as in other C-based languages such as C++ and Java), linearization is done row-wise, i.e., the rows are stored successively in memory, while in Fortran linearization is column-wise, i.e., the columns are stored successively; see Figure 4.1. The necessary indexing code is then generated accordingly by the compiler.[1]

A *vector* (abstract data type) is a one-dimensional view (access, data packet) on an array of data in memory.

A *matrix* (abstract data type) is a two-dimensional view (access, data packet) on an array of data in memory. While the view (i.e., the interpretation of memory contents) is two-dimensional, it does not prevent the programmer (nor the implementation) from accessing the very same data elements in memory also in, for example, a one-dimensional view.

Beyond ordinary rectangular dense matrices, BLAS also supports a number of special matrix types with optimized storage formats. These include the following formats:

- *Triangular matrix*: a square matrix where elements below the main diagonal are zero and thus are not explicitly stored in memory;
- *Symmetric matrix*: a square matrix where $\forall i, j: \; a_{ij} = a_{ji}$ and thus only one of them is explicitly stored;
- *Hermitean matrix*: a complex square matrix that equals its own conjugate-complex transpose;
- *Banded matrix*: a matrix where only the main diagonal and a few other diagonals may contain non-zero elements and are stored explicitly;
- *Packed sparse matrix*: a matrix where only the non-zero elements are stored in a row-compressed format.

BLAS implementations also define the storage layout for these different matrix types. In the remainder of this chapter, we will only consider dense matrices.

BLAS-2 and BLAS-3 routines can utilize the knowledge about the flattening to speed up some operations. For example, a single loop over all elements is sufficient to initialize a 2D array with a fixed value, which leads to lower loop overhead compared to the straightforward implementation using two nested loops

[1] Additional linearization methods do exist, for example recursive fractal memory layouts such as Morton-order memory layout, which tries to strike a balance between the two extreme cases row-wise and column-wise layout and lead to, on average, better data locality for cases where both row-wise and column-wise accesses to the same multidimensional array occur together. Such memory layouts are usually realized in software atop the native row-wise linearization, using an index transformation library or code transformations by a compiler.

Table 4.2: BLAS 2 Functions (Selection)

`_GBMV`	General banded matrix-vector multiplication: $y \leftarrow \alpha Ax + \beta y$
`_GEMV`	General matrix-vector multiplication: $y \leftarrow \alpha Ax + \beta y$
`_GER`, ...	Rank-1 update (outer product): $A \leftarrow \alpha xy^T + A$
`_HEMV`, `_HBMV`, ...	Hermitian (banded) matrix-vector multiplication: $y \leftarrow \alpha Ax + \beta y$
`_SYR`	Symmetric rank-1 update: $A \leftarrow \alpha xx^T + A$
`_SYR2`	Symmetric rank-2 update: $A \leftarrow \alpha xy^T + \alpha^H yx^T + A$
`_TRMV`, `_TBMV`, ...	Triangular (banded) matrix-vector multiplication: $y \leftarrow \alpha Ax + \beta y$
`_TRSV`, `_TBSV`, ...	Triangular (banded) system solving (forward/backward substitution): $x \leftarrow \alpha A^{-1}x$

4.1.5 Level-2 BLAS

A summary of the Level-2 BLAS functions is given in Table 4.2, where x and y denote vectors with m and n elements respectively, A is a $n \times m$ matrix, and α and β are scalars. Most of them are variations of *generic matrix-vector multiplication*, of forward/backward substitution for direct triangular system solving, and of outer product (*rank-1 update*) operations, for the different matrix types introduced in Section 4.1.4.

As an example, `_GEMV` denotes *generic matrix-vector multiplication*, i.e., the operation $y \leftarrow \alpha Ax + \beta y$ where x and y are vectors with m and n elements respectively, A is a $n \times m$ matrix, and α and β are scalars. This generic form becomes the ordinary matrix-vector multiply for $\alpha = 1$ and $\beta = 0$, which we further consider as an example in this subsection.

As a straightforward sequential implementation would use two nested `for` loops for matrix-vector multiplication, there exist two basic implementation alternatives resulting from the two possible loop orderings: the ij-variant and the ji-variant, which we describe next.

4.1.5.1 ij-Variant

The ij-variant of matrix-vector multiplication has the following straightforward C implementation:

```
for(i=0; i<n; i++)
   for(j=0; j<m; j++)
     y[i] = A[i][j]*x[j] + y[i];
```

Or, expressed using BLAS-1 functions:

```
for(i=0; i<n; i++)
   sdot( &y[i], A[i], x, 1.0, 0.0, 1 );
```

i.e., a loop over n dot products, with stride 1 access in both A and x.

For efficiency reasons, `y[i]` should be kept in a register during accumulation. As some compilers might not do this for array elements, we often see hand-optimized BLAS implementations where separate scalar temporary variables are used for the accumulation to ”help” the compiler allocating a register.

The i-loop easy to parallelize later, because all its iterations with their dot product calls are independent of each other. `sdot` contains a reduction and is thus somewhat harder to vectorize, hence usually only the outer loop is used for parallelization as long as the resulting degree of parallelism is large enough.

Table 4.3: BLAS 3 Functions (Selection)

`_GEMM`	General matrix-matrix multiplication: $C \leftarrow \alpha AB + \beta C$
`_HEMM`	Hermitian matrix-matrix multiplication: $C \leftarrow \alpha AB + \beta C$
`_HERK`	Hermitian rank-k update: $C \leftarrow \alpha AA^H + \beta C$
`_SYMM`	Symmetric matrix-matrix multiplication: $C \leftarrow \alpha AB + \beta C$
`_SYRK`	Symmetric rank-k update: $C \leftarrow \alpha AA^T + \beta C$
`_SYR2K`	Symmetric rank-2-k update: $C \leftarrow \alpha AB^H + \alpha^H BA^H + \beta C$
`_TRMM`	Multiple triangular matrix-matrix multiplication: $B \leftarrow \alpha AB$
`_TRSM`	Multiple triangular system solving: $B \leftarrow \alpha A^{-1}B$

4.1.5.2 *ji*-Variant

The ji–variant of matrix-vector multiplication has the following straightforward C implementation:

```
for (j=0; j<m; j++)
   for (i=0; i<n; i++)
      y[i] = A[i][j]*x[j] + y[i];
```

Or, expressed using BLAS-1 functions:

```
for (j=0; j<m; j++)
   saxpy( y, &A[0][j], x[j], 1.0, 0.0, n );
```

i.e., a loop over m vector updates. More specifically, the j-loop is a reduction loop where the accumulator variable is now an entire vector and the combine operation in this reduction (the `saxpy` operation) is performed element-wise.

The innermost loop performs a stride n (column-wise) access in A, which is not so good (in C) from a data access locality point of view if the matrix is large. The accesses to x are stride-1.

The inner loop (`saxpy` operation) is easy to vectorize (once the data is contiguous in memory), and even to parallelize. However, the outer reduction loop, if left sequential, would force us to barrier-synchronize after each iteration, which adds significant overhead. Parallelizing the outer (reduction) loop is typically less efficient than a reduction-free data-parallel loop (as in the ij-variant).

Because of the data locality issue and for convenient outer loop parallelization, the ij variant is preferred in C. In Fortran, the ji variant has better data locality and might be preferable.

4.1.6 Level-3 BLAS

A summary of level-3 BLAS operations is shown in Table 4.3. where A, B, C denote matrices and α and β denote scalar values.

The most prominent BLAS-3 operation is *general matrix-matrix multiplication* (`_GEMM`), which denotes the operation $C \leftarrow \alpha AB + \beta C$ where C is a $n \times m$ matrix, A is a $n \times q$ matrix, B is a $q \times m$ matrix, and α and β are scalar values. The generic GEMM operation degenerates to ordinary matrix-matrix multiplication for $\alpha = 1$ and $\beta = 0$.

Matrix-matrix multiplication calculates each element C_{ij} of the result matrix as a dot product of the ith row vector of operand matrix A with the jth column vector of operand matrix B, see Figure 4.3.

Figure 4.4 illustrates the invocation of DGEMM as provided in the GNU GSL BLAS library.

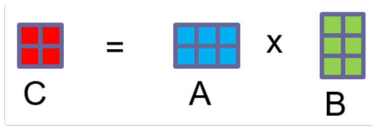


Figure 4.2: Matrix-Matrix Multiplication

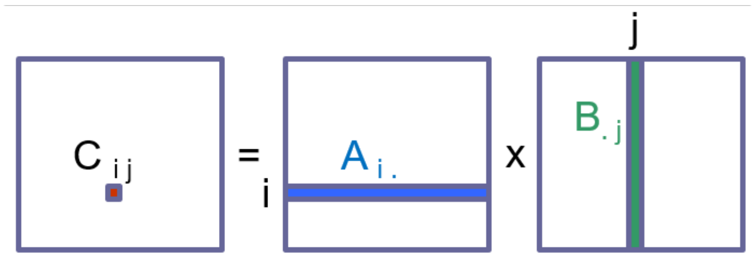


Figure 4.3: Matrix-Matrix Multiplication

```
#include <stdio.h>
#include <gsl/gsl_blas.h>

int main (void) ( )
{
  double a[] = { 0.11, 0.12, 0.13, 0.21, 0.22, 0.23 };
  double b[] = { 1011, 1012, 1021, 1022, 1031, 1032 };
  double c[] = { 0.00, 0.00, 0.00, 0.00 };
  gsl_matrix_view A = gsl_matrix_view_array( a, 2, 3 );
  gsl_matrix_view B = gsl_matrix_view_array( b, 3, 2 );
  gsl_matrix_view C = gsl_matrix_view_array( c, 2, 2 );
  /* Compute C = A B */
  gsl_blas_dgemm ( CblasNoTrans, CblasNoTrans, 1.0,
                   &A.matrix, &B.matrix, 0.0, &C.matrix );
  printf(" %g, %g \n", c[0], c[1] );
  printf(" %g, %g \n", c[2], c[3] );
  return 0;
}
```

Figure 4.4: Example code calling the GNU GSL BLAS funcction DGEMM.

