Parallel Scientific Computation

A structured approach using BSP and MPI

ROB H. BISSELING

OXFORD

PARALLEL SCIENTIFIC COMPUTATION

This page intentionally left blank

Parallel Scientific Computation

A structured approach using BSP and MPI

ROB H. BISSELING

Utrecht University

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford OX2 6DP

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide in

Oxford New York

Auckland Bangkok Buenos Aires Cape Town Chennai
Dar es Salaam Delhi Hong Kong Istanbul
Karachi Kolkata Kuala Lumpur Madrid Melbourne Mexico City Mumbai
Nairobi São Paulo Shanghai Taipei Tokyo Toronto

Published in the United States
by Oxford University Press Inc., New York

First published 2004

A catalogue record for this title is available from the British Library

Library of Congress Cataloging in Publication Data
(Data available)

ISBN 0 19 852939 2

10 9 8 7 6 5 4 3 2 1

Typeset by Newgen Imaging Systems (P) Ltd., Chennai, India
Printed in Great Britain
on acid-free paper by
Biddles Ltd, www.biddles.co.uk

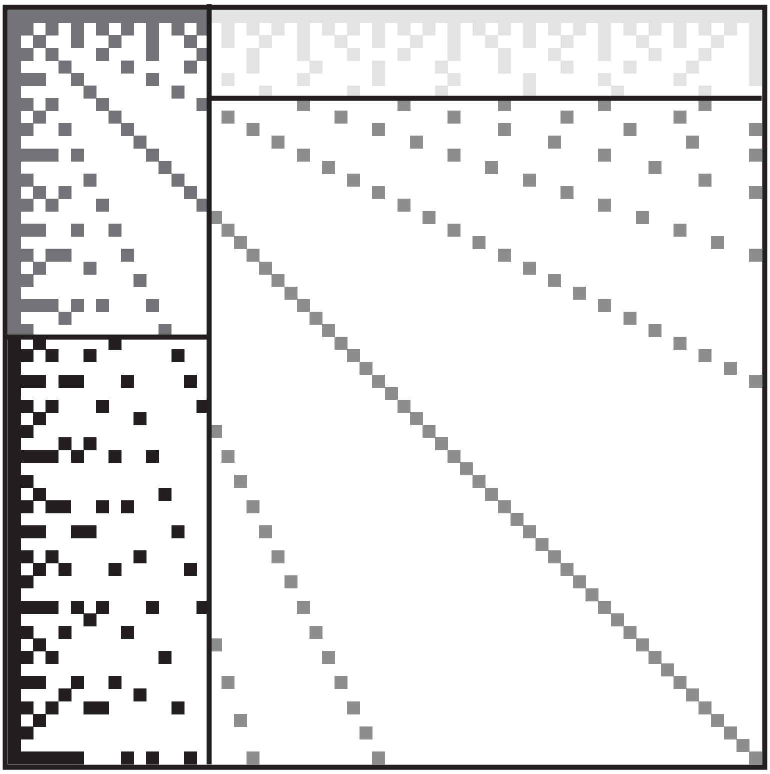


PLATE 1. Sparse matrix `prime60` distributed over four processors of a parallel computer. Cf. Chapter 4.

This page intentionally left blank

PREFACE

Why this book on parallel scientific computation? In the past two decades, several shelves full of books have been written on many aspects of parallel computation, including scientific computation aspects. My intention to add another book asks for a motivation. To say it in a few words, the time is ripe now. The pioneering decade of parallel computation, from 1985 to 1995, is well behind us. In 1985, one could buy the first parallel computer from a commercial vendor. If you were one of the few early users, you probably found yourself working excessively hard to make the computer perform well on your application; most likely, your work was tedious and the results were frustrating. If you endured all this, survived, and are still interested in parallel computation, you deserve strong sympathy and admiration!

Fortunately, the situation has changed. Today, one can theoretically develop a parallel algorithm, analyse its performance on various architectures, implement the algorithm, test the resulting program on a PC or cluster of PCs, and then run the same program with predictable efficiency on a massively parallel computer. In most cases, the parallel program is only slightly more complicated than the corresponding sequential program and the human time needed to develop the parallel program is not much more than the time needed for its sequential counterpart.

This change has been brought about by improvements in parallel hardware and software, together with major advances in the theory of parallel programming. An important theoretical development has been the advent of the Bulk Synchronous Parallel (BSP) programming model proposed by Valiant in 1989 [177,178], which provides a useful and elegant theoretical framework for bridging the gap between parallel hardware and software. For this reason, I have adopted the BSP model as the target model for my parallel algorithms. In my experience, the simplicity of the BSP model makes it suitable for teaching parallel algorithms: the model itself is easy to explain and it provides a powerful tool for expressing and analysing algorithms.

An important goal in designing parallel algorithms is to obtain a good algorithmic structure. One way of achieving this is by designing an algorithm as a sequence of large steps, called supersteps in BSP language, each containing many basic computation or communication operations and a global synchronization at the end, where all processors wait for each other to finish their work before they proceed to the next superstep. Within a superstep, the work is done in parallel, but the global structure of the algorithm is sequential. This simple structure has proven its worth in practice in many parallel applications, within the BSP world and beyond.

Recently, efficient implementations of the BSP model have become available. The first implementation that could be run on many different parallel computers, the Oxford BSP library, was developed by Miller and Reed [140,141] in 1993. The BSP Worldwide organization was founded in 1995, see `http://www.bsp-worldwide.org`, to promote collaboration between developers and users of BSP. One of the goals of BSP Worldwide is to provide a standard library for BSP programming. After extensive discussions in the BSP community, a standard called BSPlib [105] was proposed in May 1997 and an implementation by Hill and coworkers [103], the Oxford BSP toolset, was made available in the public domain, see `http://www.bsp-worldwide.org/implmnts/oxtool`. The programs in the main text of this book make use of the primitives of BSPlib.

Another communication library, which can be used for writing parallel programs and which has had a major impact on the field of parallel computing is the Message-Passing Interface (MPI), formulated in 1994 as MPI-1, and enhanced in 1997 by the MPI-2 extensions. The MPI standard has promoted the portability of parallel programs and one might say that its advent has effectively ended the era of architecture-dependent communication interfaces. The programs in the main text of this book have also been converted to MPI and the result is presented in Appendix C.

I wrote this book for students and researchers who are interested in scientific computation. The book has a dual purpose: first, it is a textbook for a course on parallel scientific computation at the final year undergraduate/first year graduate level. The material of the book is suitable for a one-semester course at a mathematics or computer science department. I tested all material in class at Utrecht University during the period 1993–2002, in an introductory course given every year on parallel scientific computation, called 'Parallel Algorithms for Supercomputers 1', see the course page `http://www.math.uu.nl/people/bisseling/pas1.html` and an advanced course given biannually, see `pas2.html`. These courses are taken by students from mathematics, physics, and computer science. Second, the book is a source of example parallel algorithms and programs for computational physicists and chemists, and for other computational scientists who are eager to get a quick start in parallel computing and want to learn a structured approach to writing parallel programs. Prerequisites are knowledge about linear algebra and sequential programming. The program texts assume basic knowledge of the programming language ANSI C.

The scope of this book is the area of numerical scientific computation; the scope also includes combinatorial scientific computation needed for numerical computations, such as in the area of sparse matrices, but it excludes symbolic scientific computation. The book treats the area of numerical scientific computation by presenting a detailed study of several important numerical problems. Through these problems, techniques are taught for designing and implementing efficient, well-structured parallel algorithms. I selected these particular problems because they are important for applications and because they give rise to a variety of important parallelization techniques. This book treats well-known subjects such as dense LU decomposition, fast Fourier transform (FFT), and sparse matrix–vector multiplication. One can view these subjects as belonging to the area of numerical linear algebra, but they are also fundamental to many applications in scientific computation in general. This choice of problems may not be highly original, but I made an honest attempt to approach these problems in an original manner and to present efficient parallel algorithms and programs for their solution in a clear, concise, and structured way.

Since this book should serve as a textbook, it covers a limited but carefully chosen amount of material; I did not strive for completeness in covering the area of numerical scientific computation. A vast amount of sequential algorithms can be found in *Matrix Computations* by Golub and Van Loan [79] and *Numerical Recipes in C: The Art of Scientific Computing* by Press, Teukolsky, Vetterling, and Flannery [157]. In my courses on parallel algorithms, I have the habit of assigning sequential algorithms from these books to my students and asking them to develop parallel versions. Often, the students go out and perform an excellent job. Some of these assignments became exercises in the present book. Many exercises have the form of programming projects, which are suitable for use in an accompanying computer-laboratory class. I have graded the exercises according to difficulty/amount of work involved, marking an exercise by an asterisk if it requires more work and by two asterisks if it requires a lot of work, meaning that it would be suitable as a final assignment. Inevitably, such a grading is subjective, but it may be helpful for a teacher in assigning problems to students. The main text of the book treats a few central topics from parallel scientific computation in depth; the exercises are meant to give the book breadth.

The structure of the book is as follows. Chapter 1 introduces the BSP model and BSPlib, and as an example it presents a simple complete parallel program. This two-page program alone already teaches half the primitives of BSPlib. (The other half is taught by the program of Chapter 4.) The first chapter is a concise and self-contained tutorial, which tells you how to get started with writing BSP programs, and how to benchmark your computer as a BSP computer. Chapters 2–4 present parallel algorithms for problems with increasing irregularity. Chapter 2 on dense LU decomposition presents a regular computation with communication patterns that are common in matrix computations. Chapter 3 on the FFT also treats a regular computation but one with a more complex flow of data. The execution time requirements of the LU decomposition and FFT algorithms can be analysed exactly and the performance of an implementation can be predicted quite accurately. Chapter 4 presents the multiplication of a sparse matrix and a dense vector. The computation involves only those matrix elements that are nonzero, so that in general it is irregular. The communication involves the components of dense input and output vectors. Although these vectors can be stored in a regular data structure, the communication pattern becomes irregular because efficient communication must exploit the sparsity. The order in which the chapters can be read is: 1, 2, then either 3 or 4, depending on your taste. Chapter 3 has the brains, Chapter 4 has the looks, and after you have finished both you know what I mean. Appendix C presents MPI programs in the order of the corresponding BSPlib programs, so that it can be read in parallel with the main text; it can also be read afterwards. I recommend reading the appendix, even if you do not intend to program in MPI, because it illustrates the various possible choices that can be made for implementing communications and because it makes the differences and similarities between BSPlib and MPI clear.

Each chapter contains: an abstract; a brief discussion of a sequential algorithm, included to make the material self-contained; the design and analysis of a parallel algorithm; an annotated program text; illustrative experimental results of an implementation on a particular parallel computer; bibliographic notes, giving historical background and pointers for further reading; theoretical and practical exercises.

My approach in presenting algorithms and program texts has been to give priority to clarity, simplicity, and brevity, even if this comes at the expense of a slight decrease in efficiency. In this book, algorithms and programs are only optimized if this teaches an important technique, or improves efficiency by an order of magnitude, or if this can be done without much harm to clarity. Hints for further optimization are given in exercises. The reader should view the programs as a starting point for achieving fast implementations.

One goal of this book is to ease the transition from theory to practice. For this purpose, each chapter includes an example program, which presents a possible implementation of the central algorithm in that chapter. The program texts form a small but integral part of this book. They are meant to be read by humans, besides being compiled and executed by computers. Studying the program texts is the best way of understanding what parallel programming is really about. Using and modifying the programs gives you valuable hands-on experience.

The aim of the section on experimental results is to illustrate the theoretical analysis. Often, one aspect is highlighted; I made no attempt to perform an exhaustive set of experiments. A real danger in trying to explain experimental results for an algorithm is that a full explanation may lead to a discussion of nitty-gritty implementation details or hardware quirks. This is hardly illuminating for the algorithm, and therefore I have chosen to keep such explanations to a minimum. For my experiments, I have used six different parallel machines, older ones as well as newer ones: parallel computers come and go quickly.

The bibliographic notes of this book are lengthier than usual, since I have tried to summarize the contents of the cited work and relate them to the topic discussed in the current chapter. Often, I could not resist the temptation to write a few sentences about a subject not fully discussed in the main text, but still worth mentioning.

The source files of the printed program texts, together with a set of test programs that demonstrate their use, form a package called BSPedupack, which is available at `http://www.math.uu.nl/people/bisseling/software.html`. The MPI version, called MPIedupack, is also available from that site. The packages are copyrighted, but freely available under the GNU General Public License, meaning that their programs can be used and modified freely, provided the source and all modifications are mentioned, and every modification is again made freely available under the same license. As the name says, the programs in BSPedupack and MPIedupack are primarily intended for teaching. They are definitely not meant to be used as a black box. If your program, or worse, your airplane crashes because BSP/MPIedupack is not sufficiently robust, it is your own responsibility. Only rudimentary error handling has been built into the programs. Other software available from my software site is the Mondriaan package [188], which is used extensively in Chapter 4. This is actual production software, also available under the GNU General Public License.

To use BSPedupack, a BSPlib implementation such as the Oxford BSP toolset [103] must have been installed on your computer. As an alternative, you can use BSPedupack on top of the Paderborn University BSP (PUB) library [28,30], which contains BSPlib as a subset. If you have a cluster of PCs, connected by a Myrinet network, you may want to use the Panda BSP library, a BSPlib implementation by Takken [173], which is soon to be released. The programs of this book have been tested extensively for the Oxford BSP toolset.

To use the first four programs of MPIedupack, you need an implementation of MPI-1. This often comes packaged with the parallel computer. The fifth program needs MPI-2. Sometimes, part of the MPI-2 extensions have been supplied by the computer vendor. A full public-domain implementation of MPI-2 is expected to become available in the near future for many different architectures as a product of the MPICH-2 project.

If you prefer to use a different communication library than BSPlib, you can port BSPlib programs to other systems such as MPI, as demonstrated by Appendix C. Porting out of BSPlib is easy, because of the limited number of BSPlib primitives and because of the well-structured programs that are the result of following the BSP model. It is my firm belief that if you use MPI or another communication library, for historical or other reasons, you can benefit tremendously from the structured approach to parallel programming taught in this book. If you use MPI-2, and in particular its one-sided communications, you are already close to the bulk synchronous parallel world, and this book may provide you with a theoretical framework.

The programming language used in this book is ANSI C [121]. The reason for this is that many students learn C as their first or second programming language and that efficient C compilers are available for many different sequential and parallel computers. Portability is the name of the game for BSP software. The choice of using C together with BSPlib will make your software run on almost every computer. Since C is a subset of C++, you can also use C++ together with BSPlib. If you prefer to use another programming language, BSPlib is also available in Fortran 90.

Finally, let me express my hope and expectation that this book will transform your barriers: your own activation barrier for parallel programming will disappear; instead, synchronization barriers will appear in your parallel programs and you will know how to use them as an effective way of designing well-structured parallel programs.

R.H. Bisseling
Utrecht University
July 2003

ACKNOWLEDGEMENTS

First of all, I would like to thank Bill McColl of Oxford University for introducing the BSP model to me in 1992 and convincing me to abandon my previous habit of developing special-purpose algorithms for mesh-based parallel computers. Thanks to him, I turned to designing general-purpose algorithms that can run on every parallel computer. Without Bill's encouragement, I would not have written this book.

Special mention should be made of Jon Hill of Oxford University, now at Sychron Ltd, who is co-designer and main implementor of BSPlib. The BSPlib standard gives the programs in the main text of this book a solid foundation. Many discussions with Jon, in particular during the course on BSP we gave together in Jerusalem in 1997, were extremely helpful in shaping this book.

Several visits abroad gave me feedback and exposure to constructive criticism. These visits also provided me the opportunity to test the parallel programs of this book on a variety of parallel architectures. I would like to thank my hosts Michael Berman of BioMediCom Ltd in Jerusalem, Richard Brent and Bill McColl of Oxford University, Iain Duff of CERFACS in Toulouse, Jacko Koster and Fredrik Manne of the University of Bergen, Satish Rao of NEC Research at Princeton, Pilar de la Torre of the University of New Hampshire, and Leslie Valiant of Harvard University, inventor of the BSP model. I appreciate their hospitality. I also thank the Engineering and Physical Sciences Research Council in the United Kingdom for funding my stay in Oxford in 2000, which enabled me to make much progress with this book.

I would like to thank the Oxford Supercomputer Centre for granting access to their SGI Origin 2000 supercomputer and I am grateful to Jeremy Martin and Bob McLatchie for their help in using this machine. I thank Sychron Ltd in Oxford for giving me access to their PC cluster. In the Netherlands, I would like to acknowledge grants of computer time and funding of two postdocs by the National Computer Facilities foundation (NCF). Patrick Aerts, director of the NCF, has tremendously stimulated the development of the high-performance computing infrastructure in the Netherlands, and it is thanks to him that I could use so many hours of computing time on so many different parallel computers. I thank the supercomputer centres HPαC in Delft and SARA in Amsterdam for access to their computers, with personal thanks to Jana Vasiljev and Willem Vermin for supporting BSPlib at these centres. I also thank Aad van der Steen for help in accessing DAS-2, the Dutch distributed supercomputer.

The Mathematics Department of Utrecht University provided me with a stimulating environment for writing this book. Henk van der Vorst started the Computational Science education programme in Utrecht in 1993, which gave me the opportunity to develop and test the material of the book. Over the years, more than a hundred students have graduated from my parallel scientific computation courses. Many have contributed, occasionally by being guinea pigs (my apologies!), but more often as partners in a genuine dialogue. These students have helped me to improve the exposition of the material and they have forced me to be as clear and brief as I can. I thank all of them, with special mention of Tammo-Jan Dijkema and Maarten Löffler who were champion proofreaders and Dik Takken who implemented a version of BSPlib for Windows just to speed up his homework assignment and later implemented a version for a Myrinet-based PC cluster, which is used in Chapter 4.

During the period of writing this book, I was joined in my parallel computing research by MSc students, PhD students, and postdocs. Discussions with them often yielded new insights and I enjoyed many working and off-working hours spent together. In particular, I would like to mention here: Márcia Alves de Inda, Ildikó Flesch, Jeroen van Grondelle, Alexander van Heukelum, Neal Hegeman, Guy Horvitz, Frank van Lingen, Jacko Koster, Joris Koster, Wouter Meesen, Bruce Stephens, Frank van der Stappen, Mark Stijnman, Dik Takken, Patrick Timmers, and Brendan Vastenhouw.

Much of my pre-BSP work has contributed to this book as well. In particular, research I carried out from 1987–93 at the Koninklijke/Shell-laboratory in Amsterdam has taught me much about parallel computing and sparse matrix computations. I became an expert in the programming language occam, which unfortunately could only be used on transputer-based parallel computers, thereby representing perhaps the antithesis of portability. In the pioneering years, however, occam was a great educational tool. Ideas from the prototype library PARPACK, developed in those years at Shell, profoundly influenced the present work. The importance of a structured approach was already apparent then; good structure was obtained in PARPACK by writing programs with communication-closed layers, the predecessor of the BSP superstep. I would like to express my debt to the enlightened management by Arie Langeveld and Theo Verheggen at Shell and to my close colleagues Daniël Loyens and Hans van de Vorst from the parallel computing group.

Going back even further, to my years 1981–6 at the Hebrew University of Jerusalem, my PhD supervisor Ronnie Kosloff aroused my interest in fast Fourier transforms, which has become the subject of Chapter 3. Ronnie seriously influenced my way of working, by injecting me with a large dose of (quantum molecular) dynamics. In Jerusalem, Larry Rudolph introduced me to the field of parallel computing. His enthusiasm and juggling acts left an imprint forever.

Comments on draft chapters of this book have been given by Márcia Alves de Inda, Richard Brent, Olivier Dulieu, Jonathan Hill, Slava Kokoouline, Jacko Koster, Frank van Lingen, Ronald Meester, Adina Milston, John Reid, Dan Stefanescu, Pilar de la Torre, Leslie Valiant, and Yael Weinbach. Aesthetic advice has been given by Ron Bertels, Lidy Bisseling, Gerda Dekker, and Gila and Joel Kantor. Thanks to all of them. Disclaimer: if you find typing errors, small flaws, serious flaws, unintended Dutch, or worse, do not blame them, just flame me! All comments are welcome at: `Rob.Bisseling@math.uu.nl`. I thank my editors at Oxford University Press, Elizabeth Johnston, Alison Jones, and Mahua Nandi, for accepting my vision of this book and for their ideas, good judgement, help, and patience.

Finally, in the writing of this book, I owe much to my family. My wife Rona showed love and sympathy, and gave support whenever needed. Our daughter Sarai, born in 1994, has already acquired quite some mathematical and computer skills. I tested a few exercises on her (admittedly, unmarked ones), and am amazed how much a nine-year old can understand about parallel computing. If she can, you can. Sarai provided me with the right amount of distraction and the proper perspective. Furthermore, one figure in the book is hers.

ABOUT THE AUTHOR

Rob Bisseling is an associate professor at the Mathematics Department of Utrecht University, the Netherlands, where he has worked in the area of scientific computation since 1993. Previously, he held positions as a senior research mathematician (1990–3) and research mathematician (1987–90) at the Koninklijke/Shell-Laboratorium, Amsterdam, the Netherlands, where he investigated the application of parallel computing in oil refinery optimization and polymer modelling. He received a BSc degree cum laude in mathematics, physics, and astronomy in 1977 and an MSc degree cum laude in mathematics in 1981, both from the Catholic University of Nijmegen, the Netherlands, and a PhD degree in theoretical chemistry in 1987 from the Hebrew University of Jerusalem, Israel.

The author has spent sabbaticals as a visiting scientist at Silicon Graphics Biomedical, Jerusalem, in 1997 and at the Programming Research Group of Oxford University, UK, in 2000. He is co-author of the BSPlib standard (1997) and the Mondriaan package for partitioning sparse matrices (2002). Since 2000, he maintains the website of the BSP Worldwide organization. His research interests are numerical and combinatorial scientific computing in general, and parallel algorithms, sparse matrix computations, and bioinformatics in particular. His research goal is to design algorithms and develop software tools that are useful in a wide range of scientific computing applications.

CONTENTS

1

INTRODUCTION

This chapter is a self-contained tutorial which tells you how to get started with parallel programming and how to design and implement algorithms in a structured way. The chapter introduces a simple target architecture for designing parallel algorithms, the bulk synchronous parallel computer. Using the computation of the inner product of two vectors as an example, the chapter shows how an algorithm is designed hand in hand with its cost analysis. The algorithm is implemented in a short program that demonstrates the most important primitives of BSPlib, the main communication library used in this book. If you understand this program well, you can start writing your own parallel programs. Another program included in this chapter is a benchmarking program that allows you to measure the BSP parameters of your parallel computer. Substituting these parameters into a theoretical cost formula for an algorithm gives a prediction of the actual running time of an implementation.

1.1 Wanted: a gentle parallel programming model

Parallel programming is easy if you use the right programming model. All you have to do is find it! Today's developers of scientific application software must pay attention to the use of computer time and memory, and also to the accuracy of their results. Since this is already quite a burden, many developers view **parallelism**, the use of more than one processor to solve a problem, as just an extra complication that is better avoided. Nevertheless, parallel algorithms are being developed and used by many computational scientists in fields such as astronomy, biology, chemistry, and physics. In industry, engineers are trying to accelerate their simulations by using parallel computers. The main motivation of all these brave people is the tremendous computing power promised by parallel computers. Since the advent of the World Wide Web, this power lies only a tempting few mouse-clicks away from every computer desk. The Grid is envisioned to deliver this power to that desk, providing computational services in a way that resembles the workings of an electricity grid.

The potential of parallel computers could be realized if the practice of parallel programming were just as easy and natural as that of sequential programming. Unfortunately, until recently this has not been the case, and parallel computing used to be a very specialized area where exotic parallel algorithms were developed for even more exotic parallel architectures, where software could not be reused and many man years of effort were wasted in developing software of limited applicability. Automatic parallelization by compilers could be a solution for this problem, but this has not been achieved yet, nor is it likely to be achieved in the near future. Our only hope of harnessing the power of parallel computing lies in actively engaging ourselves in parallel programming. Therefore, we might as well try to make parallel programming easy and effective, turning it into a natural activity for everyone who writes computer programs.

An important step forward in making parallel programming easier has been the development of **portability layers**, that is, communication software such as PVM [171] and MPI [137] that enable us to run the same parallel program on many different parallel computers without changing a single line of program text. Still, the resulting execution time behaviour of the program on a new machine is unpredictable (and can indeed be rather erratic), due to the lack of an underlying parallel programming model.

To achieve the noble goal of easy parallel programming we need a model that is simple, efficiently implementable, and acceptable to all parties involved: hardware designers, software developers, and end users. This model should not interfere with the process of designing and implementing algorithms. It should exist mainly in the background, being tacitly understood by everybody. Such a model would encourage the use of parallel computers in the same way as the Von Neumann model did for the sequential computer.

The bulk synchronous parallel (BSP) model proposed by Valiant in 1989 [177,178] satisfies all requirements of a useful parallel programming model: the BSP model is simple enough to allow easy development and analysis of algorithms, but on the other hand it is realistic enough to allow reasonably accurate modelling of real-life parallel computing; a portability layer has been defined for the model in the form of BSPlib [105] and this standard has been implemented efficiently in at least two libraries, namely the Oxford BSP toolset [103] and the Paderborn University BSP library [28,30], each running on many different parallel computers; another portability layer suitable for BSP programming is the one-sided communications part of MPI-2 [138], implementations of which are now appearing; in principle, the BSP model could be used in taking design decisions when building new hardware (in practice though, designers face other considerations); the BSP model is actually being used as the framework for algorithm design and implementation on a range of parallel computers with completely different architectures (clusters of PCs, networks of workstations, shared-memory multiprocessors, and massively parallel computers with distributed memory). The BSP model is explained in the next section.

1.2 The BSP model

The BSP model proposed by Valiant in 1989 [177,178] comprises a computer architecture, a class of algorithms, and a function for charging costs to algorithms. In this book, we use a variant of the BSP model; the differences with the original model are discussed at the end of this section.

A **BSP computer** consists of a collection of processors, each with private memory, and a communication network that allows processors to access memories of other processors. The architecture of a BSP computer is shown in Fig. 1.1. Each processor can read from or write to every memory cell in the entire machine. If the cell is local, the read or write operation is relatively fast. If the cell belongs to another processor, a message must be sent through the communication network, and this operation is slower. The access time for all nonlocal memories is the same. This implies that the communication network can be viewed as a black box, where the connectivity of the network is hidden in the interior. As users of a BSP computer, we need not be concerned with the details of the communication network. We only care about the remote access time delivered by the network, which should be uniform. By concentrating on this single property, we are able to develop **portable** algorithms, that is, algorithms that can be used for a wide range of parallel computers.

A **BSP algorithm** consists of a sequence of supersteps. A **superstep** contains either a number of computation steps or a number of communication steps, followed by a global barrier synchronization. In a **computation superstep**, each processor performs a sequence of operations on local data. In scientific computations, these are typically floating-point operations (flops). (A **flop** is a multiplication, addition, subtraction, or division of two floating-point numbers. For simplicity, we assume that all these operations take the same amount of time.) In a **communication superstep**, each processor

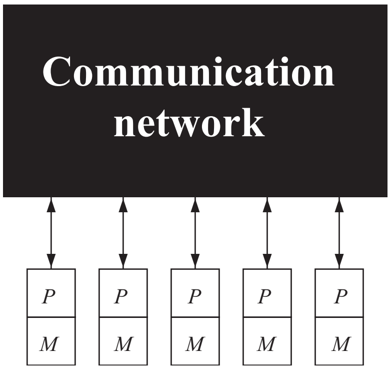


FIG. 1.1. Architecture of a BSP computer. Here, 'P' denotes a processor and 'M' a memory.

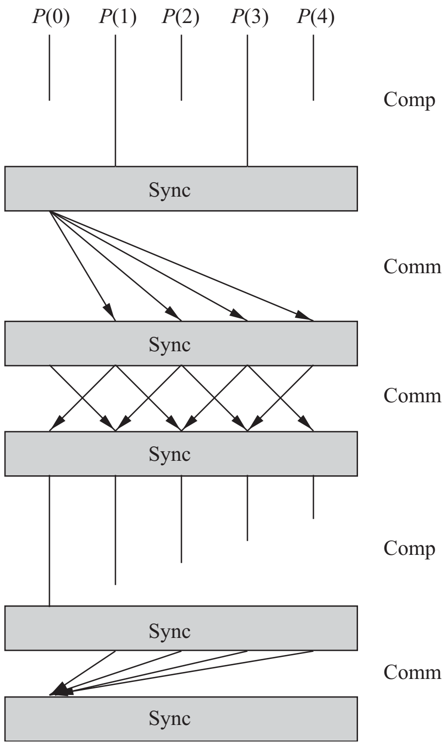


FIG. 1.2. BSP algorithm with five supersteps executed on five processors. A vertical line denotes local computation; an arrow denotes communication of one or more data words to another processor. The first superstep is a computation superstep. The second one is a communication superstep, where processor $P(0)$ sends data to all other processors. Each superstep is terminated by a global synchronization.

sends and receives a number of messages. At the end of a superstep, all processors synchronize, as follows. Each processor checks whether it has finished all its obligations of that superstep. In the case of a computation superstep, it checks whether the computations are finished. In the case of a communication superstep, it checks whether it has sent all messages that it had to send, and whether it has received all messages that it had to receive. Processors wait until all others have finished. When this happens, they all proceed to the next superstep. This form of synchronization is called **bulk synchronization**, because usually many computation or communication operations take place between successive synchronizations. (This is in contrast to pairwise synchronization, used in most message-passing systems, where each message causes a pair of sending and receiving processors to wait until the message has been transferred.) Figure 1.2 gives an example of a BSP algorithm.

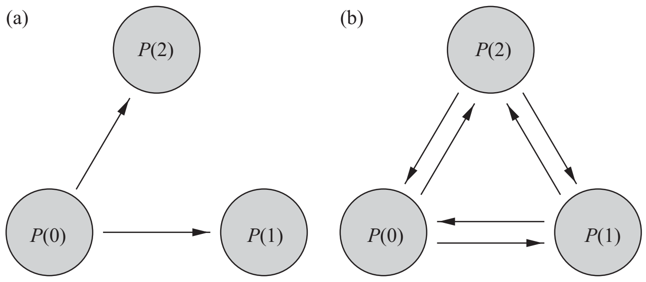


FIG. 1.3. Two different h-relations with the same h. Each arrow represents the communication of one data word. (a) A 2-relation with $h_{\rm s} = 2$ and $h_{\rm r} = 1$; (b) a 2-relation with $h_{\rm s} = h_{\rm r} = 2$.

The **BSP cost function** is defined as follows. An ***h*-relation** is a communication superstep where each processor sends at most h data words to other processors and receives at most h data words, and where at least one processor sends or receives h words. A data word is a real or an integer. We denote the maximum number of words sent by any processor by $h_{\rm s}$ and the maximum number received by $h_{\rm r}$. Therefore,

$$h = \max\{h_{\rm s}, h_{\rm r}\}. \tag{1.1}$$

This equation reflects the assumption that a processor can send and receive data simultaneously. The cost of the superstep depends solely on h. Note that two different communication patterns may have the same h, so that the cost function of the BSP model does not distinguish between them. An example is given in Fig. 1.3. Charging costs on the basis of h is motivated by the assumption that the bottleneck of communication lies at the entry or exit of the communication network, so that simply counting the maximum number of sends and receives gives a good indication of communication time. Note that for the cost function it does not matter whether data are sent together or as separate words.

The time, or **cost**, of an h-relation is

$$T_{\rm comm}(h) = hg + l, \tag{1.2}$$

where g and l are machine-dependent parameters and the time unit is the time of one flop. This cost is charged because of the expected linear increase of communication time with h. Since $g = \lim_{h\to\infty} T_{\rm comm}(h)/h$, the parameter g can be viewed as the time needed to send one data word into the communication network, or to receive one word, in the asymptotic situation of continuous message traffic. The linear cost function includes a nonzero constant l because executing an h-relation incurs a fixed overhead, which includes the cost of global synchronization, but also fixed cost components of ensuring that all data have arrived at their destination and of starting up the communication. We lump all such fixed costs together into one parameter l.

Approximate values for g and l of a particular parallel computer can be obtained by measuring the execution time for a range of **full** h-relations, that is, h-relations where each processor sends and receives exactly h data words. Figure 1.3(b) gives an example of a full 2-relation. (Can you find another full 2-relation?) In practice, the measured cost of a full h-relation will be an upper bound on the measured cost of an arbitrary h-relation. For our measurements, we usually take 64-bit reals or integers as data words.

The cost of a computation superstep is

$$T_{\text{comp}}(w) = w + l, \tag{1.3}$$

where the amount of work w is defined as the maximum number of flops performed in the superstep by any processor. For reasons of simplicity, the value of l is taken to be the same as that of a communication superstep, even though it may be less in practice. As a result, the total synchronization cost of an algorithm can be determined simply by counting its supersteps. Because of (1.2) and (1.3), the total cost of a BSP algorithm becomes an expression of the form $a + bg + cl$. Figure 1.4 displays the cost of the BSP algorithm from Fig. 1.2.

A BSP computer can be characterized by four parameters: p, r, g, and l. Here, p is the **number of processors**. The parameter r is the **single-processor computing rate** measured in flop/s (floating-point operations per second). This parameter is irrelevant for cost analysis and algorithm design, because it just normalizes the time. (But if you wait for the result of your program, you may find it highly relevant!) From a global, architectural point of view, the parameter g, which is measured in flop time units, can be seen as the ratio between the computation throughput and the communication throughput of the computer. This is because in the time period of an h-relation, phg flops can be performed by all processors together and ph data words can be communicated through the network. Finally, l is called the **synchronization cost**, and it is also measured in flop time units. Here, we slightly abuse the language, because l includes fixed costs other than the cost of synchronization as well. Measuring the characteristic parameters of a computer is called **computer benchmarking**. In our case, we do this by measuring r, g, and l. (For p, we can rely on the value advertised by the computer vendor.) Table 1.1 summarizes the BSP parameters.

We can predict the execution time of an implementation of a BSP algorithm on a parallel computer by theoretically analysing the cost of the algorithm and, independently, benchmarking the computer for its BSP performance. The predicted time (in seconds) of an algorithm with cost $a+bg+cl$ on a computer with measured parameters r, g, and l equals $(a + bg + cl)/r$, because the time (in seconds) of one flop equals $t_{\text{flop}} = 1/r$.

One aim of the BSP model is to guarantee a certain performance of an implementation. Because of this, the model states costs in terms of upper

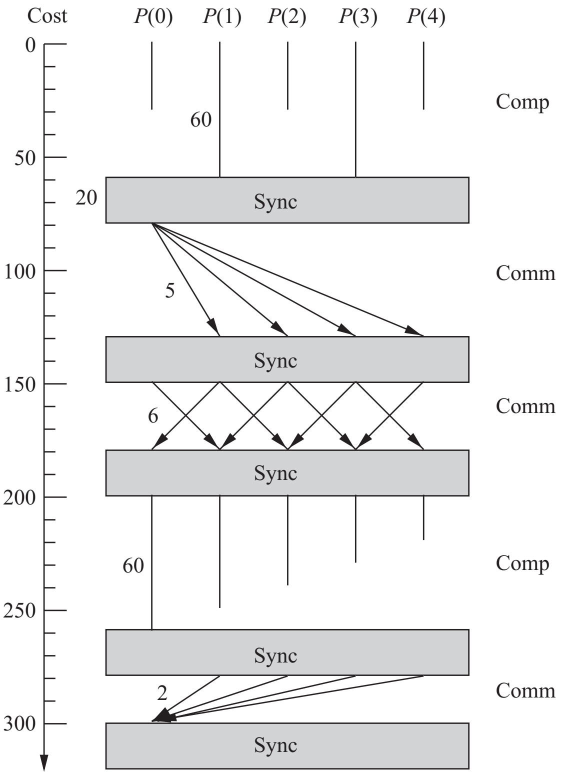


FIG. 1.4. Cost of the BSP algorithm from Fig. 1.2 on a BSP computer with $p = 5$, $g = 2.5$, and $l = 20$. Computation costs are shown only for the processor that determines the cost of a superstep (or one of them, if there are several). Communication costs are shown for only one source/destination pair of processors, because we assume in this example that the amount of data happens to be the same for every pair. The cost of the first superstep is determined by processors $P(1)$ and $P(3)$, which perform 60 flops each. Therefore, the cost is $60 + l = 80$ flop time units. In the second superstep, $P(0)$ sends five data words to each of the four other processors. This superstep has $h_s = 20$ and $h_r = 5$, so that it is a 20-relation and hence its cost is $20g + l = 70$ flops. The cost of the other supersteps is obtained in a similar fashion. The total cost of the algorithm is 320 flops.

TABLE 1.1. The BSP parameters

p	number of processors
r	computing rate (in flop/s)
g	communication cost per data word (in time units of 1 flop)
l	global synchronization cost (in time units of 1 flop)

bounds. For example, the cost function of an h-relation assumes a worst-case communication pattern. This implies that the predicted time is an upper bound on the measured time. Of course, the accuracy of the prediction depends on how the BSP cost function reflects reality, and this may differ from machine to machine.

Separation of concerns is a basic principle of good engineering. The BSP model enforces this principle by separating the hardware concerns from the software concerns. (Because the software for routing messages in the communication network influences g and l, we consider such routing software to be part of the hardware system.) On the one hand, the hardware designer can concentrate on increasing p, r and decreasing g, l. For example, he could aim at designing a BSP(p, r, g, l) computer with $p \geq 100$, $r \geq 1$ Gflop/s (the prefix G denotes Giga $= 10^9$), $g \leq 10$, $l \leq 1000$. To stay within a budget, he may have to trade off these objectives. Obviously, the larger the number of processors p and the computing rate r, the more powerful the communication network must be to keep g and l low. It may be preferable to spend more money on a better communication network than on faster processors. The BSP parameters help in quantifying these design issues. On the other hand, the software designer can concentrate on decreasing the algorithmic parameters a, b, and c in the cost expression $a + bg + cl$; in general, these parameters depend on p and the problem size n. The aim of the software designer is to obtain good scaling behaviour of the cost expression. For example, she could realistically aim at a decrease in a as $1/p$, a decrease in b as $1/\sqrt{p}$, and a constant c.

The BSP model is a distributed-memory model. This implies that both the computational work and the data are distributed. The work should be distributed evenly over the processors, to achieve a good load balance. The data should be distributed in such a way that the total communication cost is limited. Often, the data distribution determines the work distribution in a natural manner, so that the choice of data distribution must be based on two objectives: good load balance and low communication cost. In all our algorithms, choosing a data distribution is an important decision. By designing algorithms for a distributed-memory model, we do not limit ourselves to this model. Distributed-memory algorithms can be used efficiently on shared-memory parallel computers, simply by partitioning the shared memory among the processors. (The reverse is not true: shared-memory algorithms do not take data locality into account, so that straightforward distribution leads to inefficient algorithms.) We just develop a distributed-memory program; the BSP system does the rest. Therefore, BSP algorithms can be implemented efficiently on every type of parallel computer.

The main differences between our variant of the BSP model and the original BSP model [178] are:

1. The original BSP model allows supersteps to contain both computation and communication. In our variant, these operations must be split

into separate supersteps. We impose this restriction to achieve conceptual simplicity. We are not concerned with possible gains obtained by overlapping computation and communication. (Such gains are minor at best.)

2. The original cost function for a superstep with an h-relation and a maximum amount of work w is $\max(w, hg, L)$, where L is the **latency**, that is, the minimum number of time units between successive supersteps. In our variant, we charge $w + hg + 2l$ for the corresponding two supersteps. We use the synchronization cost l instead of the (related) latency L because it facilitates the analysis of algorithms. For example, the total cost in the original model of a 2-relation followed by a 3-relation equals $\max(2g, L) + \max(3g, L)$, which may have any of the outcomes $5g$, $3g + L$, and $2L$, whereas in our variant we simply charge $5g + 2l$. If g and l (or L) are known, we can substitute their values into these expressions and obtain one scalar value, so that both variants are equally useful. However, we also would like to analyse algorithms without knowing g and l (or L), and in that case our cost function leads to simpler expressions. (Valiant [178] mentions $w + hg + l$ as a possible alternative cost function for a superstep with both computation and communication, but he uses $\max(w, hg, l)$ in his analysis.)

3. The data distribution in the original model is controlled either directly by the user, or, through a randomizing hash function, by the system. The latter approach effectively randomizes the allocation of memory cells, and thereby it provides a shared-memory view to the user. Since this approach is only efficient in the rare case that g is very low, $g \approx 1$, we use the direct approach instead. Moreover, we use the data distribution as our main means of making computations more efficient.

4. The original model allowed for synchronization of a subset of all processors instead of all processors. This option may be useful in certain cases, but for reasons of simplicity we disallow it.

1.3 BSP algorithm for inner product computation

A simple example of a BSP algorithm is the following computation of the inner product α of two vectors $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$ and $\mathbf{y} = (y_0, \ldots, y_{n-1})^{\mathrm{T}}$,

$$\alpha = \sum_{i=0}^{n-1} x_i y_i. \qquad (1.4)$$

In our terminology, vectors are column vectors; to save space we write them as $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$, where the superscript 'T' denotes transposition. The vector $\mathbf{x}$ can also be viewed as an $n \times 1$ matrix. The inner product of $\mathbf{x}$ and $\mathbf{y}$ can concisely be expressed as $\mathbf{x}^{\mathrm{T}}\mathbf{y}$.

The inner product is computed by the processors $P(0), \ldots, P(p-1)$ of a BSP computer with p processors. We assume that the result is needed by all processors, which is usually the case if the inner product computation is part of a larger computation, such as in iterative linear system solvers.

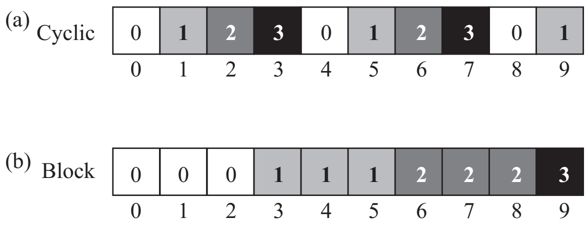


FIG. 1.5. Distribution of a vector of size ten over four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. (a) Cyclic distribution; (b) block distribution.