4.1.7 BLAS Time Complexity Analysis

For the analysis of work/time and arithmetic intensity, we consider here one example each from BLAS-1, BLAS-2 and BLAS-3.

The _AXPY operation, here the sequential straightforward implementation

```
for(i=0; i<n; i++)
   y[i] = a[i] * x + y[i];
```

performs $O(n)$ work; specifically, n floatingpoint multiplications, n floatingpoint addition, $2n + 1$ floatingpoint loads and n floatingpoint stores. Hence, the arithmetic intensity is $2n/(3n + 1) < 1$. In other words, _AXPY is a memory-bound operation. The property that memory access volume is at least on par with the amount of arithmetic operations is common to all BLAS-1 operations.

For matrix-vector multiplication, here for the standard case ($\alpha = 1$, $\beta = 0$):

```
for (i=0; i<n; i++) {
   s = 0.0;
   for (j=0; j<m; j++)
      s = a[i][j] * x[j] + s;
   y[i] = s;
}
```

the work, and thus the sequential time complexity, is $O(nm)$: nm floatingpoint multiplications and $n(m - 1)$ floatingpoint additions, $nm + n + m$ floatingpoint loads and n floatingpoint stores. The arithmetic intensity (arithmetic operations per word of memory access) is thus about 2, i.e., the operation is still rather memory-bound (dominated by the matrix load), albeit less so than common BLAS-1 operations such as axpy or dot product.

For matrix-matrix multiplication, here the straightforward implemementation of the standard case of SGEMM,

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++) {
      float s = 0.0;
      for (k=0, k<r; k++)
         s = a[i][k] * b[k][j] + s;
      y[i][j] = s;
   }
```

the work is $O(nmr)$ for rectangular matrices, and $O(n^3)$ for the case of square $n \times n$ matrices. In the latter case, we have n^3 floatingpoint multiplications and $n^3 - n^2$ additions, $2n^2$ floatingpoint loads[2] and n^2 floatingpoint stores. The arithmetic intensity is $(2n^3 - n^2)/(3n^2) \approx 3n/2 = \Theta(n)$, i.e., grows linearly in n, and thus matrix-matrix multiplication is, if properly implemented (see later), compute-bound for reasonably large n.

[2]This holds for the case shown above where the elements of C are initialized to zero. We have otherwise $3n^2$ floatingpoint loads in the case where the result matrix C is not zero-initialized but its initial value must be loaded from memory, too.

```
For calculating 1 block $C'_{ij}$ of $C$,
  For $k' = 1, ..., q$:
    Load (access) 1 block $A'_{ik}$ of $A'$ (with $S \times S = (n/q)^2$) elements)
    Load (access) 1 block $B'_{kj}$ of $B'$ (of same size)
    Multiply ($S \times S$ _gemm) and accumulate in $C'_{ij}$
    Write $C'_{ij}$ to memory ($S \times S$ elements)
```

Figure 4.5: Tiled Matrix-Matrix multiplication

4.2 Optimizing for Memory Hierarchy

Let us reconsider the analysis of $n \times n$ matrix-matrix multiplication (GEMM). The arithmetic work is $\approx 2n^3$ floatingpoint operations. The overall operand data access volume to load from (mandatory misses) and store to main memory is $3n^2$ elements—if the cache capacity at all levels of the memory hierarchy is large enough to keep all elements throughout their lifetime. At any time of the execution of the ijk GEMM algorithm described above, the working set consists of: one cache line of C (holding C_{ij}), an entire row of A (holding A_{i*}) and all rows of B simultaneously in the cache; hence, the worst-case working set size is $n^2 + O(n)$. For small caches and sufficiently large n, we cannot hold that many elements in the cache simultaneously, and capacity misses will occur at almost every access to B, which will lead to a higher actual memory access work of $O(n^3)$ actual memory accesses, and thus sets GEMM back to a memory-bound operation.

We will now introduce an optimization technique that avoids this problem by reducing the computation's worst-case working set size of the algorithm.

4.2.1 Tiling a Matrix

Continuing using the $n \times n$ matrix-matrix multiplication $C = AB$ as a running example, we now decompose each matrix in $q \times q$ blocks (submatrices). For simplicity of presentation, we assume that q divides n so we skip handling some corner cases. Hence, the block (submatrix) size is $S \times S$, where $qS = n$. We leverage now a fundamental property of Linear algebra which states that the $n \times n$ matrix-matrix multiplication is equivalent to calculating a $q \times q$ matrix-matrix product of entire $S \times S$ submatrix "elements", using $S \times S$ matrix-matrix-multiply as "element" multiply and elementwise matrix addition as "elment" addition, see Figure 4.5.

We choose q large enough (respectively, S small enough) so that at least 3 such $S \times S$ blocks (or even better, $2 + q$ blocks so we can hold an entire block row of B) fit in the (data) cache, at least in the last-level cache.

Now it remains to adapt the loop nesting and indexing structure to work in terms of entire matrix tiles rather than individual elements, see Figure 4.6: We block each loop by block size S and then interchange loops so that the outer loops iterate over the blocks (the "tiles") and the inner loops iterate within a tile. This *tiling* transformation only changes the relative order of the computation's memory accesses, multiplications and additions but does not change the overall number of arithmetic floatingpoint operations, which remains $2n^3$.

The overall number of actual memory accesses now becomes:

$$q^2\,(2q+1)\left(\frac{n}{q}\right)^2 \;=\; (2q+1)n^2$$

which is much closer to the theoretical memory access volume of $3n^2$. Obviously, we try to keep q as small as possible (and S accordingly as large as possible) so we can just keep 3 blocks in the

```
for (i=0; i<n; i++)
  for (j=0;  j<n;  j++)
    for (k=0;  k<n;  k++)
      C[i][j] += A[i][k] * B[k][j];
```

```
for (ii=0; ii<n; ii+=S)
  for (jj=0; jj<n; jj+=S)
    for (kk=0; kk<n; kk+=S)
      for (i=ii; i<S; i++)
        for (j=jj; j<S; j++)
          for (k=kk; k<S; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 4.6: The tiling transformation, applied to the ijk matrix-matrix multiplication code on the left hand side, produces the loop nest on the right-hand side. For simplicity we assume here that S divides n.

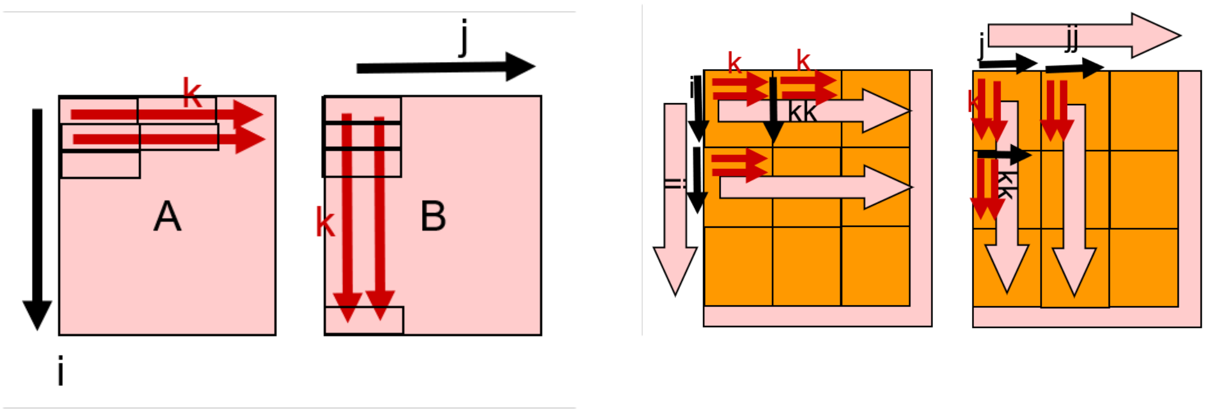


Figure 4.7: Data access locality in Matrix-Matrix Multiplication. — Left: Bad data access locality for large $n \times n$ matrix B. — Right: Tiling the loop nest and hence the operand matrices with a suitable tiling factor $S \ll n$ considerably improves data access locality by reducing the working set size.

cache.

Figure 4.7 shows graphically the different traversal orders for the A and B matrices before and after tiling.

4.2.2 Towards Performance-Portability

Different computers will have different cache sizes, hence a fixed S will not be appropriate for a BLAS-3 library where not only the code is portable but also the performance-affecting parameters (such as the tile size S) should automatically adapt to its target execution environment. This latter property is also known as *performance-portability*. Generating performance-portable code is, in general, a hard problem, but for specific, well-understood kernels such as the BLAS kernels it can be solved automatically to a great extent today.

As the different cache levels in the memory hierarchy differ in capacity, the tiling technique might actually be performed multiple times to the same code with different tile sizes, once per level of the memory hierarchy. Moreover, tiling is not the only program transformation that is highly target architecture specific. Other such transformations include, for example, loop unroll-and-jam, or the use of SIMD instructions, or the use of available hardware accelerators such as GPUs, TPUs or FPGAs.

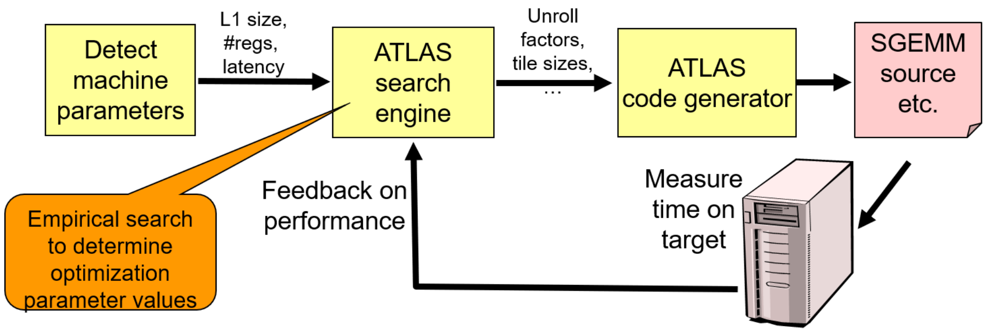


Figure 4.8: ATLAS

4.2.3 Auto-tuned BLAS

ATLAS (Automatically Tuned Linear Algebra Software) [20] is a BLAS library generator. It applies an automated iterative search process (design space exploration), see Figure 4.8, to determine the best combination of values for the main tuning parameters, such as the tile size S, loop unroll factors, and some other performance- but not correctness-relevant parameters of the implementation. For each considered combination, it instantiates a generic BLAS code template that includes parametric optimizations such as tiling to address L1 cache, loop unroll-and-jam, scalar replacement to improve register usage, and the usage of multiply-accumulate instructions and SIMD instructions where available. The generated BLAS source code for the current tuning parameter combination is then compiled and run on the target machine and the observed execution time is fed back to the search engine. This is repeated until no further improvement is possible, then the best parameter combination found is used to synthesize the final BLAS code to be used from now on on this machine. Whenever BLAS is to be installed on a new target machine, ATLAS is run once to generate a custom version of BLAS for it.

In this way, ATLAS provides automatic performance tuning (”autotuning”) for any given sequential processor architecture. The ATLAS-generated code outperformed vendor-specific BLAS implementations because the automated search procedure can encounter solutions that a human software engineer might not even think about.

4.3 Recursive Algorithms for Matrix-Matrix Multiplication

4.3.1 Simple Recursive Matrix-Matrix Multiplication

The simple recursive matrix-matrix multiplication algorithm follows the Divide-and-Conquer paradigm. For $n > 1$ we begin by decomposing the $n \times n$ matrices C, A, and B each in four quarter $(n/2 \times n/2)$ submatrices. As with tiling, we leverage the fundamental property of linear algebra stating that we can compute C using four dot products in terms of the corresponding operations on entire quarter matrices, hence, using eight $(n/2 \times n/2)$ matrix-matrix multiplications and four $(n/2 \times n/2)$ elementwise matrix additions:

$$C^{u,v} = A^{u,0} \times B^{0,v} + A^{u,1} \times B^{1,v}, \quad u = 0, 1, \quad v = 0, 1 \tag{4.1}$$

This enables the recursive application of the algorithm to these eight quarter matrix multiplications. The base case, on 1×1 matrices, is ordinary scalar floatingpoint multiplication.

The work performed for the recursive case ($n > 1$) consists mainly of 8 multiplications of $n/2 \times n/2$-submatrices plus 4 additions of $n/2 \times n/2$-submatrices

$$T_{RecMM}(n) = 8T_{RecMM}(n/2) + \Theta(n^2)$$

The base case complexity is

$$T_{RecMM}(1) = O(1)$$

Solving this recurrence equation system, we obtain

$$T_{RecMM}(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$$

i.e., an $\Theta(n^3)$ work algorithm again, thus the same asymptotic work complexity as the standard loop-based method for GEMM.

Good for data locality, effect similar as tiling

4.3.2 Strassen's Recursive Matrix-Matrix Multiplication Algorithm

Starting from the above recursive matrix-matrix multiplication method, Strassen [58] found out that, by spending some extra matrix additions and subtractions, the quarter matrices of C can be computed from the quarter matrices of A and B by only *seven* $(n/2 \times n/2)$ matrix multiplications and a few more matrix additions and subtractions as follows:

$$\begin{array}{rcl} Q^0 & = & (A^{0,0} + A^{1,1}) \times (B^{0,0} + B^{1,1}) \\ Q^1 & = & (A^{1,0} + A^{1,1}) \times B^{0,0} \\ Q^2 & = & A^{0,0} \times (B^{0,1} - B^{1,1}) \\ Q^3 & = & A^{1,1} \times (B^{1,0} - B^{0,0}) \\ Q^4 & = & (A^{0,0} + A^{0,1}) \times B^{1,1} \\ Q^5 & = & (A^{1,0} - A^{0,0}) \times (B^{0,0} + B^{0,1}) \\ Q^6 & = & (A^{0,1} - A^{1,1}) \times (B^{1,0} + B^{1,1}) \end{array} \tag{4.2}$$

Then, C can be assembled from Q^0,...,Q^6 as follows:

$$\begin{array}{rcl} C^{0,0} & = & Q^0 + Q^3 - Q^4 + Q^6 \\ C^{0,1} & = & Q^2 + Q^4 \\ C^{1,0} & = & Q^1 + Q^3 \\ C^{1,1} & = & Q^0 + Q^2 - Q^1 + Q^5 \end{array} \tag{4.3}$$

If applied recursively, this method results in a work complexity defined by the recurrence equation

$$T_{StrassenMM}(n) = 7 \cdot T_{StrassenMM}(n/2) + O(n^2) = O(n^{\log_2 7]})$$

where the first term accounts for the 7 quarter matrix multiplications performed in the Equations (4.3), and the quadratic term for all matrix additions and subtractions encountered in Equations (4.2) and (4.3).

The base case could be set for $n = 1$ and handled by an ordinary scalar multiplication:

$$T_{StrassenMM}(1) = O(1)$$

Solving this recurrence equation system, we obtain

$$T_{StrassenMM}(1) = O(n^{\log_2 7}),$$

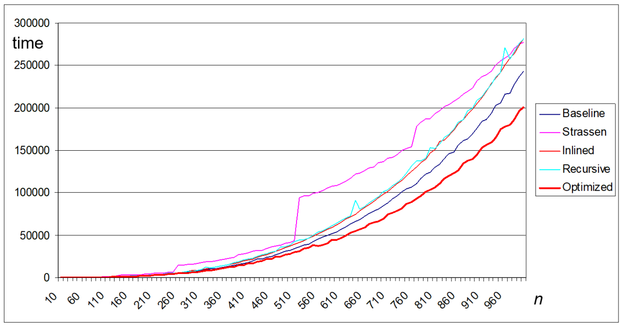


Figure 4.9: Execution times of four matrix-matrix multiplication algorithms for different problem sizes. Deep implementation selection performs better than any fixed algorithm.

Note that $\log_2 7$ is about 2.81. Even more advanced recursive methods split the matrix into more than four submatrices and apply much more complicated sets of recursive computation formulas, which lead to even better asymptotic complexities. The best currently known asymptotic bound is $O(n^{2.376})$ [16].

Of course, this is a rather theoretical result, albeit an important one because it showed that Matrix-Matrix multiplication (which occurs as a subroutine in many other algorithms, such as all-pairs shortest paths) can be computed in less than cubic time. The constant factor hidden in the $O()$ notation is much larger than for the standard $O(n^3)$ method, such that this algorithm will pay off only for very large problem sizes n.

On the other hand, recursive parallel algorithms for matrix operations have now gained importance as they tend to exploit the memory hierarchy of cache-based parallel architectures better than do their iterative counterparts.

4.3.3 Tuning the Recursive Matrix-Matrix Multiplication Algorithms

By now, we assumed for the recursive Matrix-Matrix multiplication algorithms that the base case is reached only for $n_0 \times n_0$ matrices of size $n_0 = 1$. However, we may decide at any recursive call to switch to any other Matrix-Matrix multiplication algorithm already for $n > n_0$. This can be advantageous for utilizing algorithms such as the ijk standard loop-based method that have low overheads for smaller problem sizes and can be tiled to better exploit small caches, while other algorithms (e.g. Strassen's algorithm) have better asymptotic complexity for very large problem sizes. Choosing the problem sizes to switch between algorithms, and in particular, choosing the best n_0 to stop the recursion and the best algorithm to use then for solving the larger base cases, is however very much dependent on the characteristics of the target machine, such as its cache sizes, memory and CPU speed etc., but also on the optimizations performed by the used compiler. Hardcoding a specific n_0 or algorithmic choice in the algorithm or program code is not advisable, because it will, in general, lead to suboptimal performance when deploying and running the code on a different computer system. Hence, this is another case for auto-tuning, where the decision is delegated to a system software that learns from training executions on the target system the best value for n_0 etc.

Figure 4.9 [39] shows the performance behavior of four different sequential implementations of Matrix-Matrix multiplication, including simple recursive and Strassen recursive matrix-matrix multiplication, the ijk loop based standard implementation, and a refactored implementation calling matrix-vector multiplication and dot product as subroutines. The best transtion points have been precalculated off-line for the used target system and compiler by a dynamic-programming based

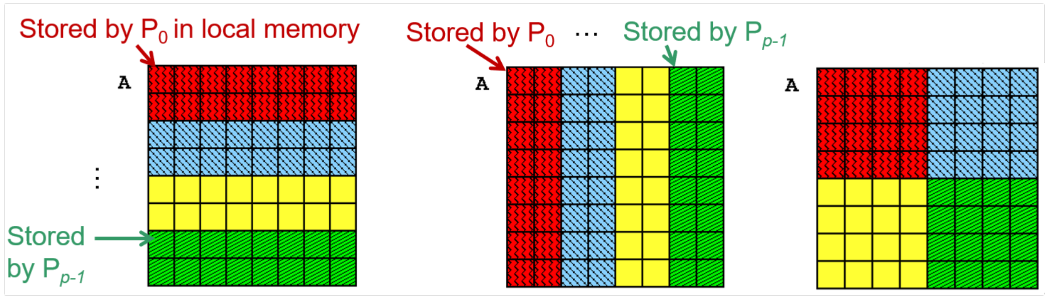

Stored by P_0 in local memory
A
Stored by P_{p-1}
Stored by P_0 … Stored by P_{p-1}
A
A

Figure 4.10: Matrix Partitioning and Distribution Schemes

optimization. It is interesting to observe that the composed hybrid algorithm beats *any* fixed implementation selection; this is due to considering algorithmic choice at *every* call in the recursive algorithms' implementations (*deep selection*).

The generalized problem, autotuned algorithm selection at each call depending on the call context, is actively researched especially since the transition to multicore architectures, because it also involves the decision between sequential, parallel or accelerator computation (where applicable). Frameworks for algorithmic choice have been proposed in high-level parallel programming environments as well as in task-based runtime systems especially for heterogeneous parallel computer systems.