The data distribution of the vectors $\mathbf{x}$ and $\mathbf{y}$ should be the same, because in that case the components x_i and y_i reside on the same processor and they can be multiplied immediately without any communication. The data distribution then determines the work distribution in a natural manner. To balance the work load of the algorithm, we must assign the same number of vector components to each processor. Card players know how to do this blindly, even without counting and in the harshest of circumstances. They always deal out their cards in a cyclic fashion. For the same reason, an optimal work distribution is obtained by the **cyclic distribution** defined by the mapping

$$x_i \longmapsto P(i \bmod p), \quad \text{for } 0 \le i < n. \tag{1.5}$$

Here, the mod operator stands for taking the remainder after division by p, that is, computing modulo p. Similarly, the div operator stands for integer division rounding down. Figure 1.5(a) illustrates the cyclic distribution for $n = 10$ and $p = 4$. The maximum number of components per processor is $\lceil n/p \rceil$, that is, n/p rounded up to the nearest integer value, and the minimum is $\lfloor n/p \rfloor = n \text{ div } p$, that is, n/p rounded down. The maximum and the minimum differ at most by one. If p divides n, every processor receives exactly n/p components. Of course, many other data distributions also lead to the best possible load balance. An example is the **block distribution**, defined by the mapping

$$x_i \longmapsto P(i \text{ div } b), \quad \text{for } 0 \le i < n, \tag{1.6}$$

with block size $b = \lceil n/p \rceil$. Figure 1.5(b) illustrates the block distribution for $n = 10$ and $p = 4$. This distribution has the same maximum number of components per processor, but the minimum can take every integer value between zero and the maximum. In Fig. 1.5(b) the minimum is one. The minimum can even be zero: if $n = 9$ and $p = 4$, then the block size is $b = 3$, and the processors receive 3, 3, 3, 0 components, respectively. Since the computation cost is determined by the maximum amount of work, this is just as good as

Algorithm 1.1. Inner product algorithm for processor $P(s)$, with $0 \le s < p$.

```
input:      $\mathbf{x}, \mathbf{y}$ : vector of length $n$,
            $\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
            with $\phi(i) = i \bmod p$, for $0 \le i < n$.
output:     $\alpha = \mathbf{x}^{\mathrm{T}}\mathbf{y}$.

(0)   $\alpha_s := 0$;
      for $i := s$ to $n - 1$ step $p$ do
            $\alpha_s := \alpha_s + x_i y_i$;

(1)   for $t := 0$ to $p - 1$ do
            put $\alpha_s$ in $P(t)$;

(2)   $\alpha := 0$;
      for $t := 0$ to $p - 1$ do
            $\alpha := \alpha + \alpha_t$;
```

the cyclic distribution, which assigns 3, 2, 2, 2 components. Intuitively, you may object to the idling processor in the block distribution, but the work distribution is still optimal!

Algorithm 1.1 computes an inner product in parallel. It consists of three supersteps, numbered (0), (1), and (2). The synchronizations at the end of the supersteps are not written explicitly. All the processors follow the same program text, but their actual execution paths differ. The path of processor $P(s)$ depends on the processor identity s, with $0 \le s < p$. This style of programming is called **single program multiple data** (SPMD), and we shall use it throughout the book.

In superstep (0), processor $P(s)$ computes the local partial inner product

$$\alpha_s = \sum_{0 \le i < n,\ i \bmod p = s} x_i y_i, \tag{1.7}$$

multiplying x_s by y_s, x_{s+p} by y_{s+p}, x_{s+2p} by y_{s+2p}, and so on, and adding the results. The data for this superstep are locally available. Note that we use global indices so that we can refer uniquely to variables without regard to the processors that own them. We access the local components of a vector by stepping through the arrays with a **stride**, or step size, p.

In superstep (1), each processor **broadcasts** its result α_s, that is, it sends α_s to all processors. We use the communication primitive 'put x in $P(t)$' in the program text of $P(s)$ to denote the one-sided action by processor $P(s)$ of storing a data element x at another processor $P(t)$. This completely determines the communication: both the source processor and the destination processor of the data element are specified. The 'put' primitive assumes that the source processor knows the memory location on the destination processor where the data must be put. The source processor is the initiator of the action, whereas the destination processor is passive. Thus, we assume implicitly that each processor allows all others to put data into its memory. Superstep (1) could also have been written as 'put α_s in $P(*)$', where we use the abbreviation $P(*)$ to denote all processors. Note that the program includes a put by processor $P(s)$ into itself. This operation is simply skipped or becomes a local memory-copy, but it does not involve communication. It is convenient to include such puts in program texts, to avoid having to specify exceptions.

Sometimes, it may be necessary to let the destination processor initiate the communication. This may happen in irregular computations, where the destination processor knows that it needs data, but the source processor is unaware of this need. In that case, the destination processor must fetch the data from the source processor. This is done by a statement of the form 'get x from $P(t)$' in the program text of $P(s)$. In most cases, however, we use the 'put' primitive. Note that using a 'put' is much simpler than using a matching 'send'/'receive' pair, as is done in message-passing parallel algorithms. The program text of such an algorithm must contain additional if-statements to distinguish between sends and receives. Careful checking is needed to make sure that pairs match in all possible executions of the program. Even if every send has a matching receive, this does not guarantee correct communication as intended by the algorithm designer. If the send/receive is done by the handshake (or kissing) protocol, where both participants can only continue their way after the handshake has finished, then it can easily happen that the sends and receives occur in the wrong order. A classic case is when two processors both want to send first and receive afterwards; this situation is called **deadlock**. Problems such as deadlock cannot happen when using puts.

In superstep (2), all processors compute the final result. This is done **redundantly**, that is, the computation is replicated so that all processors perform exactly the same operations on the same data. The complete algorithm is illustrated in Fig. 1.6.

The cost analysis of the algorithm is as follows. Superstep (0) requires a floating-point multiplication and an addition for each component. Therefore, the cost of (0) is $2\lceil n/p \rceil + l$. Superstep (1) is a $(p-1)$-relation, because each processor sends and receives $p-1$ data. (Communication between a processor and itself is not really communication and hence is not counted in determining h.) The cost of (1) is $(p-1)g + l$. The cost of (2) is $p + l$. The total cost of the inner product algorithm is

$$T_{\text{inprod}} = 2\left\lceil \frac{n}{p} \right\rceil + p + (p-1)g + 3l. \tag{1.8}$$

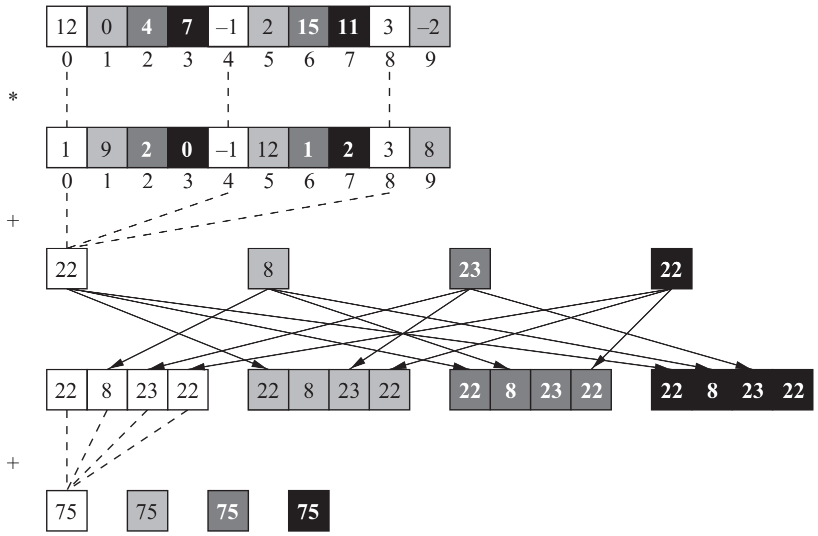


FIG. 1.6. Parallel inner product computation. Two vectors of size ten are distributed by the cyclic distribution over four processors. The processors are shown by greyshades. First, each processor computes its local inner product. For example, processor $P(0)$ computes its local inner product $12 \cdot 1 + (-1) \cdot (-1) + 3 \cdot 3 = 22$. Then the local result is sent to all other processors. Finally, the local inner products are summed redundantly to give the result 75 in every processor.

An alternative approach would be to send all partial inner products to one processor, $P(0)$, and let this processor compute the result and broadcast it. This requires four supersteps. Sending the partial inner products to $P(0)$ is a $(p-1)$-relation and therefore is just as expensive as broadcasting them. While $P(0)$ adds partial inner products, the other processors are idle and hence the cost of the addition is the same as for the redundant computation. The total cost of the alternative algorithm would be $2\lceil n/p \rceil + p + 2(p-1)g + 4l$, which is higher than that of Algorithm 1.1. The lesson to be learned: if you have to perform an h-relation with a particular h, you might as well perform as much useful communication as possible in that h-relation; other supersteps may benefit from this.

1.4 Starting with BSPlib: example program `bspinprod`

BSPlib is a standard library interface, which defines a set of primitives for writing bulk synchronous parallel programs. Currently, BSPlib has been implemented efficiently in two libraries, namely the Oxford BSP toolset [103] and the Paderborn University BSP library [28,30]. The BSPlib standard unifies and extends its two predecessors, the Oxford BSP library designed by Miller and Reed [140,141] and the Green BSP library by Goudreau *et al.* [81].

Implementations of BSPlib exist for many different architectures, including: a cluster of PCs running Linux; a network of UNIX workstations connected by an Ethernet and communicating through the TCP/IP or UDP/IP protocol; shared-memory multiprocessors running a variant of UNIX; and massively parallel computers with distributed memory such as the Cray T3E, the Silicon Graphics Origin, and the IBM SP. In addition, the library can also be used on an ordinary sequential computer to run a parallel program with $p = 1$ as a sequential program; in this case, a special version of BSPlib can be used that strips off the parallel overhead. This is advantageous because it allows us to develop and maintain only one version of the source program, namely the parallel version. It is also possible to simulate a parallel computer by running p processes in parallel on one processor, sharing the time of the common CPU. This is a useful environment for developing parallel programs, for example, on a PC.

The BSPlib library contains a set of primitive functions, which can be called from a program written in a conventional programming language such as C, C++, or Fortran 90. BSPlib was designed following the motto 'small is beautiful'. The primitives of BSPlib were carefully crafted and particular attention was paid to the question of what to exclude from the library. As a result, BSPlib contains only 20 primitive functions, which are also called the **core operations**. In this section, we present a small C program, which uses 12 different primitives. The aim of this tutorial program is to get you started using the library and to expose the main principles of writing a BSPlib program. The remainder of this book gives further examples of how the library can be used. Six primitives for so-called bulk synchronous message passing will be explained in Chapter 4, where they are first needed. Two primitives for high-performance communication are explained in an exercise in Chapter 2. They are primarily meant for programmers who want the ultimate in performance, in terms of memory and computing speed, and who are prepared to live on the edge and be responsible for the safety of their programs, instead of relying on the BSP system to provide this. A quick reference guide to the BSPlib primitives is given as Appendix B. An alternative to using BSPlib is using MPI. Appendix C discusses how to program in BSP style using MPI and presents MPI versions of all the programs from Chapters 1 to 4. For a full explanation of the BSPlib standard, see the definitive source by Hill *et al.* [105]. In the following, we assume that a BSPlib implementation has already been installed. To start with, you could install the Oxford BSP toolset [103] on your PC running Linux, or ask your systems administrator to install the toolset at your local network of workstations.

The parallel part of a BSPlib program starts with the statement

```
bsp_begin(reqprocs);
```

where `int reqprocs` is the number of processors requested. The function `bsp_begin` starts several executions of the same subprogram, where each execution takes place on a different processor and handles a different stream of data, in true SPMD style. The parallel part is terminated by

```
bsp_end();
```

Two possible modes of operation can be used. In the first mode, the whole computation is SPMD; here, the call to `bsp_begin` must be the first executable statement in the program and the call to `bsp_end` the last. Sometimes, however, one desires to perform some sequential part of the program before and after the parallel part, for example, to handle input and output. For instance, if the optimal number of processors to be used depends on the input, we want to compute it before the actual parallel computation starts. The second mode enables this: processor $P(0)$ executes the sequential parts and all processors together perform the parallel part. Processor $P(0)$ preserves the values of its variables on moving from one part to the next. The other processors do not inherit values; they can only obtain desired data values by communication. To allow the second mode of operation and to circumvent the restriction of `bsp_begin` and `bsp_end` being the first and last statement, the actual parallel part is made into a separate function `spmd` and an initializer

```
bsp_init(spmd, argc, argv);
```

is called as the first executable statement of the `main` function. Here, `int argc` and `char **argv` are the standard arguments of `main` in a C program, and these can be used to transfer parameters from a command line interface. Funny things may happen if this is not the first executable statement. Do not even think of trying it! The initializing statement is followed by: a sequential part, which may handle some input or ask for the desired number of processors (depending on the input size it may be better to use only part of the available processors); the parallel part, which is executed by `spmd`; and finally another sequential part, which may handle output. The sequential parts are optional.

The rules for I/O are simple: processor $P(0)$ is the only processor that can read from standard input or can access the file system, but all processors can write to standard output. Be aware that this may mix the output streams; use an `fflush(stdout)` statement to empty the output buffer immediately and increase the chance of obtaining ordered output (sorry, no guarantees).

At every point in the parallel part of the program, one can enquire about the total number of processors. This integer is returned by

```
bsp_nprocs();
```

The function `bsp_nprocs` also serves a second purpose: when it is used in the sequential part at the start, or in the `bsp_begin` statement, it returns the available number of processors, that is, the size of the BSP machine used. Any desired number of processors not exceeding the machine size can be assigned to the program by `bsp_begin`. The local processor identity, or processor number, is returned by

```
bsp_pid();
```

It is an integer between 0 and `bsp_nprocs()`−1. One can also enquire about the time in seconds elapsed on the local processor since `bsp_begin`; this time is given as a double-precision value by

```
bsp_time();
```

Note that in the parallel context the **elapsed time**, or wall-clock time, is often the desired metric and not the CPU time. In parallel programs, processors are often idling because they have to wait for others to finish their part of a computation; a measurement of elapsed time includes idle time, whereas a CPU time measurement does not. Note, however, that the elapsed time metric does have one major disadvantage, in particular to your fellow users: you need to claim the whole BSP computer for yourself when measuring run times.

Each superstep of the SPMD part, or **program superstep**, is terminated by a global synchronization statement

```
bsp_sync();
```

except the last program superstep, which is terminated by `bsp_end`. The structure of a BSPlib program is illustrated in Fig. 1.7. Program supersteps may be contained in loops and if-statements, but the condition evaluations of these loops and if-statements must be such that all processors pace through the same sequence of program supersteps. The rules imposed by BSPlib may seem restrictive, but following them makes parallel programming easier, because they guarantee that all processors are in the same superstep. This allows us to assume full data integrity at the start of each superstep.

The version of the BSP model presented in Section 1.2 does not allow computation and communication in the same superstep. The BSPlib system automatically separates computation and communication, since it delays communication until all computation is finished. Therefore the user does not have to separate these parts herself and she also does not have to include a `bsp_sync` for this purpose. In practice, this means that BSPlib programs can freely mix computation and communication. The automatic separation feature of BSPlib is convenient, for instance because communication often involves address calculations and it would be awkward for a user to separate these computations from the corresponding communication operations. A program superstep can thus be viewed as a sequence of computation, implicit synchronization, communication, and explicit synchronization. The computation part or the communication part may be empty. Therefore, a program superstep may contain one or two supersteps as defined in the BSP model, namely a computation superstep and/or a communication superstep. From

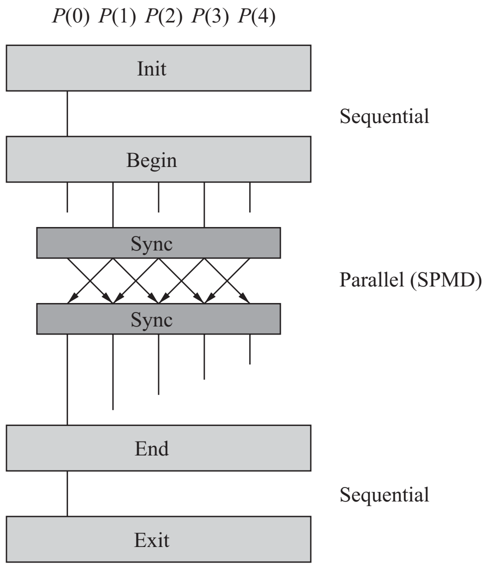


FIG. 1.7. Structure of a BSPlib program. The program first initializes the BSP machine to be used and then it performs a sequential computation on $P(0)$, followed by a parallel computation on five processors. It finishes with a sequential computation on $P(0)$.

now on, we use the shorter term 'superstep' to denote program supersteps as well, except when this would lead to confusion.

Wouldn't it be nice if we could compute and communicate at the same time? This tempting thought may have occurred to you by now. Indeed, processors could in principle compute while messages travel through the communication network. Exploiting this form of parallelism would reduce the total computation/communication cost $a + bg$ of the algorithm, but at most by a factor of two. The largest reduction would occur if the cost of each computation superstep were equal to the cost of the corresponding communication superstep, and if computation and communication could be overlapped completely. In most cases, however, either computation or communication dominates, and the cost reduction obtained by overlapping is insignificant. Surprisingly, BSPlib guarantees *not* to exploit potential overlap. Instead, delaying all communication gives more scope for optimization, since this allows the system to combine different messages from the same source to the same destination and to reorder the messages with the aim of balancing the communication traffic. As a result, the cost may be reduced by much more than a factor of two.

Processors can communicate with each other by using the `bsp_put` and `bsp_get` functions (or their high-performance equivalents `bsp_hpput` and

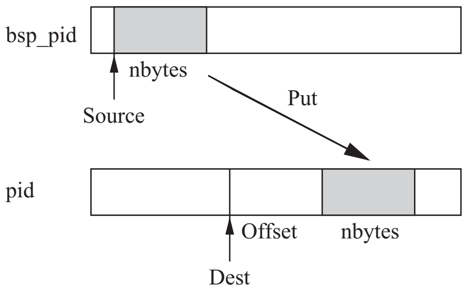


Fig. 1.8. Put operation from BSPlib. The `bsp_put` operation copies `nbytes` of data from the local processor `bsp_pid` into the specified destination processor `pid`. The pointer `source` points to the start of the data to be copied, whereas the pointer `dest` specifies the start of the memory area where the data is written. The data is written at `offset` bytes from the start.

`bsp_hpget`, see Exercise 10, or the `bsp_send` function, see Section 4.9). A processor that calls `bsp_put` reads data from its own memory and writes them into the memory of another processor. The function `bsp_put` corresponds to the put operation in our algorithms. The syntax is

```
bsp_put(pid, source, dest, offset, nbytes);
```

Here, `int pid` is the identity of the remote processor; `void *source` is a pointer to the source memory in the local processor from which the data are read; `void *dest` is a pointer to the destination memory in the remote processor into which the data are written; `int offset` is the number of bytes to be added to the address `dest` to obtain the address where writing starts; and `int nbytes` is the number of bytes to be written. The `dest` variable must have been registered previously; the registration mechanism will be explained soon. If `pid` equals `bsp_pid`, the put is done locally by a memory copy, and no data is communicated. The offset is determined by the local processor, but the destination address is part of the address space of the remote processor. The use of an offset separates the concerns of the local processor, which knows where in an array a data element should be placed, from the concerns of the remote processor, which knows the address of the array in its own address space. The `bsp_put` operation is illustrated in Fig. 1.8.

The `bsp_put` operation is safe in every sense, since the value to be put is first written into a local out-buffer, and only at the end of the superstep (when all computations in all processors are finished) it is transferred into a remote in-buffer, from which it is finally copied into the destination memory. The user can manipulate both the source and destination value without worrying about possible interference between data manipulation and transfer. Once the `bsp_put` is initiated, the user has got rid of the source data and can reuse the variable that holds them. The destination variable can be used until the end of the superstep, when it will be overwritten. It is possible to put several values into the same memory cell, but of course only one value survives and reaches the next superstep. The user cannot know which value, and he bears the responsibility for ensuring correct program behaviour. Put and get operations do not block progress within their superstep: after a put or get is initiated, the program proceeds immediately.

Although a remote variable may have the same name as a local variable, it may still have a different physical memory address because each processor could have its own memory allocation procedure. To enable a processor to write into a remote variable, there must be a way to link the local name to the correct remote address. Linking is done by the registration primitive

```
bsp_push_reg(variable, nbytes);
```

where `void *variable` is a pointer to the variable being registered. All processors must simultaneously register a variable, or the `NULL` pointer; they must also deregister simultaneously. This ensures that they go through the same sequence of registrations and deregistrations. Registration takes effect at the start of the next superstep. From that moment, all simultaneously registered variables are linked to each other. Usually, the name of each variable linked in a registration is the same, in the right SPMD spirit. Still, it is allowed to link variables with different names.

If a processor wants to put a value into a remote address, it can do this by using the local name that is linked to the remote name and hence to the desired remote address. The second registration parameter, `int nbytes`, is an upper bound on the number of bytes that can be written starting from `variable`. Its sole purpose is sanity checking: our hope is to detect insane programs in their youth.

A variable is deregistered by a call to

```
bsp_pop_reg(variable);
```

Within a superstep, the variables can be registered and deregistered in arbitrary order. The same variable may be registered several times, but with different sizes. (This may happen for instance as a result of registration of the same variable inside different functions.) A deregistration cancels the last registration of the variable concerned. The last surviving registration of a variable is the one valid in the next superstep. For each variable, a stack of registrations is maintained: a variable is pushed onto the stack when it is registered; and it is popped off the stack when it is deregistered. A **stack** is the computer science equivalent of the hiring and firing principle for teachers in the Dutch educational system: Last In, First Out (LIFO). This keeps the average stack population old, but that property is irrelevant for our book. In a sensible program, the number of registrations is kept limited. Preferably, a registered variable is reused many times, to amortize the associated overhead costs. Registration is costly because it requires a broadcast of the registered local variable to the other processors and possibly an additional synchronization.

A processor that calls the `bsp_get` function reads data from the memory of another processor and writes them into its own memory. The syntax is

```
bsp_get(pid, source, offset, dest, nbytes);
```

The parameters of `bsp_get` have the same meaning as those of `bsp_put`, except that the source memory is in the remote processor and the destination memory in the local processor and that the offset is in the source memory. The offset is again computed by the local processor. The `source` variable must have been registered previously. The value obtained by the `bsp_get` operation is the source value immediately *after* the computations of the present superstep have terminated, but *before* it can be modified by other communication operations.

If a processor detects an error, it can take action and bring down all other processors in a graceful manner by a call to

```
bsp_abort(error_message);
```

Here, `error_message` is an output string such as used by the `printf` function in C. Proper use of the abort facility makes it unnecessary to check periodically whether all processors are still alive and computing.

BSPlib contains only core operations. By keeping the core small, the BSPlib designers hoped to enable quick and efficient implementation of BSPlib on every parallel computer that appears on the market. Higher level functions such as broadcasting or global summing, generally called **collective communication**, are useful but not really necessary. Of course, users can write their own higher level functions on top of the primitive functions, giving them exactly the desired functionality, or use the predefined ones available in the Oxford BSP toolset [103]. Section 2.5 gives an example of a collective-communication function, the broadcast of a vector.

The best way of learning to use the library is to study an example and then try to write your own program. Below, we present the function `bspip`, which is an implementation of Algorithm 1.1 in C using BSPlib, and the test program `bspinprod`, which handles input and output for a test problem. Now, try to compile the program by the UNIX command

```
bspcc -o ip bspinprod.c bspedupack.c -lm
```

and run the resulting executable program `ip` on four processors by the command

```
bsprun -npes 4 ip
```

and see what happens for this particular test problem, defined by $x_i = y_i = i+1$, for $0 \leq i < n$. (If you are not running a UNIX variant, you may have to follow a different procedure.) A listing of the file `bspedupack.c` can be found

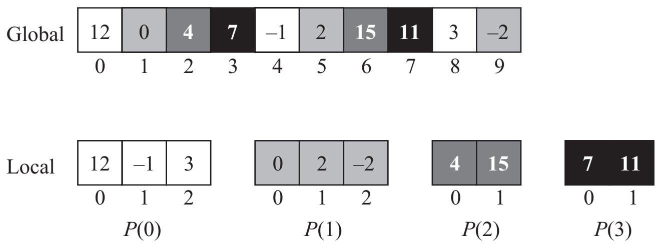


FIG. 1.9. Two different views of the same vector. The vector of size ten is distributed by the cyclic distribution over four processors. The numbers in the square cells are the numerical values of the vector components. The processors are shown by greyshades. The global view is used in algorithms, where vector components are numbered using global indices j. The local view is used in implementations, where each processor has its own part of the vector and uses its own local indices `j`.

in Appendix A. It contains functions for allocation and deallocation of vectors and matrices.

The relation between Algorithm 1.1 and the function `bspip` is as follows. The variables p, s, t, n, α of the algorithm correspond to the variables `p, s, t, n, alpha` of the function. The local inner product α_s of $P(s)$ is denoted by `inprod` in the program text of $P(s)$, and it is also put into `Inprod`[`s`] in all processors. The global index i in the algorithm equals `i * p + s`, where `i` is a local index in the program. The vector component x_i corresponds to the variable `x`[`i`] on the processor that owns x_i, that is, on processor $P(i \bmod p)$. The number of local indices on $P(s)$ is `nloc(p,s,n)`. The first $n \bmod p$ processors have $\lceil n/p \rceil$ such indices, while the others have $\lfloor n/p \rfloor$. Note the efficient way in which `nloc` is computed and check that this method is correct, by writing $n = ap + b$ with $0 \leq b < p$ and expanding the expression returned by the function.

It is convenient to describe algorithms such as Algorithm 1.1 in global variables, but to implement them using local variables. This avoids addressing by a stride and its worse alternative, the superfluous test '**if** $i \bmod p = s$' in a loop over the global index i. Using local, consecutive indices is most natural in an implementation because only a subarray of the original global array is stored locally. The difference between the global and local view is illustrated in Fig. 1.9. For all our distributions, we store the local vector components in order of increasing global index. This gives rise to a natural mapping between local and global indices.

The program printed below largely explains itself; a few additional explanations are given in the following. The included file `bspedupack.h` can be found in Appendix A. It contains inclusion statements for standard header files and also constants such as the size of a double `SZDBL`. The number of processors is first stored as a global variable `P` (global in the C sense, that is, accessible to all functions in its file), so that we are able to transfer the value of `P` from the main program to the SPMD part. Values cannot be transferred other than by using global variables, because the SPMD function is not allowed to have parameters. The function `vecallocd` from `bspedupack.c` is used to allocate an array of doubles of length `p` dynamically and `vecfreed` is used to free the array afterwards.

The offset in the first `bsp_put` is `s*SZDBL`, since the local inner product of processor `s` is put into `Inprod`[s] on every processor `t`. The processors synchronize before the time measurements by `bsp_time`, so that the measurements start and finish simultaneously.

```
#include "bspedupack.h"

/*  This program computes the sum of the first n squares, for n>=0,
        sum = 1*1 + 2*2 + ... + n*n
    by computing the inner product of x=(1,2,...,n)^T and itself.
    The output should equal n*(n+1)*(2n+1)/6.
    The distribution of x is cyclic.
*/

int P; /* number of processors requested */

int nloc(int p, int s, int n){
    /* Compute number of local components of processor s for vector
       of length n distributed cyclically over p processors. */

    return  (n+p-s-1)/p ;

} /* end nloc */

double bspip(int p, int s, int n, double *x, double *y){
    /* Compute inner product of vectors x and y of length n>=0 */

    int nloc(int p, int s, int n);
    double inprod, *Inprod, alpha;
    int i, t;

    Inprod= vecallocd(p); bsp_push_reg(Inprod,p*SZDBL);
    bsp_sync();

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++){
        inprod += x[i]*y[i];
    }

    for (t=0; t<p; t++){
        bsp_put(t,&inprod,Inprod,s*SZDBL,SZDBL);
    }
    bsp_sync();
    alpha= 0.0;
    for (t=0; t<p; t++){
        alpha += Inprod[t];
    }
    bsp_pop_reg(Inprod); vecfreed(Inprod);

    return alpha;

} /* end bspip */

void bspinprod(){

    double bspip(int p, int s, int n, double *x, double *y);
    int nloc(int p, int s, int n);
    double *x, alpha, time0, time1;
    int p, s, n, nl, i, iglob;

    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    s= bsp_pid();    /* s = processor number */
    if (s==0){
        printf("Please enter n:\n"); fflush(stdout);
        scanf("%d",&n);
        if(n<0)
            bsp_abort("Error in input: n is negative");
    }
    bsp_push_reg(&n,SZINT);
    bsp_sync();

    bsp_get(0,&n,0,&n,SZINT);
    bsp_sync();
    bsp_pop_reg(&n);

    nl= nloc(p,s,n);
    x= vecallocd(nl);
    for (i=0; i<nl; i++){
        iglob= i*p+s;
        x[i]= iglob+1;
    }
    bsp_sync();
    time0=bsp_time();

    alpha= bspip(p,s,n,x,x);
    bsp_sync();
    time1=bsp_time();

    printf("Processor %d: sum of squares up to %d*%d is %.lf\n",
            s,n,n,alpha); fflush(stdout);
    if (s==0){
        printf("This took only %.6lf seconds.\n", time1-time0);
        fflush(stdout);
    }
    vecfreed(x);
    bsp_end();

} /* end bspinprod */

int main(int argc, char **argv){

    bsp_init(bspinprod, argc, argv);

    /* sequential part */
    printf("How many processors do you want to use?\n");
    fflush(stdout);
    scanf("%d",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, not enough processors available.\n");
        fflush(stdout);
        exit(1);
    }

    /* SPMD part */
    bspinprod();

    /* sequential part */
    exit(0);

} /* end main */
```

1.5 BSP benchmarking

Computer benchmarking is the activity of measuring computer performance by running a representative set of test programs. The performance results for a particular sequential computer are often reduced in some ruthless way to one number, the computing rate in flop/s. This allows us to rank different computers according to their computing rate and to make informed decisions on what machines to buy or use. The performance of parallel computers must be expressed in more than a single number because communication and synchronization are just as important for these computers as computation. The BSP model represents machine performance by a parameter set of minimal size: for a given number of processors p, the parameters r, g, and l represent the performance for computation, communication, and synchronization. Every parallel computer can be viewed as a BSP computer, with good or bad BSP parameters, and hence can also be benchmarked as a BSP computer. In this section, we present a method for BSP benchmarking. The aim of the method is to find out what the BSP computer looks like to an average user, perhaps you or me, who writes parallel programs but does not really want to spend much time optimizing programs, preferring instead to let the compiler and the BSP system do the job. (The benchmark method for optimization enthusiasts would be very different.)

The sequential computing rate r is determined by measuring the time of a so-called DAXPY operation ('Double precision A times X Plus Y'), which has the form $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. A DAXPY with vectors of length n contains n additions and n multiplications and some overhead in the form of $\mathcal{O}(n)$ address calculations. We also measure the time of a DAXPY operation with the addition replaced by subtraction. We use 64-bit arithmetic throughout; on most machines this is called **double-precision arithmetic**. This mixture of operations is representative for the majority of scientific computations. We measure the time for a vector length, which on the one hand is large enough so that we can ignore the startup costs of vector operations, but on the other hand is small enough for the vectors to fit in the cache; a choice of $n = 1024$ is often adequate. A **cache** is a small but fast intermediate memory that allows immediate reuse of recently accessed data. Proper use of the cache considerably increases the computing rate on most modern computers. The existence of a cache makes the life of a benchmarker harder, because it leads to two different computing rates: a flop rate for in-cache computations and a rate for out-of-cache computations. Intelligent choices should be made if the performance results are to be reduced to a single meaningful figure.

The DAXPY measurement is repeated a number of times, both to obtain a more accurate clock reading and to amortize the cost of bringing the vector into the cache. We measure the sequential computing rate of each processor of the parallel computer, and report the minimum, average, and maximum rate. The difference between the minimum and the maximum indicates the accuracy of the measurement, except when the processors genuinely differ in speed. (One processor that is slower than the others can have a remarkable effect on the overall time of a parallel computation!) We take the average computing rate of the processors as the final value of r. Note that our measurement is representative for user programs that contain mostly hand-coded vector operations. To realize top performance, system-provided matrix–matrix operations should be used wherever possible, because these are often efficiently coded in assembler language. Our benchmark method does not reflect that situation.

The communication parameter g and the synchronization parameter l are obtained by measuring the time of full h-relations, where each processor sends and receives exactly h data words. To be consistent with the measurement of r, we use double-precision reals as data words. We choose a particularly demanding test pattern from the many possible patterns with the same h, which reflects the typical way most users would handle communication in their programs. The destination processors of the values to be sent

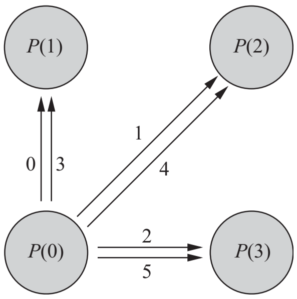


Fig. 1.10. Communication pattern of the 6-relation in the BSP benchmark. Processors send data to the other processors in a cyclic manner. Only the data sent by processor $P(0)$ are shown; other processors send data in a similar way. Each arrow represents the communication of one data word; the number shown is the index of the data word.

are determined in a cyclic fashion: $P(s)$ puts h values in remote processors $P(s+1), P(s+2), \ldots, P(p-1), P(0), \ldots, P(s-1), P(s+1), \ldots$, wrapping around at processor number p and skipping the source processor to exclude local puts that do not require communication, see Fig. 1.10. In this communication pattern, all processors receive the same number, h, of data words (this can easily be proven by using a symmetry argument). The destination processor of each communicated value is computed before the actual h-relation is performed, to prevent possibly expensive modulo operations from influencing the timing results of the h-relation. The data are sent out as separate words, to simulate the typical situation in an application program where the user does not worry about the size of the data packets. This is the task of the BSP system, after all! Note that this way of benchmarking h-relations is a test of both the machine and the BSP library for that machine. Library software can combine several data words into one packet, if they are sent in one superstep from the same source processor to the same destination processor. An efficient library will package such data automatically and choose an optimal packet size for the particular machine used. This results in a lower value of g, because the communication startup cost of a packet is amortized over several words of data.

Our variant of the BSP model assumes that the time $T_{\text{comm}}(h)$ of an h-relation is linear in h, see (1.2). In principle, it would be possible to measure $T_{\text{comm}}(h)$ for two values of h and then determine g and l. Of course, the results would then be highly sensitive to measurement error. A better way of doing this is by measuring $T_{\text{comm}}(h)$ for a range of values h_0–h_1, and then finding the best least-squares approximation, given by the values of g and l that minimize the error

$$E_{\mathrm{LSQ}} = \sum_{h=h_0}^{h_1} (T_{\mathrm{comm}}(h) - (hg + l))^2. \tag{1.9}$$

(These values are obtained by setting the partial derivatives with respect to g and l to zero, and solving the resulting 2×2 linear system of equations.) We choose $h_0 = p$, because packet optimization becomes worthwhile for $h \geq p$; we would like to capture the behaviour of the machine and the BSP system for such values of h. A value of $h_1 = 256$ will often be adequate, except if $p \geq 256$ or if the asymptotic communication speed is attained only for very large h.

Timing parallel programs requires caution since ultimately it often relies on a system timer, which may be hidden from the user and may have a low resolution. Always take a critical look at your timing results and your personal watch and, in case of suspicion, plot the output data in a graph! This may save you from potential embarrassment: on one occasion, I was surprised to find that according to an erroneous timer the computer had exceeded its true performance by a factor of four. On another occasion, I found that g was negative. The reason was that the particular computer used had a small g but a huge l, so that for $h \leq h_1$ the measurement error in $gh + l$ was much larger than gh, thereby rendering the value of g meaningless. In this case, h_1 had to be increased to obtain an accurate measurement of g.

1.6 Example program `bspbench`

The program `bspbench` is a simple benchmarking program that measures the BSP parameters of a particular computer. It is an implementation of the benchmarking method described in Section 1.5. In the following, we present and explain the program. The least-squares function of `bspbench` solves a 2×2 linear system by subtracting a multiple of one equation from the other. Dividing by zero or by a small number is avoided by subtracting the equation with the largest leading coefficient. (Solving a 2×2 linear system is a prelude to Chapter 2, where large linear systems are solved by LU decomposition.)

The computing rate r of each processor is measured by using the `bsp_time` function, which gives the elapsed time in seconds for the processor that calls it. The measurements of r are independent, and hence they do not require timer synchronization. The number of iterations `NITERS` is set such that each DAXPY pair (and each h-relation) is executed 100 times. You may decrease the number if you run out of patience while waiting for the results.

The h-relation of our benchmarking method is implemented as follows. The data to be sent are put into the array `dest` of the destination processors. The destination processor `destproc`[`i`] is determined by starting with the next processor $s+1$ and allocating the indices i to the $p-1$ remote processors in a cyclic fashion, that is, by adding i mod $(p-1)$ to the processor number $s+1$. Taking the resulting value modulo p then gives a valid processor number unequal to s. The destination index destindex[i] is chosen such that each source processor fills its own part in the dest arrays on the other processors: $P(s)$ fills locations $s, s+p, s+2p$, and so on. The locations are defined by destindex[i] $= s + ($i div $(p-1))p$, because we return to the same processor after each round of $p-1$ puts into different destination processors. The largest destination index used in a processor is at most $p-1+((h-1)$ div $(p-1))p < p + 2 \cdot$ MAXH, which guarantees that the array dest is sufficiently large.

```
#include "bspedupack.h"

/*  This program measures p, r, g, and l of a BSP computer
    using bsp_put for communication.
*/

#define NITERS 100     /* number of iterations */
#define MAXN 1024      /* maximum length of DAXPY computation */
#define MAXH 256       /* maximum h in h-relation */
#define MEGA 1000000.0

int P; /* number of processors requested */

void leastsquares(int h0, int h1, double *t, double *g, double *l){
    /* This function computes the parameters g and l of the
       linear function T(h)= g*h+l that best fits
       the data points (h,t[h]) with h0 <= h <= h1. */

    double nh, sumt, sumth, sumh, sumhh, a;
    int h;

    nh= h1-h0+1;
    /* Compute sums:
        sumt  =  sum of t[h] over h0 <= h <= h1
        sumth =         t[h]*h
        sumh  =         h
        sumhh =         h*h     */
    sumt= sumth= 0.0;
    for (h=h0; h<=h1; h++){
        sumt  += t[h];
        sumth += t[h]*h;
    }
    sumh= (h1*h1-h0*h0+h1+h0)/2;
    sumhh= ( h1*(h1+1)*(2*h1+1) - (h0-1)*h0*(2*h0-1))/6;

    /* Solve      nh*l +  sumh*g =  sumt
                sumh*l + sumhh*g = sumth */
    if(fabs(nh)>fabs(sumh)){
        a= sumh/nh;
        /* subtract a times first eqn from second eqn */
        *g= (sumth-a*sumt)/(sumhh-a*sumh);
        *l= (sumt-sumh* *g)/nh;
    } else {
        a= nh/sumh;
        /* subtract a times second eqn from first eqn */
        *g= (sumt-a*sumth)/(sumh-a*sumhh);
        *l= (sumth-sumhh* *g)/sumh;
    }

} /* end leastsquares */

void bspbench(){
    void leastsquares(int h0, int h1, double *t, double *g, double *l);
    int p, s, s1, iter, i, n, h, destproc[MAXH], destindex[MAXH];
    double alpha, beta, x[MAXN], y[MAXN], z[MAXN], src[MAXH], *dest,
           time0, time1, time, *Time, mintime, maxtime,
           nflops, r, g0, l0, g, l, t[MAXH+1];

    /**** Determine p ****/
    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    s= bsp_pid();    /* s = processor number */

    Time= vecallocd(p); bsp_push_reg(Time,p*SZDBL);
    dest= vecallocd(2*MAXH+p); bsp_push_reg(dest,(2*MAXH+p)*SZDBL);
    bsp_sync();

    /**** Determine r ****/
    for (n=1; n <= MAXN; n *= 2){
        /* Initialize scalars and vectors */
        alpha= 1.0/3.0;
        beta= 4.0/9.0;
        for (i=0; i<n; i++){
            z[i]= y[i]= x[i]= (double)i;
        }
        /* Measure time of 2*NITERS DAXPY operations of length n */
        time0=bsp_time();
        for (iter=0; iter<NITERS; iter++){
            for (i=0; i<n; i++)
                y[i] += alpha*x[i];
            for (i=0; i<n; i++)
                z[i] -= beta*x[i];
        }
        time1= bsp_time();
        time= time1-time0;
        bsp_put(0,&time,Time,s*SZDBL,SZDBL);
        bsp_sync();

        /* Processor 0 determines minimum, maximum, average
```

```
            computing rate */
        if (s==0){
            mintime= maxtime= Time[0];
            for(s1=1; s1<p; s1++){
                mintime= MIN(mintime,Time[s1]);
                maxtime= MAX(maxtime,Time[s1]);
            }
            if (mintime>0.0){
                /* Compute r = average computing rate in flop/s */
                nflops= 4*NITERS*n;
                r= 0.0;
                for(s1=0; s1<p; s1++)
                    r += nflops/Time[s1];
                r /= p;
                printf("n= %5d min= %7.3lf max= %7.3lf av= %7.3lf Mflop/s ",
                       n, nflops/(maxtime*MEGA),nflops/
                        (mintime*MEGA), r/MEGA);
                fflush(stdout);
                /* Output for fooling benchmark-detecting compilers */
                printf(" fool=%7.1lf\n",y[n-1]+z[n-1]);
            } else
                printf("minimum time is 0\n"); fflush(stdout);
        }
    }

    /**** Determine g and l ****/
    for (h=0; h<=MAXH; h++){
        /* Initialize communication pattern */
        for (i=0; i<h; i++){
            src[i]= (double)i;
            if (p==1){
                destproc[i]=0;
                destindex[i]=i;
            } else {
                /* destination processor is one of the p-1 others */
                destproc[i]= (s+1 + i%(p-1)) %p;
                /* destination index is in my own part of dest */
                destindex[i]= s + (i/(p-1))*p;
            }
        }

        /* Measure time of NITERS h-relations */
        bsp_sync();
        time0= bsp_time();
        for (iter=0; iter<NITERS; iter++){
            for (i=0; i<h; i++)
                bsp_put(destproc[i],&src[i],dest,destindex[i]*SZDBL,
                        SZDBL);
            bsp_sync();
        }
        time1= bsp_time();
        time= time1-time0;
        /* Compute time of one h-relation */
        if (s==0){
            t[h]= (time*r)/NITERS;
            printf("Time of %5d-relation= %lf sec= %8.0lf flops\n",
                   h, time/NITERS, t[h]); fflush(stdout);
        }
    }

    if (s==0){
        printf("size of double = %d bytes\n",(int)SZDBL);
        leastsquares(0,p,t,&g0,&l0);
        printf("Range h=0 to p   : g= %.1lf, l= %.1lf\n",g0,l0);
        leastsquares(p,MAXH,t,&g,&l);
        printf("Range h=p to HMAX: g= %.1lf, l= %.1lf\n",g,l);
        printf("The bottom line for this BSP computer is:\n");
        printf("p= %d, r= %.3lf Mflop/s, g= %.1lf, l= %.1lf\n",
               p,r/MEGA,g,l);
        fflush(stdout);
    }
    bsp_pop_reg(dest); vecfreed(dest);
    bsp_pop_reg(Time); vecfreed(Time);

    bsp_end();
} /* end bspbench */

int main(int argc, char **argv){

    bsp_init(bspbench, argc, argv);
    printf("How many processors do you want to use?\n");
    fflush(stdout);
    scanf("%d",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, not enough processors available.\n");
        exit(1);
    }
    bspbench();
    exit(0);

} /* end main */
```

1.7 Benchmark results

What is the cheapest parallel computer you can buy? Two personal computers connected by a cable. What performance does this configuration give you? Certainly $p = 2$, and perhaps $r = 122$ Mflop/s, $g = 1180$, and $l = 138\,324$. These BSP parameters were obtained by running the program `bspbench` on two PCs from a cluster that fills up a cabinet at the Oxford office of Sychron. This cluster consists of 11 identical Pentium-II PCs with a clock rate of 400 MHz running the Linux operating system, and a connection network of four Fast

Ethernets. Each Ethernet of this cluster consists of a Cisco Catalyst switch with 11 cables, each ending in a Network Interface Card (NIC) at the back of a different PC. (A **switch** connects pairs of PCs; pairs can communicate independently from other pairs.) Thus, each Ethernet connects all PCs. Having four Ethernets increases the communication capacity fourfold and it gives each PC four possible routes to every other PC, which is useful if some Ethernets are tied down by other communicating PCs. A Fast Ethernet is capable of transferring data at the rate of 100 Mbit/s (i.e. 12.5 Mbyte/s or 1 562 500 double-precision reals per second—ignoring overheads). The system software running on this cluster is the Sychron Virtual Private Server, which guarantees a certain specified computation/communication performance on part or all of the cluster, irrespective of use by others. On top of this system, the **portability layers** MPI and BSPlib are available. In our experiments on this machine, we used version 1.4 of BSPlib with communication optimization level 2 and the GNU C compiler with computation optimization level 3 (our compiler flags were `-flibrary-level 2 -O3`).