4.4 Basic Linear Algebra Computations on Distributed-Memory

We will now focus on computing BLAS functionality on distributed-memory (message passing) systems, in particular matrix-vector multiplication and matrix-matrix multiplication as case studies.

This will usually require that (most) of the matrices and vectors that we operate on are partitioned and distributed across the different nodes. For these, each node will thus usually only allocate memory space for one partition (plus for remote elements communicated), not for the entire data structure (which might even be too large to be reasonably stored in one place).

4.4.1 Distributed Vectors and Matrices

Common partitioning/distribution schemes for a matrix in matrix-vector multiply include:

- Row-wise (row-block wise) partitioning

  Process i stores rows $i(n/p), ..., (i+1)(n/p) - 1$ in its local memory, see Figure 4.10 (left).

- Column-wise (column-block wise) partitioning

  Process i stores columns $i(m/p), ..., (i+1)(m/p) - 1$ in its local memory, see Figure 4.10 (middle).

- Block-block wise partitioning using partitioning and distribution along both dimensions, see Figure 4.10 (right).

Further partitioning schemes for vectors and matrices exist.

Many data-parallel computations on matrices, including all BLAS operations, perform the same amount of work per array element. Hence, load distribution among p processes is perfect if the operand data is partitioned and distributed evenly. In the case of matrices, partitioning is usually

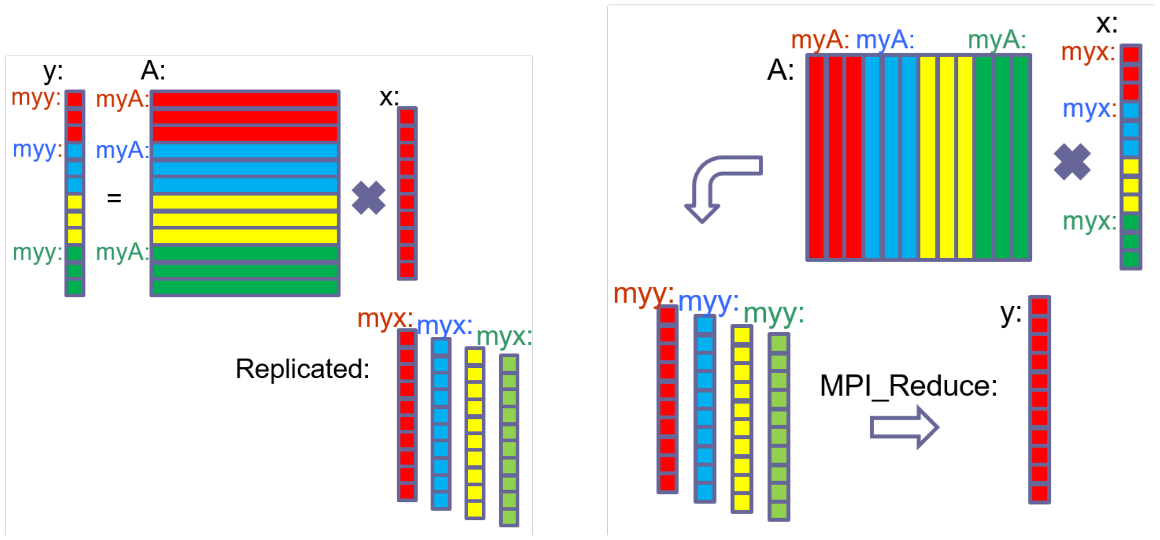


Figure 4.11: Distributed Matrix-Vector Multiplication. Left: ij-variant. Right: ji-variant.

done in terms of entire rows or columns in order to keep the computation of local and global indexes simple.

Partitioning a $n \times n$ matrix as equally as possible among p processes is straightforward if p divides n. If this is not the case, we need to take care of some corner cases. Two different methods are most common:

- We choose as partition sizes $\lceil n/p \rceil$ rows resp. columns for the first $n\%p$ processes and $\lfloor n/p \rfloor$ rows resp. columns for the last $p - n\%p$ processes.
- $\lceil n/p \rceil$ rows resp. columns for all processes i with $(i+1)\lceil n/p \rceil \leq n$;
  $n - j(n/p)$ rows resp. columns for the process j with $j\lceil n/p \rceil < n$ and $(j+1)\lceil n/p \rceil > n$; and
  no rows resp. columns for processes k with $k(n/p) > n$.

Both alternatives lead to optimal load balancing in terms of entire rows resp. columns. The second alternative might, in extreme cases, lead to unused processes, which can be less desirable because it might require extra adaptations in the program control to properly take care of these.

4.4.2 Message-passing Matrix-Vector Product

4.4.2.1 ij-Variant

The pseudocode of the distributed ij variant can be found in Figure 4.12 (left).
Analysis: The local computation time on each node is $T_{comp} = O((n/p)m)$. The communication involves three collective communication operations and their times accumulate:

$$T_{comm}(n,m,p) \;=\; T_{scatter}((n/p)m, p) + T_{broadcast}(m,p) + T_{gather}(n/p, p)$$

4.4.2.2 ji-Variant

The pseudocode of the distributed ji variant can be found in Figure 4.12 (right).
Analysis:

```
// Distributed ij-variant

// Assume matrix A is row-wise partitioned
// (otherwise, scatter it first from P0),
// my local partition is called myA

// Vector x is replicated
// (if not, broadcast it from P0),
// my local copy is called myx

// Calculate my local part of y:
   for (i=0; i<mynrows; i++)
      myy[i] = sdot( myA[i], myx, ... );

// Now gather the partial results:
   MPI_Gather( y, ..., myy, mynrows,
               ..., 0, ...);
// or MPI_Allgather / multi-broadcast myy
// if every process needs the complete y later
```

```
// Distributed ji-variant

// Assume matrix A is column-wise partitioned
// (otherwise, scatter it from P0)
// my local partition is called myA

// Vector x is block-distributed
// (if not, scatter it from P0)
// my local partition is called myx

// calculate my local contribution to y:
   myy = sgemv( myA, myx, ... );

// Now accumulate the partial products:
   MPI_Reduce( y, myy, myncols,
               PartialRowtype, MPI_Sum, ...);

// use instead MPI_Allreduce if every
//  process needs the complete y later
```

Figure 4.12: Distributed Matrix-Vector Multiplication. Left: ij-variant. Right: ji-variant.

Table 4.4: Collective communication times depending on network topology

Collective operation on p processors	Bus	Ring	d-dim. Mesh	Hypercube
(Single-)Broadcast(1 element)	$O(1)$	$O(p)$	$O(p^{1/d})$	$O(\log p)$
Scatter/Gather(1 elem.)	$O(p)$	$O(p)$	$O(p)$	$O(p/\log p)$
Reduce(1 element)	$O(p)$	$O(p)$	$O(p^{1/d})$	$O(\log p)$
Total exchange	$O(p^2)$	$O(p^2)$	$O(p(d+1)/d)$	$O(p)$

Again the local computation on each node takes time $O((n/p)m)$. The communication time for the ji-variant is

$$T_{scatter}(n(m/p),p) + T_{scatter}(m/p,p) + T_{reduce}(n,m/p,p)$$

Note that the communication times for the collective communication operations, $T_{scatter}$, T_{gather}, $T_{broadcast}$, T_{reduce}, ..., depend on the underlying network topology. For examples see Table 4.4.

4.5 Message-Passing Matrix-Matrix-Multiply

Several message-passing algorithms exist for matrix-matrix multiplication. We begin with the SUMMA algorithm (Scalable Universal Matrix Multiplication Algorithm) and consider systolic (pipelined) matrix-matrix multiplication algorithms later.

4.5.1 SUMMA Algorithm

The SUMMA algorithm [63] works well in practice on many systems; it is used in distributed-memory parallel BLAS implementations and in ScaLAPACK.

SUMMA works conceptually on a quadratic 2D mesh network of nodes (referred to as ”processors” for the rest of this chapter) and thus assumes $p = r^2$ for some integer r; see Figure 4.13

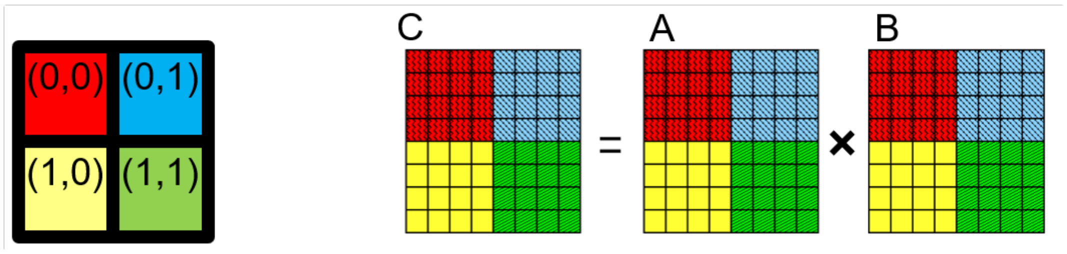


Figure 4.13: Processor mesh (left, here 2×2) and matrix distributions for the SUMMA algorithm.

```
algorithm SUMMA ( distributed matrices $C$, $A$, $B$ )
{
  The processes are organized in $r \times r$ 2D grid;
  my coordinate is $(i, j)$
  $C_{ij} = 0$
  For $l = 0, ..., n - 1$
    If I own block $A_{il}$ then broadcast it within my process row $i$
    If I own block $B_{lj}$ then broadcast it within my process column $j$
    $C_{ij} = C_{ij} + A_{il} \times B_{lj}$
}
```

Figure 4.14: Pseudocode of the SUMMA algorithm. Here, + denotes (sequential) block matrix addition and $\times$ denotes (sequential) block matrix-matrix multiply (GEMM).

(left).

Accordingly, we partition the three operand matrices A, B, C equally in both dimensions, i.e., a block-wise partitioning into $p = r \cdot r$ blocks each of size $k \times k$, where $k \cdot r = n$ (let us assume for simplicity of presentation that we are multiplying square $n \times n$ matrices and that k divides n). We then distribute the blocks, i.e., map the blocks to processes such that block (i, j) is owned by process (i, j).