Figure 1.11 shows how you can build a parallel computer from cheap commodity components such as PCs running Linux, cables, and simple switches. This figure gives us a look inside the black box of the basic BSP architecture shown in Fig. 1.1. A parallel computer built in DIY (do-it-yourself) fashion is often called a Beowulf, after the hero warrior of the Old English epic, presumably in admiration of Beowulf's success in slaying the supers of his era. The Sychron cluster is an example of a small Beowulf; larger ones of hundreds of PCs are now being built by cost-conscious user groups in industry and in academia, see [168] for a how-to guide.

Figure 1.12 shows the time of an h-relation on two PCs of the Sychron Beowulf. The benchmark was run with `MAXN` decreased from the default

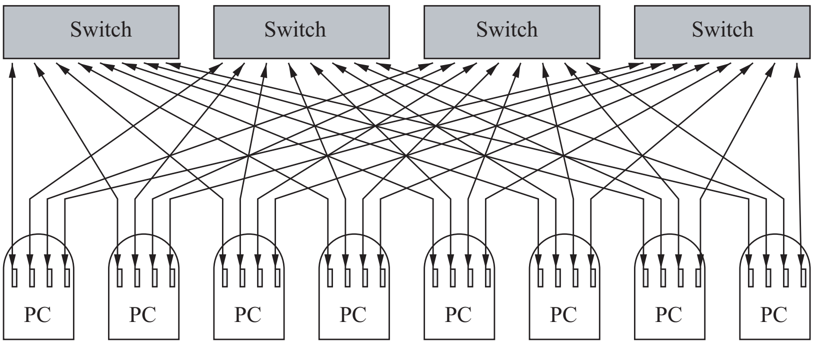


Fig. 1.11. Beowulf cluster of eight PCs connected by four switches. Each PC is connected to all switches.

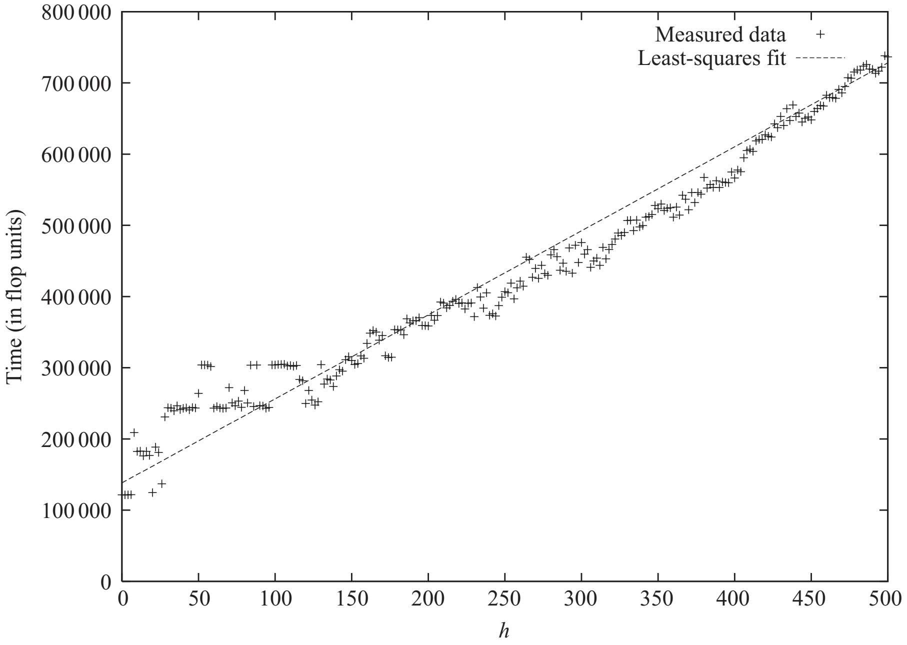


FIG. 1.12. Time of an h-relation on two connected PCs. The values shown are for even h with $h \leq 500$.

1024 to 512, to make the vectors fit in primary cache (i.e. the fastest cache); for length 1024 and above the computing rate decreases sharply. The value of `MAXH` was increased from the default 256 to 800, because in this case $g \ll l$ and hence a larger range of h-values is needed to obtain an accurate value for g from measurements of $hg + l$. Finding the right input parameters `MAXN`, `MAXH`, and `NITERS` for the benchmark program may require trial and error; plotting the data is helpful in this process. It is unlikely that one set of default parameters will yield sensible measurements for every parallel computer.

What is the most expensive parallel computer you can buy? A supercomputer, by definition. Commonly, a **supercomputer** is defined as one of today's top performers in terms of computing rate, communication/synchronization rate, and memory size. Most likely, the cost of a supercomputer will exceed a million US dollars. An example of a supercomputer is the Cray T3E, which is a massively parallel computer with distributed memory and a communication network in the form of a three-dimensional **torus** (i.e. a mesh with wraparound links at the boundaries). We have benchmarked up to 64 processors of the 128-processor machine called Vermeer, after the famous Dutch painter, which is located at the HPαC supercomputer centre of Delft University of Technology. Each node of this machine consists of a DEC Alpha 21164 processor with a clock speed of 300 MHz, an advertised peak performance of 600 Mflop/s, and 128 Mbyte memory. In our experiments, we used version 1.4 of BSPlib with optimization level 2 and the Cray C compiler with optimization level 3.

The measured single-processor computing rate is 35 Mflop/s, which is much lower than the theoretical peak speed of 600 Mflop/s. The main reason for this discrepancy is that we measure the speed for a DAXPY operation written in C, whereas the highest performance on this machine can only be obtained by performing matrix–matrix operations such as DGEMM (Double precision GEneral Matrix–Matrix multiplication) and then only when using well-tuned subroutines written in assembler language. The BLAS (Basic Linear Algebra Subprograms) library [59,60,126] provides a portable interface to a set of subroutines for the most common vector and matrix operations, such as DAXPY and DGEMM. (The terms 'DAXPY' and 'DGEMM' originate in the BLAS definition.) Efficient BLAS implementations exist for most machines. A complete BLAS list is given in [61,Appendix C]. A Cray T3E version of the BLAS is available; its DGEMM approaches peak performance. A note of caution: our initial, erroneous result on the Cray T3E was a computing rate of 140 Mflop/s. This turned out to be due to the Cray timer IRTC, which is called by BSPlib on the Cray and ran four times slower than it should. This error occurs only in version 1.4 of BSPlib, in programs compiled at BSPlib level 2 for the Cray T3E.

Figure 1.13 shows the time of an h-relation on 64 processors of the Cray T3E. The time grows more or less linearly, but there are some odd jumps, for instance the sudden significant decrease around $h = 130$. (Sending more data takes less time!) It is beyond our scope to explain every peculiarity of every benchmarked machine. Therefore, we feel free to leave some surprises, like this one, unexplained.

Table 1.2 shows the BSP parameters obtained by benchmarking the Cray T3E for up to 64 processors. The results for $p = 1$ give an indication of the overhead of running a bulk synchronous parallel program on one processor. For the special case $p = 1$, the value of `MAXH` was decreased to 16, because $l \approx g$ and hence a smaller range of h-values is needed to obtain an accurate value for l from measurements of $hg + l$. (Otherwise, even negative values of l could appear.) The table shows that g stays almost constant for $p \leq 16$ and that it grows slowly afterwards. Furthermore, l grows roughly linearly with p, but occasionally it behaves strangely: l suddenly decreases on moving from 16 to 32 processors. The explanation is hidden inside the black box of the communication network. A possible explanation is the increased use of wraparound links when increasing the number of processors. (For a small number of processors, all boundary links of a subpartition connect to other subpartitions, instead of wrapping around to the subpartition itself; thus, the subpartition

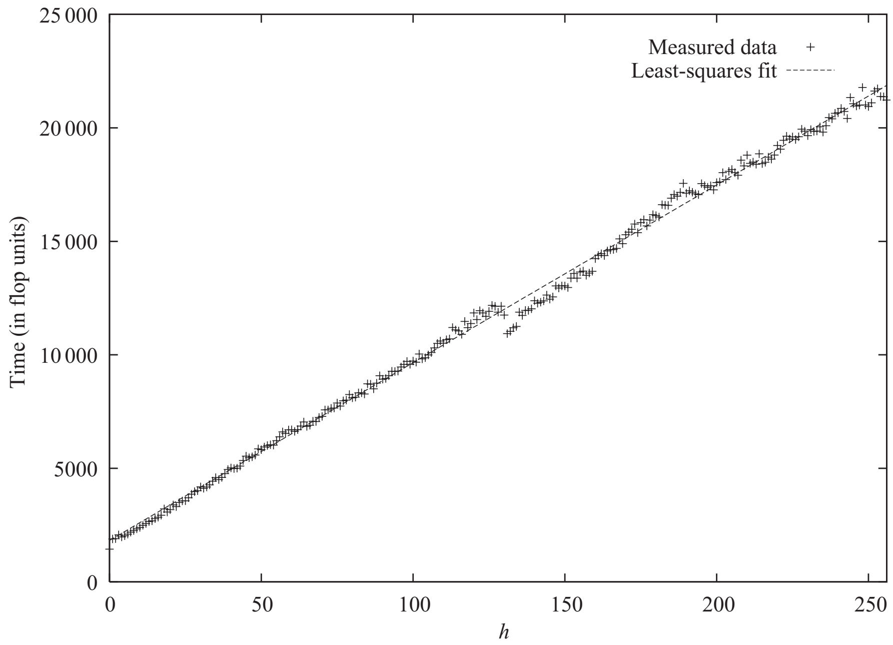


FIG. 1.13. Time of an h-relation on a 64-processor Cray T3E.

TABLE 1.2. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Cray T3E. All times are in flop units ($r = 35$ Mflop/s)

p	g	l	$T_{\text{comm}}(0)$
1	36	47	38
2	28	486	325
4	31	679	437
8	31	1193	580
16	31	2018	757
32	72	1145	871
64	78	1825	1440

is a mesh, rather than a torus. Increasing the number of processors makes the subpartition look more like a torus, with richer connectivity.) The time of a 0-relation (i.e. the time of a superstep without communication) displays a smoother behaviour than that of l, and it is presented here for comparison. This time is a lower bound on l, since it represents only part of the fixed cost of a superstep.

Another prominent supercomputer, at the time of writing these lines of course since machines come and go quickly, is the IBM RS/6000 SP. This machine evolved from the RS/6000 workstation and it can be viewed as a tightly coupled cluster of workstations (without the peripherals, of course). We benchmarked eight processors of the 76-processor SP at the SARA supercomputer centre in Amsterdam. The subpartition we used contains eight so-called thin nodes connected by a switch. Each node contains a 160 MHz PowerPC processor, 512 Mbyte of memory, and a local disk of 4.4 Gbyte. The theoretical peak rate of a processor is about 620 Mflop/s, similar to the Cray T3E above. Figure 1.14 shows the time of an h-relation on $p = 8$ processors of the SP. Note the effect of optimization by BSPlib: for $h < p$, the time of an h-relation increases rapidly with h. For $h = p$, however, BSPlib detects that the pth message of each processor is sent to the same destination as the first message, so that it can combine the messages. Every additional message can also be combined with a previous one. As a result, the number of messages does not increase further, and g grows only slowly. Also note the statistical outliers, that is, those values that differ considerably from the others. All outliers are high, indicating interference by message traffic from other users. Although the system guarantees exclusive access to the processors used, it does not guarantee the same for the communication network. This makes communication

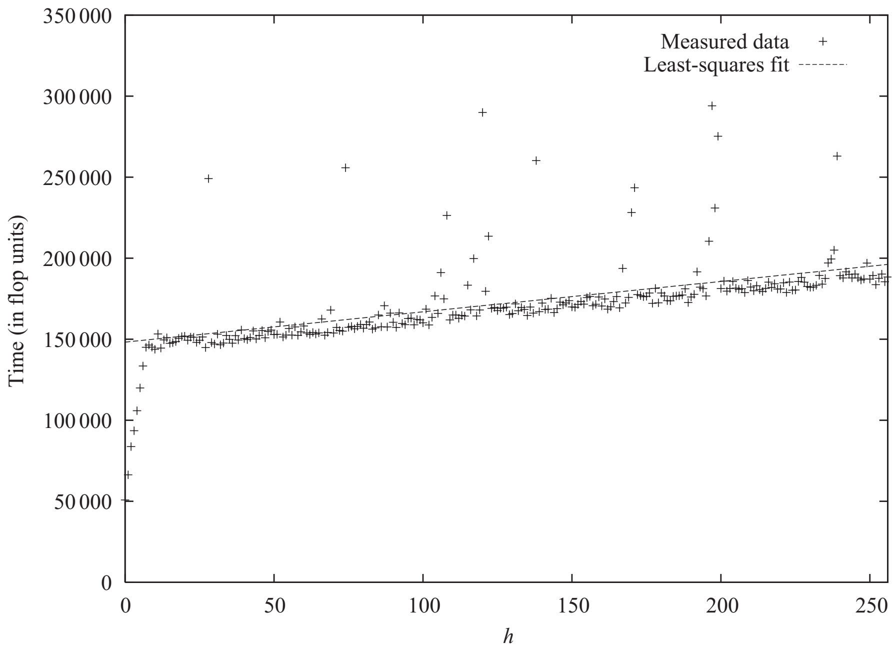


FIG. 1.14. Time of an h-relation on an 8-processor IBM SP.

benchmarking difficult. More reliable results can be obtained by repeating the benchmark experiment, or by organizing a party for one's colleagues (and sneaking out in the middle), in the hope of encountering a traffic-free period. A plot will reveal whether this has happened. The resulting BSP parameters of our experiment are $p = 8$, $r = 212$ Mflop/s, $g = 187$, and $l = 148\ 212$. Note that the interference from other traffic hardly influences g, because the slope of the fitted line is the same as that of the lower string of data (which can be presumed to be interference-free). The value of l, however, is slightly overestimated.

The last machine we benchmark is the Silicon Graphics Origin 2000. This architecture has a physically distributed memory, like the other three benchmarked computers. In principle, this promises scalability because memory, communication links, and other resources can grow proportionally with the number of processors. For ease of use, however, the memory can also be made to look like shared memory. Therefore, the Origin 2000 is often promoted as a 'Scalable Shared Memory Multiprocessor'. Following BSP doctrine, we ignore the shared-memory facility and use the Origin 2000 as a distributed-memory machine, thereby retaining the advantage of portability.

The Origin 2000 that we could lay our hands on is Oscar, a fine 86-processor parallel computer at the Oxford Supercomputer Centre of Oxford University. Each processing element of the Origin 2000 is a MIPS R10000 processor with a clock speed of 195 MHz and a theoretical peak performance of 390 Mflop/s. We compiled our programs using the SGI MIPSpro C-compiler with optimization flags switched on as before. We used eight processors in our benchmark. The systems managers were kind enough to empty the system from other jobs, and hence we could benchmark a dedicated system. The resulting BSP parameters are $p = 8$, $r = 326$ Mflop/s, $g = 297$, and $l = 95\ 686$. Figure 1.15 shows the results.

Table 1.3 presents benchmark results for three different computers with the number of processors fixed at eight. The parameters g and l are given not only in flops but also in raw microseconds to make it easy to compare machines with widely differing single-processor performance. The Cray T3E is the best balanced machine: the low values of g and l in flops mean that communication/synchronization performance of the Cray T3E is excellent, relative to the computing performance. The low values of l in microseconds tell us that in absolute terms synchronization on the Cray is still cheap. The main drawback of the Cray T3E is that it forces the user to put effort into optimizing programs, since straightforward implementations such as our benchmark do not attain top performance. For an unoptimized program, eight processors of the T3E are slower than a single processor of the Origin 2000. The SP and the Origin 2000 are similar as BSP machines, with the Origin somewhat faster in computation and synchronization.

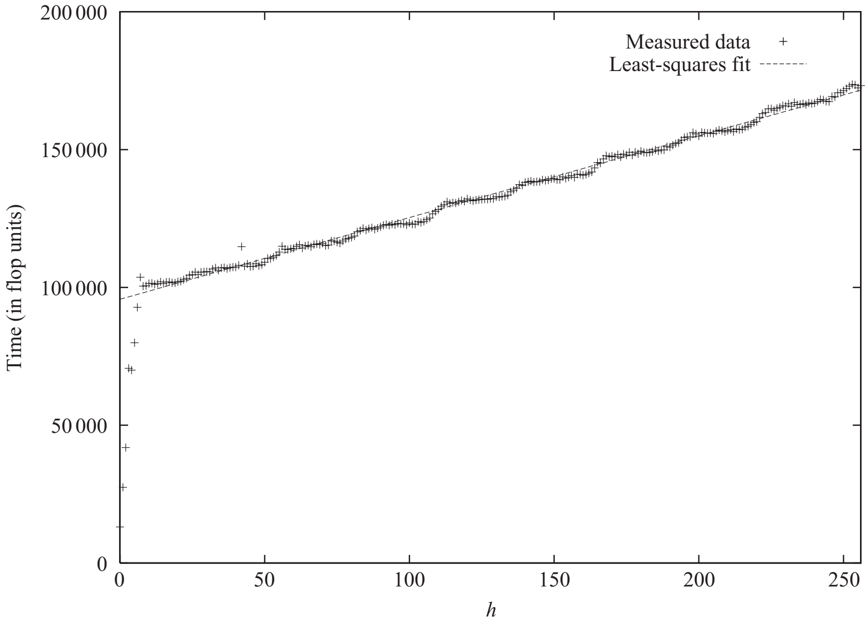


FIG. 1.15. Time of an h-relation on an 8-processor SGI Origin.

TABLE 1.3. Comparing the BSP parameters for three different parallel computers with $p = 8$

Computer	r (Mflop/s)	(flop)		(μs)	
		g	l	g	l
Cray T3E	35	31	1 193	0.88	34
IBM RS/6000 SP	212	187	148 212	0.88	698
SGI Origin 2000	326	297	95 686	0.91	294

1.8 Bibliographic notes

1.8.1 *BSP-related models of parallel computation*

Historically, the Parallel Random Access Machine (PRAM) has been the most widely studied general-purpose model of parallel computation. In this model, processors can read from and write to a shared memory. Several variants of the PRAM model can be distinguished, based on the way concurrent memory access is treated: the concurrent read, concurrent write (CRCW) variant allows full concurrent access, whereas the exclusive read, exclusive write (EREW) variant allows only one processor to access the memory at a time. The PRAM model ignores communication costs and is therefore mostly of theoretical interest; it is useful in establishing lower bounds for the cost of parallel algorithms. The PRAM model has stimulated the development of many other models, including the BSP model. The BSP variant with automatic memory management by randomization in fact reduces to the PRAM model in the asymptotic case $g = l = \mathcal{O}(1)$. For an introduction to the PRAM model, see the survey by Vishkin [189]. For PRAM algorithms, see the survey by Spirakis and Gibbons [166,167] and the book by JáJá [113].

The BSP model has been proposed by Valiant in 1989 [177]. The full description of this 'bridging model for parallel computation' is given in [178]. This article describes the two basic variants of the model (automatic memory management or direct user control) and it gives a complexity analysis of algorithms for fast Fourier transform, matrix–matrix multiplication, and sorting. In another article [179], Valiant proves that a hypercube or butterfly architecture can simulate a BSP computer with optimal efficiency. (Here, the model is called XPRAM.) The BSP model as it is commonly used today has been shaped by various authors since the original work by Valiant. The survey by McColl [132] argues that the BSP model is a promising approach to general-purpose parallel computing and that it can deliver both scalable performance and architecture independence. Bisseling and McColl [21,22] propose the variant of the model (with pure computation supersteps of cost $w + l$ and pure communication supersteps of cost $hg + l$) that is used in this book. They show how a variety of scientific computations can be analysed in a simple manner by using their BSP variant. McColl [133] analyses and classifies several important BSP algorithms, including dense and sparse matrix–vector multiplication, matrix–matrix multiplication, LU decomposition, and triangular system solution.

The LogP model by Culler *et al.* [49] is an offspring of the BSP model, which uses four parameters to describe relative machine performance: the latency L, the overhead o, the gap g, and the number of processors P, instead of the three parameters l, g, and p of the BSP model. The LogP model treats messages individually, not in bulk, and hence it does not provide the notion of a superstep. The LogP model attempts to reflect the actual machine architecture more closely than the BSP model, but the price to be paid is an increase in the complexity of algorithm design and analysis. Bilardi *et al.* [17] show that the LogP and BSP model can simulate each other efficiently so that in principle they are equally powerful.

The YPRAM model by de la Torre and Kruskal [54] characterizes a parallel computer by its latency, bandwidth inefficiency, and recursive decomposability. The decomposable BSP (D-BSP) model [55] is the same model expressed in BSP terms. In this model, a parallel computer can be decomposed into submachines, each with their own parameters g and l. The parameters g and l of submachines will in general be lower than those of the complete machine. The scaling behaviour of the submachines with p is described by functions $g(p)$ and $l(p)$. This work could provide a theoretical basis for subset synchronization within the BSP framework.

The BSPRAM model by Tiskin [174] replaces the communication network by a shared memory. At the start of a superstep, processors read data from the shared memory into their own local memory; then, they compute independently using locally held data; and finally they write local data into the shared memory. Access to the shared memory is in bulk fashion, and the cost function of such access is expressed in g and l. The main aim of the BSPRAM model is to allow programming in shared-memory style while keeping the benefits of data locality.

1.8.2 *BSP libraries*

The first portable BSP library was the Oxford BSP library by Miller and Reed [140,141]. This library contains six primitives: put, get, start of superstep, end of superstep, start of program, and end of program. The Cray SHMEM library [12] can be considered as a nonportable BSP library. It contains among others: put, strided put, get, strided get, and synchronization. The Oxford BSP library is similar to the Cray SHMEM library, but it is available for many different architectures. Neither of these libraries allows communication into dynamically allocated memory. The Green BSP library by Goudreau *et al.* [80,81] is a small experimental BSP library of seven primitives. The main difference with the Oxford BSP library is that the Green BSP library communicates by bulk synchronous message passing, which will be explained in detail in Chapter 4. The data sent in a superstep is written into a remote receive-buffer. This is one-sided communication, because the receiver remains passive when the data is being communicated. In the next superstep, however, the receiver becomes active: it must retrieve the desired messages from its receive-buffer, or else they will be lost forever. Goudreau *et al.* [80] present results of numerical experiments using the Green BSP library in ocean eddy simulation, computation of minimum spanning trees and shortest paths in graphs, n-body simulation, and matrix–matrix multiplication.

Several communication libraries and language extensions exist that enable programming in BSP style but do not fly the BSP flag. The Split-C language by Culler *et al.* [48] is a parallel extension of C. It provides put and get primitives and additional features such as global pointers and spread arrays. The Global Array toolkit [146] is a software package that allows the creation and destruction of distributed matrices. It was developed in first instance for use in computational chemistry. The Global Array toolkit and the underlying programming model include features such as one-sided communication, global synchronization, relatively cheap access to local memory, and uniformly more expensive access to remote memory. Co-Array Fortran [147], formerly called F^{--}, is a parallel extension of Fortran 95. It represents a strict version of an SPMD approach: all program variables exist on all processors; remote variables can be accessed by appending the processor number in square brackets, for example, `x(3)[2]` is the variable `x(3)` on $P(2)$. A put is concisely formulated by using such brackets in the left-hand side of an assignment, for example, `x(3)[2]=y(3)`, and a get by using them on the right-hand side. Processors can be synchronized in subsets or even in pairs. The programmer needs to be aware of the cost implications of the various types of assignments.

BSPlib, used in this book, combines the capabilities of the Oxford BSP and the Green BSP libraries. It has grown into a *de facto* standard, which is fully defined by Hill *et al.* [105]. These authors also present results for fast Fourier transformation, randomized sample sorting, and n-body simulation using BSPlib. Frequently asked questions about BSP and BSPlib, such as 'Aren't barrier synchronizations expensive?' are answered by Skillicorn, Hill, and McColl [163].

Hill and Skillicorn [106] discuss how to implement BSPlib efficiently. They demonstrate that postponing communication is worthwhile, since this allows messages to be reordered (to avoid congestion) and combined (to reduce startup costs). If the natural ordering of the communication in a superstep requires every processor to put data first into $P(0)$, then into $P(1)$, and so on, this creates congestion at the destination processors, even if the total h-relation is well-balanced. To solve this problem, Hill and Skillicorn use a $p \times p$ Latin square, that is, a matrix with permutations of $\{0, \ldots, p-1\}$ in the rows and columns, as a schedule for the communication. An example is the 4×4 Latin square

$$R = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix}. \tag{1.10}$$

The communication of the superstep is done in p rounds. In round j, processor $P(i)$ sends all data destined for $P(r_{ij})$. In another article [107], Hill and Skillicorn discuss the practical issues of implementing a global synchronization on different machines. In particular, they show how to abuse the cache-coherence mechanism of a shared-memory parallel computer to obtain an extremely cheap global synchronization. (In one case, 13.4 times faster than the vendor-provided synchronization.) Donaldson, Hill, and Skillicorn [57] show how to implement BSPlib efficiently on top of the TCP/IP and UDP/IP protocols for a Network Of Workstations (NOW) connected by an Ethernet. They found that it is important to release data packets at an optimal rate into the Ethernet. Above this rate, data packets will collide too often, in which case they must be resent; this increases g. Below the optimal rate, the network capacity is underused. Hill, Donaldson, and Lanfear [102] present an implementation of BSPlib that continually migrates p processes around a NOW, taking care to run them on the least used workstations. After the work loads change, for example, because the owner returns from lunch, the next global synchronization provides a natural point to stop the computation, dump all data onto disk, and restart the computation on a less busy set of workstations. As an additional benefit, this provides fault tolerance, for instance the capability to recover from hardware failures.

The Oxford BSP toolset [103] contains an implementation of the C and Fortran 90 versions of the BSPlib standard. This library is used in most numerical experiments of this book. The toolset also contains profiling tools [101,104] that can be used to measure and visualize the amount of computation and the amount of incoming and outgoing data of each processor during a sequence of supersteps.

The Paderborn University BSP (PUB) library [28,30] is an implementation of the C version of the BSPlib standard with extensions. One extension of PUB is **zero-cost synchronization** [5], also called oblivious synchronization, which exploits the fact that for certain regular computations each receiving processor $P(s)$ knows the number of data words $h_{\mathrm{r}}(s)$ it will receive. Processor $P(s)$ performs a `bsp_oblsync(`$h_{\mathrm{r}}(s)$`)` operation at the end of the superstep, instead of a `bsp_sync`. This type of synchronization is cheap because no communication is needed to determine that processors can move on to the next superstep. Another extension is partitioning (by using the primitive `bsp_partition`), which is decomposing a BSP machine into submachines, each of them again a BSP machine. The submachines must be rejoined later (by using the primitive `bsp_done`). Each submachine can again be partitioned, in a recursive fashion. The processors of a submachine must be numbered consecutively. The partitioning mechanism provides a disciplined form of subset synchronization. The PUB library also includes a large set of collective-communication functions. Bonorden *et al.* [29] compare the performance on the Cray T3E of PUB with that of the Oxford BSP toolset and MPI.

The one-sided communications added by MPI-2 [138] to the orginal MPI standard can also be viewed as comprising a BSP library. The MPI-2 primitives for one-sided communications are put, get, and accumulate; their use is demonstrated in the fifth program, `mpimv`, of Appendix C. For the complete MPI-2 standard with annotations, see [83]. For a tutorial introduction, see the book by Gropp, Lusk, and Thakur [85].

1.8.3 *The non-BSP world: message passing*

In contrast to the one-sided communication of BSP programming, traditional message passing uses two-sided communication, which involves both an active sender and an active receiver. The underlying model is that of communicating sequential processes (CSP) by Hoare [108]. Several libraries support this type of communication. The first portable communication library, parallel virtual machine (PVM), was developed by Sunderam [171]. PVM is a message-passing library that enables computing on **heterogeneous networks**, that is, networks of computers with different architectures. PVM has evolved into a standard [75] and PVM implementations are available for many different parallel computers. The two strong points of PVM compared with other systems such as BSPlib and MPI-1 are its support of heterogeneous architectures and its capability of dynamic process creation. This means that processors can be added (or removed) during a computation. These features make PVM attractive for certain Beowulf clusters, in particular heterogeneous ones. Geist, Kohl, and Papadopoulus [76] compare PVM with MPI-1.

The message-passing interface (MPI) [137], based on traditional message passing, was defined in 1994 by the MPI Forum, a committee of users and manufacturers of parallel computers. The initial definition is now known as MPI-1. For the complete, most recent MPI-1 standard with annotations, see [164]. For a tutorial introduction, see the book by Gropp, Lusk, and Skjellum [84]. An introduction to parallel programming that uses MPI-1 is the textbook by Pacheco [152]. An introduction that uses PVM and MPI-1 for distributed-memory parallel programming and Pthreads for shared-memory programming is the textbook by Wilkinson and Allen [191]. An introduction to parallel computing that uses MPI-1 for distributed-memory parallel programming and Pthreads and OpenMP for shared-memory programming is the textbook by Grama *et al.* [82]. Today, MPI-1 is the most widely used portability layer for parallel computers.

1.8.4 *Benchmarking*

The BSPlib definition [105] presents results obtained by the optimized benchmarking program `bspprobe`, which is included in the Oxford BSP toolset [103]. The values of r and l that were measured by `bspprobe` agree well with those of `bspbench`. The values of g, however, are much lower: for instance, the value $g = 1.6$ for 32-bit words at $r = 47$ Mflop/s given in [105,Table 1] corresponds to $g = 0.07\,\mu$s for 64-bit words, which is 12.5 times less than the 0.88 μs measured by `bspbench`, see Table 1.3. This is due to the high optimization level of `bspprobe`: data are sent in blocks instead of single words and high-performance puts are used instead of buffered puts. The goal of `bspprobe` is to measure communication performance for optimized programs and hence its bottom line takes as g-value the asymptotic value for large blocks. The effect of such optimizations will be studied in Chapter 2. The program `bspprobe` measures g for two different h-relations with the same h: (i) a local cyclic shift, where every processor sends h data to the next higher-numbered processor; (ii) a global all-to-all procedure where every processor sends $h/(p-1)$ data to every one of the others. In most cases, the difference between the two g-values is small. This validates the basic assumption of the BSP model, namely that costs can be expressed in terms of h. The program `bspprobe` takes as l-value the cost of a synchronization in the absence of communication, that is, $T_{\mathrm{comm}}(0)$, see Table 1.2. BSP parameters obtained for the Green BSP library are given by Goudreau *et al.* [80].

Benchmark results for machines ranging from personal computers to massively parallel supercomputers are collected and regularly updated by Dongarra [58]. These results represent the total execution rates for solving a dense $n \times n$ linear system of equations with $n = 100$ and $n = 1000$ by the LINPACK software and with unlimited n by any suitable software. The slowest machine included is an HP48 GX pocket calculator which achieves 810 flop/s on the $n = 100$ benchmark. A PC based on the 3.06 GHz Intel Pentium-IV chip achieves a respectable 1.41 Gflop/s for $n = 100$ and 2.88 Gflop/s for $n = 1000$ (1 Gflop = 1 Gigaflop = 10^9 flop). Most interesting is the unrestricted benchmark, see [58,Table 3], which allows supercomputers to show off and demonstrate their capabilities. The table gives: r_{max}, the maximum rate achieved; r_{peak}, the theoretical peak rate; n_{max}, the size of the system at which r_{max} is achieved; and $n_{1/2}$, the size at which half of r_{max} is obtained. The $n_{1/2}$ parameter is widely used as a measure of startup overhead. Low $n_{1/2}$ values promise top rates already for moderate problem sizes, see *The Science of Computer Benchmarking* by Hockney [109]. The value of r_{max} is the basis for the Top 500 list of supercomputer sites, see `http://www.top500.org`.

To be called a supercomputer, at present a computer must achieve at least 1 Tflop/s (1 Tflop = 1 Teraflop = 10^{12} flop). The fastest existing number cruncher is the Earth Simulator, which was custom-built by NEC for the Japan Marine Science and Technology Center. This computer occupies its own building, has 5120 processors, and it solves a linear system of size $n =$ 1 075 200 at $r_{\mathrm{max}} = 36$ Tflop/s. Of course, computing rates increase quickly, and when you read these lines, the fastest computer may well have passed the Pflop/s mark (1 Pflop = 1 Petaflop = 10^{15} flop).

1.9 Exercises

1. Algorithm 1.1 can be modified to combine the partial sums into one global sum by a different method. Let $p = 2^q$, with $q \geq 0$. Modify the algorithm to combine the partial sums by repeated pairing of processors. Take care that every processor obtains the final result. Formulate the modified algorithm exactly, using the same notation as in the original algorithm. Compare the BSP cost of the two algorithms. For which ratio l/g is the pairwise algorithm faster?

2. Analyse the following operations and derive the BSP cost for a parallel algorithm. Let $\mathbf{x}$ be the input vector (of size n) of the operation and $\mathbf{y}$ the output vector. Assume that these vectors are block distributed over p processors, with $p \leq n$. Furthermore, k is an integer with $1 \leq k \leq n$. The operations are:

(a) Minimum finding: determine the index j of the component with the minimum value and subtract this value from every component: $y_i = x_i - x_j$, for all i.

(b) Shifting (to the right): assign $y_{(i+k) \bmod n} = x_i$.

(c) Smoothing: replace each component by a moving average $y_i = 1/(k+1) \sum_{j=i-k/2}^{i+k/2} x_j$, where k is even.

(d) Partial summing: compute $y_i = \sum_{j=0}^{i} x_j$, for all i. (This problem is an instance of the **parallel prefix** problem.)

(e) Sorting by counting: sort $\mathbf{x}$ by increasing value and place the result in $\mathbf{y}$. Each component x_i is an integer in the range 0–k, where $k \ll n$.

3. Get acquainted with your parallel computer before you use it.

(a) Run the program `bspbench` on your parallel computer. Measure the values of g and l for various numbers of processors. How does the performance of your machine scale with p?

(b) Modify `bspbench` to measure `bsp_gets` instead of `bsp_puts`. Run the modified program for various p. Compare the results with those of the original program.

4. Since their invention, computers have been used as tools in cryptanalytic attacks on secret messages; parallel computers are no exception. Assume a plain text has been encrypted by the classic method of monoalphabetic substitution, where each letter from the alphabet is replaced by another one and where blanks and punctuation characters are deleted. For such a simple encryption scheme, we can apply statistical methods to uncover the message. See Bauer [14] for more details and also for a fascinating history of cryptology.

(a) Let $\mathbf{t} = (t_0, \ldots, t_{n-1})^{\mathrm{T}}$ be a cryptotext of n letters and $\mathbf{t}'$ another cryptotext, of the same length, language, and encryption alphabet. With a bit of luck, we can determine the language of the texts by computing Friedman's Kappa value, also called the index of coincidence,

$$\kappa(\mathbf{t}, \mathbf{t}') = \frac{1}{n} \sum_{i=0}^{n-1} \delta(t_i, t_i').$$

Here, $\delta(x, y) = 1$ if $x = y$, and $\delta(x, y) = 0$ otherwise. The value of κ tells us how likely it is that two letters in the same position of the texts are identical. Write a parallel program that reads an encrypted text, splits it into two parts $\mathbf{t}$ and $\mathbf{t}'$ of equal size (dropping the last letter if necessary), and computes $\kappa(\mathbf{t}, \mathbf{t}')$. Motivate your choice of data distribution.

(b) Find a suitable cryptotext as input and compute its κ. Guess its language by comparing the result with the κ-values found by Kullback (reproduced in [14]): Russian 5.29%, English 6.61%, German 7.62%, French 7.78%.
(c) Find out whether Dutch is closer to English or German.
(d) Extend your program to compute all letter frequencies in the input text. In English, the 'e' is the most frequent letter; its frequency is about 12.5%.
(e) Run your program on some large plain texts in the language just determined to obtain a frequency profile of that language. Run your program on the cryptotext and establish its letter frequences. Now break the code.
(f) Is parallelization worthwhile in this case? When would it be?

5. (*) Data compression is widely used to reduce the size of data files, for instance texts or pictures to be transferred over the Internet. The LZ77 algorithm by Ziv and Lempel [193] passes through a text and uses the most recently accessed portion as a reference dictionary to shorten the text, replacing repeated character strings by pointers to their first occurrence. The popular compression programs PKZIP and gzip are based on LZ77.

Consider the text

'yabbadabbadoo'

(Fred Flintstone, Stone Age). Assume we arrive at the second occurrence of the string 'abbad'. By going back 5 characters, we find a matching string of length 5. We can code this as the triple of decimal numbers (5,5,111), where the first number in the triple is the number of characters we have to go back and the second number the length of the matching string. The number 111 is the ASCII code for the lower-case 'o', which is the next character after the second 'abbad'. (The lower-case characters 'a'–'z' are numbered 97–122 in the ASCII set.) Giving the next character ensures progress, even if no match was found. The output for this example is: (0,0,121), (0,0,97), (0,0,98), (1,1,97), (0,0,100), (5,5,111), (1,1,−1). The '−1' means end of input. If more matches are possible, the longest one is taken. For longer texts, the search for a match is limited to the last m characters before the current character (the search window); the string to be matched is limited to the first n characters starting at the current character (the look-ahead window).

(a) Write a sequential function that takes as input a character sequence and writes as output an LZ77 sequence of triples (o, l, c), where o is the offset, that is, number of characters to be moved back, l the length of the matching substring, and c is the code for the next character. Use suitable data types for o, l, and c to save space. Take $m = n = 512$.

Also write a sequential function that decompresses the LZ77 sequence. Which is fastest, compression or decompression?

(b) Design and implement a parallel LZ77 compression algorithm. You may adapt the original algorithm if needed for parallelization as long as the output can be read by the sequential LZ77 program. Hint: make sure that each processor has all input data it needs, before it starts compressing.

(c) Now design a parallel algorithm that produces exactly the same output sequence as the sequential algorithm. You may need several passes through the data.

(d) Compare the compression factor of your compression programs with that of `gzip`. How could you improve the performance?

(e) Is it worthwhile to parallelize the decompression?

6. (∗) A **random number generator** (RNG) produces a sequence of real numbers, in most cases uniformly distributed over the interval [0,1], that are uncorrelated and at least appear to be random. (In fact, the sequence is usually generated by a computer in a completely deterministic manner.) A simple and widely used type of RNG is the **linear congruential generator** based on the integer sequence defined by

$$x_{k+1} = (ax_k + b) \bmod m,$$

where a, b, and m are suitable constants. The starting value x_0 is called the **seed**. The choice of constants is critical for the quality of the generated sequence. The integers x_k, which are between 0 and $m-1$, are converted to real values $r_k \in [0, 1)$ by $r_k = x_k/m$.

(a) Express x_{k+p} in terms of x_k and some constants. Use this expression to design a parallel RNG, that is, an algorithm, which generates a different, hopefully uncorrelated, sequence for each of the p processors of a parallel computer. The local sequence of a processor should be a subsequence of the x_k.

(b) Implement your algorithm. Use the constants proposed by Lewis, Goodman, and Miller [129]: $a = 7^5$, $b = 0$, $m = 2^{31} - 1$. This is a simple multiplicative generator. Do not be tempted to use zero as a seed!

(c) For the statistically sophisticated. Design a statistical test to check the randomness of the local sequence of a processor, for example, based on the χ^2 test. Also design a statistical test to check the independence of the different local sequences, for example, for the case $p = 2$. Does the parallel RNG pass these tests?

(d) Use the sequential RNG to simulate a random walk on the two-dimensional integer lattice $\mathbf{Z}^2$, where the walker starts at (0,0), and at each step moves north, east, south, or west with equal probability 1/4. What is the expected distance to the origin after 100 steps? Create a large number of walks to obtain a good estimate. Use the parallel RNG to accelerate your simulation.

(e) Improve the quality of your parallel RNG by adding a local shuffle to break up short distance correlations. The numbers generated are written to a buffer array of length 64 instead of to the output. The buffer is filled at startup; after that, each time a random number is needed, one of the array values is selected at random, written to the output, and replaced by a new number x_k. The random selection of the buffer element is done based on the last output number. The shuffle is due to Bays and Durham [16]. Check whether this improves the quality of the RNG. Warning: the resulting RNG has limited applicability, because m is relatively small. Better parallel RNGs exist, see for instance the SPRNG package [131], and in serious work such RNGs must be used.

7. (∗) The sieve of Eratosthenes (276–194 BC) is a method for generating all prime numbers up to a certain bound n. It works as follows. Start with the integers from 2 to n. The number 2 is a prime; cross out all larger multiples of 2. The smallest remaining number, 3, is a prime; cross out all larger multiples of 3. The smallest remaining number, 5, is a prime, etc.

(a) When can we stop?

(b) Write a sequential sieve program. Represent the integers by a suitable array.

(c) Analyse the cost of the sequential algorithm. Hint: the probability of an arbitrary integer $x \geq 2$ to be prime is about $1/\log x$, where $\log = \log_e$ denotes the natural logarithm. Estimate the total number of cross-out operations and use some calculus to obtain a simple formula. Add operation counters to your program to check the accuracy of your formula.

(d) Design a parallel sieve algorithm. Would you distribute the array over the processors by blocks, cyclically, or in some other fashion?

(e) Write a parallel sieve program `bspsieve` and measure its execution time for $n = 1000,\ 10\,000,\ 100\,000,\ 1\,000\,000$ and $p = 1, 2, 4, 8$, or use as many processors as you can lay your hands on.

(f) Estimate the BSP cost of the parallel algorithm and use this estimate to explain your time measurements.

(g) Can you reduce the cost further? Hints: for the prime q, do you need to start crossing out at $2q$? Does every processor cross out the same number of integers? Is all communication really necessary?

(h) Modify your program to generate twin primes, that is, pairs of primes that differ by two, such as (5, 7). (It is unknown whether there are infinitely many twin primes.)

(i) Extend your program to check the Goldbach conjecture: every even $k > 2$ is the sum of two primes. Choose a suitable range of integers to check. Try to keep the number of operations low. (The conjecture has been an open question since 1742.)

2

LU DECOMPOSITION

This chapter presents a general Cartesian scheme for the distribution of matrices. Based on BSP cost analysis, the square cyclic distribution is proposed as particularly suitable for matrix computations such as LU decomposition. Furthermore, the chapter introduces two-phase broadcasting of vectors, which is a useful method for sending copies of matrix rows or columns to a group of processors. These techniques are demonstrated in the specific case of LU decomposition, but they are applicable to almost all parallel matrix computations. After having read this chapter, you are able to design and implement parallel algorithms for a wide range of matrix computations, including symmetric linear system solution by Cholesky factorization and eigensystem solution by QR decomposition or Householder tridiagonalization.

2.1 The problem

Take a close look at your favourite scientific computing application. Whether it originates in ocean modelling, oil refinery optimization, electronic circuit simulation, or in another application area, most likely you will find on close inspection that its core computation consists of the solution of large systems of linear equations. Indeed, solving linear systems is the most time-consuming part of many scientific computing applications. Therefore, we start with this problem.

Consider a system of linear equations

$$A\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where A is a given $n \times n$ nonsingular matrix, $\mathbf{b}$ a given vector of length n, and $\mathbf{x}$ the unknown solution vector of length n. One method for solving this system is by using **LU decomposition**, that is, decomposition of the matrix A into an $n \times n$ unit lower triangular matrix L and an $n \times n$ upper triangular matrix U such that

$$A = LU. \tag{2.2}$$

An $n \times n$ matrix L is called **unit lower triangular** if $l_{ii} = 1$ for all i, $0 \leq i < n$, and $l_{ij} = 0$ for all i, j with $0 \leq i < j < n$. An $n \times n$ matrix U is called **upper triangular** if $u_{ij} = 0$ for all i, j with $0 \leq j < i < n$. Note that we always start counting at zero—my daughter Sarai was raised that way—and this will turn out to be an advantage later in life, when encountering parallel computations. (For instance, it becomes easier to define the cyclic distribution.)

Example 2.1

$$\text{For } A = \begin{bmatrix} 1 & 4 & 6 \\ 2 & 10 & 17 \\ 3 & 16 & 31 \end{bmatrix}, \quad \text{the decomposition is } L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix},$$

$$U = \begin{bmatrix} 1 & 4 & 6 \\ 0 & 2 & 5 \\ 0 & 0 & 3 \end{bmatrix}.$$

The linear system $A\mathbf{x} = \mathbf{b}$ can be solved by first decomposing A into $A = LU$ and then solving the triangular systems $L\mathbf{y} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$. The advantage of LU decomposition over similar methods such as Gaussian elimination is that the factors L and U can be reused, to solve different systems $A\mathbf{x} = \mathbf{b}'$ with the same matrix but different right-hand sides. The main text of the present chapter only deals with LU decomposition. Exercise 9 treats the parallel solution of triangular systems.

2.2 Sequential LU decomposition

In this section, we derive the sequential algorithm that is the basis for developing our parallel algorithm. By expanding (2.2) and using the fact that $l_{ir} = 0$ for $i < r$ and $u_{rj} = 0$ for $r > j$, we get

$$a_{ij} = \sum_{r=0}^{n-1} l_{ir} u_{rj} = \sum_{r=0}^{\min(i,j)} l_{ir} u_{rj}, \quad \text{for } 0 \le i, j < n. \tag{2.3}$$

In the case $i \le j$, we split off the ith term and substitute $l_{ii} = 1$, to obtain

$$u_{ij} = a_{ij} - \sum_{r=0}^{i-1} l_{ir} u_{rj}, \quad \text{for } 0 \le i \le j < n. \tag{2.4}$$

Similarly,

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{r=0}^{j-1} l_{ir} u_{rj} \right), \quad \text{for } 0 \le j < i < n. \tag{2.5}$$

Equations (2.4) and (2.5) lead to a method for computing the elements of L and U. For convenience, we first define the intermediate $n \times n$ matrices $A^{(k)}$, $0 \le k \le n$, by

$$a_{ij}^{(k)} = a_{ij} - \sum_{r=0}^{k-1} l_{ir} u_{rj}, \quad \text{for } 0 \le i, j < n. \tag{2.6}$$

Algorithm 2.1. Sequential LU decomposition.

```
input:      $A^{(0)}$ : $n \times n$ matrix.
output:     $L$ : $n \times n$ unit lower triangular matrix,
            $U$ : $n \times n$ upper triangular matrix,
            such that $LU = A^{(0)}$.

for $k := 0$ to $n-1$ do
        for $j := k$ to $n-1$ do
                $u_{kj} := a_{kj}^{(k)}$;
        for $i := k+1$ to $n-1$ do
                $l_{ik} := a_{ik}^{(k)}/u_{kk}$;
        for $i := k+1$ to $n-1$ do
                for $j := k+1$ to $n-1$ do
                        $a_{ij}^{(k+1)} := a_{ij}^{(k)} - l_{ik}u_{kj}$;
```

Note that $A^{(0)} = A$ and $A^{(n)} = 0$. In this notation, (2.4) and (2.5) become

$$u_{ij} = a_{ij}^{(i)}, \quad \text{for } 0 \le i \le j < n, \tag{2.7}$$

and

$$l_{ij} = \frac{a_{ij}^{(j)}}{u_{jj}}, \quad \text{for } 0 \le j < i < n. \tag{2.8}$$

Algorithm 2.1 produces the elements of L and U in stages. Stage k first computes the elements u_{kj}, $j \ge k$, of row k of U and the elements l_{ik}, $i > k$, of column k of L. Then, it computes $A^{(k+1)}$ in preparation for the next stage. Since only values $a_{ij}^{(k)}$ with $i, j \ge k$ are needed in stage k, only the values $a_{ij}^{(k+1)}$ with $i, j \ge k+1$ are prepared. It can easily be verified that this order of computation is indeed feasible: in each assignment of the algorithm, the values of the right-hand side have already been computed.

Figure 2.1 illustrates how computer memory can be saved by storing all currently available elements of L, U, and $A^{(k)}$ in one working matrix, which we call A. Thus, we obtain Algorithm 2.2. On input, A contains the original matrix $A^{(0)}$, whereas on output it contains the values of L below the diagonal and the values of U above and on the diagonal. In other words, the output matrix equals $L - I_n + U$, where I_n denotes the $n \times n$ **identity matrix**, which has ones on the diagonal and zeros everywhere else. Note that stage $n-1$ of the algorithm does nothing, so we can skip it.

This is a good moment for introducing our matrix/vector notation, which is similar to the MATLAB [100] notation commonly used in the field of numerical linear algebra. This notation makes it easy to describe submatrices and

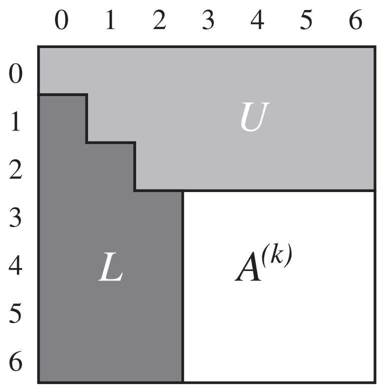


FIG. 2.1. LU decomposition of a 7×7 matrix at the start of stage $k = 3$. The values of L and U computed so far and the computed part of $A^{(k)}$ fit exactly in one matrix.

Algorithm 2.2. Memory-efficient sequential LU decomposition.

```
input:      $A$ : $n \times n$ matrix, $A = A^{(0)}$.
output:     $A$ : $n \times n$ matrix, $A = L - I_n + U$, with
            $L$ : $n \times n$ unit lower triangular matrix,
            $U$ : $n \times n$ upper triangular matrix,
            such that $LU = A^{(0)}$.

for $k := 0$ to $n - 1$ do
        for $i := k + 1$ to $n - 1$ do
                $a_{ik} := a_{ik}/a_{kk}$;
        for $i := k + 1$ to $n - 1$ do
                for $j := k + 1$ to $n - 1$ do
                        $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

subvectors. The subvector $x(i_0 \colon i_1)$ is a vector of length $i_1 - i_0 + 1$, which contains all components of $\mathbf{x}$ from i_0 up to and including i_1. The (noncontiguous) subvector $x(i_0 \colon s \colon i_1)$ contains the components $i_0, i_0 + s, i_0 + 2s, \ldots$ not exceeding i_1. Here, s is the **stride** of the subvector. The subvector $x(*)$ contains all components and hence it equals $\mathbf{x}$. The subvector $x(i)$ contains one component, the ith. The submatrix $A(i_0 \colon i_1, j_0 \colon j_1)$ contains all elements a_{ij} with $i_0 \le i \le i_1$ and $j_0 \le j \le j_1$. The ranges for the matrix indices can be written in the same way as for the vector indices. For example, the submatrix $A(i, *)$ denotes row i of the matrix A. Using our matrix/vector notation, we can write the submatrix used to store elements of $A^{(k)}$ as $A(k \colon n-1, k \colon n-1)$.

We can also write the part of U computed in stage k as $U(k, k\colon n-1)$ and the part of L computed in stage k as $L(k+1\colon n-1, k)$.

Example 2.2 The matrix A of Example 2.1 is transformed into a matrix holding the L and U factors, as follows:

$$A = \begin{bmatrix} 1 & 4 & 6 \\ 2 & 10 & 17 \\ 3 & 16 & 31 \end{bmatrix} \xrightarrow{(0)} \begin{bmatrix} 1 & 4 & 6 \\ 2 & 2 & 5 \\ 3 & 4 & 13 \end{bmatrix} \xrightarrow{(1)} \begin{bmatrix} 1 & 4 & 6 \\ 2 & 2 & 5 \\ 3 & 2 & 3 \end{bmatrix} = L - I_n + U.$$

Example 2.3 No LU decomposition exists for

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

The last example shows that Algorithm 2.2 may break down, even in the case of a nonsingular matrix. This happens if $a_{kk} = 0$ for a certain k, so that division by zero is attempted. A remedy for this problem is to permute the rows of the matrix A in a suitable way, giving a matrix PA, before computing an LU decomposition. This yields

$$PA = LU, \tag{2.9}$$

where P is an $n \times n$ **permutation matrix**, that is, a matrix obtained by permuting the rows of I_n. A useful property of a permutation matrix is that its inverse equals its transpose, $P^{-1} = P^{\mathrm{T}}$. The effect of multiplying A from the left by P is to permute the rows of A.

Every permutation matrix corresponds to a unique permutation, and vice versa. Let $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ be a permutation. We define the permutation matrix P_σ corresponding to σ as the $n \times n$ matrix with elements

$$(P_\sigma)_{ij} = \begin{cases} 1 & \text{if } i = \sigma(j) \\ 0 & \text{otherwise,} \end{cases} \qquad \text{for } 0 \le i, j < n. \tag{2.10}$$

This means that column j of P_σ has an element one in row $\sigma(j)$, and zeros everywhere else.

Example 2.4 Let $n = 3$ and $\sigma(0) = 1$, $\sigma(1) = 2$, and $\sigma(2) = 0$. Then

$$P_\sigma = \begin{bmatrix} \cdot & \cdot & 1 \\ 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \end{bmatrix},$$

where the dots in the matrix denote zeros.

The matrix P_σ has the following useful properties:

Lemma 2.5 *Let* $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ *be a permutation. Let* $\mathbf{x}$ *be a vector of length* n *and* A *an* $n \times n$ *matrix. Then*

$$\begin{aligned} (P_\sigma \mathbf{x})_i &= x_{\sigma^{-1}(i)}, \text{ for } 0 \le i < n, \\ (P_\sigma A)_{ij} &= a_{\sigma^{-1}(i),j}, \text{ for } 0 \le i,j < n, \\ (P_\sigma A P_\sigma^{\mathrm{T}})_{ij} &= a_{\sigma^{-1}(i),\sigma^{-1}(j)}, \text{ for } 0 \le i,j < n. \end{aligned}$$

Lemma 2.6 *Let* $\sigma, \tau : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ *be permutations. Then*

$$P_\tau P_\sigma = P_{\tau\sigma} \quad \text{and} \quad (P_\sigma)^{-1} = P_{\sigma^{-1}}.$$

Here, $\tau\sigma$ *denotes* σ *followed by* τ.

Proof The proofs follow immediately from the definition of P_σ. □

Usually, it is impossible to determine a suitable complete row permutation before the LU decomposition has been carried out, because the choice may depend on the evolving computation of L and U. A common procedure, which works well in practice, is **partial row pivoting**. The computation starts with the original matrix A. At the start of stage k, a **pivot element** a_{rk} is chosen with the largest absolute value in column k, among the elements a_{ik} with $i \ge k$. We express this concisely in the program text by stating that $r = \text{argmax}(|a_{ik}| : k \le i < n)$, that is, r is the argument (or index) of the maximum. If A is nonsingular, it is guaranteed that $a_{rk} \ne 0$. (Taking the largest element, instead of an arbitrary nonzero, keeps us farthest from dividing by zero and hence improves the numerical stability.) Swapping row k and the **pivot row** r now makes it possible to perform stage k.

LU decomposition with partial row pivoting produces the L and U factors of a permuted matrix PA, for a given input matrix A. These factors can then be used to solve the linear system $PA\mathbf{x} = P\mathbf{b}$ by permuting the vector $\mathbf{b}$ and solving two triangular systems. To perform the permutation, we need to know P. We can represent P by a permutation vector π of length n. We denote the components of π by π_i or $\pi(i)$, whichever is more convenient in the context. We determine P by registering the swaps executed in the stages of the computation. For instance, we can start with the identity permutation stored as a vector $\mathbf{e} = (0, 1, \ldots, n-1)^{\mathrm{T}}$ in π. We swap the components k and r of π whenever we swap a row k and a row r of the working matrix. On output, the working matrix holds the L and U factors of PA, and π holds the vector $P\mathbf{e}$. Assume $P = P_\sigma$ for a certain permutation σ. Applying Lemma 2.5 gives $\pi(i) = (P_\sigma \mathbf{e})_i = \mathbf{e}_{\sigma^{-1}(i)} = \sigma^{-1}(i)$, for all i. Therefore, $\sigma = \pi^{-1}$ and $P_{\pi^{-1}} A = LU$. Again applying the lemma, we see that this is equivalent with $a_{\pi(i),j} = (LU)_{ij}$, for all i, j.

The resulting LU decomposition with partial pivoting is given as Algorithm 2.3.

Algorithm 2.3. Sequential LU decomposition with partial row pivoting.

```
input:      $A$ :  $n \times n$ matrix, $A = A^{(0)}$.
output:     $A$ :  $n \times n$ matrix, $A = L - I_n + U$, with
            $L$ :  $n \times n$ unit lower triangular matrix,
            $U$ :  $n \times n$ upper triangular matrix,
            $\pi$ : permutation vector of length $n$,
            such that $a^{(0)}_{\pi(i),j} = (LU)_{ij}$, for $0 \le i, j < n$.

for $i := 0$ to $n-1$ do
        $\pi_i := i$;
for $k := 0$ to $n-1$ do
        $r := \mathrm{argmax}(|a_{ik}| : k \le i < n)$;
        swap$(\pi_k, \pi_r)$;
        for $j := 0$ to $n-1$ do
                swap$(a_{kj}, a_{rj})$;
        for $i := k+1$ to $n-1$ do
                $a_{ik} := a_{ik}/a_{kk}$;
        for $i := k+1$ to $n-1$ do
                for $j := k+1$ to $n-1$ do
                        $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Its cost is determined as follows. The floating-point operations in stage k are: $n-k-1$ divisions, $(n-k-1)^2$ multiplications, and $(n-k-1)^2$ subtractions. We ignore all other operations, such as comparisons, assignments, and integer operations, because taking these into account would make our analysis laborious and unnecessarily complicated. The cost of Algorithm 2.3, measured in flops, is therefore:

$$T_{\mathrm{seq}} = \sum_{k=0}^{n-1}(2(n-k-1)^2 + n-k-1) = \sum_{k=0}^{n-1}(2k^2+k) = \frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}. \quad (2.11)$$

In the summation, we used two formulae so important for analysing the complexity of matrix computations, that you should know them by heart:

Lemma 2.7 *Let $n \ge 0$ be an integer. Then*

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}, \qquad \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Proof By induction on n. □

2.3 Basic parallel algorithm

Design your parallel algorithms backwards! We follow this motto by first transforming a sequential step into a computation superstep and then inserting preceding communication supersteps to obtain nonlocal data where needed.

The design process of our parallel LU decomposition algorithm is as follows. First, we introduce a general data distribution scheme, which reflects the problem and restricts the possible communication patterns, but which also leaves sufficient freedom for optimization. Second, we derive a basic parallel algorithm, directly from the sequential algorithm and the data distribution scheme; we do this mostly in the backward direction. Third, we analyse the cost of the basic parallel algorithm and use the results of this analysis to choose a data distribution with optimal load balance and low communication overhead. Fourth, we restructure the algorithm to reduce its cost further. This section presents the first three phases of the design process; the fourth phase is presented in the next section.

The data to be distributed for parallel LU decomposition are the matrix A and the vector π. Clearly, the most important decision is how to distribute A. The bulk of the computational work in stage k of the sequential algorithm is the modification of the matrix elements a_{ij} with $i, j \geq k+1$. Therefore, our choice of distribution will be based on an analysis of this part of the algorithm. It is easy to distribute the computational work of this part evenly over the processors; this can simply be done by evenly distributing the corresponding data. Distribution of the matrix elements over different processors, however, will give rise to communication, because in general the matrix elements a_{ij}, a_{ik}, and a_{kj} involved in an update $a_{ij} := a_{ij} - a_{ik}a_{kj}$ will not reside on the same processor. There are $(n-k-1)^2$ elements a_{ij} to be updated, using only $n-k-1$ elements a_{ik} from column k of A and $n-k-1$ elements a_{kj} from row k. Therefore, to prevent communication of large amounts of data, the update $a_{ij} := a_{ij} - a_{ik}a_{kj}$ must be performed by the processor that contains a_{ij}. This implies that only elements of column k and row k of A need to be communicated in stage k. This approach is illustrated in Fig. 2.2.

An important observation is that the modification of the elements in row $A(i, k+1\colon n-1)$ uses only one value from column k of A, namely a_{ik}. If we distribute each matrix row over a limited set of N processors, then the communication of an element from column k can be restricted to a broadcast to N processors. Similarly, the modification of the elements in column $A(k+1\colon n-1, j)$ uses only one value from row k of A, namely a_{kj}. If we distribute each matrix column over a limited set of M processors, then the communication of an element from row k can be restricted to a broadcast to M processors.

For matrix computations, it is natural to number the processors by two-dimensional identifiers $P(s,t)$, $0 \leq s < M$ and $0 \leq t < N$, where $p = MN$ is the number of processors. We define **processor row** $P(s,*)$

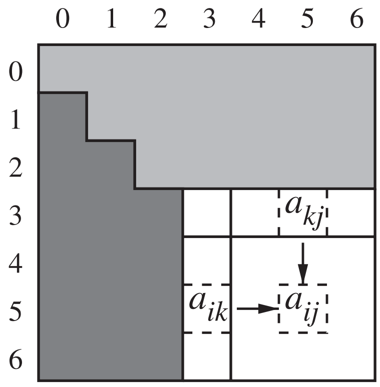


FIG. 2.2. Matrix update by operations $a_{ij} := a_{ij} - a_{ik}a_{kj}$ at the end of stage $k = 3$. Arrows denote communication.

as the group of N processors $P(s,t)$ with $0 \le t < N$, and **processor column** $P(*,t)$ as the group of M processors $P(s,t)$ with $0 \le s < M$. This is just a two-dimensional numbering of the processors and has no physical meaning in the BSP model. Any resemblance to actual parallel computers, such as a rectangular processor network, is purely coincidental and, for the sake of portability, such resemblance should not be exploited. To make it easier to resist the temptation, BSP veterans always tell newcomers to the BSP world that BSPlib software randomly renumbers the processors before it starts.

A **matrix distribution** is a mapping

$$\phi : \ \{(i,j) : 0 \le i,j < n\} \to \{(s,t) : 0 \le s < M \ \wedge \ 0 \le t < N\}$$

from the set of matrix index pairs to the set of processor identifiers. The mapping function ϕ has two coordinates,

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)), \quad \text{for } 0 \le i,j < n. \tag{2.12}$$

A matrix distribution is called **Cartesian** if $\phi_0(i,j)$ is independent of j and $\phi_1(i,j)$ is independent of i, so that we can write

$$\phi(i,j) = (\phi_0(i), \phi_1(j)), \quad \text{for } 0 \le i,j < n. \tag{2.13}$$

Figure 2.3 shows a Cartesian distribution of a 7×7 matrix over 2×3 processors. Cartesian distributions allocate matrix rows to processor rows. This is good for LU decomposition, because in stage k an element a_{ik} of column k needs to be communicated only to the owners of matrix row i, that is, to processor row $P(\phi_0(i), *)$, which is a group of N processors. Similarly, Cartesian distributions allocate matrix columns to processor columns, which reduces the communication of an element from row k to a broadcast to M processors.

	$t=0$	2	1	2	0	1	0
$s=0$	00	02	01	02	00	01	00
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10

FIG. 2.3. A Cartesian distribution of a 7×7 matrix over 2×3 processors. The label 'st' in a cell denotes its owner, processor $P(s,t)$.

In both cases, the destination is only a subset of all the processors. Therefore, we decide to use a Cartesian matrix distribution. For this moment, we do not specify the distribution further, to leave us the freedom of tailoring it to our future needs.

An initial parallel algorithm can be developed by parallelizing the sequential algorithm step by step, using data parallelism to derive computation supersteps and the **need-to-know principle** to obtain the necessary communication supersteps. According to this principle, exactly those nonlocal data that are needed in a computation superstep should be fetched in preceding communication supersteps.

One parallelization method based on this approach is to allocate a computation to the processor that possesses the variable on the left-hand side of an assignment and to communicate beforehand the nonlocal data appearing in the right-hand side. An example is the superstep pair (10)–(11) of Algorithm 2.4, which is a parallel version of the matrix update from stage k of the LU decomposition. (The superstep numbering corresponds to that of the complete basic parallel algorithm.) In superstep (11), the local elements a_{ij} with $i,j \geq k+1$ are modified. In superstep (10), the elements a_{ik} and a_{kj} with $i,j \geq k+1$ are communicated to the processors that need them. It is guaranteed that all values needed have been sent, but depending on the distribution and the stage k, certain processors actually may not need all of the communicated elements. (This mild violation of the strict need-to-know principle is common in *dense* matrix computations, where all matrix elements are treated as nonzero; for *sparse* matrices, however, where many matrix elements are zero, the communication operations should be precisely targeted, see Chapter 4.) Another example of this parallelization method is the superstep pair (8)–(9). In superstep (9), the local elements of column k are divided by a_{kk}. This division is performed only by processors in processor

Algorithm 2.4. Parallel matrix update in stage k for $P(s,t)$.

```
(8)   if $\phi_0(k) = s \wedge \phi_1(k) = t$ then put $a_{kk}$ in $P(*,t)$;

(9)   if $\phi_1(k) = t$ then for all $i : k < i < n \wedge \phi_0(i) = s$ do
          $a_{ik} := a_{ik}/a_{kk}$;

(10)  if $\phi_1(k) = t$ then for all $i : k < i < n \wedge \phi_0(i) = s$ do
          put $a_{ik}$ in $P(s,*)$;
      if $\phi_0(k) = s$ then for all $j : k < j < n \wedge \phi_1(j) = t$ do
          put $a_{kj}$ in $P(*,t)$;

(11)  for all $i : k < i < n \wedge \phi_0(i) = s$ do
          for all $j : k < j < n \wedge \phi_1(j) = t$ do
              $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Algorithm 2.5. Parallel pivot search in stage k for $P(s,t)$.

```
(0)   if $\phi_1(k) = t$ then $r_s := \text{argmax}(|a_{ik}| : k \le i < n \wedge \phi_0(i) = s)$;

(1)   if $\phi_1(k) = t$ then put $r_s$ and $a_{r_s,k}$ in $P(*,t)$;

(2)   if $\phi_1(k) = t$ then
          $s_{\max} := \text{argmax}(|a_{r_q,k}| : 0 \le q < M)$;
          $r := r_{s_{\max}}$;

(3)   if $\phi_1(k) = t$ then put $r$ in $P(s,*)$;
```

column $P(*, \phi_1(k))$, since these processors together possess matrix column k. In superstep (8), the element a_{kk} is obtained.

An alternative parallelization method based on the same need-to-know approach is to allocate a computation to the processor that contains part or all of the data of the right-hand side, and then to communicate partial results to the processors in charge of producing the final result. This may be more efficient if the number of result values is less than the number of input data values involved. An example is the sequence of supersteps (0)–(3) of Algorithm 2.5, which is a parallel version of the pivot search from stage k of the LU decomposition. First a local element with maximum absolute value is determined, whose index and value are then sent to all processors in $P(*, \phi_1(k))$. (In our cost model, this takes the same time as sending them to

Algorithm 2.6. Index and row swaps in stage k for $P(s,t)$.

```
(4)   if $\phi_0(k) = s \wedge t = 0$ then put $\pi_k$ as $\hat{\pi}_k$ in $P(\phi_0(r), 0)$;
      if $\phi_0(r) = s \wedge t = 0$ then put $\pi_r$ as $\hat{\pi}_r$ in $P(\phi_0(k), 0)$;

(5)   if $\phi_0(k) = s \wedge t = 0$ then $\pi_k := \hat{\pi}_r$;
      if $\phi_0(r) = s \wedge t = 0$ then $\pi_r := \hat{\pi}_k$;

(6)   if $\phi_0(k) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
          put $a_{kj}$ as $\hat{a}_{kj}$ in $P(\phi_0(r), t)$;
      if $\phi_0(r) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
          put $a_{rj}$ as $\hat{a}_{rj}$ in $P(\phi_0(k), t)$;

(7)   if $\phi_0(k) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
          $a_{kj} := \hat{a}_{rj}$;
      if $\phi_0(r) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
          $a_{rj} := \hat{a}_{kj}$;
```

only one master processor $P(0, \phi_1(k))$; a similar situation occurs for the inner product algorithm in Section 1.3.) All processors in the processor column redundantly determine the processor $P(s_{\max}, \phi_1(k))$ and the global row index r of the maximum value. The index r is then broadcast to all processors.

The part of stage k that remains to be parallelized consists of index and row swaps. To parallelize the index swaps, we must first choose the distribution of π. It is natural to store π_k together with row k, that is, somewhere in processor row $P(\phi_0(k), *)$; we choose $P(\phi_0(k), 0)$ as the location. Alternatively, we could have replicated π_k and stored a copy in every processor of $P(\phi_0(k), *)$. (Strictly speaking, this is not a distribution any more.) The index swaps are performed by superstep pair (4)–(5) of Algorithm 2.6. The components π_k and π_r of the permutation vector are swapped by first putting each component into its destination processor and then assigning it to the appropriate component of the array π. Temporary variables (denoted by hats) are used to help distinguishing between the old and the new contents of a variable. The same is done for the row swaps in supersteps (6)–(7).

To make the algorithm efficient, we must choose a distribution ϕ that incurs low BSP cost. To do this, we first analyse stage k of the algorithm and identify the main contributions to its cost. Stage k consists of 12 supersteps, so that its synchronization cost equals $12l$. Sometimes, a superstep may be empty so that it can be deleted. For example, if $N = 1$, superstep (3) is empty. In the extreme case $p = 1$, all communication supersteps can be deleted and the remaining computation supersteps can be combined into one superstep. For $p > 1$, however, the number of supersteps in one stage remains a small constant, which should not influence the choice of distribution. Therefore, we consider $12nl$ to be an upper bound on the total synchronization cost of the algorithm, and we exclude terms in l from the following analysis of the separate supersteps.

The computation and communication cost can concisely be expressed using

$$R_k = \max_{0 \le s < M} |\{i : k \le i < n \wedge \phi_0(i) = s\}|, \tag{2.14}$$

that is, R_k is the maximum number of local matrix rows with index $\ge k$, and

$$C_k = \max_{0 \le t < N} |\{j : k \le j < n \wedge \phi_1(j) = t\}|, \tag{2.15}$$

that is, C_k is the maximum number of local matrix columns with index $\ge k$.

Example 2.8 In Fig. 2.3, $R_0 = 4, C_0 = 3$ and $R_4 = 2, C_4 = 2$.

Lower bounds for R_k and C_k are given by

$$R_k \ge \left\lceil \frac{n-k}{M} \right\rceil, \qquad C_k \ge \left\lceil \frac{n-k}{N} \right\rceil. \tag{2.16}$$

Proof Assume $R_k < \lceil (n-k)/M \rceil$. Because R_k is integer, we even have that $R_k < (n-k)/M$ so that each processor row has less than $(n-k)/M$ matrix rows. Therefore, the M processor rows together possess less than $n-k$ matrix rows, which contradicts the fact that they hold the whole range $k \le i < n$. A similar proof holds for C_k. □

The computation supersteps of the algorithm are (0), (2), (5), (7), (9), and (11). Supersteps (0), (2), (5), and (7) are for free in our benign cost model, since they do not involve floating-point operations. (A more detailed analysis taking all types of operations into account would yield a few additional lower-order terms.) Computation superstep (9) costs R_{k+1} time units, since each processor performs at most R_{k+1} divisions. Computation superstep (11) costs $2R_{k+1}C_{k+1}$ time units, since each processor performs at most $R_{k+1}C_{k+1}$ multiplications and $R_{k+1}C_{k+1}$ subtractions. The cost of (11) clearly dominates the total computation cost.

Table 2.1 presents the cost of the communication supersteps of the basic parallel LU decomposition. It is easy to verify the cost values given by the table. For the special case $N = 1$, the h_r value given for (3) in the table should in fact be 0 instead of 1, but this does not affect the resulting value of h. A similar remark should be made for supersteps (4), (8), and (10). During most of the algorithm, the largest communication superstep is (10), while the next-largest one is (6). Near the end of the computation, (6) becomes dominant.

To minimize the total BSP cost of the algorithm, we must take care to minimize the cost of both computation and communication. First we consider

TABLE 2.1. Cost (in g) of communication supersteps in stage k of basic parallel LU decomposition

Superstep	h_s	h_r	$h = \max\{h_s, h_r\}$
(1)	$2(M-1)$	$2(M-1)$	$2(M-1)$
(3)	$N-1$	1	$N-1$
(4)	1	1	1
(6)	C_0	C_0	C_0
(8)	$M-1$	1	$M-1$
(10)	$R_{k+1}(N-1)+$ $C_{k+1}(M-1)$	$R_{k+1}+C_{k+1}$	$R_{k+1}(N-1)+$ $C_{k+1}(M-1)$

the computation cost, and in particular the cost of the dominant computation superstep,

$$T_{(11)} = 2R_{k+1}C_{k+1} \geq 2\left\lceil\frac{n-k-1}{M}\right\rceil \left\lceil\frac{n-k-1}{N}\right\rceil. \tag{2.17}$$

This cost can be minimized by distributing the matrix rows cyclically over the M processor rows and the matrix columns cyclically over the N processor columns. In that case, matrix rows $k+1$ to $n-1$ are evenly or nearly evenly divided over the processor rows, with at most a difference of one matrix row between the processor rows, and similarly for the matrix columns. Thus,

$$T_{(11),\mathrm{cyclic}} = 2\left\lceil\frac{n-k-1}{M}\right\rceil \left\lceil\frac{n-k-1}{N}\right\rceil. \tag{2.18}$$

The resulting matrix distribution is the $M \times N$ **cyclic distribution**, defined by

$$\phi_0(i) = i \bmod M, \quad \phi_1(j) = j \bmod N, \text{ for } 0 \leq i, j < n. \tag{2.19}$$

Figure 2.4 shows the 2×3 cyclic distribution of a 7×7 matrix.

The cost of (11) for the $M \times N$ cyclic distribution is bounded between

$$\begin{aligned}\frac{2(n-k-1)^2}{p} \leq T_{(11),\mathrm{cyclic}} &< 2\left(\frac{n-k-1}{M}+1\right)\left(\frac{n-k-1}{N}+1\right)\\ &= \frac{2(n-k-1)^2}{p} + \frac{2(n-k-1)}{p}(M+N)+2,\end{aligned}$$

where we have used that $MN = p$. The upper bound is minimal if $M = N = \sqrt{p}$, that is, if the distribution is **square**. The resulting second-order term $4(n-k-1)/\sqrt{p}$ in the upper bound can be viewed as the additional computation cost caused by imbalance of the work load.

	$t=0$	1	2	0	1	2	0
$s=0$	00	01	02	00	01	02	00
1	10	11	12	10	11	12	10
0	00	01	02	00	01	02	00
1	10	11	12	10	11	12	10
0	00	01	02	00	01	02	00
1	10	11	12	10	11	12	10
0	00	01	02	00	01	02	00

FIG. 2.4. The 2×3 cyclic distribution of a 7×7 matrix.

Next, we examine the cost of the dominant communication superstep,

$$\begin{aligned} T_{(10)} &= (R_{k+1}(N-1) + C_{k+1}(M-1))g \\ &\geq \left(\left\lceil \frac{n-k-1}{M} \right\rceil (N-1) \; + \; \left\lceil \frac{n-k-1}{N} \right\rceil (M-1) \right) g \\ &= T_{(10),\mathrm{cyclic}}. \end{aligned} \tag{2.20}$$

Again, we can minimize the cost by using the $M \times N$ cyclic distribution. To find optimal values for M and N, we consider the upper bound

$$\begin{aligned} T_{(10),\mathrm{cyclic}} &< \left(\left(\frac{n-k-1}{M} + 1 \right) N + \left(\frac{n-k-1}{N} + 1 \right) M \right) g \\ &= \left((n-k-1) \left(\frac{N}{M} + \frac{M}{N} \right) + M + N \right) g. \end{aligned} \tag{2.21}$$

We now minimize this simple upper bound on the cost, instead of the more complicated true cost itself. (This approximation is valid because the bound is not too far from the true cost.) From $(M-N)^2 \geq 0$, it immediately follows that $N/M + M/N = (M^2+N^2)/(MN) \geq 2$. For $M = N = \sqrt{p}$, the inequality becomes an equality. For this choice, the term $N/M + M/N$ is minimal. The same choice also minimizes the term $M + N$ under the constraint $MN = p$. This implies that the square cyclic distribution is a good choice for the basic LU decomposition algorithm, on the grounds of both computation cost and communication cost.

2.4 Two-phase broadcasting and other improvements

Can the basic parallel algorithm be improved? The computation cost in flops cannot be reduced by much, because the computation part is already well-balanced and little is computed redundantly. Therefore, the question is whether the communication and synchronization cost can be reduced. To answer this, we take a closer look at the communication supersteps.

The **communication volume** V of an h-relation is defined as the total number of data words communicated. Using a one-dimensional processor numbering, we can express this as

$$V = \sum_{s=0}^{p-1} h_{\mathrm{s}}(s) = \sum_{s=0}^{p-1} h_{\mathrm{r}}(s), \tag{2.22}$$

where $h_{\mathrm{s}}(s)$ is the number of data words sent by processor $P(s)$ and $h_{\mathrm{r}}(s)$ is the number received. In this notation, $\max_s h_{\mathrm{s}}(s) = h_{\mathrm{s}}$ and $\max_s h_{\mathrm{r}}(s) = h_{\mathrm{r}}$. Note that $V \leq \sum_{s=0}^{p-1} h = ph$. We call an h-relation **balanced** if $V = ph$, that is, $h = V/p$. Equality can only hold if $h_{\mathrm{s}}(s) = h$ for all s. Therefore, a balanced h-relation has $h_{\mathrm{s}}(s) = h$ for all s, and, similarly, $h_{\mathrm{r}}(s) = h$ for all s. These necessary conditions for balance are also sufficient and hence an h-relation is balanced if and only if every processor sends and receives exactly h words. But this is precisely the definition of a full h-relation, see Section 1.2. It is just a matter of viewpoint whether we call an h-relation balanced or full. The communication volume provides us with a measure for load imbalance: we call $h - V/p$ the **communicational load imbalance**. This is analogous to the **computational load imbalance**, which commonly (but often tacitly) is defined as $w - w_{\mathrm{seq}}/p$, where w denotes work. If an h-relation is balanced, then $h = h_{\mathrm{s}} = h_{\mathrm{r}}$. The reverse is not true: it is possible that $h = h_{\mathrm{s}} = h_{\mathrm{r}}$ but that the h-relation is still unbalanced: some processors may be overloaded sending and some receiving. In that case, $h > V/p$. To reduce communication cost, one can either reduce the volume, or improve the balance for a fixed volume.

Consider the basic parallel LU decomposition algorithm with the cyclic distribution. Assume for diagnostic purposes that the distribution is square. (Later, in developing our improved algorithm, we shall assume the more general $M \times N$ cyclic distribution.) Supersteps (3), (8), and (10) perform h-relations with $h_{\mathrm{s}} \gg h_{\mathrm{r}}$, see Table 2.1. Such a discrepancy between h_{s} and h_{r} is a clear symptom of imbalance. The three unbalanced supersteps are candidates for improvement. We concentrate our efforts on the dominant communication superstep, (10), which has $h_{\mathrm{s}} = (\sqrt{p} - 1)h_{\mathrm{r}}$ and $h \approx 2(n - k - 1)$, see (2.20). The contribution of superstep (10) to the total communication cost of the basic algorithm is about $\sum_{k=0}^{n-1} 2(n - k - 1)g = 2g \sum_{k=0}^{n-1} k = 2g(n - 1)n/2 \approx n^2 g$, irrespective of the number of processors. With an increasing number of processors, the fixed contribution of $n^2 g$ to the total communication cost will soon dominate the total computation cost of roughly $T_{\mathrm{seq}}/p \approx 2n^3/3p$, see (2.11). This back-of-the-envelope analysis suffices to reveal the undesirable scaling behaviour of the row and column broadcasts.

The unbalance in the broadcasts of superstep (10) is caused by the fact that only $2\sqrt{p} - 1$ out of p processors send data: the sending processors are $P(*, \phi_1(k)) = P(*, k \bmod \sqrt{p})$ and $P(\phi_0(k), *) = P(k \bmod \sqrt{p}, *)$. The receives are spread better: the majority of the processors receive $2R_{k+1}$ data elements, or one or two elements less. The communication volume equals $V = 2(n-k-1)(\sqrt{p}-1)$, because $n-k-1$ elements of row k and column k must be broadcast to $\sqrt{p}-1$ processors. It is impossible to reduce the communication volume significantly: all communication operations are really necessary, except in the last few stages of the algorithm. The communication balance, however, has potential for improvement.

To find ways to improve the balance, let us first examine the problem of broadcasting a vector $\mathbf{x}$ of length n from a processor $P(0)$ to all p processors of a parallel computer, where $n \geq p$. For this problem, we use a one-dimensional processor numbering. The simplest approach is that processor $P(0)$ creates $p-1$ copies of each vector component and sends these copies out. This method concentrates all sending work at the source processor. A better balance can be obtained by sending each component to a randomly chosen intermediate processor and making this processor responsible for copying and sending the copies to the final destination. (This method is similar to two-phase randomized routing [176], where packets are sent from source to destination through a randomly chosen intermediate location, to avoid congestion in the routing network.) The new method splits the original h-relation into two phases: phase 0, an unbalanced h-relation with small volume that randomizes the location of the data elements; and phase 1, a well-balanced h-relation that performs the broadcast itself. We call the resulting pair of h-relations a **two-phase broadcast**.

An optimal balance during phase 1 can be guaranteed by choosing the intermediate processors deterministically instead of randomly. For instance, this can be achieved by spreading the vector in phase 0 according to the block distribution, defined by (1.6). (An equally suitable choice is the cyclic distribution.) The resulting two-phase broadcast is given as Algorithm 2.7; it is illustrated by Fig. 2.5. The notation $\text{repl}(\mathbf{x}) = P(*)$ means that $\mathbf{x}$ is replicated such that each processor has a copy. (This is in contrast to $\text{distr}(\mathbf{x}) = \phi$, which means that $\mathbf{x}$ is distributed according to the mapping ϕ.) Phase 0 is an h-relation with $h = n - b$, where $b = \lceil n/p \rceil$ is the block size, and phase 1 has $h = (p-1)b$. Note that both phases cost about ng. The total cost of the two-phase broadcast of a vector of length n to p processors is

$$T_{\text{broadcast}} = \left(n + (p-2) \left\lceil \frac{n}{p} \right\rceil \right) g + 2l \approx 2ng + 2l. \qquad (2.23)$$

This is much less than the cost $(p-1)ng + l$ of the straightforward **one-phase broadcast** (except when l is large).

The two-phase broadcast can be used to broadcast column k and row k in stage k of the parallel LU decomposition. The broadcasts are performed in

Algorithm 2.7. Two-phase broadcast for $P(s)$.

```
input:          x : vector of length $n$, repl(x) = $P(0)$.
output:         x : vector of length $n$, repl(x) = $P(*)$.
function call:  broadcast(x, $P(0)$, $P(*)$).

$b := \lceil n/p \rceil$;
{ Spread the vector. }
(0)     if $s = 0$ then for $t := 0$ to $p-1$ do
                    for $i := tb$ to $\min\{(t+1)b, n\} - 1$ do
                        put $x_i$ in $P(t)$;

{ Broadcast the subvectors. }
(1)     for $i := sb$ to $\min\{(s+1)b, n\} - 1$ do
            put $x_i$ in $P(*)$;
```

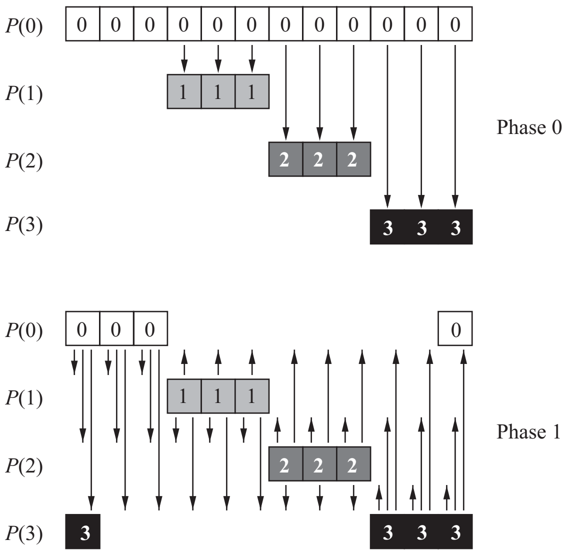


FIG. 2.5. Two-phase broadcast of a vector of size twelve to four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. The block size is $b = 3$. The arrows denote communication. In phase 0, the vector is spread over the four processors. In phase 1, each processor broadcasts its subvector to all processors. To avoid clutter, only a few of the destination cells of phase 1 are shown.

supersteps (6) and (7) of the final algorithm, Algorithm 2.8. The column part to be broadcast from processor $P(s, k \bmod N)$ is the subvector $(a_{ik} : k < i < n \wedge i \bmod M = s)$, which has length R_{k+1} or $R_{k+1} - 1$, and this subvector is broadcast to the whole processor row $P(s, *)$. Every processor row performs its own broadcast of a column part. The row broadcast is done similarly. Note the identical superstep numbering '(6)/(7)' of the two broadcasts, which is a concise way of saying that phase 0 of the row broadcast is carried out together with phase 0 of the column broadcast and phase 1 with phase 1 of the column broadcast. This saves two synchronizations. (In an implementation, such optimizations are worthwhile, but they harm modularity: the complete broadcast cannot be invoked by one function call; instead, we need to make the phases available as separately callable functions.)

The final algorithm has eight supersteps in the main loop, whereas the basic algorithm has twelve. The number of supersteps has been reduced as follows. First, we observe that the row swap of the basic algorithm turns element a_{rk} into the pivot element a_{kk}. The element a_{rk}, however, is already known by all processors in $P(*, k \bmod N)$, because it is one of the elements broadcast in superstep (1). Therefore, we divide column k immediately by a_{rk}, instead of dividing by a_{kk} after the row swap. This saves the pivot broadcast (8) of the basic algorithm and the synchronization of superstep (9). For readability, we introduce the convention of writing the condition '**if** $k \bmod N = t$ **then**' only once for supersteps (0)–(3), even though we want the test to be carried out in every superstep. This saves space and makes the algorithm better readable; in an implementation, the test must be repeated in every superstep. (Furthermore, we must take care to let all processors participate in the global synchronization, and not only those that test positive.) Second, the index and row swaps are now combined and performed in two supersteps, numbered (4) and (5). This saves two synchronizations. Third, the last superstep of stage k of the algorithm is combined with the first superstep of stage $k+1$. We express this by numbering the last superstep as $(0')$, that is, superstep (0) of the next stage.

The BSP cost of the final algorithm is computed in the same way as before. The cost of the separate supersteps is given by Table 2.2. Now, $R_{k+1} = \lceil (n-k-1)/M \rceil$ and $C_{k+1} = \lceil (n-k-1)/N \rceil$, because we use the $M \times N$ cyclic distribution. The cost expressions for supersteps (6) and (7) are obtained as in the derivation of (2.23).

The dominant computation superstep in the final algorithm remains the matrix update; the choice $M = N = \sqrt{p}$ remains optimal for computation. The costs of the row and column broadcasts do not dominate the other communication costs any more, since they have decreased to about $2(R_{k+1} + C_{k+1})g$ in total, which is of the same order as the cost $C_0 g$ of the row swap. To find optimal values of M and N for communication, we consider

Algorithm 2.8. Final parallel LU decomposition algorithm for $P(s,t)$.