Following a principle called the *owner computes* rule, we let process (i, j) calculate partition $C_{i,j}$. For that, it will need all blocks of A in block row i and all blocks of B in block column j (see Figure 4.15 for illustration). The pseudocode of the SUMMA algorithm is shown in Figure 4.14.

The `broadcast` communication along rows of A and columns of B is illustrated in Figure 4.15.

The amount of floatingpoint operations performed is $2k^3 + k^2 + O(1) \in O(k^3)$, thus $O(n^3/p)$ per process. For analyzing the communication time, we see that each process performs 2 `broadcasts` (including $2r$ receives) of k^2 elements each over r processes. As an example (see Table 4.4), we assume a hypercube network, where

$$T_{broadcast}(M, r) = O(bM + g) \log_2 r$$

using the Delay model with per-process bandwidth b and message latency g as the cost model. If not further bandwidth-limited, all p processes can run their sequence of $2r$ receives of $M = k^2$ elements in parallel. Hence, the overall communication time becomes

$$T_{comm} = 2r(bk^2 + g) \log_2 r = 2n(bk + g/k) \log_2 r = n \log_2 p(bn/\sqrt{p} + g\sqrt{p}/n)$$

T_{comm} scales quite well for large p, as p only occurs sublinearly in the term for communication

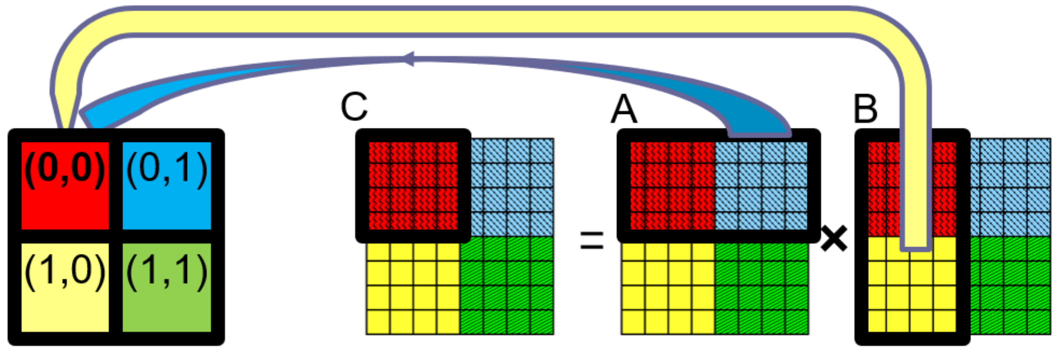


Figure 4.15: Communication in the SUMMA algorithm: Broadcasting partitions of A along the rows and partitions of B along the columns of the processor mesh.

time.

The relative speedup is thereby:

$$\begin{aligned} S(p,n) &= T_{GEMM}(n)/T_{SUMMA}(p,n) \\ &= 2n^3/(2n^3/p + n\log_2 p(bn/\sqrt{p} + g\sqrt{p}/n)) \\ &= n^3/(n^3/p + 0.5n\log_2 p(bn/\sqrt{p} + g\sqrt{p}/n)) \end{aligned}$$

and the relative efficiency is:

$$\begin{aligned} E(p,n) &= S(p,n)/p \\ &= 1/(1 + \log_2 p(b\sqrt{p}/n + gp\sqrt{p}/n^3)) \end{aligned}$$

Considering this complicated term, we can mostly ignore the $\log p$ factor in the scalability analysis because it grows very slowly in comparison to $\sqrt{p}$ or n. Hence, SUMMA scales well even for large machine sizes p as long as $\sqrt{p} \in O(n)$ (strong scaling) and even further if we can co-scale the problem size n accordingly (weak scaling). However, the communication will become inefficient as p approaches n^2 as there will be very many small messages.

Another potential issue is that SUMMA needs efficient global communication (namely, the `broadcast` operations), which might not be well supported e.g. on mesh-based architectures where nearest-neighbor communication is more efficient. We will now consider two message-passing matrix-matrix multiplication algorithms that are suitable for 2D mesh-based networks as they do not involve broadcasting. The key idea is to organize the algorithm as a 2D pipeline and move data only among nearest neighbor nodes in the mesh in each time step.

4.6 Systolic Algorithms for Matrix-Matrix Multiplication

Systolic algorithms are pipelined distributed algorithms defined for processor mesh networks where data is flowing step by step through the processor network, very much like blood is pulsed through the blood vessels in the body. Systolic algorithms thus need no global collective communication such as broadcast or reduction, and instead use nearest-neighbor point-to-point communication.

Systolic algorithms working on a 2D mesh network are thus particularly suitable for an implementation in hardware ("systolic processor array"). This principle is used today e.g. in the Google *Tensor Processing Unit*, which contains a hardware accelerator for matrix-matrix multiplication that consists of 256×256 8-bit MAC (multiply-accumulate) units.

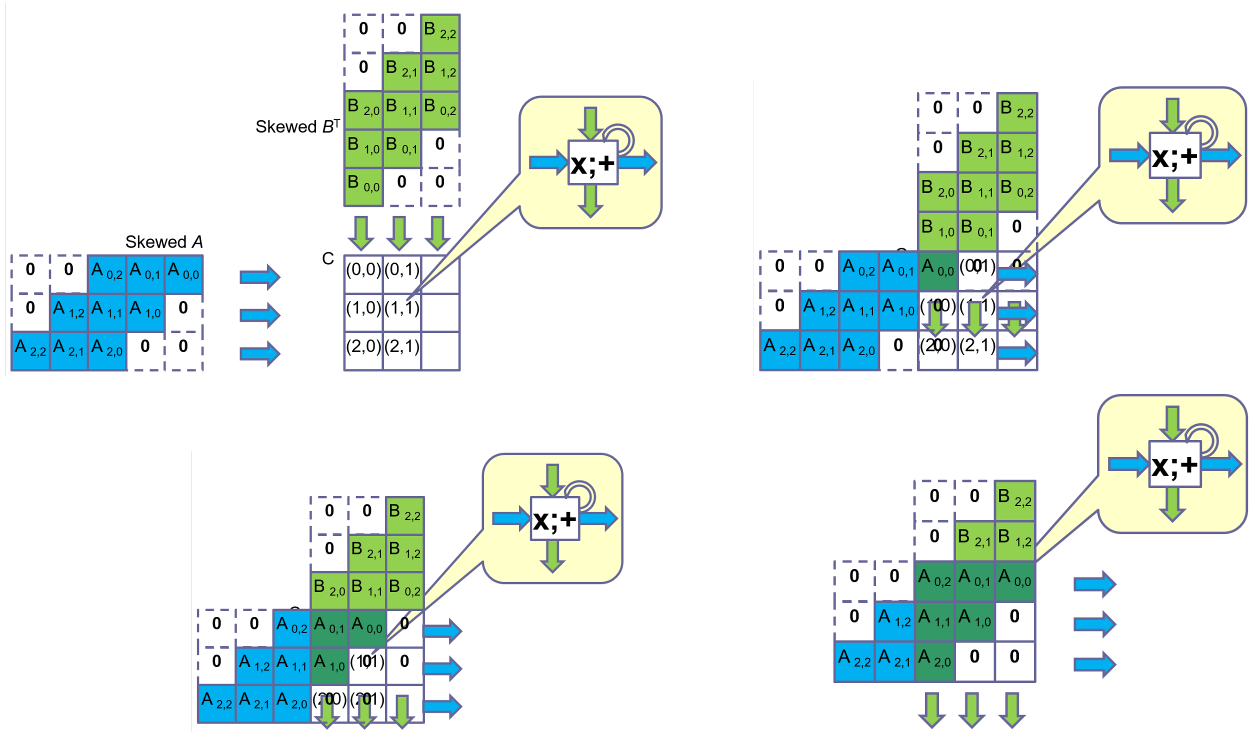


Figure 4.16: Illustration of the Kung-Leiserson algorithm for matrix-matrix multiplication on a distributed memory system organized as a 3×3 processor mesh. — Left upper: Start configuration after skewing the operand matrices. Right upper: The first computation step after shifting the operands. Left lower: The second computation step. Right lower: Third computation step.

We will now consider two systolic algorithms for message-passing Matrix-Matrix-multiply, the Kung-Leiserson systolic (pipelined) algorithm [43] (Section 4.6.1) and Cannon's Algorithm [13] (Section 4.6.2).

In both cases we assume a square 2D processor mesh resp. torus network with $p = r \times r$ nodes (processors).

4.6.1 Systolic Matrix-Matrix Multiplication by Kung/Leiserson

The Kung-Leiserson algorithm works on $r \times r$ skewed $k \times k$ blocks of the operand matrices. It is organized as a 2D pipeline. see Figure 4.16.

Initially, operand matrix A is skewed along the block rows dimension such that it forms a $r \times (2r-1)$ matrix A' with $A_{i,j} = A'_{i,i+j}$, where the unused blocks in A' are padded with zeroed blocks, as shown in Figure 4.16 (top left). Likewise, B is skewed and zero-padded along the block columns dimension into a $(2r-1) \times r$ matrix, with $B_{i,j} = B'_{i+j,j}$.