```
input:     $A$ : $n \times n$ matrix, $A = A^{(0)}$, distr($A$) = $M \times N$ cyclic.
output:    $A$ : $n \times n$ matrix, distr($A$) = $M \times N$ cyclic, $A = L - I_n + U$, with
           $L$ : $n \times n$ unit lower triangular matrix,
           $U$ : $n \times n$ upper triangular matrix,
           $\pi$ : permutation vector of length $n$, distr($\pi$) = cyclic in $P(*,0)$,
           such that $a^{(0)}_{\pi(i),j} = (LU)_{ij}$, for $0 \le i,j < n$.

if $t = 0$ then for all $i : 0 \le i < n \wedge i \bmod M = s$ do
        $\pi_i := i$;
for $k := 0$ to $n-1$ do
        if $k \bmod N = t$ then
(0)             $r_s :=$ argmax$(|a_{ik}| : k \le i < n \wedge i \bmod M = s)$;
(1)             put $r_s$ and $a_{r_s,k}$ in $P(*,t)$;
(2)             $s_{\max} :=$ argmax$(|a_{r_q,k}| : 0 \le q < M)$;
                $r := r_{s_{\max}}$;
                for all $i : k \le i < n \wedge i \bmod M = s \wedge i \ne r$ do
                        $a_{ik} := a_{ik}/a_{rk}$;
(3)             put $r$ in $P(s,*)$;

(4)     if $k \bmod M = s$ then
                if $t = 0$ then put $\pi_k$ as $\hat{\pi}_k$ in $P(r \bmod M, 0)$;
                for all $j : 0 \le j < n \wedge j \bmod N = t$ do
                        put $a_{kj}$ as $\hat{a}_{kj}$ in $P(r \bmod M, t)$;
        if $r \bmod M = s$ then
                if $t = 0$ then put $\pi_r$ as $\hat{\pi}_r$ in $P(k \bmod M, 0)$;
                for all $j : 0 \le j < n \wedge j \bmod N = t$ do
                        put $a_{rj}$ as $\hat{a}_{rj}$ in $P(k \bmod M, t)$;

(5)     if $k \bmod M = s$ then
                if $t = 0$ then $\pi_k := \hat{\pi}_r$;
                for all $j : 0 \le j < n \wedge j \bmod N = t$ do
                        $a_{kj} := \hat{a}_{rj}$;
        if $r \bmod M = s$ then
                if $t = 0$ then $\pi_r := \hat{\pi}_k$;
                for all $j : 0 \le j < n \wedge j \bmod N = t$ do
                        $a_{rj} := \hat{a}_{kj}$;

(6)/(7) broadcast($(a_{ik} : k < i < n \wedge i \bmod M = s), P(s, k \bmod N), P(s,*)$);
(6)/(7) broadcast($(a_{kj} : k < j < n \wedge j \bmod N = t), P(k \bmod M, t), P(*,t)$);

(0′)    for all $i : k < i < n \wedge i \bmod M = s$ do
                for all $j : k < j < n \wedge j \bmod N = t$ do
                        $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

TABLE 2.2. Cost of supersteps in stage k of the final parallel LU decomposition algorithm

Superstep	Cost
(0)	l
(1)	$2(M-1)g+l$
(2)	R_k+l
(3)	$(N-1)g+l$
(4)	$(C_0+1)g+l$
(5)	l
(6)	$(R_{k+1}-\lceil R_{k+1}/N\rceil+C_{k+1}-\lceil C_{k+1}/M\rceil)g+l$
(7)	$((N-1)\lceil R_{k+1}/N\rceil+(M-1)\lceil C_{k+1}/M\rceil)g+l$
(0′)	$2R_{k+1}C_{k+1}$

the upper bound

$$\begin{aligned} R_{k+1}+C_{k+1} &< \left(\frac{n-k-1}{M}+1\right)+\left(\frac{n-k-1}{N}+1\right) \\ &= (n-k-1)\frac{M+N}{p}+2, \end{aligned} \tag{2.24}$$

which is minimal for $M=N=\sqrt{p}$. The row swap in superstep (4) prefers large values of N, because $C_0=\lceil n/N\rceil$. The degenerate choice $N=p$ even gives a free swap, but at the price of an expensive column broadcast. Overall, the choice $M=N=\sqrt{p}$ is close to optimal and we shall adopt it in the following analysis.

The total BSP cost of the final algorithm with the square cyclic distribution is obtained by summing the contributions of all supersteps. This gives

$$\begin{aligned} T_{\mathrm{LU}} = \sum_{k=0}^{n-1}(2R_{k+1}^2+R_k) + 2\sum_{k=0}^{n-1}\left(R_{k+1}+(\sqrt{p}-2)\left\lceil\frac{R_{k+1}}{\sqrt{p}}\right\rceil\right)g \\ + (C_0+3\sqrt{p}-2)ng+8nl. \end{aligned} \tag{2.25}$$

To compute $\sum_{k=0}^{n-1}R_k=\sum_{k=0}^{n-1}\lceil(n-k)/\sqrt{p}\rceil=\sum_{k=1}^{n}\lceil k/\sqrt{p}\rceil$ and the sums of R_{k+1} and R_{k+1}^2, we need the following lemma.

Lemma 2.9 *Let $n,q\geq 1$ be integers with $n \bmod q = 0$. Then*

$$\sum_{k=0}^{n}\left\lceil\frac{k}{q}\right\rceil=\frac{n(n+q)}{2q}, \qquad \sum_{k=0}^{n}\left\lceil\frac{k}{q}\right\rceil^2=\frac{n(n+q)(2n+q)}{6q^2}.$$

Proof

$$\sum_{k=0}^{n} \left\lceil \frac{k}{q} \right\rceil = \left\lceil \frac{0}{q} \right\rceil + \left(\left\lceil \frac{1}{q} \right\rceil + \cdots + \left\lceil \frac{q}{q} \right\rceil \right) + \cdots + \left(\left\lceil \frac{n-q+1}{q} \right\rceil + \cdots + \left\lceil \frac{n}{q} \right\rceil \right)$$

$$= q \cdot 1 + q \cdot 2 + \cdots + q \cdot \frac{n}{q} = q \sum_{k=1}^{n/q} k = q \frac{n}{2q} \left(\frac{n}{q} + 1 \right), \tag{2.26}$$

where we have used Lemma 2.7. The proof of the second equation is similar. □

Provided $n \bmod \sqrt{p} = 0$, the resulting sums are:

$$\sum_{k=0}^{n-1} R_k = \frac{n(n+\sqrt{p})}{2\sqrt{p}}, \tag{2.27}$$

$$\sum_{k=0}^{n-1} R_{k+1} = \frac{n(n+\sqrt{p})}{2\sqrt{p}} - \frac{n}{\sqrt{p}}, \tag{2.28}$$

$$\sum_{k=0}^{n-1} R_{k+1}^2 = \frac{n(n+\sqrt{p})(2n+\sqrt{p})}{6p} - \frac{n^2}{p}. \tag{2.29}$$

To compute the sum of $\lceil R_{k+1}/\sqrt{p} \rceil = \lceil \lceil (n-k-1)/\sqrt{p} \rceil / \sqrt{p} \rceil$, we need the following lemma, which may be useful in other contexts as well.

Lemma 2.10 *Let k, q, r be integers with $q, r \geq 1$. Then*

$$\left\lceil \frac{\lceil k/q \rceil}{r} \right\rceil = \left\lceil \frac{k}{qr} \right\rceil.$$

Proof Write $k = aqr + bq + c$, with $0 \leq b < r$ and $0 \leq c < q$. If $b = c = 0$, both sides of the equation equal a. Otherwise, they equal $a + 1$, as can easily be verified. □

The remaining sum equals

$$\sum_{k=0}^{n-1} \left\lceil \frac{R_{k+1}}{\sqrt{p}} \right\rceil = \sum_{k=0}^{n-1} \left\lceil \frac{\lceil (n-k-1)/\sqrt{p} \rceil}{\sqrt{p}} \right\rceil = \sum_{k=0}^{n-1} \left\lceil \frac{\lceil k/\sqrt{p} \rceil}{\sqrt{p}} \right\rceil = \sum_{k=0}^{n-1} \left\lceil \frac{k}{p} \right\rceil$$

$$= \frac{n(n+p)}{2p} - \frac{n}{p}. \tag{2.30}$$

The last equality follows from Lemma 2.9 with $q = p$, which can be applied if we assume that $n \bmod p = 0$ (this also guarantees that $n \bmod \sqrt{p} = 0$). This assumption is made solely for the purpose of simplifying our analysis; in an implementation, such a restriction would hinder practical application and hence it should be avoided there.

The total BSP cost of the final algorithm with the square cyclic distribution is obtained by substituting the results of (2.27)–(2.30) and the value $C_0 = n/\sqrt{p}$ into (2.25). This gives

$$T_{\rm LU} = \frac{2n^3}{3p} + \left(\frac{3}{2\sqrt{p}} - \frac{2}{p}\right) n^2 + \frac{5n}{6} + \left(\left(\frac{3}{\sqrt{p}} - \frac{2}{p}\right) n^2 + \left(4\sqrt{p} - \frac{4}{\sqrt{p}} + \frac{4}{p} - 3\right) n\right) g + 8nl. \qquad (2.31)$$

For many purposes, it suffices to approximate the total cost of an algorithm by taking into account only the highest-order computation term and the highest-order **overhead** terms, that is, the terms representing load imbalance, communication, and synchronization. In this case, an approximate cost estimate is

$$T_{\rm LU} \approx \frac{2n^3}{3p} + \frac{3n^2}{2\sqrt{p}} + \frac{3n^2 g}{\sqrt{p}} + 8nl. \qquad (2.32)$$

Note that in the final algorithm the row swaps, row broadcasts, and column broadcasts each contribute $n^2 g/\sqrt{p}$ to the communication cost.

2.5 Example function `bsplu`

This section presents the program texts of the function `bsplu`, which is a BSPlib implementation of Algorithm 2.8, and the collective-communication function `bsp_broadcast` used in `bsplu`.

The function `bsp_broadcast` implements a slight generalization of Algorithm 2.7: it broadcasts the vector `x` from processor $P(\texttt{src})$ instead of $P(0)$, and the set of destination processors is $\{P(\texttt{s0}+t*\texttt{stride}) : 0 \le t < \texttt{p0}\}$ instead of the set of all p processors. (Sometimes, the term '**multicast**' is used to describe such an operation with a limited number of destination processors; the term 'broadcast' is then reserved for the case with p destination processors. We do not make this distinction.) Within the broadcast function, processors are numbered in one-dimensional fashion. The broadcast as we formulate it is flexible and it can be applied in many situations: for instance, it can be used to broadcast a vector within one processor row of a parallel computer numbered in two-dimensional fashion; for the standard identification $P(s,t) \equiv P(s+tM)$, the parameter `s0` equals the number of the processor row involved, `stride` $= M$, and `p0` $= N$. The function can also be used to perform several broadcasts simultaneously, for example, one broadcast within each processor row. In that case, $P(s,t)$ executes the function with the parameter `s0` $= s$. This feature must be used with caution: the simultaneous broadcast works well as long as the set of processors can be partitioned into disjoint subsets, each including a source and a destination set. The processors within a subset should all be able to determine their source, destination set, and vector length uniquely from the function parameters. A processor can then decide to participate as source and/or destination in its subset or to remain idle. In the LU decomposition program, processors are partitioned into processor rows for the purpose of column broadcasts, and into processor columns for row broadcasts.

The broadcast function is designed such that it can perform the phases of the broadcast separately; a complete broadcast is done by calling the function twice, first with a value `phase` $= 0$, then with `phase` $= 1$. The synchronization terminating the phase is not done by the broadcast function itself, but is left to the calling program. The advantage of this approach is that unnecessary synchronizations are avoided. For instance, phase 0 of the row and column broadcasts can be combined into one superstep, thus needing only one synchronization.

The program text of the broadcast function is a direct implementation of Algorithm 2.7 in the general context described above. Note that the size of the data vector to be put is the minimum of the block size b and the number $n - tb$ of components that would remain if all preceding processors had put b components. The size thus computed may be negative or zero, so that we must make sure that the put is carried out only for positive size. In phase 1, all processors avoid sending data back to the source processor. This optimization has no effect on the BSP cost, since the (cost-determining) source processor itself does not benefit, as can be seen by studying the role of $P(0)$ in Fig. 2.5. Still, the optimization reduces the overall communication volume, making it equal to that of the one-phase broadcast. This may make believers in other models than BSP happy, and BSP believers with cold feet as well!

The basic structure of the LU decomposition algorithm and the function `bsplu` are the same, except that supersteps (0)–(1) of the algorithm are combined into `Superstep 0` of the function, supersteps (2)–(3) are combined into `Superstep 1`, and supersteps (4)–(5) into `Superstep 2`. For the pairs (0)–(1) and (2)–(3) this could be done because BSPlib allows computation and communication to be mixed; for (4)–(5), this could be done because `bsp_puts` are buffered automatically, so that we do not have to take care of that ourselves. Note that the superstep (0′)–(0) of the algorithm is delimited quite naturally in the program text by the common terminating `bsp_sync` of `Superstep 0`. As a result, each stage of the function `bsplu` has five supersteps.

The relation between the variables of the algorithm and those of the function `bsplu` is as follows. The variables $M, N, s, t, n, k, s_{\max}, r$ of the algorithm correspond to the variables `M`, `N`, `s`, `t`, `n`, `k`, `smax`, `r` of the function. The global row index used in the algorithm is $i = \texttt{i} * \texttt{M} + \texttt{s}$, where `i` is the local row index used in the function. The global column index is $j = \texttt{j} * \texttt{N} + \texttt{t}$, where `j` is the local column index. The matrix element a_{ij} corresponds to `a[i][j]` on the processor that owns a_{ij}, and the permutation component π_i corresponds to `pi[i]`. The global row index of the local element in column k with largest absolute value is $r_s = \texttt{imax} * \texttt{M} + \texttt{s}$. The numerical value $a_{r_s,k}$ of this element corresponds to the variable `max` in the function. The arrays `Max` and `Imax` store the local maxima and their index for the M processors that contain part of column k. The maximum for processor $P(s, k \bmod N)$ is stored in $\texttt{Max}[\texttt{s}]$. The global row index of the overall winner is $r = \texttt{Imax}[\texttt{smax}] * \texttt{M} + \texttt{smax}$ and its value is $a_{rk} = \texttt{pivot}$. The arrays `uk` and `lk` store, starting from index 0, the local parts to be broadcast from row k of U and column k of L.

The function `nloc` introduced in the program `bspinprod` of Section 1.4 is used here as well, for instance to compute the number of local rows $\texttt{nloc}(\texttt{M},\texttt{s},\texttt{n})$ of processor row $P(s,*)$ for an $n \times n$ matrix in the $M \times N$ cyclic distribution. The local rows have local indices $\texttt{i} = 0, 1, \ldots, \texttt{nloc}(\texttt{M},\texttt{s},\texttt{n}) - 1$. In general, it holds that $\texttt{i} < \texttt{nloc}(\texttt{M},\texttt{s},\texttt{k})$ if and only if $i < k$. Thus, $\texttt{i} = \texttt{nloc}(\texttt{M},\texttt{s},\texttt{k})$ is the first local row index for which the corresponding global row index satisfies $i \geq k$.

Variables must have been registered before they can be used as the destination of put operations. For example, the arrays `lk`, `uk`, `Max`, `Imax` are registered immediately upon allocation. Thus, registration takes place outside the main loop, which is much cheaper than registering an array each time it is used in communication. The exact length of the array is also registered so that we can be warned if we attempt to put data beyond the end of the array. (Sloppy, lazy, or overly confident programmers sometimes register `INT_MAX`, defined in the standard C file `limits.h`, instead of the true length; a practice not to be recommended.) The permutation array `pi` must be allocated outside `bsplu` because it is an output array. Nevertheless, its registration takes place inside `bsplu`, because this is the only place where it is used in communication. The matrix A itself is the target of put operations, namely in the row swaps. The easiest and cheapest way of registering a two-dimensional array is by exploiting the fact that the utility function `matallocd` from `bspedupack.c` (see Appendix A) allocates a contiguous array of length mn to store an $m \times n$ matrix. We can address this matrix in a one-dimensional fashion, if we wish to do so, and communication is one of the few occasions where this is worthwhile. We introduce a variable `pa`, which stands for 'pointer to A'; it is a pointer that stores the address of the first matrix row, $\texttt{a}[0]$. We can put data into every desired matrix row `i` by putting them into the space pointed to by `pa` and using a suitable offset $\texttt{i} * \texttt{nlc}$, where `nlc` is the row length of the destination processor. (Watch out: the putting processor must know the row length of the remote processor. Fortunately, in the present case the local and remote row lengths are the same, because the rows are swapped.) This way of registering a matrix requires only one registration, instead of one registration per row. This saves much communication time, because registration is expensive: each registration costs at least $(p-1)g$ because every processor broadcasts the address of its own variable to all other processors.

The supersteps of the function `bsplu` are a straightforward implementation of the supersteps in the LU decomposition algorithm. A few details need additional explanation. The division by the pivot in `Superstep 1` is also carried out for the pivot element itself, yielding $a_{rk}/a_{rk} = 1$, despite the fact that this element should keep its original value a_{rk}. Later, this value must be swapped into a_{kk}. This problem is solved by simply reassigning the original value stored in the temporary variable `pivot`.

In the matrix update, elements `lk[i − kr1]` are used instead of `lk[i]`, because array `lk` was filled starting from position 0. If desired, the extra index calculations can be saved by shifting the contents of `lk` by `kr1` positions to the right just before the update loop. A similar remark holds for `uk`.

The program contains only rudimentary error handling. For the sake of brevity, we have only included a crude test for numerical singularity. If no pivot can be found with an absolute value larger than `EPS`, then the program is aborted and an error message is printed. The complete program text is:

```
#include "bspedupack.h"

#define EPS 1.0e-15

void bsp_broadcast(double *x, int n, int src, int s0, int stride, int p0,
                   int s, int phase){
    /* Broadcast the vector x of length n from processor src to
       processors s0+t*stride, 0 <= t < p0. Here n >= 0, p0 >= 1.
       The vector x must have been registered previously.
       Processors are numbered in one-dimensional fashion.
       s = local processor identity.
       phase= phase of two-phase broadcast (0 or 1)
       Only one phase is performed, without synchronization.
    */

    int b, t, t1, dest, nbytes;

    b= ( n%p0==0 ?  n/p0 : n/p0+1 ); /* block size */

    if (phase==0 && s==src){
        for (t=0; t<p0; t++){
            dest= s0+t*stride;
            nbytes= MIN(b,n-t*b)*SZDBL;
            if (nbytes>0)
                bsp_put(dest,&x[t*b],x,t*b*SZDBL,nbytes);
        }
    }

    if (phase==1 && s%stride==s0%stride){
        t=(s-s0)/stride; /* s = s0+t*stride */
        if (0<=t && t<p0){
            nbytes= MIN(b,n-t*b)*SZDBL;
            if (nbytes>0){
                for (t1=0; t1<p0; t1++){
                    dest= s0+t1*stride;
                    if (dest!=src)
                        bsp_put(dest,&x[t*b],x,t*b*SZDBL,nbytes);
                }
            }
        }
    }

} /* end bsp_broadcast */

int nloc(int p, int s, int n){
    /* Compute number of local components of processor s for vector
       of length n distributed cyclically over p processors. */

    return  (n+p-s-1)/p ;

} /* end nloc */

void bsplu(int M, int N, int s, int t, int n, int *pi, double **a){
    /* Compute LU decomposition of n by n matrix A with partial pivoting.
       Processors are numbered in two-dimensional fashion.
       Program text for P(s,t) = processor s+t*M,
       with 0 <= s < M and 0 <= t < N.
       A is distributed according to the M by N cyclic distribution.
    */

    int nloc(int p, int s, int n);
    double *pa, *uk, *lk, *Max;
    int nlr, nlc, k, i, j, r, *Imax;

    nlr=  nloc(M,s,n); /* number of local rows */
    nlc=  nloc(N,t,n); /* number of local columns */

    bsp_push_reg(&r,SZINT);
    if (nlr>0)
        pa= a[0];
    else
        pa= NULL;
    bsp_push_reg(pa,nlr*nlc*SZDBL);
    bsp_push_reg(pi,nlr*SZINT);
    uk= vecallocd(nlc); bsp_push_reg(uk,nlc*SZDBL);
    lk= vecallocd(nlr); bsp_push_reg(lk,nlr*SZDBL);
    Max= vecallocd(M); bsp_push_reg(Max,M*SZDBL);
    Imax= vecalloci(M); bsp_push_reg(Imax,M*SZINT);

    /* Initialize permutation vector pi */
    if (t==0){
        for(i=0; i<nlr; i++)
            pi[i]= i*M+s; /* global row index */
    }
    bsp_sync();
```

```
for (k=0; k<n; k++){
    int kr, kr1, kc, kc1, imax, smax, s1, t1;
    double absmax, max, pivot;

    /****** Superstep 0 ******/
    kr=  nloc(M,s,k); /* first local row with global index >= k */
    kr1= nloc(M,s,k+1);
    kc=  nloc(N,t,k);
    kc1= nloc(N,t,k+1);

    if (k%N==t){   /* k=kc*N+t */
        /* Search for local absolute maximum in column k of A */
        absmax= 0.0; imax= -1;
        for (i=kr; i<nlr; i++){
            if (fabs(a[i][kc])>absmax){
                absmax= fabs(a[i][kc]);
                imax= i;
            }
        }
        if (absmax>0.0){
            max= a[imax][kc];
        } else {
            max= 0.0;
        }

        /* Broadcast value and local index of maximum to P(*,t) */
        for(s1=0; s1<M; s1++){
            bsp_put(s1+t*M,&max,Max,s*SZDBL,SZDBL);
            bsp_put(s1+t*M,&imax,Imax,s*SZINT,SZINT);
        }
    }
    bsp_sync();

    /****** Superstep 1 ******/
    if (k%N==t){
        /* Determine global absolute maximum (redundantly) */
        absmax= 0.0;
        for(s1=0; s1<M; s1++){
            if (fabs(Max[s1])>absmax){
                absmax= fabs(Max[s1]);
                smax= s1;
            }
        }
        if (absmax > EPS){
            r= Imax[smax]*M+smax; /* global index of pivot row */
            pivot= Max[smax];
            for(i=kr; i<nlr; i++)
                a[i][kc] /= pivot;
            if (s==smax)
                a[imax][kc]= pivot; /* restore value of pivot */
            /* Broadcast index of pivot row to P(*,*) */
            for(t1=0; t1<N; t1++)
                bsp_put(s+t1*M,&r,&r,0,SZINT);
        } else {
            bsp_abort("bsplu at stage %d: matrix is singular\n",k);
        }
    }
    bsp_sync();

    /****** Superstep 2 ******/
    if (k%M==s){
        /* Store pi(k) in pi(r) on P(r%M,0) */
        if (t==0)
            bsp_put(r%M,&pi[k/M],pi,(r/M)*SZINT,SZINT);
        /* Store row k of A in row r on P(r%M,t) */
        bsp_put(r%M+t*M,a[k/M],pa,(r/M)*nlc*SZDBL,nlc*SZDBL);
    }
    if (r%M==s){
        if (t==0)
            bsp_put(k%M,&pi[r/M],pi,(k/M)*SZINT,SZINT);
        bsp_put(k%M+t*M,a[r/M],pa,(k/M)*nlc*SZDBL,nlc*SZDBL);
    }
    bsp_sync();

    /****** Superstep 3 ******/
    /* Phase 0 of two-phase broadcasts */
    if (k%N==t){
        /* Store new column k in lk */
        for(i=kr1; i<nlr; i++)
            lk[i-kr1]= a[i][kc];
    }
    if (k%M==s){
        /* Store new row k in uk */
        for(j=kc1; j<nlc; j++)
            uk[j-kc1]= a[kr][j];
    }
    bsp_broadcast(lk,nlr-kr1,s+(k%N)*M,  s,M,N,s+t*M,0);
    bsp_broadcast(uk,nlc-kc1,(k%M)+t*M,t*M,1,M,s+t*M,0);
    bsp_sync();

    /****** Superstep 4 ******/
    /* Phase 1 of two-phase broadcasts */
    bsp_broadcast(lk,nlr-kr1,s+(k%N)*M,  s,M,N,s+t*M,1);
    bsp_broadcast(uk,nlc-kc1,(k%M)+t*M,t*M,1,M,s+t*M,1);
    bsp_sync();

    /****** Superstep 0 ******/
    /* Update of A */
    for(i=kr1; i<nlr; i++){
        for(j=kc1; j<nlc; j++)
            a[i][j] -= lk[i-kr1]*uk[j-kc1];
    }
}
```

```
    bsp_pop_reg(Imax); vecfreei(Imax);
    bsp_pop_reg(Max); vecfreed(Max);
    bsp_pop_reg(lk); vecfreed(lk);
    bsp_pop_reg(uk); vecfreed(uk);
    bsp_pop_reg(pi);
    bsp_pop_reg(pa);
    bsp_pop_reg(&r);

} /* end bsplu */
```

2.6 Experimental results on a Cray T3E

Experiment does to computation models what the catwalk does to fashion models: it subjects the models to critical scrutiny, exposes their good and bad sides, and makes the better models stand out. In this section, we put the predictions of the BSP model for LU decomposition to the test. We perform numerical experiments to check whether the theoretical benefits of two-phase broadcasting can be observed in practice. To do this, we measure the performance of the function `bsplu` with the two-phase broadcasting function `bsp_broadcast`, and also with a one-phase broadcasting function.

We performed our experiments on 64 processors of the Cray T3E introduced in Section 1.7. We compiled our LU program using the standard Cray ANSI C compiler and BSPlib version 1.4 with optimization flags `-flibrary-level 2 -O3 -bspfifo 10000 -fcombine-puts`. We multiplied the times produced by the (faulty) timer by 4.0, to obtain correct time measurements, as was done in Section 1.7. A good habit is to run the benchmark program `bspbench` just before running an application program such as `bsplu`. This helps detecting system changes (improvements or degradations) and tells you what BSP machine you have today. The BSP parameters of our computer are $p = 64$, $r = 38.0$ Mflop/s, $g = 87$, $l = 2718$.

In the experiments, the test matrix A is distributed by the 8×8 cyclic distribution. The matrix is chosen such that the pivot row in stage k is row $k + 1$; this forces a row swap with communication in every stage of the algorithm, because rows k and $k + 1$ reside on different processor rows.

Table 2.3 presents the total execution time of LU decomposition with one-phase and two-phase broadcasts. The difference between the two cases is small but visible. For $n < 4000$, LU decomposition with the one-phase broadcast is faster because it requires less synchronization; this is important for small problems. For $n > 4000$, LU decomposition with the two-phase broadcast is faster, which is due to better spreading of the communication. The savings in broadcast time, however, are insignificant compared with the total execution time. The break-even point for the two types of broadcast lies at about $n = 4000$.

Why is the difference in total execution time so small? Timing the supersteps is an excellent way of answering such questions. By inserting a `bsp_time` statement after every `bsp_sync`, taking the time difference between subsequent

TABLE 2.3. Time (in s) of LU decomposition on a 64-processor Cray T3E using one-phase and two-phase broadcasts

n	One-phase	Two-phase
1000	1.21	1.33
2000	7.04	7.25
3000	21.18	21.46
4000	47.49	47.51
5000	89.90	89.71
6000	153.23	152.79
7000	239.21	238.25
8000	355.84	354.29
9000	501.92	499.74
10 000	689.91	689.56

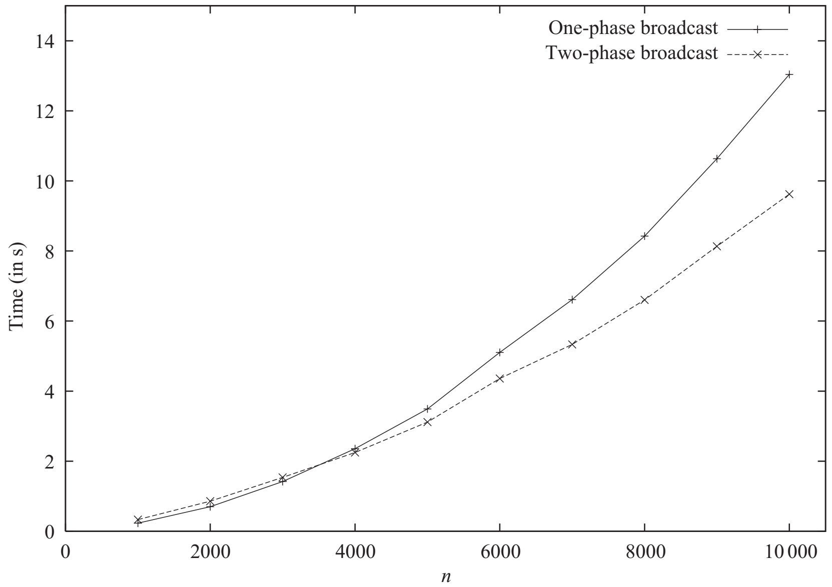


FIG. 2.6. Total broadcast time of LU decomposition on a 64-processor Cray T3E using one-phase and two-phase broadcasts.

synchronizations, and adding the times for the same program superstep, we obtain the total time spent in each of the supersteps of the program. By adding the total time of program supersteps 3 and 4, we compute the total broadcast time, shown in Fig. 2.6. In this figure, it is easy to see that for large matrices the two-phase broadcast is significantly faster than the one-phase broadcast, thus confirming our theoretical analysis. For small matrices, with $n < 4000$, the vectors to be broadcast are too small to justify the extra synchronization. Note that for $n = 4000$, each processor has a local submatrix of size 500×500, so that it broadcasts two vectors of size 499 in stage 0, and this size decreases until it reaches 1 in stage 498; in all stages, the vectors involved are relatively small. The theoretical asymptotic gain factor in broadcast time for large matrices is about $\sqrt{p}/2 = 4$; the observed gain factor of about 1.4 at $n = 10\,000$ is still far from that asymptotic value. Our results imply that for $n = 4000$ the broadcast time represents only about 4.8% of the total time; for larger n this fraction is even less. Thus, the significant improvement in broadcast time in the range $n =$4000–10 000 becomes insignificant compared to the total execution time, explaining the results of Table 2.3. (On a different computer, with faster computation compared to communication and hence a higher g, the improvement would be felt also in the total execution time.)

Program supersteps 2, 3, and 4 account for almost all of the communication carried out by `bsplu`. These supersteps perform the row swaps, phase 0 of the row and column broadcasts, and phase 1, respectively. The BSP model predicts that these widely differing operations have the same total cost,

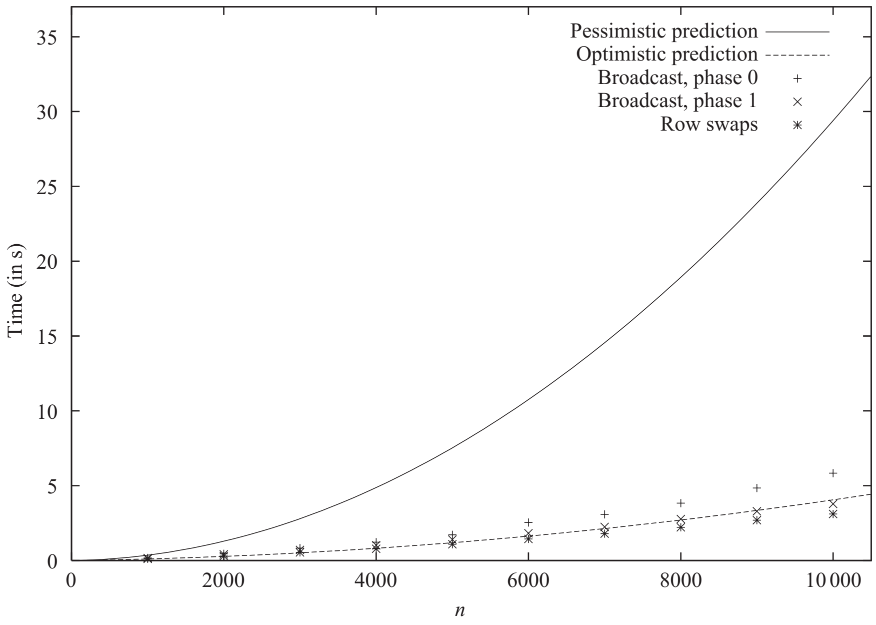


FIG. 2.7. Total measured time (shown as data points) of row swaps, broadcast phases 0 and broadcast phases 1 of LU decomposition on a 64-processor Cray T3E. Also given is the total predicted time (shown by lines).

$n^2g/\sqrt{p}+nl$. Figure 2.7 shows the measured time for these operations. In general, the three timing results for the same problem size are reasonably close to each other, which at least qualitatively confirms the prediction of the BSP model. This is particularly encouraging in view of the fact that the communication volumes involved are quite different: for instance, the communication volume of phase 1 is about $\sqrt{p}-1 = 7$ times that of phase 0. (We may conclude that communication volume is definitely not a good predictor of communication time, and that the BSP cost is a much better predictor.) We can also use the theoretical cost $n^2g/\sqrt{p}+nl$ together with benchmark results for r, g, and l, to predict the time of the three supersteps in a more precise, quantitative way. To do this, the BSP cost in flops is converted into a time in seconds by multiplying with $t_{\rm flop} = 1/r$. The result for the values obtained by `bspbench` is plotted in Fig. 2.7 as 'pessimistic prediction'. The reason for this title is obvious from the plot.

To explain the overestimate of the communication time, we note that the theoretical BSP model does not take **header overhead** into account, that is, the cost of sending address information together with the data themselves. The BSP cost model is solely based on the amount of data sent, not on that of the associated headers. In most practical cases, this matches reality, because the header overhead is often insignificant. If we benchmark g also in such a situation, and use this (lower) value of g, the BSP model will predict communication time well. We may call this value the **optimistic g-value**. This value is measured by `bspprobe`, see Section 1.8.3. If, however, the data are communicated by put or get operations of very small size, say less than five reals, such overhead becomes significant. In the extreme case of single words as data, for example, one real, we have a high header overhead, which is proportional to the amount of data sent. We can then just include this overhead in the cost of sending the data themselves, which leads to a higher, **pessimistic g-value**. This is the value measured by `bspbench`. In fact, the header overhead includes more than just the cost of sending header information; for instance, it also includes the overhead of a call to the `bsp_put` function. Such costs are conveniently lumped together. In the transition range, for data size in the range 1–5 reals, the BSP model does not accurately predict communication time, but we have an upper and a lower bound. It would be easy to extend the model to include an extra parameter (called the block size B in the BSP* model [15]), but this would be at the expense of simplicity. We shall stick to the simple BSP model, and rely on our common sense to choose between optimism and pessimism.

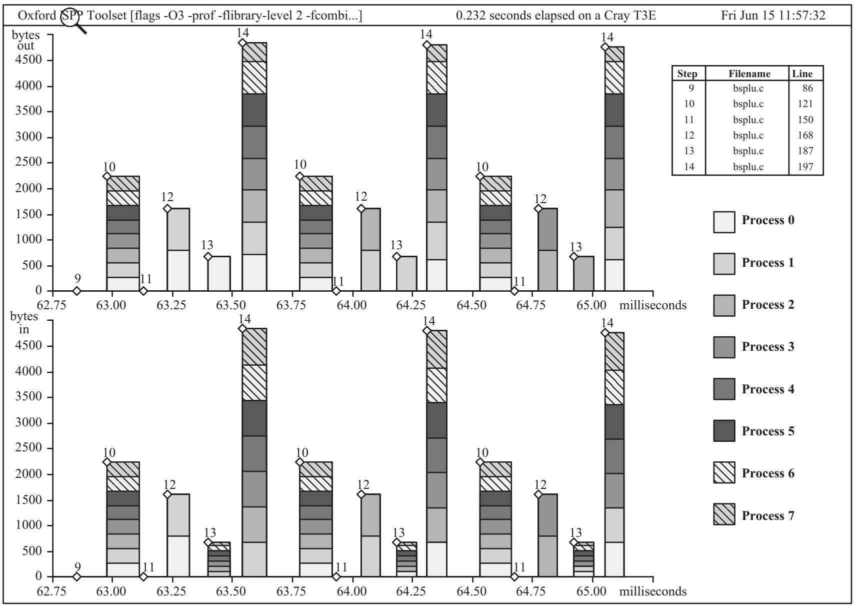
For LU decomposition, the optimistic g-value is appropriate since we send data in large blocks. Here, we measure the optimistic g-value by modifying `bspbench` to use puts of 16 reals (each of 64 bits), instead of single reals. This gives $g = 10.1$. Results with this g are plotted as 'optimistic prediction'. The figure shows that the optimistic prediction matches the measurements reasonably well.

We have looked at the total time of the LU decomposition and the total time of the different supersteps. Even more insight can be gained by examining the individual supersteps. The easiest way of doing this is to use the Oxford BSP toolset profiler, which is invoked by compiling with the option `-prof`. Running the resulting program creates `PROF.bsp`, a file that contains the statistics of every individual superstep carried out. This file is converted into a plot in Postscript format by the command

```
bspprof PROF.bsp
```

An example is shown in Fig. 2.8. We have zoomed in on the first three stages of the algorithm by using the zooming option, which specifies the starting and finishing time in seconds of the profile plot:

```
bspprof -zoom 0.06275,0.06525 PROF.bsp
```



Step	Filename	Line
9	bsplu.c	86
10	bsplu.c	121
11	bsplu.c	150
12	bsplu.c	168
13	bsplu.c	187
14	bsplu.c	197

FIG. 2.8. Profile of stages $k = 0, 1, 2$ of an LU decomposition with two-phase broadcast on an 8-processor Cray T3E, for $n = 100$, $M = 8$, $N = 1$. The horizontal axis shows the time. The vertical axis shows the communication of a superstep in bytes sent (top) and received (bottom). The height of the bar represents the total communication volume. The shaded parts of the bar represent the values $h_{\rm s}(s)$ (top) and $h_{\rm r}(s)$ (bottom) in bytes for the different processors $P(s)$. The width of the bar represents the communication time of the corresponding superstep. The distance between two bars represents the computation time of the corresponding superstep. The supersteps are numbered based on the program supersteps.

The absolute times given in the profile must be taken with a grain of salt, since the profiling itself adds some extra time.

The BSP profile of a program tells us where the communication time is spent, which processor is busiest communicating, and whether more time is spent communicating or computing. Because our profiling example concerns an LU decomposition with a row distribution ($M = 8, N = 1$), it is particularly easy to recognize what happens in the supersteps. Before reading on, try to guess which superstep is which.

Superstep 10 in the profile corresponds to program superstep 0, superstep 11 corresponds to program superstep 1, and so on. Superstep 10 communicates the local maxima found in the pivot search. For the small problem size of $n = 100$ this takes a significant amount of time, but for larger problems the time needed becomes negligible compared to the other parts of the program. Superstep 11 contains no communication because $N = 1$, so that the broadcast of the pivot index within a processor row becomes a copy operation within a processor. Superstep 12 represents the row swap; it has two active processors, namely the owners of rows k and $k + 1$. In stage 0, these are processors $P(0)$ and $P(1)$; in stage 1, $P(1)$ and $P(2)$; and in stage 2, $P(2)$ and $P(3)$. Superstep 13 represents the first phase of a row broadcast. In stage 0, $P(0)$ sends data and all other processors receive; in stage 1, $P(1)$ is the sender. Superstep 14 represents the second phase. In stage 0, $P(0)$ sends $\lceil 99/8 \rceil = 13$ row elements to seven other processors; $P(1)$–$P(6)$ each send 13 elements to six other processors (not to $P(0)$); and $P(7)$ sends the remaining 8 elements to six other processors. The number of bytes sent by $P(0)$ is $13 \cdot 7 \cdot 8 = 728$; by each of $P(1)$–$P(6)$, 624; and by $P(7)$, 384. The total is 4856 bytes. These numbers agree with the partitioning of the bars in the top part of the plot.

Returning to the fashion world, did the BSP model survive the catwalk? Guided by the BSP model, we obtained a theoretically superior algorithm with a better spread of the communication tasks over the processors. Our experiments show that this algorithm is also superior in practice, but that the benefits occur only in a certain range of problem sizes and that their impact is limited on our particular computer. The BSP model helped explaining our experimental results, and it can tell us when to expect significant benefits. The superstep concept of the BSP model helped us zooming in on certain parts of the computation and enabled us to understand what happens in those parts. Qualitatively speaking, we can say that the BSP model passed an important test. The BSP model also gave us a rough indication of the expected time for different parts of the algorithm. Unfortunately, to obtain this indication, we had to distinguish between two types of values for the communication parameter g, reflecting whether or not the put operations are extremely small. In most cases, we can (and should) avoid extremely small put operations, at least in the majority of our communication operations, so that we can use the optimistic g-value for predictions. Even then, the resulting prediction can easily be off by 50%, see Fig. 2.7.

A simple explanation for the remaining discrepancy between prediction and experiment is that there are lies, damned lies, and benchmarks. Substituting a benchmark result in a theoretical time formula gives an *ab initio* prediction, that is, a prediction from basic principles, and though this may be useful as an indication of expected performance, it will hardly ever be an accurate estimate. There are just too many possibilities for quirks in the hardware and the system software, ranging from obscure cache behaviour to inadequate implementation of certain communication primitives. Therefore, we should not have unrealistic quantitative expectations of a computation model.

2.7 Bibliographic notes

2.7.1 *Matrix distributions*

Almost all matrix distributions that have been proposed for use in parallel matrix computations are Cartesian. The term 'Cartesian' was first used in this context by Bisseling and van de Vorst [23] and Van de Velde [181] in articles on LU decomposition. Van de Velde even defines a matrix distribution as being Cartesian. (Still, non-Cartesian matrix distributions exist and they may sometimes be useful, see Exercise 1 and Chapter 4.) An early example of a Cartesian distribution is the cyclic row distribution, which is just the $p \times 1$ cyclic distribution. Chu and George [42] use this distribution to show that explicit row swaps lead to a good load balance during the whole LU decomposition and that the resulting savings in computation time outweigh the additional communication time. Geist and Romine [77] present a method for reducing the number of explicit row swaps (by a factor of two) while preserving good load balance. They relax the constraint of maintaining a strict cyclic distribution of the rows, by demanding only that successive sets of p pivot rows are evenly distributed over the p processors.

In 1985, O'Leary and Stewart [149] introduced the square cyclic distribution in the field of parallel matrix computations; they called it **torus assignment**. In their scheme, the matrix elements and the corresponding computations are assigned to a torus of processors, which executes a dataflow algorithm. Over the years, this distribution has acquired names such as **cyclic storage** [115], **grid distribution** [182], **scattered square decomposition** [71], and **torus-wrap mapping** [7], see also [98], and it has been used in a wide range of matrix computations. It seems that the name 'cyclic distribution' has been generally accepted now, and therefore we use this term.