Now the algorithm is properly set up for the pipelined computation. In each step, one block in each block row of A' is moved by one position to the right in the processor mesh, and one block in each block column of B' is moved by one position downwards in the processor mesh. The first three steps of the pipeline are shown in Figure 4.16 top right, bottom left and bottom right, respectively. In each step, each process (i, j) multiplies the received blocks of A' and B' (using GEMM), accumulates the product block element-wise with its locally stored block of C_{ij}, and forwards the blocks of A' and B' to its neighbors $(i, j+1)$ to the right and $(i+1, j)$ below, respectively. The pseudocode can be found in Figure 4.17.

```
algorithm SystolicMatMul ( Matrices $A$, $B$ )
{
  Skew and pad matrices $A$ and $B$ as described in the text.
  Process $(i, j)$ calculates $C_{ij}$ in $2r - 1$ rounds:
  In each round:
    Receive an A-block $A$ from left neighbor process $(i, j - 1)$
    Receive a B-block $B$ from upper neighbor process $(i - 1, j)$
    $C_{ij} = C_{ij} + A'B'$
    // i.e., sequential GEMM$(C', A', B'^T)$ followed by elementwise accumulation of $C'$ in $C_{ij}$)
    Forward block $A'$ to right neighbor process $(i, j + 1)$
    Forward block $B'$ to lower neighbor process $(i + 1, j)$
  After $2r - 1$ rounds, $C_{ij}$ contains $\sum_l A_{il} B_{lj}$
}
```

Figure 4.17: The systolic matrix-matrix multiply algorithm by Kung/Leiserson [43]

Due to the skewing of A and B, the blocks A_{il} and B_{lj} will meet at process (i, j) in time step $i + j + l$ of the pipeline execution (if counting steps from 0), and their product is added to the block value accumulated in process (i, j). Hence, after $2r - 1$ steps, process (i, j) has calculated $C_{ij} = \sum_l A_{il} B_{lj}$.

The analysis of time and speedup is left as an exercise.

We remark that the algorithm only uses local (nearest-neighbor) communication, and that it could use up to n^2 processors. A disadvantage is the need to skew and pad the operand matrices A and B, and the resulting higher number of time steps of $2r - 1$. We will now see that we can modify the algorithm in order to perform the actual computation pipeline in r steps only.

4.6.2 Cannon's Algorithm

Cannon's Algorithm [13] is a systolic algorithm similar to Kung/Leiserson. Instead of a $r \times r$ processor 2D mesh, it assumes a $r \times r$ processor 2D torus network.

In the setup phase, we skew the given operand matrices A and B by shifting *cyclically* along rows/columns into the skewed start configuration (Figure 4.18). Hence, padding with zero blocks as in the Kung/Leiserson algorithm is not required.

The pipelined computation proceeds similar to the Kung/Leiserson algorithm: after local multiplication, process (i, j) forwards its current block of A along the rows dimension to destination process $(i + 1) \mod r, j)$ and its current block of B along the columns dimension to destination process $(i, (j + 1) \mod r)$. Hence, again, only local, nearest-neighbor point-to-point communication is used. Overall, only r pipeline steps with local computation are needed. The analysis of time and speedup is left as an exercise.

As an example, we consider a 3×3 processor torus network, hence Cannon's algorithm performs $r = 3$ pipelined steps.

Step 0: We begin by preparing the start configuration by horizontal cyclic shifting of A, where we shift each partition A_{il} along the i-th processor row by i processors to the left (modulo r), and by vertical cyclic shifting of B, where we similarly shift partition B_{lj} along the j-th processor column by j processors upwards modulo r, see Figure 4.18.

Also, we initialize the blocks of the C matrix locally on each process:

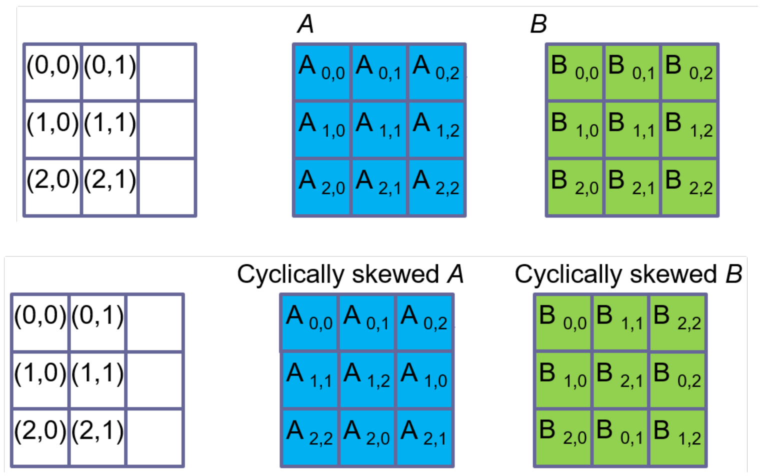


Figure 4.18: Top: Initial state of operand matrices. — Bottom: The start configuration for Cannon's Algorithm after skewing the operand matrices.

$C_{00} = 0$
$C_{01} = 0$ etc.
Step 1: Each processor (i, j) multiplies (using GEMM) and accumulates (using elementwise addition) the local partitions of A and B: $C_{ij} += A' \times B'$. Hence,
$C_{00} = A_{00}B_{00}$
$C_{01} = A_{01}B_{11}$
etc.
It then sends its partition $A' = A_{il}$ along the i-th row to its left neighbor modulo r, and its partition $B' = B_{lj}$ along the j-th column to its upper neighbor modulo r.
Step 2: performs the same; we obtain:
$C_{00} = A_{00}B_{00} + A_{01}B_{10}$
$C_{01} = A_{01}B_{11} + A_{02}B_{21}$
etc.
Step 3: performs the same, we obtain:
$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$
$C_{01} = A_{01}B_{11} + A_{02}B_{21} + A_{00}B_{01}$
etc., i.e., we are done, and the operand matrices A, B are back to their distribution in the start configuration (cyclically skewed as before Step 1).

4.7 Chapter Summary

This chapter studied efficient sequential and parallel implementations of basic linear algebra computations such as the BLAS operations, with special emphasis on matrix-vector and matrix-matrix multiplication.

From studying optimized implementations of BLAS and parallelizing BLAS operations, we can extract some rules of thumb in scientific computing:

- It is easy but pointless to parallelize inefficient algorithms.

  Good scalability (relative speedup) does not automatically imply good performance. If starting from an inefficient, unoptimized sequential implementation (for example, the ji-variant of matrix-vector multiplication in C), the overheads of parallelization (such as thread creation or synchronization) will be a smaller percentage of the overall work, and the bloated grain size will lead to good relative speedups. But absolute speedup relative to an optimized sequential implementation will be poor, and we waste expensive parallel computing resources. Instead, parallelization should start from efficient sequential algorithms and implementations where available.

- Keep the number of floatingpoint operations low— both asymptotically and in per-element work.

  For example, optimized GEMV and GEMM implementations have special code variants for the special cases $\alpha = 1$ or $\beta = 0$ to avoid useless multiplications and/or additions.

- Keep the number of memory accesses low.

  E.g., using a scalar accumulator variable in ij-GEMV enables the compiler to use a register.

- For each (off-chip) memory reference, perform as many floatingpoint operations as possible.

  For example, this can be achieved, where applicable, by improving temporal locality by reducing the working set size, such as in matrix tiling (see Section 4.2).

- Exploit spatial locality.

  This can be achieved by careful programming to have most arrays accessed with stride 1 in innermost loops where applicable, as in the ij variant of matrix-vector multiplication (in C).

- Avoid conditional branches (other than loop control) in inner loops.

  For example, the check for important special cases, such as GEMM with $a = 1$, should be placed at top level, with a separate code branch dedicated this case, instead of repeatedly checking the value of a in each inner iteration.

In particular, we considered tiling as an optimization for improved data access locality, and techniques for automated performance tuning by predicting the best tiling factor and the best switching points for recursive algorithms.

We considered several parallel algorithms for matrix-vector multiply and for matrix-matrix multiply for distributed-memory computation. For the latter, we presented and analyzed the SUMMA algorithm, which is suitable in systems with efficient broadcast, and two systolic algorithms which better fit systems with a 2D mesh resp. 2D torus network.

4.8 Bibliographical Notes *

TBD

4.9 Exercises

1. The ATLAS autotuning approach can, in principle, be applied to any BLAS operation. However, it is most effective for BLAS-3 operations such as matrix-matrix multiplication. Why?

2. Elaborate on the *shared-memory* parallelizations of the ij and ji variants of matrix-vector multiplication, for example using OpenMP. Apart from the memory hierarchy issues already discussed, which of the two variants is likely to lead to more efficient parallel code (less parallel overhead)? Justify your answer.

3. Generalize the SUMMA pseudocode of Section 4.5.1 for computing on HPC clusters consisting of shared-memory nodes.

4. If you need to compute a matrix-matrix multiplication for large $n \times n$ matrices as fast as you possibly can, given a shared-memory system (e.g., a PRAM) with as many processors as you want, would you consider a systolic algorithm? Justify your answer.

5. Analyze the parallel time and speedup of the Kung-Leiserson algorithm and of Cannon's algorithm.

6. Following the approach by Kung/Leiserson, develop a systolic algorithm for matrix-vector multiplication. (Hint: How many dimensions should the processor mesh have?)

7. To be continued ...

Appendix A

Notation for the Growth of Functions: O, Θ, Ω, o, ω

A.1 How Functions Grow

The following table illustrates the growth rates of some common functions:

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$
...	...	...	...	...	...

Assuming a pretty fast (sequential) computer doing on average one time step (operation) per nanosecond, an execution time function of 2^n would, for a modest problem size of $n = 64$, mean an execution time of $1.84 \cdot 10^{19}$ns $= 2.14 \cdot 10^5$ days $= 585$ years.

A.2 Dominance Relation

Consider two growing functions f, g from natural numbers to positive real numbers, as shown in Figure A.1.

f dominates g iff $f(n)/g(n)$ increases without bounds for $n \to \infty$,
that is, for a given constant factor $c > 0$,
there is some threshold value $n_0 \in \mathbb{N}$
such that $f(n) > c \cdot g(n)$ for all $n > n_0$.
Example: $f(n) = n^2$ dominates $g(n) = 7n$.

A.3 Big-Oh Notation

Big-Oh notation is a shorthand notation for the asymptotic growth of functions. It allows to more easily compare the growth rates of increasing functions. When discussing algorithms, it allows to estimate the relative efficiency of different algorithms by reference to simple functions. It abstracts from constant factors and lower-order (dominated) terms and thereby defines classes (i.e., sets) of functions with similar growth behavior.