Several algorithms based on the square cyclic distribution have been proposed for parallel LU decomposition. Van de Vorst [182] compares the square cyclic distribution with other distributions, such as the square block distribution, which allocates square submatrices of size $n/\sqrt{p} \times n/\sqrt{p}$ to processors.

The square block distribution leads to a bad load balance, because more and more processors become idle when the computation proceeds. As a result, the computation takes three times longer than with the square cyclic distribution. Fox *et al.* [71,Chapter 20] present an algorithm for LU decomposition of a banded matrix. They perform a theoretical analysis and give experimental results on a hypercube computer. (A matrix A is **banded** with **upper bandwidth** b_U and **lower bandwidth** b_L if $a_{ij} = 0$ for $i < j - b_U$ and $i > j + b_L$. A dense matrix can be viewed as a degenerate special case of a banded matrix, with $b_L = b_U = n - 1$.) Bisseling and van de Vorst [23] prove optimality with respect to load balance of the square cyclic distribution, within the class of Cartesian distributions. They also show that the communication time on a square mesh of processors is of the same order, $\mathcal{O}(n^2/\sqrt{p})$, as the load imbalance and that on a complete network the communication volume is $\mathcal{O}(n^2\sqrt{p})$. (For a BSP computer this would imply a communication cost of $\mathcal{O}(n^2 g/\sqrt{p})$, provided the communication can be balanced.) Extending these results, Hendrickson and Womble [98] show that the square cyclic distribution is advantageous for a large class of matrix computations, including LU decomposition, QR decomposition, and Householder tridiagonalization. They present experimental results for various ratios N/M of the $M \times N$ cyclic distribution.

A straightforward generalization of the cyclic distribution is the **block-cyclic distribution**, where the cyclic distribution is used to assign rectangular submatrices to processors instead of assigning single matrix elements. O'Leary and Stewart [150] proposed this distribution already in 1986, giving it the name **block-torus assignment**. It is now widely used, for example, in ScaLAPACK (Scalable Linear Algebra Package) [24,25,41] and in the object-oriented package PLAPACK (Parallel Linear Algebra Package) [4,180]. The $M \times N$ **block-cyclic distribution** with block size $b_0 \times b_1$ is defined by

$$\phi_0(i) = (i \text{ div } b_0) \bmod M, \; \phi_1(j) = (j \text{ div } b_1) \bmod N, \; \text{ for } 0 \leq i, j < n. \tag{2.33}$$

A good choice of the block size can improve efficiency. In general, larger block sizes lead to lower synchronization cost, but also to higher load imbalance. Usually, the block size does not affect the communication performance. The characteristics of the particular machine used and the problem size determine the best block size. In many cases, synchronization time is insignificant so that the best choice is $b_0 = b_1 = 1$, that is, the square cyclic distribution.

Note that blocking of a distribution has nothing to do with blocking of an algorithm, which is merely a way of organizing the (sequential or parallel) algorithm, usually with the aim of formulating it in terms of matrix operations, rather than operations on single elements or vectors. Blocking of algorithms makes it possible to attain peak computing speeds and hence algorithms are usually blocked in packages such as LAPACK (Linear Algebra PACKage) [6]; see also Exercise 10. Hendrickson, Jessup, and Smith [94] present a blocked parallel eigensystem solver based on the square cyclic distribution, and they argue in favour of blocking of algorithms but not of distributions.

2.7.2 *Collective communication*

The idea of spreading data before broadcasting them was already used by Barnett *et al.* [9,10], who call the two phases of their broadcast 'scatter' and 'collect'. The interprocessor Collective Communication (iCC) library is an implementation of broadcasting and related algorithms; it also implements hybrid methods for broadcasting vectors of intermediate length. In the BSP context, the cost analysis of a two-phase broadcast becomes easy. Juurlink [116] presents and analyses a set of communication primitives including broadcasts for different vector lengths. Bisseling [20] implements a two-phase broadcast and shows that it reduces communication time significantly in LU decomposition. This implementation, however, has not been optimized by sending data in blocks, and hence the g-values involved are pessimistic. The timing results in the present chapter are much better than in [20] (but as a consequence the gains of a two-phase broadcast are less clearly visible).

2.7.3 *Parallel matrix computations*

The handbook on matrix computations by Golub and Van Loan [79] is a standard reference for the field of numerical linear algebra. It treats sequential LU decomposition in full detail, discussing issues such as roundoff errors, partial and complete pivoting, matrix scaling, and iterative improvement. The book provides a wealth of references for further study. Chapter 6 of this handbook is devoted to parallel matrix computations, with communication by message passing using sends and receives. As is often done in message passing, it is assumed that communication takes $\alpha + \beta n$ time units for a message of length n, with α the startup time of the message and β the time per data word. The book by Dongarra, Duff, Sorensen, and van der Vorst [61] is a very readable introduction to numerical linear algebra on high-performance computers. It treats architectures, parallelization techniques, and the direct and iterative solution of linear systems and eigensystems, with particular attention to sparse systems.

Modern software for sequential LU decomposition and other matrix computations is provided by LAPACK [6], a parallel version of which is ScaLAPACK [24,25,41]. These widely used packages contain solvers for linear systems, eigenvalue systems, and linear least-squares problems, for dense symmetric and unsymmetric matrices. The packages also contain several solvers for banded, triangular, and tridiagonal matrices. ScaLAPACK is available on top of the portability layers PVM and MPI; Horvitz and Bisseling [110] discuss how ScaLAPACK can be ported to BSPlib and they show for LU decomposition that savings in communication time can be obtained by using BSPlib.

Several BSP algorithms have been designed for LU decomposition. Gerbessiotis and Valiant [78] present a BSP algorithm for Gauss–Jordan elimination, a matrix computation that is quite similar to LU decomposition; their algorithm uses the square cyclic distribution and a tree-based broadcast (which is less efficient than a two-phase broadcast). McColl [133] presents a BSP algorithm for LU decomposition without pivoting that is very different from other LU decomposition algorithms. He views the computation as a directed acyclic graph forming a regular three-dimensional mesh of $n \times n \times n$ vertices, where vertex (i, j, k) represents matrix element a_{ij} in stage k. The absence of pivoting allows for more flexibility in scheduling the computation. Instead of processing the vertices layer by layer, that is, stage after stage, as is done in other algorithms, McColl's algorithm processes the vertices in cubic blocks, with p blocks handled in parallel. This algorithm has time complexity $\mathcal{O}(n^3/p + n^2 g/\sqrt{p} + \sqrt{p}l)$, which may be attractive because of the low synchronization cost. The computation and communication cost are a constant factor higher than for Algorithm 2.8.

2.8 Exercises

1. Find a matrix distribution for parallel LU decomposition that is optimal with respect to computational load balance in all stages of the computation. The distribution need not be Cartesian. When would this distribution be applicable?

2. The ratio $N/M = 1$ is close to optimal for the $M \times N$ cyclic distribution used in Algorithm 2.8 and hence this ratio was assumed in our cost analysis. The optimal ratio, however, may be slightly different. This is mainly due to an asymmetry in the communication requirements of the algorithm. Explain this by using Table 2.2. Find the ratio N/M with the lowest communication cost, for a fixed number of processors $p = MN$. What is the reduction in communication cost for the optimal ratio, compared with the cost for the ratio $N/M = 1$?

3. Algorithm 2.8 contains a certain amount of unnecessary communication, because the matrix elements a_{rj} with $j > k$ are first swapped out and then spread and broadcast. Instead, they could have been spread already from their original location.

(a) How would you modify the algorithm to eliminate superfluous communication? How much communication cost is saved for the square cyclic distribution?

(b) Modify the function `bsplu` by incorporating this algorithmic improvement. Test the modified program for $n = 1000$. What is the resulting reduction in execution time? What is the price to be paid for this optimization?

4. Take a critical look at the benefits of two-phase broadcasting by first running `bsplu` on your own parallel computer for a range of problem sizes and then replacing the two-phase broadcast by a simple one-phase broadcast. Measure the run time for both programs and explain the difference. Gain more insight by timing the broadcast parts separately.

5. $(*)$ The **Cholesky factor** of a symmetric positive definite matrix A is defined as the lower triangular matrix L with positive diagonal entries that satisfies $A = LL^{\mathrm{T}}$. (A matrix A is **symmetric** if it equals its transpose, $A = A^{\mathrm{T}}$, and it is **positive definite** if $\mathbf{x}^{\mathrm{T}}A\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$.)

(a) Derive a sequential Cholesky factorization algorithm that is similar to the sequential LU decomposition algorithm without pivoting, Algorithm 2.2. The Cholesky algorithm should save half the flops, because it does not have to compute the upper triangular matrix L^{T}. Furthermore, pivoting is not needed in the symmetric positive definite case, see [79].

(b) Design a parallel Cholesky factorization algorithm that uses the $M \times N$ cyclic distribution. Take care to communicate only where needed. Analyse the BSP cost of your algorithm.

(c) Implement and test your algorithm.

6. $(*)$ Matrix–matrix multiplication is a basic building block in linear algebra computations. The highest computing rates can be achieved by constructing algorithms based on this operation. Consider the matrix product $C = AB$, where A, B, and C are $n \times n$ matrices. We can divide the matrices into submatrices of size $n/q \times n/q$, where we assume that $n \bmod q = 0$. Thus we can write $A = (A_{st})_{0 \le s,t < q}$, and similarly for B and C. We can express the matrix product in terms of submatrix products,

$$C_{st} = \sum_{u=0}^{q-1} A_{su}B_{ut}, \quad \text{for } 0 \le s,t < q. \tag{2.34}$$

On a sequential computer with a cache, this is even the best way of computing C, since we can choose the submatrix size such that two submatrices and their product fit into the cache. On a parallel computer, we can compute the product in two-dimensional or three-dimensional fashion; the latter approach was proposed by Aggarwal, Chandra, and Snir [3], as an example of the use of their LPRAM model. (See also [1] for experimental results and [133] for a BSP analysis.)

(a) Let $p = q^2$ be the number of processors. Assume that the matrices are distributed by the square block distribution, so that processor $P(s,t)$ holds the submatrices A_{st} and B_{st} on input, and C_{st} on output. Design a BSP algorithm for the computation of $C = AB$ where $P(s,t)$ computes C_{st}. Analyse the cost and memory use of your algorithm.

(b) Let $p = q^3$ be the number of processors. Design a BSP algorithm where processor $P(s,t,u)$ with $0 \le s,t,u < q$ computes the product $A_{su}B_{ut}$.

Choose a suitable distribution for the input and output matrices that spreads the data evenly. Analyse the cost and memory use of your algorithm. When is the three-dimensional algorithm preferred? Hint: assume that on input the submatrix A_{su} is distributed over $P(s, *, u)$.

7. $(**)$ Once upon a time, there was a Mainframe computer that had great difficulty in multiplying floating-point numbers and preferred to add or subtract them instead. So the Queen decreed that computations should be carried out with a minimum of multiplications. A young Prince, Volker Strassen [170], set out to save multiplications in the Queen's favourite pastime, computing the product of 2×2 matrices on the Royal Mainframe,

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}. \tag{2.35}$$

At the time, this took eight multiplications and four additions. The young Prince slew one multiplication, but at great cost: fourteen new additions sprang up. Nobody knew how he had obtained his method, but there were rumours [46], and indeed the Prince had drunk from the magic potion. Later, three additions were beheaded by the Princes Paterson and Winograd and the resulting Algorithm 2.9 was announced in the whole Kingdom. The Queen's subjects happily noted that the new method, with seven multiplications and fifteen additions, performed the same task as before. The Queen herself lived happily ever after and multiplied many more 2×2 matrices.

(a) Join the inhabitants of the Mainframe Kingdom and check that the task is carried out correctly.

(b) Now replace the matrix elements by submatrices of size $n/2 \times n/2$,

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}. \tag{2.36}$$

The Strassen method can be applied here as well, because it does not rely on the commutativity of the real numbers. (Commutativity, $ab = ba$, holds for real numbers a and b, but in general not for matrices A and B.) This should be beneficial, because multiplication of submatrices is much more expensive than addition of submatrices (and not only on mainframes!). The method requires the multiplication of smaller submatrices. This can again be done using Strassen's method, and so on, until the remaining problem is small and traditional matrix–matrix multiplication is used. The resulting matrix–matrix multiplication algorithm is the Strassen algorithm. We say that the method is applied **recursively**, that is, calling itself on smaller problem sizes. The number of times the original matrix size must be halved to reach the current matrix size is called the current **level** of the

Algorithm 2.9. Strassen 2×2 matrix–matrix multiplication.

```
input:     A : 2 × 2 matrix.
           B : 2 × 2 matrix.
output:    C : 2 × 2 matrix, C = AB.

$l_0 := a_{00}$;                    $r_0 := b_{00}$;
$l_1 := a_{01}$;                    $r_1 := b_{10}$;
$l_2 := a_{10} + a_{11}$;           $r_2 := b_{01} - b_{00}$;
$l_3 := a_{00} - a_{10}$;           $r_3 := b_{11} - b_{01}$;
$l_4 := l_2 - a_{00}$;              $r_4 := r_3 + b_{00}$;
$l_5 := a_{01} - l_4$;              $r_5 := b_{11}$;
$l_6 := a_{11}$;                    $r_6 := b_{10} - r_4$;

for $i := 0$ to 6 do
        $m_i := l_i r_i$;

$t_0 := m_0 + m_4$;
$t_1 := t_0 + m_3$;

$c_{00} := m_0 + m_1$;
$c_{01} := t_0 + m_2 + m_5$;
$c_{10} := t_1 + m_6$;
$c_{11} := t_1 + m_2$;
```

recursion. Assume that n is a power of two. Derive a formula that expresses $T(n)$, the time needed to multiply two $n \times n$ matrices, in terms of $T(n/2)$. Use this formula to count the number of flops of the Strassen algorithm with a switch to the traditional method at size $r \times r$, $1 \leq r \leq n$. (The traditional method computes an $r \times r$ matrix product by performing $r - 1$ floating-point additions and r floating-point multiplications for each of the r^2 elements of the output matrix, thus requiring a total of $2r^3 - r^2$ flops.)

(c) Prove that the Strassen algorithm with $r = 1$ requires $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$ flops, which scales better than the $\mathcal{O}(n^3)$ flops of traditional matrix–matrix multiplication. What is the optimal value of the switching parameter r and the corresponding total number of flops?

(d) Write a traditional matrix–matrix multiplication function and a recursive sequential function that implements the Strassen algorithm. Compare their accuracy for some test matrices.

(e) Recursion is a gift to the programmer, because it takes the burden of data management from his shoulders. For parallelism, recursion is not such a blessing, because it zooms in on one task to be carried out while a parallel computation tries to perform many tasks at the same time. (This is explained in more detail in Section 3.3, where a recursive fast Fourier transform is cast into nonrecursive form to prepare the ground for parallelization.) Write a nonrecursive sequential Strassen function that computes all matrices of one recursion level in the splitting part together, and does the same in the combining part. Hint: in the splitting part, the input of a level is an array $L_0, \ldots, L_{q-1}$ of $k \times k$ matrices and a corresponding array $R_0, \ldots, R_{q-1}$, where $q = 7^{\log_2(n/k)}$. The output is an array $L_0, \ldots, L_{7q-1}$ of $k/2 \times k/2$ matrices and a corresponding array $R_0, \ldots, R_{7q-1}$. How do the memory requirements grow with an increase in level? Generalize the function `matallocd` from BSPedupack to the three-dimensional case and use the resulting function to allocate space for arrays of matrices. Matrices of size $r \times r$ are not split any more; instead the products $M_i = L_i R_i$ are computed in the traditional manner. After these multiplications, the results are combined, in the reverse direction of the splittings.

(f) The addition of two $n/2 \times n/2$ submatrices of an $n \times n$ matrix A, such as the computation of $A_{10} + A_{11}$, is a fundamental operation in the Strassen algorithm. Assume this is done in parallel by p processors, with p a power of four and $\sqrt{p} \leq n/2$. Which distribution for A is better: the square block distribution or the square cyclic distribution? Why? (Loyens and Moonen [130] answered this question first.)

(g) Implement a parallel nonrecursive Strassen algorithm. Use the distribution found above for all matrices that occur in the splitting and combining parts of the Strassen algorithm. Redistribute the data before and after the multiplications $M_i := L_i R_i$ at matrix size $r' \times r'$. (This approach was proposed by McColl [134].) Perform each multiplication on one processor, using the sequential recursive Strassen algorithm and switching to the traditional algorithm at size $r \times r$, where $r \leq r'$.

(h) Analyse the BSP cost of the parallel algorithm. Find an upper bound for the load imbalance that occurs because p (a power of four) does not divide the number of multiplication tasks (a power of seven). Discuss the trade-off between computational load imbalance and communication cost and their scaling behaviour as a function of the number of splitting levels. In practice, how would you choose r'?

(i) Measure the run time of your program for different parameters n, p, r, r'. Explain your results.

8. $(**)$ Householder tridiagonalization decomposes an $n \times n$ symmetric matrix A into $A = Q_1 T Q_1^{\mathrm{T}}$, where Q_1 is an $n \times n$ orthogonal matrix and T an $n \times n$ symmetric tridiagonal matrix. (To recall some linear algebra: a square matrix Q is **orthogonal** if $Q^{\mathrm{T}} Q = I$, which is equivalent to $Q^{-1} = Q^{\mathrm{T}}$; a matrix T

is **tridiagonal** if $t_{ij} = 0$ for $|i - j| > 1$. Do not confuse the T denoting the tridiagonal matrix with the superscript denoting transposition!)

Tridiagonalization is often a stepping stone in achieving a larger goal, namely solving a real symmetric eigensystem. This problem amounts to decomposing a real symmetric matrix A into $A = QDQ^{\mathrm{T}}$, where Q is orthogonal and D is **diagonal**, that is, $d_{ij} = 0$ for $i \neq j$. The eigenvalues of A are the diagonal elements of D and the corresponding eigenvectors are the columns of Q. It is much easier to solve a symmetric tridiagonal eigensystem and decompose T into $T = Q_2 D Q_2^{\mathrm{T}}$ than to solve the original system. As a result, we obtain $A = (Q_1Q_2)D(Q_1Q_2)^{\mathrm{T}}$, which has the required form. Here, we concentrate on the tridiagonalization part of the eigensystem solution, which accounts for the majority of the flops. See [79] for more details.

(a) The central operation in Householder tridiagonalization is the application of a Householder reflection

$$P_{\mathbf{v}} = I_n - \frac{2}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{v}^{\mathrm{T}}. \tag{2.37}$$

Here, $\mathbf{v} \neq 0$ is a vector of length n with **Euclidean norm** $\|\mathbf{v}\| = \|\mathbf{v}\|_2 = (\sum_{i=0}^{n-1} v_i^2)^{1/2}$. For brevity, we drop the subscript '2' from the norm. Like all our vectors, $\mathbf{v}$ is a column vector, and hence it can also be viewed as an $n \times 1$ matrix. Note that $\mathbf{v}\mathbf{v}^{\mathrm{T}}$ represents an $n \times n$ matrix, in contrast to $\mathbf{v}^{\mathrm{T}}\mathbf{v}$, which is the scalar $\|\mathbf{v}\|^2$. Show that $P_{\mathbf{v}}$ is symmetric and orthogonal. We can apply $P_{\mathbf{v}}$ to a vector $\mathbf{x}$ and obtain

$$P_{\mathbf{v}}\mathbf{x} = \mathbf{x} - \frac{2}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{v}^{\mathrm{T}}\mathbf{x} = \mathbf{x} - \frac{2\mathbf{v}^{\mathrm{T}}\mathbf{x}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\mathbf{v}. \tag{2.38}$$

(b) Let $\mathbf{e} = (1, 0, 0, \ldots, 0)^{\mathrm{T}}$. Show that the choice $\mathbf{v} = \mathbf{x} - \|\mathbf{x}\|\mathbf{e}$ implies $P_{\mathbf{v}}\mathbf{x} = \|\mathbf{x}\|\mathbf{e}$. This means that we have an orthogonal transformation that sets all components of $\mathbf{x}$ to zero, except the first.

(c) Algorithm 2.10 is a sequential algorithm that determines a vector $\mathbf{v}$ such that $P_{\mathbf{v}}\mathbf{x} = \|\mathbf{x}\|\mathbf{e}$. For convenience, the algorithm also outputs the corresponding scalar $\beta = 2/\|\mathbf{v}\|^2$ and the norm of the input vector $\mu = \|\mathbf{x}\|$. The vector has been normalized such that $v_0 = 1$. For the memory-conscious, this can save one memory cell when storing $\mathbf{v}$. The algorithm contains a clever trick proposed by Parlett [154] to avoid subtracting nearly equal quantities (which would result in so-called **subtractive cancellation** and severe loss of significant digits). Now design and implement a parallel version of this algorithm. Assume that the input vector $\mathbf{x}$ is distributed by the cyclic distribution over p processors. The output vector $\mathbf{v}$ should become available in the same distribution. Try to keep communication to a minimum. What is the BSP cost?

Algorithm 2.10. Sequential Householder reflection.

```
input:          $\mathbf{x}$ : vector of length $n$, $n \geq 2$, $x(1\colon n-1) \neq 0$.
output:         $\mathbf{v}$ : vector of length $n$, such that $v_0 = 1$ and $P_{\mathbf{v}}\mathbf{x} = \|\mathbf{x}\|\mathbf{e}$,
                $\beta = 2/\|\mathbf{v}\|^2$,
                $\mu = \|\mathbf{x}\|$.
function call:  $(\mathbf{v}, \beta, \mu) :=$ Householder$(\mathbf{x}, n)$.

{Compute $\alpha = \|x(1\colon n-1)\|^2$ and $\mu$}
$\alpha := 0$;
for $i := 1$ to $n-1$ do
        $\alpha := \alpha + x_i^2$;
$\mu := \sqrt{x_0^2 + \alpha}$;

{Compute $\mathbf{v} = \mathbf{x} - \|\mathbf{x}\|\mathbf{e}$}
if $x_0 \leq 0$ then $v_0 := x_0 - \mu$ else $v_0 := \frac{-\alpha}{x_0+\mu}$;
for $i := 1$ to $n-1$ do
        $v_i := x_i$;

{Compute $\beta$ and normalize $\mathbf{v}$}
$\beta := \frac{2v_0^2}{v_0^2+\alpha}$;
for $i := 1$ to $n-1$ do
        $v_i := v_i/v_0$;
$v_0 := 1$;
```

(d) In stage k of the tridiagonalization, a Householder vector $\mathbf{v}$ is determined for column k of the current matrix A below the diagonal. The matrix is then transformed into $P_k A P_k$, where $P_k = \text{diag}(I_{k+1}, P_{\mathbf{v}})$ is a symmetric matrix. (The notation $\text{diag}(A_0, \ldots, A_r)$ stands for a block-diagonal matrix with blocks $A_0, \ldots, A_r$ on the diagonal.) This sets the elements a_{ik} with $i > k+1$ to zero, and also the elements a_{kj} with $j > k+1$; furthermore, it sets $a_{k+1,k}$ and $a_{k,k+1}$ to $\mu = \|A(k+1\colon n-1, k)\|$. The vector $\mathbf{v}$ without its first component can be stored precisely in the space of the zeros in column k. Our memory-frugality paid off! As a result, we obtain the matrix $Q_1 = P_{n-3} \cdots P_0$, but only in factored form: we have a record of all the $P_{\mathbf{v}}$ matrices used in the process. In most cases, this suffices and Q_1 never needs to be computed explicitly. To see how the current matrix is transformed efficiently into $P_k A P_k$, we only have to look at the submatrix $B = A(k+1\colon n-1, k+1\colon n-1)$, which is transformed into $P_{\mathbf{v}} B P_{\mathbf{v}}$. Prove that this matrix equals $B - \mathbf{v}\mathbf{w}^{\mathrm{T}} - \mathbf{w}\mathbf{v}^{\mathrm{T}}$, where $\mathbf{w} = \mathbf{p} - ((\beta \mathbf{p}^{\mathrm{T}}\mathbf{v})/2)\mathbf{v}$ with $\mathbf{p} = \beta B \mathbf{v}$.

(e) Algorithm 2.11 is a sequential tridiagonalization algorithm based on Householder reflections. Verify that this algorithm executes the method just described. The algorithm does not make use of symmetry yet, but

Algorithm 2.11. Sequential Householder tridiagonalization.

```
input:    A : n × n symmetric matrix, $A = A^{(0)}$.
output:   A : n × n symmetric matrix, $A = V + T + V^{\mathrm{T}}$, with
          T : n × n symmetric tridiagonal matrix,
          V : n × n matrix with $v_{ij} = 0$ for $i \le j+1$,
          such that $Q_1 T Q_1^{\mathrm{T}} = A^{(0)}$, where $Q_1 = P_{n-3}, \ldots, P_0$, with
          $P_k = \mathrm{diag}(I_{k+1}, P_{\mathbf{v}^{(k)}})$ and $\mathbf{v}^{(k)} = \begin{bmatrix} 1 \\ V(k+2\colon n-1, k) \end{bmatrix}$.

for $k := 0$ to $n-3$ do
    $(\mathbf{v}', \beta, \mu) :=$ Householder$(A(k+1\colon n-1, k), n-k-1)$;
    for $i := k+1$ to $n-1$ do
        $v_i := {v'}_{i-k-1}$; {shift for easier indexing}

    for $i := k+1$ to $n-1$ do
        $p_i := 0$;
        for $j := k+1$ to $n-1$ do
            $p_i := p_i + \beta a_{ij} v_j$;

    $\gamma := 0$;
    for $i := k+1$ to $n-1$ do
        $\gamma := \gamma + p_i v_i$;

    for $i := k+1$ to $n-1$ do
        $w_i := p_i - \frac{\beta\gamma}{2} v_i$;

    $a_{k+1,k} := \mu$; $a_{k,k+1} := \mu$;
    for $i := k+2$ to $n-1$ do
        $a_{ik} := v_i$; $a_{ki} := v_i$;
    for $i := k+1$ to $n-1$ do
        for $j := k+1$ to $n-1$ do
            $a_{ij} := a_{ij} - v_i w_j - w_i v_j$;
```

this is easy to achieve, by only performing operations on the lower triangular and diagonal part of A.

(f) Design, implement, and test a parallel version of this algorithm that exploits the symmetry. Assume that A is distributed by the square cyclic distribution, for similar reasons as in the case of LU decomposition. Why do these reasons apply here as well? Choose a suitable vector distribution, assuming that the vectors $\mathbf{v}$, $\mathbf{p}$, and $\mathbf{w}$ are distributed over all p processors, and that this is done in the same way for the three vectors. (We could have chosen to distribute the vectors in the same way as the input vector, that is, the column part $A(k+1\colon n-1, k)$. Why is this a bad idea?) Design the communication supersteps by following the need-to-know principle. Pay particular attention to the matrix–vector multiplication in the computation of $\mathbf{p}$. How many supersteps does this multiplication require? (Matrix–vector multiplication will be discussed extensively in Chapter 4.)

9. (**) Usually, the decomposition $PA = LU$ is followed by the solution of two triangular systems, $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$; this solves the linear system $A\mathbf{x} = \mathbf{b}$. In the parallel case, the distribution in which the triangular matrices L and U are produced by the LU decomposition must be used when solving the triangular systems, because it would be too expensive to redistribute the n^2 matrix elements involved, compared with the $2n^2$ flops required by the triangular system solutions.

(a) Design a basic parallel algorithm for the solution of a lower triangular system $L\mathbf{x} = \mathbf{b}$, where L is an $n \times n$ lower triangular matrix, $\mathbf{b}$ a given vector of length n, and $\mathbf{x}$ the unknown solution vector of length n. Assume that the number of processors is $p = M^2$ and that the matrix is distributed by the square cyclic distribution. Hint: the computation and communication can be organized in a wavefront pattern, where in stage k of the algorithm, computations are carried out for matrix elements l_{ij} with $i+j = k$. After these computations, communication is performed: the owner $P(s,t)$ of an element l_{ij} on the wavefront puts x_j into $P((s+1) \bmod M, t)$, which owns $l_{i+1,j}$, and it also puts $\sum_{r=0}^{j} l_{ir}x_r$ into $P(s, (t+1) \bmod M)$, which owns $l_{i,j+1}$.
(b) Reduce the amount of communication. Communicate only when this is really necessary.
(c) Which processors are working in stage k? Improve the load balance. Hint: procrastinate!
(d) Determine the BSP cost of the improved algorithm.
(e) Now assume that the matrix is distributed by the square block-cyclic distribution, defined by eqn (2.33) with $M = N = \sqrt{p}$ and $b_0 = b_1 = \beta$. How would you generalize your algorithm for solving lower triangular systems to this case? Determine the BSP cost of the generalized algorithm and find the optimal block size parameter β for a computer with given BSP parameters p, g, and l.
(f) Implement your algorithm for the square cyclic distribution in a function `bspltriang`. Write a similar function `bsputriang` that solves upper triangular systems. Combine `bsplu`, `bspltriang`, and `bsputriang` into one program `bsplinsol` that solves a linear system of equations $A\mathbf{x} = \mathbf{b}$. The program has to permute $\mathbf{b}$ into $P_{\pi^{-1}}\mathbf{b}$, where π is the partial pivoting permutation produced by the LU decomposition. Measure the execution time of the LU decomposition and the triangular system solutions for various p and n.

10. (**) The LU decomposition function `bsplu` is, well, educational. It teaches important distribution and communication techniques, but it is far from optimal. Our goal now is to turn `bsplu` into a fast program that is suitable for a production environment where every flop/s counts. We optimize

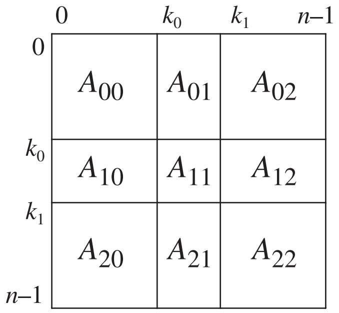


FIG. 2.9. Submatrices created by combining the operations from stages $k_0 \le k < k_1$ of the LU decomposition.

the program gradually, taking care that we can observe the effect of each modification. Measure the gains (or losses) achieved by each modification and explain your results.

(a) In parallel computing, postponing work until it can be done in bulk quantity creates opportunities for optimization. This holds for computation work as well as for communication work. For instance, we can combine several consecutive stages k, $k_0 \le k < k_1$, of the LU decomposition algorithm. As a first optimization, postpone all operations on the submatrix $A(*, k_1 : n-1)$ until the end of stage $k_1 - 1$, see Fig. 2.9. This concerns two types of operations: swapping elements as part of row swaps and modifying elements as part of matrix updates. Operations on the submatrix $A(*, 0 : k_1 - 1)$ are done as before. Carry out the postponed work by first permuting all rows involved in the row swaps and then performing a sequence of row broadcasts and matrix updates. This affects only the submatrices A_{12} and A_{22}. (We use the names of the submatrices as given in the figure when this is more convenient.)

To update the matrix correctly, the values of the columns $A(k+1 : n-1, k)$, $k_0 \le k < k_1$, that are broadcast must be stored in an array L immediately after the broadcast. Caution: the row swaps inside the submatrix $A(*, k_0 : k_1 - 1)$ will be carried out on the submatrix itself, but not on the copies created by the column broadcasts. At the end of stage $k_1 - 1$, the copy $L(i, k_0 : k_1 - 1)$ of row i, with elements frozen at various stages, may not be the same as the current row $A(i, k_0 : k_1 - 1)$. How many rows are affected in the worst case? Rebroadcast those rows. Do you need to rebroadcast in two phases? What is the extra cost incurred? Why is the new version of the algorithm still an improvement?

(b) Not all flops are created equal. Flops from matrix operations can often be performed at much higher rates than flops from vector or scalar operations. We can exploit this by postponing and then combining all updates of the submatrix A_{22} from the stages $k_0 \leq k < k_1$. Updates of the submatrix A_{12} are still carried out as in (a). Let $b = k_1 - k_0$ be the **algorithmic block size**. Formulate the combined matrix update in terms of the multiplication of an $(n - k_1) \times b$ matrix by a $b \times (n - k_1)$ matrix. Modify `bsplu` accordingly, and write your own version of the `DGEMM` function from the BLAS library to perform the multiplication.

The syntax of `DGEMM` is

```
DGEMM (transa, transb, m, n, k, alpha, a, lda,
       b, ldb, beta, c, ldc);
```

This corresponds to the operation $C := \alpha \hat{A}\hat{B} + \beta C$, where $\hat{A}$ is an $m \times k$ matrix, $\hat{B}$ a $k \times n$ matrix, C an $m \times n$ matrix, and α and β are scalars. Here, $\hat{A} = A$ if `transa='n'` and $\hat{A} = A^{\mathrm{T}}$ if `transa='t'`, where the matrix A is the matrix that is actually stored in the memory, and similarly for $\hat{B}$. The integer `lda` is the leading dimension of the matrix A, that is, `lda` determines how the two-dimensional matrix A is stored in the one-dimensional array `a`: element a_{ij} is stored in position $i + j\cdot$`lda`. Similarly, `ldb` and `ldc` determine the storage format of B and C.

A complication arises because the data format assumed by `DGEMM` is that of the Fortran language, with matrices stored by columns, whereas in the C language matrices are stored by rows. To hand over a matrix A stored by rows from the calling C program to the Fortran-speaking `DGEMM` subroutine, we just tell the subroutine that it receives the matrix A^{T} stored by columns. Therefore, the subroutine should perform the update $C^{\mathrm{T}} := \alpha \hat{B}^{\mathrm{T}}\hat{A}^{\mathrm{T}} + \beta C^{\mathrm{T}}$.

(c) Try to find a version of `DGEMM` that is tailored to your machine. (Most machine vendors provide extremely efficient BLAS in assembler language; `DGEMM` is their showcase function which should approach theoretical peak performance.) Find the optimal block size b using the vendor's `DGEMM`, for a suitable choice of n and p. Explain the relation between block size and performance.

(d) How can you achieve a perfect load balance in the update of the matrix A_{22}?

(e) Where possible, replace other computations by calls to BLAS functions. In the current version of the program, how much time is spent in computation? How much in communication?

(f) Postpone all row swaps in the submatrix $A(*, 0\colon k_0 - 1)$ until the end of stage $k_1 - 1$, and then perform them in one large row permutation. Avoid synchronization by combining the resulting superstep with another superstep. This approach may be called **superstep piggybacking**.

(g) The high-performance put function bsp_hpput of BSPlib has exactly the same syntax as the bsp_put function:

```
bsp_hpput(pid, source, dest, offset, nbytes);
```

It does not provide the safety of buffering at the source and destination that bsp_put gives. The read and write operations can in principle occur at any time during the superstep. Therefore the user must ensure safety by taking care that different communication operations do not interfere. The primary aim of using this primitive is to save the memory of the buffers. Sometimes, this makes the difference between being able to solve a problem or not. A beneficial side effect is that this saves time as well. There also exists a bsp_hpget operation, with syntax

```
bsp_hpget(pid, source, offset, dest, nbytes);
```

which should be used with the same care as bsp_hpput. In the LU decomposition program, matrix data are often put into temporary arrays and not directly into the matrix itself, so that there is no need for additional buffering by the system. Change the bsp_puts into bsp_hpputs, wherever this is useful and allowed, perhaps after a few minor modifications. What is the effect?

(h) For short vectors, a one-phase broadcast is faster than a two-phase broadcast. Replace the two-phase broadcast in stage k of row elements a_{kj} with $k < j < k_1$ by a one-phase broadcast. For which values of b is this an improvement?

(i) As already observed in Exercise 3, a disadvantage of the present approach to row swaps and row broadcasts in the submatrix $A(k_0 : n-1, k_1 : n-1)$ is that elements of pivot rows move three times: each such element is first moved into the submatrix A_{12} as part of a permutation; then it is moved as part of the data spreading operation in the first phase of the row broadcast; and finally it is copied and broadcast in the second phase. This time, there are b rows instead of one that suffer from excessive mobility, and they can be dealt with together. Instead of moving the local part of a row into A_{12}, you should spread it over the M processors of its processor column (in the same way for every row). As a result, A_{12} becomes distributed over all p processors in a column distribution. Updating A_{12} becomes a local operation, provided each processor has a copy of the lower triangular part of A_{11}. How much does this approach save?

(j) Any ideas for further improvement?

3

THE FAST FOURIER TRANSFORM

This chapter demonstrates the use of different data distributions in different phases of a computation: we use both the block and cyclic distributions of a vector and also intermediates between them. Each data redistribution is a permutation that requires communication. By making careful choices, the number of such redistributions can be kept to a minimum. This approach is demonstrated for the fast Fourier transform (FFT), a regular computation with a predictable but challenging data access pattern. Furthermore, the chapter shows how permutations with a regular pattern can be implemented more efficiently. These techniques are demonstrated in the specific case of the FFT, but they are applicable to other regular computations on data vectors as well. After having read this chapter, you are able to design and implement parallel algorithms for a range of related regular computations, including wavelet transforms, sine and cosine transforms, and convolutions; you will also be able to incorporate these algorithms in larger programs, for example, for weather forecasting or signal processing. Furthermore, you will be able to present the results of numerical experiments in a meaningful manner using the metrics of speedup and efficiency.

3.1 The problem

Fourier analysis studies the decomposition of functions into their frequency components. The functions may represent a piano sonata by Mozart recorded 50 years ago, a blurred picture of a star taken by the Hubble Space Telescope before its mirrors were repaired, or a Computerized Tomography (CT) scan of your chest. It is often easier to improve a function if we can work directly with its frequency components. Enhancing desired frequencies or removing undesired ones makes the music more pleasing to your ears. Fourier methods help deblurring the satellite picture and they are crucial in reconstructing a medical image from the tomographic measurements.

Let $f : \mathbf{R} \to \mathbf{C}$ be a ***T*-periodic** function, that is, a function with $f(t+T) = f(t)$ for all $t \in \mathbf{R}$. The Fourier series associated with f is defined by

$$\tilde{f}(t) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{2\pi ikt/T}, \tag{3.1}$$

where the Fourier coefficients c_k are given by

$$c_k = \frac{1}{T}\int_0^T f(t)\mathrm{e}^{-2\pi ikt/T}\,dt \tag{3.2}$$

and i denotes the complex number with $i^2 = -1$. (To avoid confusion, we ban the index i from this chapter.) Under relatively mild assumptions, such as piecewise smoothness, it can be proven that the Fourier series converges for every t. (A function is called **smooth** if it is continuous and its derivative is also continuous. A property is said to hold **piecewise** if each finite interval of its domain can be cut up into a finite number of pieces where the property holds; it need not hold in the end points of the pieces.) A piecewise smooth function satisfies $\tilde{f}(t) = f(t)$ in points of continuity; in the other points, $\tilde{f}$ is the average of the left and right limit of f. (For more details, see [33].) If f is real-valued, we can use Euler's formula $\mathrm{e}^{i\theta} = \cos\theta + i\sin\theta$, and eqns (3.1) and (3.2) to obtain a real-valued Fourier series expressed in sine and cosine functions.

On digital computers, signal or image functions are represented by their values at a finite number of sample points. A compact disc contains 44 100 sample points for each second of recorded music. A high-resolution digital image may contain 1024 by 1024 picture elements (**pixels**). On an unhappy day in the future, you might find your chest being cut by a CT scanner into 40 slices, each containing 512 by 512 pixels. In all these cases, we obtain a discrete approximation to the continuous world.

Suppose we are interested in computing the Fourier coefficients of a T-periodic function f which is sampled at n points $t_j = jT/n$, with $j = 0, 1, \ldots, n-1$. Using the trapezoidal rule for numerical integration on the interval $[0, T]$ and using $f(0) = f(T)$, we obtain an approximation

$$\begin{aligned} c_k &= \frac{1}{T}\int_0^T f(t)\mathrm{e}^{-2\pi ikt/T}\,dt \\ &\approx \frac{1}{T}\cdot\frac{T}{n}\left(\frac{f(0)}{2} + \sum_{j=1}^{n-1} f(t_j)\mathrm{e}^{-2\pi ikt_j/T} + \frac{f(T)}{2}\right) \\ &= \frac{1}{n}\sum_{j=0}^{n-1} f(t_j)\mathrm{e}^{-2\pi ijk/n}. \end{aligned} \tag{3.3}$$

The **discrete Fourier transform** (DFT) of a vector $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}} \in \mathbf{C}^n$ can be defined as the vector $\mathbf{y} = (y_0, \ldots, y_{n-1})^{\mathrm{T}} \in \mathbf{C}^n$ with

$$y_k = \sum_{j=0}^{n-1} x_j \mathrm{e}^{-2\pi ijk/n}, \quad \text{for } 0 \le k < n. \tag{3.4}$$

(Different conventions exist regarding the sign of the exponent.) Thus, eqn (3.3) has the form of a DFT, with $x_j = f(t_j)/n$ for $0 \le j < n$. It is easy to see that the inverse DFT is given by

$$x_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \mathrm{e}^{2\pi ijk/n}, \quad \text{for } 0 \le j < n. \tag{3.5}$$

A straightforward implementation of eqn (3.4) would require $n-1$ complex additions and n complex multiplications for each vector component y_k, assuming that factors of the form $\mathrm{e}^{-2\pi im/n}$ have been precomputed and are available in a table. A complex addition has the form $(a+bi)+(c+di) = (a+c)+(b+d)i$, which requires two real additions. A complex multiplication has the form $(a+bi)(c+di) = (ac-bd)+(ad+bc)i$, which requires one real addition, one real subtraction, and four real multiplications, that is, a total of six flops. Therefore, the straightforward computation of the DFT costs $n(2(n-1)+6n) = 8n^2 - 2n$ flops.

It is often convenient to use matrix notation to express DFT algorithms. Define the $n \times n$ **Fourier matrix** F_n by

$$(F_n)_{jk} = \omega_n^{jk}, \quad \text{for } 0 \le j, k < n, \tag{3.6}$$

where

$$\omega_n = \mathrm{e}^{-2\pi i/n}. \tag{3.7}$$

Figure 3.1 illustrates the powers of ω_n occurring in the Fourier matrix; these are sometimes called the **roots of unity**.

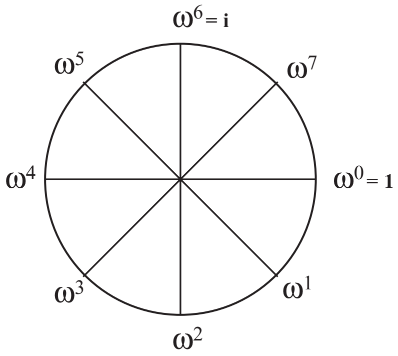


FIG. 3.1. Roots of unity ω^k, with $\omega = \omega_8 = \mathrm{e}^{-2\pi i/8}$, shown in the complex plane.

Example 3.1 Let $n = 4$. Because $\omega_4 = \mathrm{e}^{-2\pi i/4} = \mathrm{e}^{-\pi i/2} = -i$, it follows that

$$F_4 = \begin{bmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}. \tag{3.8}$$

Useful properties of the Fourier matrix are $F_n^{\mathrm{T}} = F_n$ and $F_n^{-1} = \overline{F_n}/n$, where the bar on F_n denotes the complex conjugate. The transform $\mathbf{y} := \mathrm{DFT}(\mathbf{x})$ can be written as $\mathbf{y} := F_n\mathbf{x}$, so that the DFT becomes a matrix–vector multiplication. The computation of the DFT is the problem studied in this chapter.

3.2 Sequential recursive fast Fourier transform

The FFT is a fast algorithm for the computation of the DFT. The basic idea of the algorithm is surprisingly simple, which does not mean that it is easy to discover if you have not seen it before. In this section, we apply the idea recursively, as was done by Danielson and Lanczos [51] in 1942. (A method is **recursive** if it invokes itself, usually on smaller problem sizes.)

Assume that n is even. We split the sum of eqn (3.4) into sums of even-indexed and odd-indexed terms, which gives

$$y_k = \sum_{j=0}^{n-1} x_j\omega_n^{jk} = \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2jk} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{(2j+1)k}, \quad \text{for } 0 \le k < n. \tag{3.9}$$

By using the equality $\omega_n^2 = \omega_{n/2}$, we can rewrite (3.9) as

$$y_k = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk} + \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk}, \quad \text{for } 0 \le k < n. \tag{3.10}$$

In the first sum, we recognize a Fourier transform of length $n/2$ of the even components of $\mathbf{x}$. To cast the sum exactly into this form, we must restrict the output indices to the range $0 \le k < n/2$. In the second sum, we recognize a transform of the odd components. This leads to a method for computing the set of coefficients y_k, $0 \le k < n/2$, which uses two Fourier transforms of length $n/2$. To obtain a method for computing the remaining coefficients y_k, $n/2 \le k < n$, we have to rewrite eqn (3.10). Let $k' = k - n/2$, so that $0 \le k' < n/2$. Substituting $k = k' + n/2$ in eqn (3.10) gives

$$y_{k'+n/2} = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{j(k'+n/2)} + \omega_n^{k'+n/2} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{j(k'+n/2)}, \quad \text{for } 0 \le k' < n/2. \tag{3.11}$$

By using the equalities $\omega_{n/2}^{n/2} = 1$ and $\omega_n^{n/2} = -1$, and by dropping the primes we obtain

$$y_{k+n/2} = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk} - \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk}, \quad \text{for } 0 \leq k < n/2. \qquad (3.12)$$

Comparing eqns (3.10) and (3.12), we see that the sums appearing in the right-hand sides are the same; if we add the sums we obtain y_k and if we subtract them we obtain $y_{k+n/2}$. Here, the savings become apparent: we need to compute the sums only once.

Following the basic idea, we can compute a Fourier transform of length n by first computing two Fourier transforms of length $n/2$ and then combining the results. Combining requires $n/2$ complex multiplications, $n/2$ complex additions, and $n/2$ complex subtractions, that is, a total of $(6+2+2) \cdot n/2 = 5n$ flops. If we use the DFT for the half-length Fourier transforms, the total flop count is already reduced from $8n^2 - 2n$ to $2 \cdot [8(n/2)^2 - 2(n/2)] + 5n = 4n^2 + 3n$, thereby saving almost a factor of two in computing time. Of course, we can apply the idea recursively, computing the half-length transforms by the same splitting method. The recursion ends when the input length becomes odd; in that case, we switch to a straightforward DFT algorithm. If the original input length is a power of two, the recursion ends with a DFT of length one, which is just a trivial copy operation $y_0 := x_0$. Figure 3.2 shows how the problem is split up recursively for $n = 8$. Algorithm 3.1 presents the recursive FFT algorithm for an arbitrary input length.

For simplicity, we assume from now on that the original input length is a power of two. The flop count of the recursive FFT algorithm can be computed

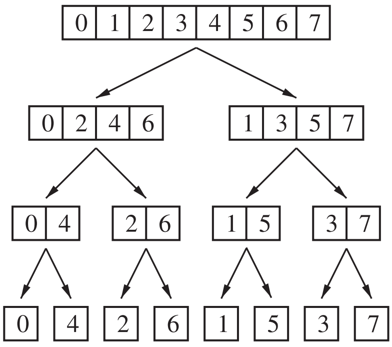


FIG. 3.2. Recursive computation of the DFT for $n = 8$. The numbers shown are the indices in the original vector, that is, the number j denotes the index of the vector component x_j (and not the numerical value). The arrows represent the splitting operation. The combining operation is executed in the reverse direction of the arrows.

Algorithm 3.1. Sequential recursive FFT.

```
input:          x : vector of length $n$.
output:         y : vector of length $n$, $\mathbf{y} = F_n\mathbf{x}$.
function call:  $\mathbf{y} := \mathrm{FFT}(\mathbf{x}, n)$.

if $n \bmod 2 = 0$ then
        $\mathbf{x}^{\mathrm{e}} := x(0\colon 2\colon n-1)$;
        $\mathbf{x}^{\mathrm{o}} := x(1\colon 2\colon n-1)$;
        $\mathbf{y}^{\mathrm{e}} := \mathrm{FFT}(\mathbf{x}^{\mathrm{e}}, n/2)$;
        $\mathbf{y}^{\mathrm{o}} := \mathrm{FFT}(\mathbf{x}^{\mathrm{o}}, n/2)$;
        for $k := 0$ to $n/2 - 1$ do
                $\tau := \omega_n^k y_k^{\mathrm{o}}$;
                $y_k := y_k^{\mathrm{e}} + \tau$;
                $y_{k+n/2} := y_k^{\mathrm{e}} - \tau$;
else $\mathbf{y} := \mathrm{DFT}(\mathbf{x}, n)$;
```

as follows. Let $T(n)$ be the number of flops of an FFT of length n. Then

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n, \tag{3.13}$$

because an FFT of length n requires two FFTs of length $n/2$ and the combination of the results requires $5n$ flops. Since the half-length FFTs are split again, we substitute eqn (3.13) in itself, but with n replaced by $n/2$. This gives

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + 5\frac{n}{2}\right) + 5n = 4T\left(\frac{n}{4}\right) + 2\cdot 5n. \tag{3.14}$$

Repeating this process until the input length becomes one, and using $T(1) = 0$, we obtain

$$T(n) = nT(1) + (\log_2 n)\cdot 5n = 5n\log_2 n. \tag{3.15}$$

The gain of the FFT compared with the straightforward DFT is huge: only $5n\log_2 n$ flops are needed instead of $8n^2 - 2n$. For example, you may be able to process a sound track of $n = 32\,768$ samples (about 0.74 s on a compact disc) on your personal computer in real time by using FFTs, but you would have to wait 43 min if you decided to use DFTs instead.

3.3 Sequential nonrecursive algorithm

With each recursive computation, a computational tree is associated. The nodes of the tree are the calls to the recursive function that performs the computation. The root of the tree is the first call and the leaves are the calls that do not invoke the recursive function themselves. Figure 3.2 shows the tree for an FFT of length eight; the tree is binary, since each node has at most two children. The tree-like nature of recursive computations may lead you into thinking that such algorithms are straightforward to parallelize. Indeed, it is clear that the computation can be split up easily. A difficulty arises, however, because a recursive algorithm traverses its computation tree sequentially, visiting different subtrees one after the other. For a parallel algorithm, we ideally would like to access many subtrees simultaneously. A first step towards parallelization of a recursive algorithm is therefore to reformulate it in nonrecursive form. The next step is then to split and perhaps reorganize the computation. In this section, we derive a nonrecursive FFT algorithm, which is known as the Cooley–Tukey algorithm [45].

Van Loan [187] presents a unifying framework in which the Fourier matrix F_n is factorized as the product of permutation matrices and structured sparse matrices. This helps in concisely formulating FFT algorithms, classifying the huge amount of existing FFT variants, and identifying the fundamental variants. We adopt this framework in deriving our parallel algorithm.

The computation of $F_n\mathbf{x}$ by the recursive algorithm can be expressed in matrix language as

$$F_n\mathbf{x} = \begin{bmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{bmatrix} \begin{bmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{bmatrix} \begin{bmatrix} x(0\colon 2\colon n-1) \\ x(1\colon 2\colon n-1) \end{bmatrix}. \tag{3.16}$$

Here, Ω_n denotes the $n \times n$ diagonal matrix with the first n powers of ω_{2n} on the diagonal,

$$\Omega_n = \mathrm{diag}(1, \omega_{2n}, \omega_{2n}^2, \ldots, \omega_{2n}^{n-1}). \tag{3.17}$$

Please verify that eqn (3.16) indeed corresponds to Algorithm 3.1.

We examine the three parts of the right-hand side of eqn (3.16) starting from the right. The rightmost part is just the vector $\mathbf{x}$ with its components sorted into even and odd components. We define the **even–odd sort matrix** S_n by

$$S_n = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ & & \vdots & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ & & \vdots & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}, \tag{3.18}$$

that is, S_n is the $n \times n$ permutation matrix that contains the even rows of I_n followed by the odd rows. (Note that the indices start at zero, so that the even rows are rows $0, 2, 4, \ldots, n-2$.) Using this notation, we can write

$$S_n\mathbf{x} = \begin{bmatrix} x(0\colon 2\colon n-1) \\ x(1\colon 2\colon n-1) \end{bmatrix}. \tag{3.19}$$

The middle part of the right-hand side of eqn (3.16) is a block-diagonal matrix with two identical blocks $F_{n/2}$ on the diagonal. The off-diagonal blocks, which are zero, can be interpreted as 0 times the submatrix $F_{n/2}$. The matrix therefore consists of four submatrices that are scaled copies of the submatrix $F_{n/2}$. In such a situation, it is convenient to use the Kronecker matrix product notation. If A is a $q \times r$ matrix and B an $m \times n$ matrix, then the **Kronecker product** (also called **tensor product**, or **direct product**) $A \otimes B$ is the $qm \times rn$ matrix defined by

$$A \otimes B = \begin{bmatrix} a_{00}B & \cdots & a_{0,r-1}B \\ \vdots & & \vdots \\ a_{q-1,0}B & \cdots & a_{q-1,r-1}B \end{bmatrix}. \tag{3.20}$$

Example 3.2 Let $A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \end{bmatrix}$. Then

$$A \otimes B = \begin{bmatrix} 0 & B \\ 2B & 4B \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 4 & 4 & 0 & 8 \\ 0 & 2 & 0 & 0 & 4 & 0 \end{bmatrix}.$$

The Kronecker product has many useful properties (but, unfortunately, it does not possess commutativity). For an extensive list, see Van Loan [187]. Here, we only mention the three properties that we shall use.

Lemma 3.3 *Let A, B, C be matrices. Then*

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

Lemma 3.4 *Let A, B, C, D be matrices such that AC and BD are defined. Then*

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD).$$

Lemma 3.5 *Let $m, n \in \mathbf{N}$. Then*

$$I_m \otimes I_n = I_{mn}.$$

Proof Boring. □

Lemma 3.3 saves some ink because we can drop brackets and write $A \otimes B \otimes C$ instead of having to give an explicit evaluation order such as $(A \otimes B) \otimes C$.

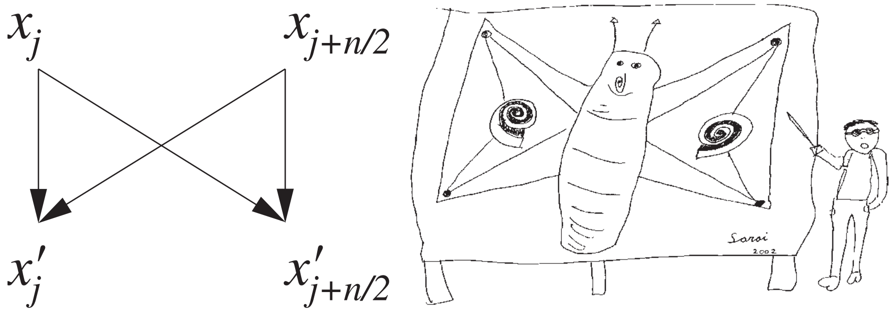


Fig. 3.3. Butterfly operation transforming an input pair $(x_j, x_{j+n/2})$ into an output pair $(x'_j, x'_{j+n/2})$. Right butterfly: © 2002 Sarai Bisseling, reproduced with sweet permission.

Using the Kronecker product notation, we can write the middle part of the right-hand side of eqn (3.16) as

$$I_2 \otimes F_{n/2} = \begin{bmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{bmatrix}. \tag{3.21}$$

The leftmost part of the right-hand side of eqn (3.16) is the $n \times n$ **butterfly matrix**

$$B_n = \begin{bmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{bmatrix}. \tag{3.22}$$

The butterfly matrix obtains its name from the butterfly-like pattern in which it transforms input pairs $(x_j, x_{j+n/2}), 0 \leq j < n/2$, into output pairs, see Fig. 3.3. The butterfly matrix is sparse because only $2n$ of its n^2 elements are nonzero. It is also structured, because its nonzeros form three diagonals.

Example 3.6

$$B_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & i \end{bmatrix}.$$

Using our new notations, we can rewrite eqn (3.16) as

$$F_n\mathbf{x} = B_n(I_2 \otimes F_{n/2})S_n\mathbf{x}. \tag{3.23}$$

Since this holds for all vectors $\mathbf{x}$, we obtain the matrix factorization

$$F_n = B_n(I_2 \otimes F_{n/2})S_n, \tag{3.24}$$

which expresses the Fourier matrix in terms of a smaller Fourier matrix. We can reduce the size of the smaller Fourier matrix further by repeatedly factorizing the middle factor of the right-hand side. For a factor of the form $I_k \otimes F_{n/k}$, this is done by applying Lemmas 3.4 (twice), 3.3, and 3.5, giving

$$\begin{aligned} I_k \otimes F_{n/k} &= [I_k I_k I_k] \otimes \left[B_{n/k}(I_2 \otimes F_{n/(2k)})S_{n/k}\right] \\ &= \left(I_k \otimes B_{n/k}\right)([I_k I_k] \otimes \left[(I_2 \otimes F_{n/(2k)})S_{n/k}\right]) \\ &= (I_k \otimes B_{n/k})(I_k \otimes I_2 \otimes F_{n/(2k)})(I_k \otimes S_{n/k}) \\ &= (I_k \otimes B_{n/k})(I_{2k} \otimes F_{n/(2k)})(I_k \otimes S_{n/k}). \end{aligned} \tag{3.25}$$

After repeatedly eating away at the middle factor, from both sides, we finally reach $I_n \otimes F_{n/n} = I_n \otimes I_1 = I_n$. Collecting the factors produced in this process, we obtain the following theorem, which is the so-called **decimation in time** (DIT) variant of the Cooley–Tukey factorization. (The name 'DIT' comes from splitting—decimating—the samples taken over time, cf. eqn (3.9).)

Theorem 3.7 (Cooley and Tukey [45]—DIT) *Let n be a power of two with $n \geq 2$. Then*

$$F_n = (I_1 \otimes B_n)(I_2 \otimes B_{n/2})(I_4 \otimes B_{n/4}) \cdots (I_{n/2} \otimes B_2)R_n,$$

where

$$R_n = (I_{n/2} \otimes S_2) \cdots (I_4 \otimes S_{n/4})(I_2 \otimes S_{n/2})(I_1 \otimes S_n).$$

Note that the factors $I_k \otimes S_{n/k}$ are permutation matrices, so that R_n is a permutation matrix.

Example 3.8

$$R_8 = (I_4 \otimes S_2)(I_2 \otimes S_4)(I_1 \otimes S_8) = (I_4 \otimes I_2)(I_2 \otimes S_4)S_8 = (I_2 \otimes S_4)S_8$$

$$= \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}.$$

The permutation matrix R_n is known as the **bit-reversal** matrix. Multiplying an input vector by this matrix first permutes the vector by splitting it into even and odd components, moving the even components to the front; then treats the two parts separately in the same way, splitting each half into its own even and odd components; and so on.

The name 'bit reversal' stems from viewing this permutation in terms of binary digits. We can write an index j, $0 \le j < n$, as

$$j = \sum_{k=0}^{m-1} b_k 2^k, \tag{3.26}$$

where $b_k \in \{0, 1\}$ is the kth bit and $n = 2^m$. We call b_0 the **least significant bit** and b_{m-1} the **most significant bit**. We express the binary expansion by the notation

$$(b_{m-1} \cdots b_1 b_0)_2 = \sum_{k=0}^{m-1} b_k 2^k. \tag{3.27}$$

Example 3.9

$$(10100101)_2 = 2^7 + 2^5 + 2^2 + 2^0 = 165.$$

Multiplying a vector by R_n starts by splitting the vector into a subvector of components $x_{(b_{m-1}\cdots b_0)_2}$ with $b_0 = 0$ and a subvector of components with $b_0 = 1$. This means that the most significant bit of the new position of a component becomes b_0. Each subvector is then split according to bit b_1, and so on. Thus, the final position of the component with index $(b_{m-1} \cdots b_0)_2$ becomes $(b_0 \cdots b_{m-1})_2$, that is, the bit reverse of the original position; hence the name. The splittings of the bit reversal are exactly the same as those of the recursive procedure, but now they are lumped together. For this reason, Fig. 3.2 can also be viewed as an illustration of the bit-reversal permutation, where the bottom row gives the bit reverses of the vector components shown at the top.

The following theorem states formally that R_n corresponds to a bit-reversal permutation ρ_n, where the correspondence between a permutation σ and a permutation matrix P_σ is given by eqn (2.10). The permutation for $n = 8$ is displayed in Table 3.1.

Theorem 3.10 *Let $n = 2^m$, with $m \ge 1$. Let $\rho_n : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ be the bit-reversal permutation defined by*

$$\rho_n((b_{m-1} \cdots b_0)_2) = (b_0 \cdots b_{m-1})_2.$$

Then

$$R_n = P_{\rho_n}.$$

Proof First, we note that

$$(S_n \mathbf{x})_{(b_{m-1}\cdots b_0)_2} = x_{(b_{m-2}\cdots b_0 b_{m-1})_2}, \tag{3.28}$$

TABLE 3.1. Bit-reversal permutation for $n = 8$

j	$(b_2b_1b_0)_2$	$(b_0b_1b_2)_2$	$\rho_8(j)$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

for all binary numbers $(b_{m-1}\cdots b_0)_2$ of m bits. This is easy to see: if $j = (b_{m-1}\cdots b_0)_2 < n/2$, then $b_{m-1} = 0$ and $j = (b_{m-2}\cdots b_0)_2$ so that $(S_n\mathbf{x})_j = x_{2j} = x_{(b_{m-2}\cdots b_0 b_{m-1})_2}$. The proof for $j \geq n/2$ is similar. Equation (3.28) says that the original position of a component of the permuted vector is obtained by shifting the bits of its index one position to the left in a circular fashion. Second, we generalize our result and obtain

$$((I_{n/2^t} \otimes S_{2^t})\mathbf{x})_{(b_{m-1}\cdots b_0)_2} = x_{(b_{m-1}\cdots b_t b_{t-2}\cdots b_0 b_{t-1})_2}, \tag{3.29}$$

for all binary numbers $(b_{m-1}\cdots b_0)_2$ of m bits. This is because we can apply eqn (3.28) with t bits instead of m to the subvector of length 2^t of $\mathbf{x}$ starting at index $j = (b_{m-1}\cdots b_t)_2 \cdot 2^t$. Here, only the t least significant bits participate in the circular shift. Third, using the definition $R_n = (I_{n/2} \otimes S_2)\cdots(I_1 \otimes S_n)$ and applying eqn (3.29) for $t = 1, 2, 3, \ldots, m$ we obtain

$$\begin{aligned}
(R_n\mathbf{x})_{(b_{m-1}\cdots b_0)_2} &= ((I_{n/2} \otimes S_2)(I_{n/4} \otimes S_4)\cdots(I_1 \otimes S_n)\mathbf{x})_{(b_{m-1}\cdots b_0)_2} \\
&= ((I_{n/4} \otimes S_4)\cdots(I_1 \otimes S_n)\mathbf{x})_{(b_{m-1}\cdots b_1 b_0)_2} \\
&= ((I_{n/8} \otimes S_8)\cdots(I_1 \otimes S_n)\mathbf{x})_{(b_{m-1}\cdots b_2 b_0 b_1)_2} \\
&= ((I_{n/16} \otimes S_{16})\cdots(I_1 \otimes S_n)\mathbf{x})_{(b_{m-1}\cdots b_3 b_0 b_1 b_2)_2} \\
&\quad \cdots \\
&= ((I_1 \otimes S_n)\mathbf{x})_{(b_{m-1} b_0 b_1 b_2 \cdots b_{m-2})_2} = x_{(b_0\cdots b_{m-1})_2}.
\end{aligned} \tag{3.30}$$

Therefore $(R_n\mathbf{x})_j = x_{\rho_n(j)}$, for all j. Using $\rho_n = \rho_n^{-1}$ and applying Lemma 2.5 we arrive at $(R_n\mathbf{x})_j = x_{\rho_n^{-1}(j)} = (P_{\rho_n}\mathbf{x})_j$. Since this holds for all j, we have $R_n\mathbf{x} = P_{\rho_n}\mathbf{x}$. Since this in turn holds for all $\mathbf{x}$, it follows that $R_n = P_{\rho_n}$. $\square$

Algorithm 3.2 is an FFT algorithm based on the Cooley–Tukey theorem. The algorithm overwrites the input vector $\mathbf{x}$ with the output vector $F_n\mathbf{x}$. For

Algorithm 3.2. Sequential nonrecursive FFT.

```
input:          $\mathbf{x}$ : vector of length $n = 2^m$, $m \geq 1$, $\mathbf{x} = \mathbf{x}_0$.
output:         $\mathbf{x}$ : vector of length $n$, such that $\mathbf{x} = F_n\mathbf{x}_0$.
function call:  FFT$(\mathbf{x}, n)$.

{ Perform bit reversal $\mathbf{x} := R_n\mathbf{x}$. Function call bitrev$(\mathbf{x}, n)$ }
for $j := 0$ to $n-1$ do
        { Compute $r := \rho_n(j)$ }
        $q := j$;
        $r := 0$;
        for $k := 0$ to $\log_2 n - 1$ do
                $b_k := q \bmod 2$;
                $q := q \operatorname{div} 2$;
                $r := 2r + b_k$;
        if $j < r$ then swap$(x_j, x_r)$;

{ Perform butterflies. Function call UFFT$(\mathbf{x}, n)$ }
$k := 2$;
while $k \leq n$ do
        { Compute $\mathbf{x} := (I_{n/k} \otimes B_k)\mathbf{x}$ }
        for $r := 0$ to $\frac{n}{k} - 1$ do
                { Compute $x(rk\colon rk+k-1) := B_k x(rk\colon rk+k-1)$ }
                for $j := 0$ to $\frac{k}{2} - 1$ do
                        { Compute $x_{rk+j} \pm \omega_k^j x_{rk+j+k/2}$}
                        $\tau := \omega_k^j x_{rk+j+k/2}$;
                        $x_{rk+j+k/2} := x_{rk+j} - \tau$;
                        $x_{rk+j} := x_{rk+j} + \tau$;
        $k := 2k$;
```

later use, we make parts of the algorithm separately callable: the function bitrev$(\mathbf{x}, n)$ performs a bit reversal of length n and the function UFFT$(\mathbf{x}, n)$ performs an **unordered FFT** of length n, that is, an FFT without bit reversal. Note that multiplication by the butterfly matrix B_k combines components at distance $k/2$, where k is a power of two. In the inner loop, the subtraction $x_{rk+j+k/2} := x_{rk+j} - \tau$ is performed before the addition $x_{rk+j} := x_{rk+j} + \tau$, because the old value of x_{rk+j} must be used in the computation of $x_{rk+j+k/2}$. (Performing these statements in the reverse order would require the use of an extra temporary variable.) A simple count of the floating-point operations shows that the cost of the nonrecursive FFT algorithm is the same as that of the recursive algorithm.

The symmetry of the Fourier matrix F_n and the bit-reversal matrix R_n gives us an alternative form of the Cooley–Tukey FFT factorization, the so-called **decimation in frequency** (DIF) variant.

Corollary 3.11 (Cooley and Tukey [45]—DIF) *Let n be a power of two with $n \geq 2$. Then*

$$F_n = R_n(I_{n/2} \otimes B_2^{\mathrm{T}})(I_{n/4} \otimes B_4^{\mathrm{T}})(I_{n/8} \otimes B_8^{\mathrm{T}}) \cdots (I_1 \otimes B_n^{\mathrm{T}}).$$

Proof

$$\begin{aligned} F_n = F_n^{\mathrm{T}} &= \left[(I_1 \otimes B_n)(I_2 \otimes B_{n/2})(I_4 \otimes B_{n/4}) \cdots (I_{n/2} \otimes B_2)R_n\right]^{\mathrm{T}} \\ &= R_n^{\mathrm{T}}(I_{n/2} \otimes B_2)^{\mathrm{T}} \cdots (I_4 \otimes B_{n/4})^{\mathrm{T}}(I_2 \otimes B_{n/2})^{\mathrm{T}}(I_1 \otimes B_n)^{\mathrm{T}} \\ &= R_n(I_{n/2} \otimes B_2^{\mathrm{T}}) \cdots (I_4 \otimes B_{n/4}^{\mathrm{T}})(I_2 \otimes B_{n/2}^{\mathrm{T}})(I_1 \otimes B_n^{\mathrm{T}}). \end{aligned}$$

□

3.4 Parallel algorithm

The first question to answer when parallelizing an existing sequential algorithm is: which data should be distributed? The vector $\mathbf{x}$ must certainly be distributed, but it may also be necessary to distribute the table containing the powers of ω_n that are used in the computation, the so-called **weights**. (A table must be used because it would be too expensive to compute the weights each time they are needed.) We defer the weights issue to Section 3.5, and for the moment we assume that the table of weights is replicated on every processor.

A suitable distribution should make the operations of the algorithm local. For the FFT, the basic operation is the butterfly, which modifies a pair $(x_j, x_{j'})$ with $j' = j + k/2$, where k is the butterfly size. This is done in **stage** k of the algorithm, that is, the iteration corresponding to parameter k of the main loop of Algorithm 3.2. Let the length of the Fourier transform be $n = 2^m$, with $m \geq 1$, and let the number of processors be $p = 2^q$, with $0 \leq q < m$. We restrict the number of processors to a power of two because this matches the structure of our sequential FFT algorithm. Furthermore, we require that $q < m$, or equivalently $p < n$, because in the pathological case $p = n$ there is only one vector component per processor, and we need at least a pair of vector components to perform a sensible butterfly operation.

A block distribution of the vector $\mathbf{x}$, with n/p components per processor, makes butterflies with $k \leq n/p$ local. This is because k and n/p are powers of two, so that $k \leq n/p$ implies that k divides n/p, thus making each butterfly block $x(rk\colon rk + k - 1)$ fit completely into a processor block $x(sn/p\colon sn/p + n/p - 1)$; no butterfly blocks cross the processor boundaries.

Example 3.12 Let $n = 8$ and $p = 2$ and assume that $\mathbf{x}$ is block distributed. Stage $k = 2$ of the FFT algorithm is a multiplication of $\mathbf{x} = x(0\colon 7)$ with

$$I_4 \otimes B_2 = \begin{bmatrix} 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & -1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & -1 \end{bmatrix}.$$

The butterfly blocks of $\mathbf{x}$ that are multiplied by blocks of $I_4 \otimes B_2$ are $x(0\colon 1)$, $x(2\colon 3)$, $x(4\colon 5)$, and $x(6\colon 7)$. The first two blocks are contained in processor block $x(0\colon 3)$, which belongs to $P(0)$. The last two blocks are contained in processor block $x(4\colon 7)$, which belongs to $P(1)$.

In contrast to the block distribution, the cyclic distribution makes butterflies with $k \geq 2p$ local, because $k/2 \geq p$ implies that $k/2$ is a multiple of p, so that the vector components x_j and $x_{j'}$ with $j' = j + k/2$ reside on the same processor.

Example 3.13 Let $n = 8$ and $p = 2$ and assume that $\mathbf{x}$ is cyclically distributed. Stage $k = 8$ of the FFT algorithm is a multiplication of $\mathbf{x} = x(0\colon 7)$ with

$$B_8 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \omega & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \omega^2 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \omega^3 \\ 1 & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & -\omega & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & -\omega^2 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & -\omega^3 \end{bmatrix},$$

where $\omega = \omega_8 = e^{-\pi i/4} = (1 - i)/\sqrt{2}$. The component pairs (x_0, x_4) and (x_2, x_6) are combined on $P(0)$, whereas the pairs (x_1, x_5) and (x_3, x_7) are combined on $P(1)$.

Now, a parallelization strategy emerges: start with the block distribution and finish with the cyclic distribution. If $p \leq n/p$ (i.e. $p \leq \sqrt{n}$), then these two distributions suffice for the butterflies: we need to redistribute only once and we can do this at any desired time after stage p but before stage $2n/p$. If $p > n/p$, however, we are lucky to have so many processors to solve such a small problem, but we are unlucky in that we have to use more distributions. For the butterflies of size $n/p < k \leq p$, we need one or more intermediates between the block and cyclic distribution.

The **group-cyclic distribution** with cycle c is defined by the mapping

$$x_j \longmapsto P\left(\left(j \text{ div } \left\lceil \frac{cn}{p} \right\rceil\right) c + \left(j \bmod \left\lceil \frac{cn}{p} \right\rceil\right) \bmod c\right), \quad \text{for } 0 \le j < n. \tag{3.31}$$

This distribution is defined for every c with $1 \le c \le p$ and $p \bmod c = 0$. It first splits the vector $\mathbf{x}$ into blocks of size $\lceil cn/p \rceil$, and then assigns each block to a group of c processors using the cyclic distribution. Note that for $c = 1$ this reduces to the block distribution, see (1.6), and for $c = p$ to the cyclic distribution, see (1.5). In the special case $n \bmod p = 0$, the group-cyclic distribution reduces to

$$x_j \longmapsto P\left(\left(j \text{ div } \frac{cn}{p}\right) c + j \bmod c\right), \quad \text{for } 0 \le j < n. \tag{3.32}$$

This case is relevant for the FFT, because n and p are both powers of two. Figure 3.4 illustrates the group-cyclic distribution for $n = 8$ and $p = 4$. The group-cyclic distribution is a new generalization of the block and cyclic distributions; note that it differs from the block-cyclic distribution introduced earlier,

$$x_j \longmapsto P((j \text{ div } b) \bmod p), \quad \text{for } 0 \le j < n, \tag{3.33}$$

where b is the block size.

In the FFT case, n and p and hence c are powers of two and we can write each global index j as the sum of three terms,

$$j = j_2 \frac{cn}{p} + j_1 c + j_0, \tag{3.34}$$

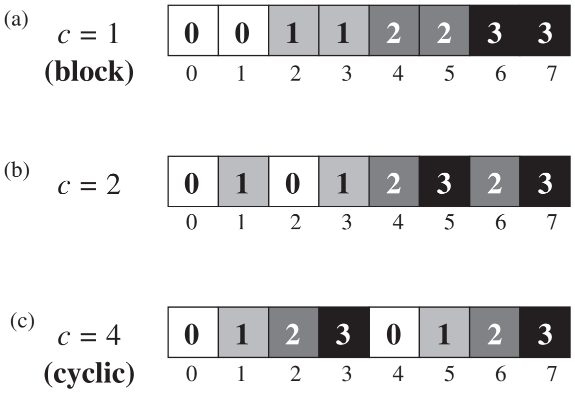


FIG. 3.4. Group-cyclic distribution with cycle c of a vector of size eight over four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. (a) $c = 1$; (b) $c = 2$; and (c) $c = 4$.

where $0 \leq j_0 < c$ and $0 \leq j_1 < n/p$. The processor that owns the component x_j in the group-cyclic distribution with cycle c is $P(j_2c + j_0)$; the processor allocation is not influenced by j_1. As always, the components are stored locally in order of increasing global index. Thus x_j ends up in local position $\mathtt{j} = j_1 = (j \bmod cn/p) \operatorname{div} c$ on processor $P(j_2c + j_0)$. (The relation between global and local indices is explained in Fig. 1.9.)

To make a butterfly of size k local in the group-cyclic distribution with cycle c, two constraints must be satisfied. First, the butterfly block $x(rk\colon rk+k-1)$ should fit completely into one block of size cn/p, which is assigned to a group of c processors. This is guaranteed if $k \leq cn/p$. Second, $k/2$ must be a multiple of the cycle c, so that the components x_j and $x_{j'}$ with $j' = j + k/2$ reside on the same processor from the group. This is guaranteed if $k/2 \geq c$. As a result, we find that a butterfly of size k is local in the group-cyclic distribution with cycle c if

$$2c \leq k \leq \frac{n}{p}c. \tag{3.35}$$

This result includes as a special case our earlier results for the block and cyclic distributions. In Fig. 3.4(a), it can be seen that for $c = 1$ butterflies of size $k = 2$ are local, since these combine pairs (x_j, x_{j+1}). In Fig. 3.4(b) this can be seen for $c = 2$ and pairs (x_j, x_{j+2}), and in Fig. 3.4(c) for $c = 4$ and pairs (x_j, x_{j+4}). In this particular example, the range of (3.35) consists of only one value of k, namely $k = 2c$.

A straightforward strategy for the parallel FFT is to start the butterflies with the group-cyclic distribution with cycle $c = 1$, and continue as long as possible with this distribution, that is, in stages $k = 2, 4, \ldots, n/p$. At the end of stage n/p, the vector $\mathbf{x}$ is redistributed into the group-cyclic distribution with cycle $c = n/p$, and then stages $k = 2n/p, 4n/p, \ldots, n^2/p^2$ are performed. Then c is again multiplied by n/p, $\mathbf{x}$ is redistributed, stages $k = 2n^2/p^2, 4n^2/p^2, \ldots, n^3/p^3$ are performed, and so on. Since $n/p \geq 2$ the value of c increases monotonically. When multiplying c by n/p would lead to a value $c = (n/p)^t \geq p$, the value of c is set to $c = p$ instead and the remaining stages $2(n/p)^t, \ldots, n$ are performed in the cyclic distribution.

Until now, we have ignored the bit reversal preceding the butterflies. The bit reversal is a permutation, which in general requires communication. We have been liberal in allowing different distributions in different parts of our algorithm, so why not use different distributions before and after the bit reversal? This way, we might be able to avoid communication.

Let us try our luck and assume that we have the cyclic distribution before the bit reversal. This is the preferred starting distribution of the overall computation, because it is the distribution in which the FFT computation ends. It is advantageous to start and finish with the same distribution, because then it is easy to apply the FFT repeatedly. This would make it possible, for instance, to execute a parallel inverse FFT by using the parallel forward FFT with conjugated weights; this approach is based on the property $F_n^{-1} = \overline{F_n}/n$.

Consider a component x_j with index $j = (b_{m-1} \cdots b_0)_2$ of a cyclically distributed vector $\mathbf{x}$. This component is stored on processor $P((b_{q-1} \cdots b_0)_2)$ in location $\mathrm{j} = (b_{m-1} \cdots b_q)_2$. In other words, the least significant q bits of j determine the processor number and the most significant $m - q$ bits the local index. Since our aim is to achieve a global bit reversal, it seems natural to start with a local bit reversal; this reverses already part of the bits and it does not incur communication. The local bit reversal moves x_j into local position $(b_q \cdots b_{m-1})_2$, which consists of the least significant $m - q$ bits of the global destination index $\rho_n(j) = (b_0 \cdots b_{m-1})_2$. The least significant $m - q$ bits also happen to be those that determine the local position in the block distribution. Therefore, the local position would be correct if we declared the distribution to be by blocks. Unfortunately, the processor would still be wrong: in the block distribution after the global bit reversal, the original x_j should find itself in processor $P(b_0 \cdots b_{q-1})$, determined by the most significant q bits of the destination index $(b_0 \cdots b_{m-1})_2$, but of course x_j did not leave the original processor $P((b_{q-1} \cdots b_0)_2)$. Note that the correct processor number is the bit-reverse of the actual processor number. Therefore, we call the current distribution after the local bit reversal the **block distribution with bit-reversed processor numbering**.

Can we use the current distribution instead of the block distribution for the first set of butterflies, with size $k = 2, 4, \ldots, n/p$? We have to check whether these butterflies remain local when using the block distribution with bit-reversed processor numbering. Fortunately, for these k, every processor carries out exactly the same operations, but on its own data. This is because $k \le n/p$, so that every butterfly block fits completely into a processor block. As a consequence, the processor renumbering does not affect the local computations. At the first redistribution, the vector is moved into the standard group-cyclic distribution without bit-reversed processor numbering, and then the algorithm returns to the original strategy outlined above.

By a stroke of luck, we now have a complete parallel FFT algorithm that starts and ends with the same distribution, the cyclic distribution, and that performs only a limited number of redistributions. The result is given as Algorithm 3.3. In stage k of the algorithm, there are n/k butterfly blocks, so that each of the p/c processor groups handles $\mathit{nblocks} = (n/k)/(p/c)$ blocks. Each processor of a group participates in the computations of every block of its group. A boolean variable *rev* is used which causes a reversal of the processor number during the first redistribution.

The redistribution function used in Algorithm 3.3 is given as Algorithm 3.4. This function puts every component x_j in the destination processor determined by the new distribution with cycle c. We do not specify where exactly in the destination processor the component x_j is placed. Strictly speaking this has nothing to do with the distribution; rather, it is related to the data structure used to store the local values. (Still, our convention that components are ordered locally by increasing global index determines the local

Algorithm 3.3. Parallel FFT algorithm for processor $P(s)$.

```
input:  x : vector of length $n = 2^m$, $m \geq 1$, $\mathbf{x} = \mathbf{x}_0$,
        distr(x) = cyclic over $p = 2^q$ processors with $0 \leq q < m$.
output: x : vector of length $n$, distr(x) = cyclic, such that $\mathbf{x} = F_n \mathbf{x}_0$.

        bitrev($x(s: p: n-1), n/p$);
        { distr(x) = block with bit-reversed processor numbering }

        $k := 2$;
        $c := 1$;
        $rev := true$;
        while $k \leq n$ do
(0)         $j_0 := s \bmod c$;
            $j_2 := s \operatorname{div} c$;
            while $k \leq \frac{n}{p}c$ do
                $nblocks := \frac{nc}{kp}$;
                for $r := j_2 \cdot nblocks$ to $(j_2 + 1) \cdot nblocks - 1$ do
                    { Compute local part of $x(rk: (r+1)k - 1)$ }
                    for $j := j_0$ to $\frac{k}{2} - 1$ step $c$ do
                        $\tau := \omega_k^j x_{rk+j+k/2}$;
                        $x_{rk+j+k/2} := x_{rk+j} - \tau$;
                        $x_{rk+j} := x_{rk+j} + \tau$;
                $k := 2k$;
            if $c < p$ then
                $c_0 := c$;
                $c := \min(\frac{n}{p}c, p)$;
(1)             redistr($\mathbf{x}, n, p, c_0, c, rev$);
                $rev := false$;
                { distr(x) = group-cyclic with cycle $c$ }
```

index, ($j \bmod cn/p$) div c, as we saw above, and this will be used later in the implementation.) We want the redistribution to work also in the trivial case $p = 1$, and therefore we need to define ρ_1; the definition in Theorem 3.10 omitted this case. By convention, we define ρ_1 to be the identity permutation of length one, which is the permutation that reverses zero bits.

Finally, we determine the cost of Algorithm 3.3. First, we consider the synchronization cost. Each iteration of the main loop has two supersteps: a computation superstep that computes the butterflies and a communication superstep that redistributes the vector $\mathbf{x}$. The computation superstep of the first iteration also includes the local bit reversal. The last iteration, with $c = p$, does not perform a redistribution any more. The total number of iterations equals $t+1$, where t is the smallest integer such that $(n/p)^t \geq p$. By taking the $\log_2$ of both sides and using $m = 2^n$ and $p = 2^q$, we see that this inequality

Algorithm 3.4. Redistribution from group-cyclic distribution with cycle c_0 to cycle c_1 for processor $P(s)$.

```
input:          x : vector of length $n = 2^m$, $m \geq 0$,
                distr(x) = group-cyclic with cycle $c_0$ over $p = 2^q$
                processors with $0 \leq q \leq m$.
                If rev is true, the processor numbering is bit-reversed,
                otherwise it is standard.
output:         x : vector of length $n$, distr(x) = group-cyclic with cycle $c_1$
                with standard processor numbering.
function call:  redistr($\mathbf{x}, n, p, c_0, c_1, rev$);

(1)             if rev then
                        $j_0 := \rho_p(s) \bmod c_0$;
                        $j_2 := \rho_p(s) \operatorname{div} c_0$;
                else
                        $j_0 := s \bmod c_0$;
                        $j_2 := s \operatorname{div} c_0$;
                for $j := j_2\frac{c_0 n}{p} + j_0$ to $(j_2+1)\frac{c_0 n}{p} - 1$ step $c_0$ do
                        $dest := (j \operatorname{div} \frac{c_1 n}{p})c_1 + j \bmod c_1$;
                        put $x_j$ in $P(dest)$;
```

is equivalent to $t \geq q/(m-q)$, so that $t = \lceil q/(m-q) \rceil$. Therefore, the total synchronization cost is

$$T_{\text{sync}} = \left(2\left\lceil \frac{q}{m-q} \right\rceil + 1\right) l. \tag{3.36}$$

Second, we examine the communication cost. Communication occurs only within the redistribution, where in the worst case all n/p old local vector components are sent away, and n/p new components are received from another processor. Each vector component is a complex number, which consists of two real numbers. The redistribution is therefore a $2n/p$-relation. (The cost is actually somewhat lower than $2ng/p$, because certain data remain local. For example, component x_0 remains on $P(0)$ in all group-cyclic distributions, even when the processor numbering is bit-reversed. This is a small effect, which we can neglect.) Thus, the communication cost is

$$T_{\text{comm}} = \left\lceil \frac{q}{m-q} \right\rceil \cdot \frac{2n}{p} g. \tag{3.37}$$

Third, we focus on the computation cost. Stage k handles $nblocks = nc/(kp)$ blocks, each of size k. Each processor handles a fraction $1/c$ of the $k/2$ component pairs in a block. Each pair requires a complex multiplication, addition, and subtraction, that is, a total of 10 real flops. We do not count indexing arithmetic, nor the computation of the weights. As a consequence, the total number of flops per processor in stage k equals $(nc/(kp)) \cdot (k/(2c)) \cdot 10 = 5n/p$. Since there are m stages, the computation cost is

$$T_{\mathrm{comp}} = 5mn/p. \tag{3.38}$$

The total BSP cost of the algorithm as a function of n and p is obtained by summing the three costs and substituting $m = \log_2 n$ and $q = \log_2 p$, giving

$$T_{\mathrm{FFT}} = \frac{5n\log_2 n}{p} + 2 \cdot \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil \cdot \frac{n}{p} g + \left(2 \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil + 1\right) l. \tag{3.39}$$

As you may know, budgets for the acquisition of parallel computers are often tight, but you, the user of a parallel computer, may be insatiable in your computing demands. In that case, p remains small, n becomes large, and you may find yourself performing FFTs with $1 < p \leq \sqrt{n}$. The good news is that then you only need one communication superstep and two computation supersteps. The BSP cost of the FFT reduces to

$$T_{\mathrm{FFT},\ 1<p\leq\sqrt{n}} = \frac{5n\log_2 n}{p} + 2\frac{n}{p}g + 3l. \tag{3.40}$$

This happens because $p \leq \sqrt{n}$ implies $p \leq n/p$ and hence $\log_2 p \leq \log_2(n/p)$, so that the ceiling expression in (3.39) becomes one.

3.5 Weight reduction

The weights of the FFT are the powers of ω_n that are needed in the FFT computation. These powers $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n/2-1}$ are usually precomputed and stored in a table, thus saving trigonometric evaluations when repeatedly using the same power of ω_n. This table can be reused in subsequent FFTs. In a sequential computation, the table requires a storage space of $n/2$ complex numbers, which is half the space needed for the vector $\mathbf{x}$; such an overhead is usually acceptable. For small n, the $\mathcal{O}(n)$ time of the weight initializations may not be negligible compared with the $5n\log_2 n$ flops of the FFT itself. The reason is that each weight computation requires the evaluation of two trigonometric functions,

$$\omega_n^j = \cos\frac{2\pi j}{n} - i\sin\frac{2\pi j}{n}, \tag{3.41}$$

which typically costs about five flops per evaluation [151] in single precision and perhaps ten flops in double precision.