Let f, g denote growing functions from the natural numbers to positive real numbers. The function sets $O(g)$, $\Omega(g)$ and $\Theta(g)$ are defined as follows:

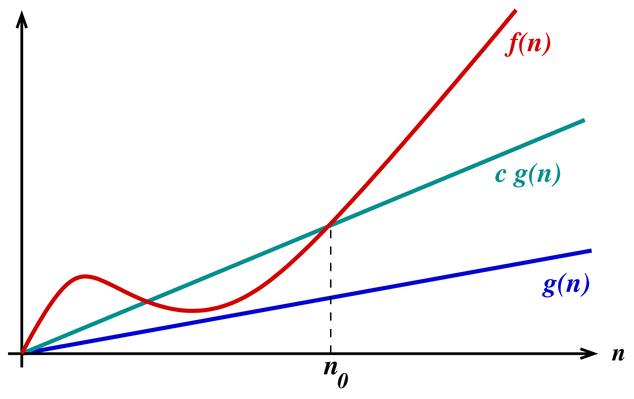


Figure A.1: Dominance relation

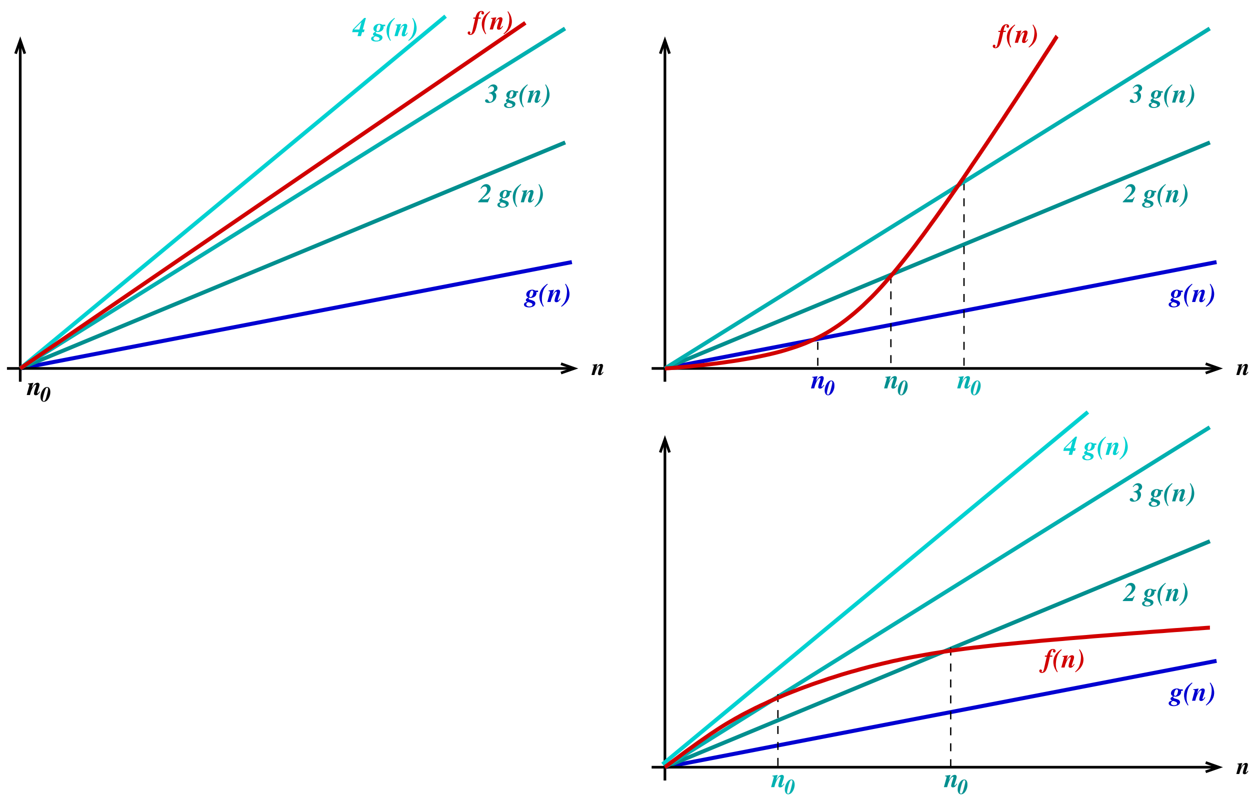


Figure A.2: Illustrations of the definitions of $O()$, Ω and Θ.

- f **is (in)** $O(g)$ iff there exist $c > 0$, $n_0 \geq 0$ such that $f(n) \leq c\ g(n)$ for all $n > n_0$.

  Intuition: Apart from constant factors, f grows at most as quickly as g. See also Figure A.2 (lower right).

- f **is (in)** $\Omega(g)$ iff there exist $c > 0$, $n_0 \geq 0$ such that $f(n) \geq c\ g(n)$ for all $n > n_0$

  Intuition: Apart from constant factors, f grows at least as quickly as g. See also Figure A.2 (upper right).

  $\Omega()$ is the converse of O, i.e. f **is in** $\Omega(g)$ iff g **is in** $O(f)$

- f **is (in)** $\Theta(g)$ iff $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

  Intuition: Apart from constant factors, f grows exactly as quickly as g. See also Figure A.2 (upper left).

A.3.1 Examples

The sequential linear search algorithm (see Figure A.4) has time complexity

$$t_{LinearSearch}(n) = n * (t_1 + t_2 + t_3) + t_4 = k_1 \cdot n + k_2$$

hence,

$t_{LinearSearch}(n) \in O(n)$ (Why?)
$t_{LinearSearch}(n) \in \Omega(n)$ (Why?)
$t_{LinearSearch}(n) \notin O(\log\ n)$ (Why?)
$t_{LinearSearch}(n) \in \Theta(n)$ (Why?)

The time complexity of binary search is $t_{BinarySearch}(n) = c_1 \cdot (\lfloor \log_2(n) \rfloor) + c_2$
hence:

$t_{BinarySearch}(n) \in O(\log\ n)$
$t_{BinarySearch}(n) \in O(n)$ (Why?)

A.3.2 Exclusive Growth Classes

$O(f)$, $\Omega(f)$ and $\Theta(f)$ are inclusive growth classes in that they include f itself. It is of course possible to define sets of functions with growth rates that are strictly stronger or strictly weaker than f:

- f **is (in) o**(g), i.e., f *is dominated by* g, iff for **any** $c > 0$ there is an $n_0 > 0$ such that $g(n) > cf(n)$ for all $n > n_0$

  Intuition: g grows *more quickly* than f.

  Hence, if $f \in o(g)$ then $f \in O(g)$, but not vice versa.

  As an example, $n \in o(n^2)$.

- f is (in) $\omega(g)$, i.e., g *is dominated by* f, iff for **any** $c > 0$ there is an $n_0 > 0$ such that $g(n) < cf(n)$ for all $n > n_0$

  Intuition: g grows *more quickly* than f.

  Hence, if $f \in \omega(g)$ then $f \in \Omega(g)$, but not vice versa.

A.3.3 Comparing Growth Rates of Some Simple Functions

Some simple facts:

- For the same base, the growth rate of power functions is determined by the exponents:

  $n^\alpha \in O(n^\beta)$ iff $\alpha \leq \beta$ $(\alpha, \beta > 0)$ $[n^\alpha \in o(n^\beta)$ iff $\alpha < \beta]$

- Power functions grow more quickly than logarithms:

  $\log_b n \in o(n^\alpha)$ for any $b, \alpha > 0$

- Exponential functions grow more quickly than power functions (and, thus, more quickly than all polynomial functions):

  $n^\alpha \in o(c^n)$ for any $\alpha > 0$, $c > 1$

- The growth rate of logarithms of various bases is equal:

  $\log_a n \in \Theta(\log_b n)$ for any a and b

  For this reason, we write shortly $\Theta(\log n)$, omitting the basis.

- $c^n \in O(d^n)$ iff $c \leq d$, $[c^n \in o(d^n)$ iff $c < d]$

- Any constant function $f(n) = c$ is in $O(1)$.

A.3.4 Properties of Big-Oh Notation

The transitivity of $O()$ etc. follows from the transitivity of set inclusion:

If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.

The growth rate depends only on fastest growing components:

If $f \in O(g)$ then also $f + g \in O(g)$.

$O()$ is closed under multiplication:

If $f \in O(f')$ and $g \in O(g')$ then $f \cdot g \in O(f' \cdot g')$

$O()$ is closed under (positive) linear transformations:

If there are d, $n_0 > 0$ such that $f(n) \geq d$ for all $n \geq n_0$,
then $k \cdot f(n) + c \in O(f)$ for all constants k, c.

A.3.5 Comparing Functions

In order to check whether $f \in O(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$, we analyze

$$l = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- $f \in O(g)$ iff $l < \infty$
- $f \in \Omega(g)$ iff $l > 0$
- $f \in \Theta(g)$ iff $0 < l < \infty$

```
**int** FACTORIAL ( **int** $n$)
{
  **if** $n = 0$
    **then return** 1
  **else**
    **return** $n \cdot$ FACTORIAL($n - 1$)
}
```

Figure A.3: The recursive FACTORIAL algorithm.

A.4 Analysis of Recursive Programs

Consider the program in Figure A.3 for calculating the factorial function:

For analyzing the execution time $T(n)$ of FACTORIAL with argument n, we can define the following cost model parameters

- time for comparison: t_c
- time for multiplication: t_m
- time for function call and return: t_f

which we can consider all being bounded by constants, i.e., are in $O(1)$.

We obtain the following recurrence equation system defining T:

$$\begin{aligned} T(0) &= t_f + t_c (\text{base case}) \\ T(n) &= t_f + t_c + t_m + T(n-1), \text{ if } n > 0 \text{ (recursive case).} \end{aligned}$$

Hence, for $n > 0$,

$$\begin{aligned} T(n) &= (t_f + t_c + t_m) + (t_f + t_c + t_m) + T(n-2) \\ &= (t_f + t_c + t_m) + (t_f + t_c + t_m) + (t_f + t_c + t_m) + T(n-3) \\ &= ... \\ &= \underbrace{(t_f + t_c + t_m) + ... + (t_f + t_c + t_m)}_{n \text{ times}} + t_f + t_c \\ &= n \cdot (t_f + t_c + t_m) + t_f + t_c \in O(n). \end{aligned}$$

In general, we could apply the following techniques:

1. Characterize the execution time by a recurrence equation system.
2. Find the solution (closed-form, non-recursive) of the recurrence relation.

   If the solution cannot be derived from the Master theorem (Sect. A.4.1) and is not listed elsewhere (e.g., in a textbook), we could:

   - First, unroll the recurrence relation a few times (as above) to get a hypothesis for a possible closed-form solution: $T(n) = ...$
   - Then, prove the hypothesis for $T(n)$ by induction. If that fails, we need to modify the hypothesis and try again ...

```
int LinearSearch ( int $T[0..n-1]$, int $K$ )
{
  for $i$ from 0 to $n-1$ do
    if $T[i] = K$ then return $i$
  return $-1$  // $K$ not found in $T$
}
```

Figure A.4: Linear Search algorithm for finding a key (here, integer) K in an unordered array T.

A.4.1 The Master Theorem

We summarize the Master theorem as provided in Cormen et al. [17], where also a proof can be found.

Theorem A.1 *(**Master Theorem** [17]) Let $a \geq 1$ and $b > 1$ be constants, and let f be a function defined over the natural numbers, and T be a function defined over the natural numbers including 0, which is defined by the following recurrence equation:*

$$\begin{aligned} T(n) &= a \cdot T(n/b) + f(n) \quad \text{if } n > 0 \\ T(1) &= \Theta(1) \end{aligned}$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, then T can, depending on the structure of f, be bounded asymptotically as follows:

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*
- *If $f(n) = O(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

As an example, for sequential mergesort we have the recurrence equation $T(n) = 2T(n/2) + \Theta(n)$, thus $a = b = 2$ and $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$, and hence the middle case of the Master theorem yields the closed-form solution for T: $T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$.

A.5 Average Case Analysis *

Let us reconsider LinearSearch(): the sequential search for some value through an unordered array. The pseudocode is given in Figure A.4.

Assume now that the input argument (the value we search for) happens to be one of the table elements, and that it is chosen with *equal probability* for all elements.

The *expected search time* is

$$\frac{1+2+3+...+n}{n}\, t_c \;=\; \frac{n(n+1)}{2n}\, t_c \;\in\; O(n)$$

In general, for an average case analysis, we *have to* know the probability distribution for the input data. Average case analysis gives no information about the worst cases.

A.6 Amortized Analysis *

Amortized analysis is done for entire *sequences* of multiple operations and input data. The average time per operation then only holds in the context of the sequence, not for the individual operations in general. Some operations will take longer than this average, while others go faster. While we cannot give tight time bounds for each individual operation, we can give such guarantees for the entire sequence.

For example, given an unsorted array $T[0..n-1]$, consider the sequence of linear search operations for all elements of T in some given permutation order. Then, the total time for the linear searches of *all* n elements is

$$\frac{n(n+1)}{2}(t_f + t_c)$$

and the amortized search time per element is $(n+1)(t_f + t_c)/2$.

A.7 Bibliographical Notes *

The Master theorem formulation in Section A.4.1 is based on Cormen et al. [17], who adapted it from Bentley, Haken and Saxe [6].

A.8 Exercises

1. Prove or disprove:
   $(n+1)^2 \in O(n^3)$
   $(n-1)^3 \in O(n^2)$
   $3^{n-1} \in O(2^n)$
   $\sqrt{n^5} \in O(n^2)$

2. Explain how $o(f)$ relates to $O(f)$ and $\omega(f)$ to $\Omega(f)$. Why is there no need for a $\theta(f)$ (lower-case Θ)?

3. Use the Master theorem to derive the time and work complexities of the bitonic sorting algorithm (see Section 3.3).

4. Use the Master theorem to derive the time and work complexities of Strassen's recursive matrix-matrix multiplication algorithm (see Section 4.3.2).

Bibliography

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer J.*, 36(8):756–762, Dec. 1993.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. 15th Annual Symposium on the Theory of Computing (STOC'83)*, pages 1–9, New York, 1983. ACM.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1994.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, 1967.

[5] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference, Vol. 32*, pages 307–314, Reston, Va., 1968. AFIPS Press.

[6] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[7] R. Bisseling. *Parallel Scientific Computation – A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

[8] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, K. Remington, and R. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Soft.*, 28(2):135–151, 2002.

[9] G. E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Trans. Comput.*, 38(11):1526–1538, Nov. 1989.

[10] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207, 2003.

[11] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[12] D. K. G. Campbell. On the CLUMPS model of parallel computation. *Inform. Process. Lett.*, 66:231–236, 1998.

[13] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.

[14] D. Cederman and P. Tsigas. GPU-Quicksort: a practical Quicksort algorithm for graphics processors. *ACM J. of Experimental Algorithmics*, 14(1.4), July 2009.

[15] R. Cole. Parallel Merge Sort. *SIAM J. Comput.*, 17(4):770–785, Aug. 1988.

[16] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th Annual ACM Symp. Theory of Computing*, pages 1–6, 1987.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[18] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: a Practical Model of Parallel Computation. *Comm. ACM*, 39(11), Nov. 1996.

[19] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM Symp. on Comput. Geometry*, pages 298–307, 1993.

[20] J. Demmel, J. Dongarra, V. Eikhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, Feb. 2005.

[21] J. Dongarra, J. DuCroz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16(1):1–17, 1990.

[22] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14(1):1–32, 1988.

[23] M. Forsell. A scalable high-performance computing solution for networks on chips. *IEEE Micro*, 22(5), sep/oct 2002.

[24] M. Forsell, J. Roivainen, and V. Leppänen. Prototyping the MBTAC processor for the REPLICA CMP. In *Proc. IPDPS Workshops*. IEEE, May 2014.

[25] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symp. Theory of Computing*, pages 114–118, 1978.

[26] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[28] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[29] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to parallel computing, Second Edition*. Addison Wesley, 2003.

[30] J. L. Gustafsson. Re-evaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[31] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[32] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[33] D. Johansson, M. Eriksson, and C. Kessler. Bulk-synchronous parallel computing on the CELL processor. In *Proc. PARS'07: 21. PARS - Workshop, Hamburg, Germany, May 31-Jun 1, 2007. GI/ITG-Fachgruppe Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*, 2008.

[34] H. F. Jordan and G. Alaghband. *Fundamentals of parallel processing*. Prentice Hall, 2003.

[35] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress*, pages 471–475, Aug. 1974.

[36] R. M. Karp, A. Sabay, E. E. Santos, and K.-E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. SPAA'93*, 1993.

[37] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley, New York, 2001.

[38] C. Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.

[39] C. Kessler and W. Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, Apr. 2012.

[40] C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.

[41] C. W. Keßler. Fork homepage, with compiler, SBPRAM simulator, system software, tools, and documentation. www.ida.liu.se/∼chrke/fork/, 2001.

[42] C. W. Keßler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. J. Parallel Programming*, 25(1):17–50, Feb. 1997.

[43] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI systems*, pages 271–294. Addison-Wesley, 1980.

[44] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, Dec. 1999.

[45] R. E. Ladner and M. J. Fisher. Parallel Prefix Computations. *J. ACM*, 27(4):831–838, Oct. 1980.

[46] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software*, 5:308–325, 1979.

[47] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *Proc. Int. Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2010.

[48] C. E. Leiserson. Programming Irregular Parallel Applications in Cilk. In *Proc. IRREGULAR'97 Int. Symp. Solving Irregularly Structured Problems in Parallel*, pages 61–71. Springer LNCS 1253, June 1997.

[49] W. F. McColl. General Purpose Parallel Computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.

[50] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[51] W. J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichtenau, and J. Röhrig. Real PRAM programming. In *Proc. Int. Euro-Par Conf.'02*, Aug. 2002.

[52] A. G. Ranade. How to emulate shared memory. In *Proc. 28th Annual IEEE Symp. Foundations of Computer Science*, pages 185–194, 1987.

[53] A. G. Ranade, S. N. Bhatt, and S. L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, Cambridge, MA, 1988. MIT Press.

[54] F. J. Seinstra and D. Koelma. Incorporating memory layout in the modeling of message passing programs. *J. Systems Architecture*, 49:109–121, 2003.

[55] O. Sinnen. *Task Scheduling for Parallel Systems.* Wiley, 2007.

[56] D. B. Skillicorn. miniBSP: a BSP Language and Transformation System. Technical report, Dept. of Computing and Information Sciences, Queens's University, Kingston, Canada, Oct. 22 1996. http://www.qucis.queensu.ca/home/skill/mini.ps.

[57] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[58] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[59] J. L. Träff. Parallel quicksort without pairwise element exchange. CoRR abs/1804.07494, 2018.

[60] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. In *Proc. 11th Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, 2003.

[61] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8), Aug. 1990.

[62] L. G. Valiant. A bridging model for multi-core computing. *J. Comp. Syst. Sciences*, 77(1), 2011.

[63] R. A. van de Geijn and J. Watts. Summa: Scalable universal matrix multipliation algorithm. Technical Report CS-95-286, University of Tennessee, May 1995.

[64] T. van Dijk and J. van de Pol. Lace: non-blocking split deque for work-stealing. In L. Lopez et al., editor, *Euro-Par 2014 Workshops, Part II, Springer LNCS 8806*, pages 206–217, 2014.

[65] U. Vishkin et al. Explicit multi-threading (XMT). Project web page, users.umiacs.umd.edu/~vishkin/XMT, Sept. 2018.

[66] X. Wen and U. Vishkin. Fpga-based prototype of a PRAM-on-chip processor. In *Proc. 5th conf. on Computing Frontiers (CF'08)*, May 2008.

[67] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Computer Science Department. Cornell University, 1979. Technical Report TR-79-387.

[68] A. Yzelman and R. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012.

Index