The precomputation of the weights can be accelerated by using symmetries. For example, the property

$$\omega_n^{n/2-j} = -\overline{(\omega_n^j)} \tag{3.42}$$

implies that only the weights ω_n^j with $0 \leq j \leq n/4$ have to be computed. The remaining weights can then be obtained by negation and complex conjugation, which are cheap operations. Symmetry can be exploited further by using the property

$$\omega_n^{n/4-j} = -i\overline{(\omega_n^j)}, \tag{3.43}$$

which is also cheap to compute. The set of weights can thus be computed by eqn (3.41) with $0 \leq j \leq n/8$, eqn (3.43) with $0 \leq j < n/8$, and eqn (3.42) with $0 < j < n/4$. This way, the initialization of the $n/2$ weights in double precision costs about $2 \cdot 10 \cdot n/8 = 2.5n$ flops.

An alternative method for precomputation of the weights is to compute the powers of ω_n by successive multiplication, computing $\omega_n^2 = \omega_n \cdot \omega_n$, $\omega_n^3 = \omega_n \cdot \omega_n^2$, and so on. Unfortunately, this propagates roundoff errors and hence produces less accurate weights and a less accurate FFT. This method is not recommended [187].

In the parallel case, the situation is more complicated than in the sequential case. For example, in the first iteration of the main loop of Algorithm 3.3, $c = 1$ and hence $j_0 = 0$ and $j_2 = s$, so that all processors perform the same set of butterfly computations, but on different data. Each processor performs an unordered sequential FFT of length n/p on its local part of $\mathbf{x}$. This implies that the processors need the same weights, so that the weight table for these butterflies must be replicated, instead of being distributed. The local table should at least contain the weights $\omega_{n/p}^j = \omega_n^{jp}$, $0 \leq j < n/(2p)$, so that the total memory used by all processors for this iteration alone is already $n/2$ complex numbers. Clearly, in the parallel case care must be taken to avoid excessive memory use and initialization time.

A brute-force approach would be to store on every processor the complete table of all $n/2$ weights that could possibly be used during the computation. This has the disadvantage that every processor has to store almost the same amount of data as needed for the whole sequential problem. Therefore, this approach is not scalable in terms of memory usage. Besides, it is also unnecessary to store all weights on every processor, since not all of them are needed. Another disadvantage is that the $2.5n$ flops of the weight initializations can easily dominate the $(5n \log_2 n)/p$ flops of the FFT itself.

At the other extreme is the simple approach of recomputing the weights whenever they are needed, thus discarding the table. This attaches a weight computation of about 20 flops to the 10 flops of each pairwise butterfly operation, thereby approximately tripling the total computing time. This approach wastes a constant factor in computing time, but it is scalable in terms of memory usage.

Our main aim in this section is to find a scalable approach in terms of memory usage that adds few flops to the overall count. To achieve this, we try to find structure in the local computations of a processor and to express them by using sequential FFTs. This has the additional benefit that we can make use of the tremendous amount of available sequential FFT software, including highly tuned programs provided by computer vendors for their hardware.

The **generalized discrete Fourier transform** (GDFT) [27] is defined by

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{j(k+\alpha)}, \quad \text{for } 0 \le k < n, \tag{3.44}$$

where α is a fixed real parameter, not necessarily an integer, which represents a frequency shift. (For instance, taking $\alpha = -1$ shifts the components $(y_0, \ldots, y_{n-1})$ of a standard DFT into $(y_{n-1}, y_0, y_1, \ldots, y_{n-2})$.) For $\alpha = 0$, the GDFT becomes the DFT. The GDFT can be computed by a fast algorithm, called GFFT, which can be derived in the same way as we did for the FFT. We shall give the main results of the derivation in the generalized case, but omit the proofs.

The sum in the GDFT definition (3.44) can be split into even and odd components, giving

$$y_k = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{j(k+\alpha)} \; + \; \omega_n^{k+\alpha} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{j(k+\alpha)}, \quad \text{for } 0 \le k < n/2, \tag{3.45}$$

and

$$y_{k+n/2} = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{j(k+\alpha)} \; - \; \omega_n^{k+\alpha} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{j(k+\alpha)}, \quad \text{for } 0 \le k < n/2. \tag{3.46}$$

These equations can be converted to matrix notation by defining the $n \times n$ **generalized Fourier matrix** F_n^α with parameter α,

$$(F_n^\alpha)_{jk} = \omega_n^{j(k+\alpha)}, \quad \text{for } 0 \le j, k < n, \tag{3.47}$$

the $n \times n$ diagonal matrix

$$\Omega_n^\alpha = \text{diag}(\omega_{2n}^\alpha, \omega_{2n}^{1+\alpha}, \omega_{2n}^{2+\alpha}, \ldots, \omega_{2n}^{n-1+\alpha}), \tag{3.48}$$

and the $n \times n$ **generalized butterfly matrix**

$$B_n^\alpha = \begin{bmatrix} I_{n/2} & \Omega_{n/2}^\alpha \\ I_{n/2} & -\Omega_{n/2}^\alpha \end{bmatrix} \tag{3.49}$$

giving the decomposition

$$F_n^\alpha = B_n^\alpha (I_2 \otimes F_{n/2}^\alpha) S_n. \tag{3.50}$$

Repeatedly applying (3.50) until we reach the middle factor $I_n \otimes F_1^\alpha = I_n$ yields the following generalization of Cooley and Tukey's theorem.

Theorem 3.14 *Let n be a power of two. Then*

$$F_n^\alpha = (I_1 \otimes B_n^\alpha)(I_2 \otimes B_{n/2}^\alpha)(I_4 \otimes B_{n/4}^\alpha) \cdots (I_{n/2} \otimes B_2^\alpha) R_n.$$

Our ultimate goal is to express the local computations of the parallel FFT algorithm in terms of sequential FFTs, but for the moment we settle for less and try to express the computations of a superstep in terms of a GFFT with suitable α. For this moderate goal, we have to inspect the inner loops of Algorithm 3.3. The j-loop takes a local subvector $x(rk + k/2 + j_0 : c : (r+1)k - 1)$ of length $k/(2c)$, multiplies it by the diagonal matrix

$$\begin{aligned} &\mathrm{diag}(\omega_k^{j_0}, \omega_k^{c+j_0}, \omega_k^{2c+j_0}, \ldots, \omega_k^{k/2-c+j_0}) \\ &\quad = \mathrm{diag}(\omega_{k/c}^{j_0/c}, \omega_{k/c}^{1+j_0/c}, \omega_{k/c}^{2+j_0/c}, \ldots, \omega_{k/c}^{k/(2c)-1+j_0/c}) \\ &\quad = \Omega_{k/(2c)}^{j_0/c}, \end{aligned} \tag{3.51}$$

and then adds it to the subvector $x(rk+j_0 : c : rk+k/2-1)$, and also subtracts it. This means that a generalized butterfly $B_{k/c}^{j_0/c}$ is performed on the local subvector $x(rk + j_0 : c : (r+1)k - 1)$. The r-loop takes care that the same generalized butterfly is performed for all $nc/(kp)$ local subvectors. Thus, stage k in the group-cyclic distribution with cycle c computes

$$(I_{nc/(kp)} \otimes B_{k/c}^{j_0/c}) \cdot x\left(j_2 \frac{nc}{p} + j_0 : c : (j_2+1)\frac{nc}{p} - 1\right).$$

A **complete** sequence of butterfly stages is a sequence of maximal length, $k = 2c, 4c, \ldots, (n/p)c$. Such a sequence multiplies the local vector by

$$(I_1 \otimes B_{n/p}^{j_0/c})(I_2 \otimes B_{n/(2p)}^{j_0/c}) \cdots (I_{n/(2p)} \otimes B_2^{j_0/c}) = F_{n/p}^{j_0/c} R_{n/p}, \tag{3.52}$$

where the equality follows from Theorem 3.14. This implies that superstep (0) is equivalent to an unordered GFFT applied to the local vector, with shift parameter $\alpha = j_0/c = (s \bmod c)/c$.

One problem that remains is the computation superstep of the last iteration. This superstep may not perform a complete sequence of butterfly stages, in which case we cannot find a simple expression for the superstep. If, however, we would start with an incomplete sequence such that all the following computation supersteps perform complete sequences, we would have an easier task, because at the start $c = 1$ so that $\alpha = 0$ and we perform standard butterflies. We can then express a sequence of stages $k = 2, 4, \ldots, k_1$ by the matrix product

$$\begin{aligned} &(I_{n/(k_1 p)} \otimes B_{k_1}) \cdots (I_{n/(4p)} \otimes B_4)(I_{n/(2p)} \otimes B_2) \\ &\quad = I_{n/(k_1 p)} \otimes ((I_1 \otimes B_{k_1}) \cdots (I_{k_1/4} \otimes B_4)(I_{k_1/2} \otimes B_2)) \\ &\quad = I_{n/(k_1 p)} \otimes (F_{k_1} R_{k_1}). \end{aligned} \tag{3.53}$$

Algorithm 3.5. Restructured parallel FFT algorithm for processor $P(s)$.

```
input:   x : vector of length $n = 2^m$, $m \geq 1$, $\mathbf{x} = \mathbf{x}_0$,
         distr(x) = cyclic over $p = 2^q$ processors with $0 \leq q < m$.
output:  x : vector of length $n$, distr(x) = cyclic, such that $\mathbf{x} = F_n\mathbf{x}_0$.

(0)      bitrev$(x(s\colon p\colon n-1), n/p)$;
         { distr(x) = block with bit-reversed processor numbering }

         $t := \lceil \frac{\log_2 p}{\log_2 (n/p)} \rceil$;
         $k_1 := \frac{n}{(n/p)^t}$;
         $rev := true$;
         for $r := s \cdot \frac{n}{k_1 p}$ to $(s+1) \cdot \frac{n}{k_1 p} - 1$ do
                 UFFT$(x(rk_1\colon (r+1)k_1 - 1), k_1)$;
         $c_0 := 1$;
         $c := k_1$;
         while $c \leq p$ do
(1)              redistr$(\mathbf{x}, n, p, c_0, c, rev)$;
                 { distr(x) = group-cyclic with cycle $c$ }
(2)              $rev := false$;
                 $j_0 := s \bmod c$;
                 $j_2 := s \operatorname{div} c$;
                 UGFFT$(x(j_2\frac{nc}{p} + j_0\colon c\colon (j_2+1)\frac{nc}{p} - 1), n/p, j_0/c)$;
                 $c_0 := c$;
                 $c := \frac{n}{p}c$;
```

Now, superstep (0) of the first iteration is equivalent to $n/(k_1p)$ times an unordered FFT applied to a local subvector of size k_1. Thus, we decide to restructure the main loop of our algorithm and use distributions with $c = 1, k_1, k_1(n/p),\ k_1(n/p)^2, \ldots, k_1(n/p)^{t-1} = p$, where $k_1 = n/(n/p)^t$ and $t = \lceil \log_2 p / \log_2 (n/p) \rceil$.

The resulting algorithm is given as Algorithm 3.5. The BSP cost of the algorithm is the same as that of Algorithm 3.3; the cost is given by eqn (3.39). The function UGFFT$(\mathbf{x}, n, \alpha)$ performs an unordered GFFT with parameter α on a vector $\mathbf{x}$ of length n. This function is identical to UFFT$(\mathbf{x}, n)$, see Algorithm 3.2, except that the term ω_k^j in the inner loop is replaced by $\omega_k^{j+\alpha}$. Table 3.2 gives an example of the shift parameters α that occur in a particular parallel computation.

The advantage of the restructured algorithm is increased modularity: the algorithm is based on smaller sequential modules, namely unordered FFTs and GFFTs of size at most n/p. Thus, we can benefit from existing efficient

TABLE 3.2. Shift parameters α on eight processors for $n = 32$

Iteration		Processor							
	c	$P(0)$	$P(1)$	$P(2)$	$P(3)$	$P(4)$	$P(5)$	$P(6)$	$P(7)$
0	1	0	0	0	0	0	0	0	0
1	2	0	1/2	0	1/2	0	1/2	0	1/2
2	8	0	1/8	1/4	3/8	1/2	5/8	3/4	7/8

sequential implementations for these modules. For the FFT many implementations are available, see Section 3.8, but for the GFFT this is not the case. We are willing to accept a small increase in flop count if this enables us to use the FFT as the computational workhorse instead of the GFFT. We can achieve this by writing the GDFT as

$$y_k = \sum_{j=0}^{n-1} (x_j \omega_n^{j\alpha}) \omega_n^{jk}, \quad \text{for } 0 \le k < n, \tag{3.54}$$

which can be viewed as a multiplication of $\mathbf{x}$ by an $n \times n$ diagonal **twiddle matrix**

$$T_n^\alpha = \text{diag}(1, \omega_n^\alpha, \omega_n^{2\alpha}, \ldots, \omega_n^{(n-1)\alpha}), \tag{3.55}$$

followed by a standard DFT. In matrix notation,

$$F_n^\alpha = F_n T_n^\alpha. \tag{3.56}$$

Twiddling with the data vector is a worthwhile preparation for the execution of the butterflies. The extra cost of twiddling is n/p complex multiplications, or $6n/p$ real flops, in every computation superstep, except the first. This is usually much less than the $(5n/p)\log_2(n/p)$ flops of a computation superstep.

A slight complication arises because the twiddling relation (3.56) between the DFT and the GDFT is for ordered transforms, whereas we need unordered ones. Using the fact that $R_n = R_n^{-1}$, we can write the unordered GDFT as

$$F_n^\alpha R_n = F_n T_n^\alpha R_n = F_n R_n (R_n T_n^\alpha R_n) = (F_n R_n) \tilde{T}_n^\alpha. \tag{3.57}$$

Here, $\tilde{T}_n^\alpha = R_n T_n^\alpha R_n$ is the diagonal matrix obtained by permuting the diagonal entries of T_n^α according to ρ_n,

$$(\tilde{T}_n^\alpha)_{jj} = (T_n^\alpha)_{\rho_n(j),\rho_n(j)}, \quad \text{for } 0 \le j < n, \tag{3.58}$$

cf. Lemma 2.5.

The maximum amount of memory space per processor used by Algorithm 3.5, expressed in reals, equals

$$M_{\mathrm{FFT}} = \left(2 \cdot \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil + 3\right) \cdot \frac{n}{p}. \tag{3.59}$$

This memory is needed for: n/p complex numbers representing $\mathbf{x}$; $n/(2p)$ complex numbers representing the weights of an FFT of length n/p; and n/p complex numbers representing a diagonal twiddle matrix, for each of the $\lceil \log_2 p / \log_2(n/p) \rceil$ GFFTs performed by a processor.

The amount of memory required by the FFT has been reduced from $\mathcal{O}(n)$ to $\mathcal{O}((n \log_2 p)/(p \log_2(n/p)))$. To see whether the new memory requirement is 'scalable', we need to define precisely what we mean by this term. We call the memory requirements of a BSP algorithm **scalable** if the maximum memory space $M(n,p)$ required per processor satisfies

$$M(n,p) = \mathcal{O}\left(\frac{M_{\mathrm{seq}}(n)}{p} + p\right), \tag{3.60}$$

where $M_{\mathrm{seq}}(n)$ is the memory space required by the sequential algorithm for an input size n, and p is the number of processors. This definition allows for $\mathcal{O}(p)$ overhead, reflecting the philosophy that BSP algorithms are based on all-to-all communication supersteps, where each processor deals with $p-1$ others, and also reflecting the practice of current BSP implementations where each processor stores several arrays of length p. (For example, each registration of a variable by the BSPlib primitive `bsp_push_reg` gives rise to an array of length p on every processor that contains the p addresses of the variable on all processors. Another example is the common implementation of a communication superstep where the number of data to be sent is announced to each destination processor before the data themselves are sent. This information needs to be stored in an array of length p on the destination processor.)

For $p \leq n/p$, only one twiddle array has to be stored, so that the total memory requirement is $M(n,p) = 5n/p$, which is scalable by the definition above. For $p > n/p$, we need $t-1$ additional iterations each requiring a twiddle array. Fortunately, we can find a simple upper bound on the additional memory requirement, namely

$$\frac{2(t-1)n}{p} = 2\left(\frac{n}{p} + \frac{n}{p} + \cdots + \frac{n}{p}\right) \leq 2\frac{n}{p} \cdot \frac{n}{p} \cdots \frac{n}{p} = 2\left(\frac{n}{p}\right)^{t-1} = \frac{2p}{k_1} \leq p. \tag{3.61}$$

Thus, the total memory use in this case is $M(n,p) \leq 5n/p + p$, which is also scalable. We have achieved our initial aim.

Note that some processors may be able to use the same twiddle array in several subsequent supersteps, thus saving memory. An extreme case is $P(0)$, which always has $\alpha = 0$ and in fact would need no twiddle memory. Processor

$P(p-1)$, however, has $\alpha = ((p-1) \bmod c)/c = (c-1)/c = 1 - 1/c$, so that each time it needs a different twiddle array. We assume that all processors are equal in memory size (as well as in computing rate), so that it is not really worthwhile to save memory for some processors when this is not possible for the others, as in the present case. After all, this will not increase the size of the largest problem that can be solved. Also note that $P(0)$ can actually save the extra flops needed for twiddling, but this does not make the overall computation faster, because $P(0)$ still has to wait for the other processors to complete their work. The moral: stick to the SPMD style of programming and do not try to be clever saving memory space or computing time for specific processors.

3.6 Example function `bspfft`

This section presents the program text of the function `bspfft`, which is a straightforward implementation of Algorithm 3.5. This function was written to explain the implementation of the algorithm, and hence its formulation emphasizes clarity and brevity rather than efficiency, leaving room for further optimization (mainly in the computation part). Throughout, the data structure used to store a complex vector of length n is a real array of size $2n$ with alternating real and imaginary parts, that is, with $\mathrm{Re}(x_j)$ stored as `x[2 * j]` and $\mathrm{Im}(x_j)$ as `x[2 * j + 1]`. The function `bspfft` can also compute an inverse FFT, and it does this by performing all operations of the forward FFT with conjugated weights and scaling the output vector by $1/n$. Before we can use `bspfft`, the function `bspfft_init` must have been called to initialize three weight tables and two bit-reversal tables.

The function `ufft` is a faithful implementation of the UFFT function in Algorithm 3.2. The loop index runs through the range `j` $= 0, 2, \ldots,$ `k` $- 2$, which corresponds to the range $j = 0, 1, \ldots, k/2 - 1$ in the algorithm; this is a consequence of the way complex vectors are stored. The function `ufft_init` initializes a weight table of $n/2$ complex weights ω_n^k, $0 \le k < n/2$, using eqns (3.41)–(3.43).

The function `permute` permutes a vector by a given permutation σ that swaps component pairs independently; an example of such a permutation is the bit-reversal permutation ρ_n. This type of permutation has the advantage that it can be done in-place, requiring only two reals as extra storage but no additional temporary array. The condition `j < sigma[j]` ensures that the swap is executed only if the indices involved are different and that this is done only once per pair. Without the condition the overall effect of the function would be nil!

The function `twiddle` multiplies **x** and **w** componentwise, $x_j := w_j x_j$, for $0 \le j < n$. The function `twiddle_init` reveals the purpose of `twiddle`: **w** is the diagonal of the diagonal matrix $\tilde{T}_n^\alpha = R_n T_n^\alpha R_n$ for a given α.

The bit-reversal initialization function `bitrev_init` fills an array `rho` with bit reverses of indices used in the FFT. Of course, the bit reverse of an index can be computed when needed, saving the memory of `rho`, but this can be costly in computer time. In the local bit reversal, for instance, the reverse of n/p local indices is computed. Each reversal of an index costs of the order $\log_2(n/p)$ integer operations. The total number of such operations is therefore of the order $(n/p)\log_2(n/p)$, which for small p is of the same order as the $(5n/p)\log_2 n$ floating-point operations of the butterfly computations. The fraction of the total time spent in the bit reversal could easily reach 20%. This justifies using a table so that the bit reversal needs to be computed only once and its cost can be amortized over several FFTs. To reduce the cost also in case the FFT is called only once, we optimize the inner loop of the function by using bit operations on unsigned integers (instead of integer operations as used everywhere else in the program). To obtain the last bit b_i of the remainder $(b_{m-1}\cdots b_i)_2$, an 'and'-operation is carried out with 1, which avoids the expensive modulo operation that would occur in the alternative formulation, `lastbit= rem%2`. After that, the remainder is shifted one position to the right, which is equivalent to `rem /=2`. It depends on the compiler and the chosen optimization level whether the use of explicit bit operations gives higher speed. (A good compiler will make such optimizations superfluous!) In my not so humble opinion, bit operations should only be used sparingly in scientific computation, but here is an instance where there is a justification.

The function `k1_init` computes k_1 from n, p by finding the first $c = (n/p)^t \geq p$. Note that the body of the c-loop consists of an empty statement, since we are only interested in the final value of the counter c. The counter c takes on $t+1$ values, which is the number of iterations of the main loop of the FFT. As a consequence, $k_1(n/p)^t = n$ so that $k_1 = n/c$.

The function `bspredistr` redistributes the vector **x** from group-cyclic distribution with cycle c_0 to the group-cyclic distribution with cycle c_1, for a ratio $c_1/c_0 \geq 1$, as illustrated in Fig. 3.5. (We can derive a similar redistribution function for $c_1/c_0 < 1$, but we do not need it.) The function is an implementation of Algorithm 3.4, but with one important optimization (I could not resist the temptation!): vector components to be redistributed are sent in blocks, rather than individually. This results in blocks of $nc_0/(pc_1)$ complex numbers. If $nc_0 < pc_1$, components are sent individually. The aim is, of course, to reach a communication rate that corresponds to optimistic values of g, see Section 2.6.

The parallel FFT, like the parallel LU decomposition, is a **regular parallel algorithm**, for which the communication pattern can be predicted exactly, and each processor can determine exactly where every communicated data element goes. In such a case, it is always possible for the user to combine data for the same destination in a block, or **packet**, and communicate them using one put operation. In general, this requires packing at the source processor and unpacking at the destination processor. No identifying information needs to be sent together with the data since the receiver knows their meaning. (In

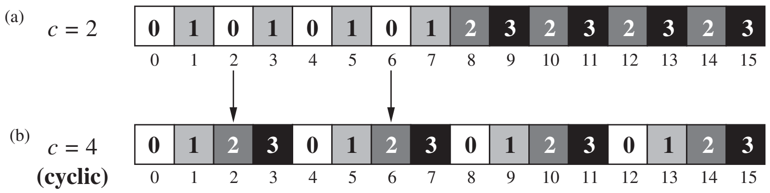


FIG. 3.5. Redistribution from group-cyclic distribution with cycle $c_0 = 2$ to cycle $c_1 = 4$ for a vector of size $n = 16$. The number of processors is $p = 4$. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. The array indices are global indices. The initial distribution has two blocks of size eight; the final distribution has one block of size 16. The arrows show a packet sent by $P(0)$ to $P(2)$.

contrast, for certain irregular algorithms, sending such information cannot be avoided, in which case there is no advantage in packaging by the user and this is better left up to the BSP system. If, furthermore, the size of the data items is small compared with the identifying information, we must sadly communicate at a rate corresponding to pessimistic values of g.)

To perform the packing, we have to answer the question: which vector components move to the same processor? Consider two components, x_j and $x_{j'}$, that reside on the same processor in the old distribution with cycle c_0. Obviously, these components are in the same block of size nc_0/p handled by a group of c_0 processors. Because $c_0 \le c_1$, the block size increases on moving to the new distribution, or stays the same, and because n, p, c_0, c_1 are powers of two, each block of the old distribution fits entirely in a block of the new distribution. Thus, x_j and $x_{j'}$ will automatically be in the same new block of size nc_1/p handled by a group of c_1 processors. Furthermore, they will be on the same processor in the new distribution if and only if j and j' differ by a multiple of c_1. Using eqn (3.34) with $c = c_0$ for j and j', and noting that $j_0 = j'_0$ and $j_2 = j'_2$ because these indices depend only on the processor number, we see that this is equivalent to $j_1 c_0$ and $j'_1 c_0$ differing a multiple of c_1, that is, j_1 and j'_1 differing a multiple of $\texttt{ratio} = c_1/c_0$. Thus we can pack components with local indices $\texttt{j}, \texttt{j} + \texttt{ratio}, \texttt{j} + 2 * \texttt{ratio}, \ldots$, into a temporary array and then put all of these together, as a packet, into the destination processor, which is determined by component `j`. The destination index must be chosen such that two different processors do no write into the same memory location. In general, this can be done by assigning on each receiving processor a piece of memory for each of the other processors. Because of the regularity, the starting point and size of such a location can be computed by the source processor. Since this computation is closely related to the unpacking, we shall treat it together with that operation.

To perform the unpacking, we have to move data from the location they were put into, to their final location on the same processor. Let x_j and $x_{j'}$ be two adjacent components in a packet, with $\mathtt{j}' = \mathtt{j} + \mathtt{ratio}$, and hence $j' = j + (c_1/c_0)c_0 = j + c_1$. Since these components are in the same new block, and their global indices differ by c_1, their new local indices differ by one, $\mathtt{j}' = \mathtt{j} + 1$. We are lucky: if we put x_j into its final location, and the next component of the packet into the next location, and so on, then all components of the packet immediately reach their final destination. In fact, this means that we do not have to unpack!

The function `bspfft` performs the FFT computation itself. It follows Algorithm 3.5 and contains no surprises. The function `bspfft_init` initializes all tables used. It assumes that a suitable amount of storage has been allocated for the tables in the calling program. For the twiddle weights, this amount is $2n/p + p$ reals, cf. eqn (3.61).

The program text is:

```
#include "bspedupack.h"

/****************** Sequential functions ********************************/
void ufft(double *x, int n, int sign, double *w){

    /* This sequential function computes the unordered discrete Fourier
       transform of a complex vector x of length n, stored in a real array
       of length 2n as pairs (Re x[j], Im x[j]), 0 <= j < n.
       n=2^m, m >= 0.
       If sign = 1, then the forward unordered dft FRx is computed;
       if sign =-1, the backward unordered dft conjg(F)Rx is computed,
       where F is the n by n Fourier matrix and R the n by n bit-reversal
       matrix. The output overwrites x.
       w is a table of n/2 complex weights, stored as pairs of reals,
          exp(-2*pi*i*j/n), 0 <= j < n/2,
       which must have been initialized before calling this function.
    */

    int k, nk, r, rk, j, j0, j1, j2, j3;
    double wr, wi, taur, taui;

    for(k=2; k<=n; k *=2){
        nk= n/k;
        for(r=0; r<nk; r++){
            rk= 2*r*k;
            for(j=0; j<k; j +=2){
                wr= w[j*nk];
                if (sign==1) {
                    wi= w[j*nk+1];
                } else {
                    wi= -w[j*nk+1];
                }
                j0= rk+j;
                j1= j0+1;
                j2= j0+k;
                j3= j2+1;
                taur= wr*x[j2] - wi*x[j3];
                taui= wi*x[j2] + wr*x[j3];
                x[j2]= x[j0]-taur;
                x[j3]= x[j1]-taui;
                x[j0] += taur;
                x[j1] += taui;
            }
        }
    }

} /* end ufft */

void ufft_init(int n, double *w){

    /* This function initializes the n/2 weights to be used
       in a sequential radix-2 FFT of length n.
       n=2^m, m >= 0.
       w is a table of n/2 complex weights, stored as pairs of reals,
          exp(-2*pi*i*j/n), 0 <= j < n/2.
    */

    int j, n4j, n2j;
    double theta;

    if (n==1)
        return;
    theta= -2.0 * M_PI / (double)n;
    w[0]= 1.0;
    w[1]= 0.0;
    if (n==4){
        w[2]=  0.0;
        w[3]= -1.0;
    } else if (n>=8) {
        /* weights 1 .. n/8 */
        for(j=1; j<=n/8; j++){
            w[2*j]=   cos(j*theta);
            w[2*j+1]= sin(j*theta);
        }
        /* weights n/8+1 .. n/4 */
        for(j=0; j<n/8; j++){
            n4j= n/4-j;
            w[2*n4j]=   -w[2*j+1];
            w[2*n4j+1]= -w[2*j];
        }
        /* weights n/4+1 .. n/2-1 */
        for(j=1; j<n/4; j++){
            n2j= n/2-j;
            w[2*n2j]=   -w[2*j];
            w[2*n2j+1]=  w[2*j+1];
        }
    }
} /* end ufft_init */

void twiddle(double *x, int n, int sign, double *w){

    /* This sequential function multiplies a complex vector x
       of length n, stored as pairs of reals, componentwise
       by a complex vector w of length n, if sign=1, and
       by conjg(w), if sign=-1. The result overwrites x.
    */

    int j, j1;
    double wr, wi, xr, xi;

    for(j=0; j<2*n; j +=2){
        j1= j+1;
        wr= w[j];
        if (sign==1) {
            wi= w[j1];
        } else {
            wi= -w[j1];
        }
        xr= x[j];
        xi= x[j1];
        x[j]=  wr*xr - wi*xi;
        x[j1]= wi*xr + wr*xi;
    }

} /* end twiddle */

void twiddle_init(int n, double alpha, int *rho, double  *w){

    /* This sequential function initializes the weight table w
       to be used in twiddling with a complex vector of length n,
       stored as pairs of reals.
       n=2^m, m >= 0.
       alpha is a real shift parameter.
       rho is the bit-reversal permutation of length n,
       which must have been initialized before calling this function.
       The output w is a table of n complex values, stored as pairs of
           reals,
          exp(-2*pi*i*rho(j)*alpha/n), 0 <= j < n.
    */

    int j;
    double theta;

    theta= -2.0 * M_PI * alpha / (double)n;
    for(j=0; j<n; j++){
        w[2*j]=   cos(rho[j]*theta);
        w[2*j+1]= sin(rho[j]*theta);
    }

} /* end twiddle_init */
```

```
void permute(double *x, int n, int *sigma){

    /* This in-place sequential function permutes a complex vector x
       of length n >= 1, stored as pairs of reals, by the permutation sigma,
           y[j] = x[sigma[j]], 0 <= j < n.
       The output overwrites the vector x.
       sigma is a permutation of length n that must be decomposable
       into disjoint swaps.
    */

    int j, j0, j1, j2, j3;
    double tmpr, tmpi;

    for(j=0; j<n; j++){
        if (j<sigma[j]){
            /* swap components j and sigma[j] */
            j0= 2*j;
            j1= j0+1;
            j2= 2*sigma[j];
            j3= j2+1;
            tmpr= x[j0];
            tmpi= x[j1];
            x[j0]= x[j2];
            x[j1]= x[j3];
            x[j2]= tmpr;
            x[j3]= tmpi;
        }
    }

} /* end permute */

void bitrev_init(int n, int *rho){

    /* This function initializes the bit-reversal permutation rho
       of length n, with n=2^m, m >= 0.
    */

    int j;
    unsigned int n1, rem, val, k, lastbit, one=1;

    if (n==1){
        rho[0]= 0;
        return;
    }
    n1= n;
    for(j=0; j<n; j++){
        rem= j; /* j= (b(m-1), ... ,b1,b0) in binary */
        val= 0;
        for (k=1; k<n1; k <<= 1){
            lastbit= rem & one; /* lastbit = b(i) with i= log2(k) */
            rem >>= 1;          /* rem = (b(m-1), ... , b(i+1)) */
            val <<= 1;
            val |= lastbit;     /* val = (b0, ... , b(i)) */
        }
        rho[j]= (int)val;
   }

} /* end bitrev_init */

/****************** Parallel functions *******************************/
int k1_init(int n, int p){

    /* This function computes the largest butterfly size k1 of the first
       superstep in a parallel FFT of length n on p processors with p < n.
    */

    int np, c, k1;

    np= n/p;
    for(c=1; c<p; c *=np)
        ;
    k1= n/c;

    return k1;

} /* end k1_init */

void bspredistr(double *x, int n, int p, int s, int c0, int c1,
                char rev, int *rho_p){

    /* This function redistributes the complex vector x of length n,
       stored as pairs of reals, from group-cyclic distribution
       over p processors with cycle c0 to cycle c1, where
       c0, c1, p, n are powers of two with 1 <= c0 <= c1 <= p <= n.
       s is the processor number, 0 <= s < p.
       If rev=true, the function assumes the processor numbering
       is bit reversed on input.
       rho_p is the bit-reversal permutation of length p.
    */

    double *tmp;
    int np, j0, j2, j, jglob, ratio, size, npackets, destproc, destindex, r;

    np= n/p;
    ratio= c1/c0;
    size= MAX(np/ratio,1);
    npackets= np/size;
    tmp= vecallocd(2*size);

    if (rev) {
        j0= rho_p[s]%c0;
        j2= rho_p[s]/c0;
    } else {
        j0= s%c0;
        j2= s/c0;
    }
    for(j=0; j<npackets; j++){
        jglob= j2*c0*np + j*c0 + j0;
        destproc=  (jglob/(c1*np))*c1 + jglob%c1;
        destindex= (jglob%(c1*np))/c1;
        for(r=0; r<size; r++){
            tmp[2*r]=   x[2*(j+r*ratio)];
            tmp[2*r+1]= x[2*(j+r*ratio)+1];
        }
        bsp_put(destproc,tmp,x,destindex*2*SZDBL,size*2*SZDBL);
    }
    bsp_sync();
    vecfreed(tmp);

} /* end bspredistr */

void bspfft(double *x, int n, int p, int s, int sign, double *w0, double *w,
            double *tw, int *rho_np, int *rho_p){

    /* This parallel function computes the discrete Fourier transform
       of a complex array x of length n=2^m, m >= 1, stored in a real array
       of length 2n as pairs (Re x[j], Im x[j]), 0 <= j < n.
       x must have been registered before calling this function.
       p is the number of processors, p=2^q, 0 <= q < m.
       s is the processor number, 0 <= s < p.
       The function uses three weight tables:
           w0 for the unordered fft of length k1,
           w  for the unordered fft of length n/p,
           tw for a number of twiddles, each of length n/p.
       The function uses two bit-reversal permutations:
           rho_np of length n/p,
           rho_p of length p.
       The weight tables and bit-reversal permutations must have been
       initialized before calling this function.
       If sign = 1, then the dft is computed,
           y[k] = sum j=0 to n-1 exp(-2*pi*i*k*j/n)*x[j], for 0 <= k < n.
       If sign =-1, then the inverse dft is computed,
           y[k] = (1/n) sum j=0 to n-1 exp(+2*pi*i*k*j/n)*x[j], for 0 <= k < n.
       Here, i=sqrt(-1). The output vector y overwrites x.
    */

    char rev;
    int np, k1, r, c0, c, ntw, j;
    double ninv;

    np= n/p;
    k1= k1_init(n,p);
    permute(x,np,rho_np);
    rev= TRUE;
    for(r=0; r<np/k1; r++)
        ufft(&x[2*r*k1],k1,sign,w0);

    c0= 1;
    ntw= 0;
    for (c=k1; c<=p; c *=np){
        bspredistr(x,n,p,s,c0,c,rev,rho_p);
        rev= FALSE;
        twiddle(x,np,sign,&tw[2*ntw*np]);
        ufft(x,np,sign,w);
        c0= c;
        ntw++;
    }

    if (sign==-1){
        ninv= 1 / (double)n;
        for(j=0; j<2*np; j++)
            x[j] *= ninv;
    }

} /* end bspfft */

void bspfft_init(int n, int p, int s, double *w0, double *w, double *tw,
                 int *rho_np, int *rho_p){

    /* This parallel function initializes all the tables used in the FFT. */

    int np, k1, ntw, c;
    double alpha;

    np= n/p;
    bitrev_init(np,rho_np);
    bitrev_init(p,rho_p);

    k1= k1_init(n,p);
    ufft_init(k1,w0);
    ufft_init(np,w);

    ntw= 0;
    for (c=k1; c<=p; c *=np){
        alpha= (s%c) / (double)(c);
        twiddle_init(np,alpha,rho_np,&tw[2*ntw*np]);
        ntw++;
    }

} /* end bspfft_init */
```

3.7 Experimental results on an SGI Origin 3800

In this section, we take a critical look at parallel computing. We discuss various ways of presenting experimental results, and we shall experience that things are not always what they seem to be.

The experiments of this section were performed on up to 16 processors of Teras, the national supercomputer in the Netherlands, located in Amsterdam. This Silicon Graphics Origin 3800 machine has 1024 processors and is the successor of the SGI Origin 2000 benchmarked in Section 1.7. The machine consists of six smaller subsystems, the largest of which has 512 processors. Each processor has a MIPS RS14000 CPU with a clock rate of 500 MHz and a theoretical peak performance of 1 Gflop/s, so that the whole machine has a peak computing rate of 1 Tflop/s. (The name Teras comes from this Teraflop/s speed, but also from the Greek word for 'monster', $\tau\epsilon\rho\alpha\varsigma$.) Each processor has a primary data cache of 32 Kbyte, a secondary cache of 8 Mbyte, and a memory of 1 Gbyte. The machine is sometimes classified as a Cache Coherent Non-Uniform Memory Access (CC-NUMA) machine, which means that the user views a shared memory (where the cache is kept coherent—hence the CC) but that physically the memory is distributed (where memory access time is not uniform—hence the NUMA). The BSP architecture assumes uniform access time for remote memory, see Section 1.2, but it assumes faster access to local memory. BSP architectures can therefore be considered NUMA, but with a strict two-level memory hierarchy. (We always have to be careful when using the word 'uniform' and state precisely what it refers to.) As in the case of the Origin 2000, we ignore the shared-memory facility and use the Origin 3800 as a distributed-memory machine.

The Origin 3800 was transformed into a BSP computer by using version 1.4 of BSPlib. We compiled our programs using the standard SGI ANSI C-compiler with optimization flags switched on. These flags are `-O3` for computation and `-flibrary-level 2 bspfifo 10000 -fcombine-puts -fcombine-puts-buffer 256K,128M,4K` for communication. The `-fcombine-puts` flag tells the compiler that puts must be combined, meaning that different messages from the same source to the same destination in the same superstep must be sent together in one packet. This saves time, but requires buffer memory. The `-fcombine-puts-buffer 256K,128M,4K` flag was added to change the default buffer sizes, telling the compiler that the buffer space available on each processor for combining put operations is $2(p-1)\times 256$ Kbyte, that is, a local send and receive buffer of 256 Kbyte for each of the remote processors. The flag also tells the compiler that no more than 128 Mbyte should be used for all buffers of all processors together and that combining puts should be switched off when this upper bound leads to less than 4 Kbyte buffer memory on each processor; this may happen for a large number of processors. We changed the default, which was tailored to the previous Origin 2000 machine, to use the larger memory available on the Origin 3800 and to prevent early switchoff.

The BSP parameters of the Origin 3800 obtained by `bspbench` for various values of p are given by Table 3.3. Note that g is fairly constant, but that l grows quickly with p; this makes larger numbers of processors less useful for

TABLE 3.3. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Silicon Graphics Origin 3800. All times are in flop units ($r = 285$ Mflop/s)

p	g	l	$T_{\mathrm{comm}}(0)$
1	99	55	378
2	75	5118	1414
4	99	12 743	2098
8	126	32 742	4947
16	122	93 488	15 766

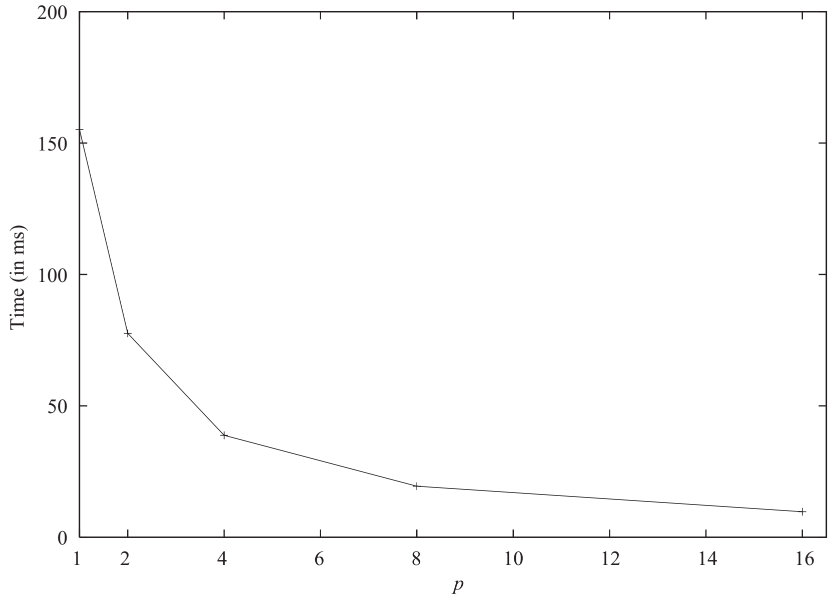


FIG. 3.6. Time of a parallel FFT of length 262 144.

certain applications. For $p = 16$, the synchronization time is already about 100 000 flop units, or 0.3 ms.

For the benchmarking, we had to reduce the compiler optimization level from `-O3` to `-O2`, because the aggressive compilation at level `-O3` removed part of the computation from the benchmark, leading to inexplicably high computing rates. At that level, we cannot fool the compiler any more about our true intention of just measuring computing speed. For the actual FFT timings, however, gains by aggressive optimization are welcome and hence the FFT measurements were carried out at level `-O3`. Note that the predecessor of the Origin 3800, the Origin 2000, was benchmarked in Section 1.7 as having a computing rate $r = 326$ Mflop/s. This rate is higher than the 285 Mflop/s of

TABLE 3.4. Time $T_p(n)$ (in ms) of sequential and parallel FFT on p processors of a Silicon Graphics Origin 3800

p		Length n			
		4096	16 384	65 536	262 144
1	(seq)	1.16	5.99	26.6	155.2
1	(par)	1.32	6.58	29.8	167.4
2		1.06	4.92	22.9	99.4
4		0.64	3.15	13.6	52.2
8		1.18	2.00	8.9	29.3
16		8.44	11.07	9.9	26.8

the Origin 3800, which is partly due to the fact that the predecessor machine was used in dedicated mode, but it must also be due to artefacts from aggressive optimization. This should serve as a renewed warning about the difficulties of benchmarking.

Figure 3.6 shows a set of timing results $T_p(n)$ for an FFT of length $n = 262\,144$. What is your first impression? Does the performance scale well? It seems that for larger numbers of processors the improvement levels off. Well, let me reveal that this figure represents the time of a theoretical, perfectly parallelized FFT, based on a time of 155.2 ms for $p = 1$. Thus, we conclude that presenting results in this way may deceive the human eye.

Table 3.4 presents the raw data of our time measurements for the program `bspfft` of Section 3.6. These data have to be taken with a grain of salt, since timings may suffer from interference by programs from other users (caused for instance by sharing of communication links). To get meaningful results, we ran each experiment three times, and took the best result, assuming that the corresponding run would suffer less from interference. Often we found that the best two timings were within 5% of each other, and that the third result was worse.

Figure 3.7 compares the actual measured execution time for $n = 262\,144$ on an Origin 3800 with the ideal time. Note that for this large n, the measured time is reasonably close to the ideal time, except perhaps for $p = 16$.

The **speedup** $S_p(n)$ of a parallel program is defined as the increase in speed of the program running on p processors compared with the speed of a sequential program (with the same level of optimization),

$$S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)}. \tag{3.62}$$

Note that we do not take the time of the parallel program with $p = 1$ as reference time, since this may be too flattering; obtaining a good speedup with such a reference for comparison may be reason for great pride, but it is

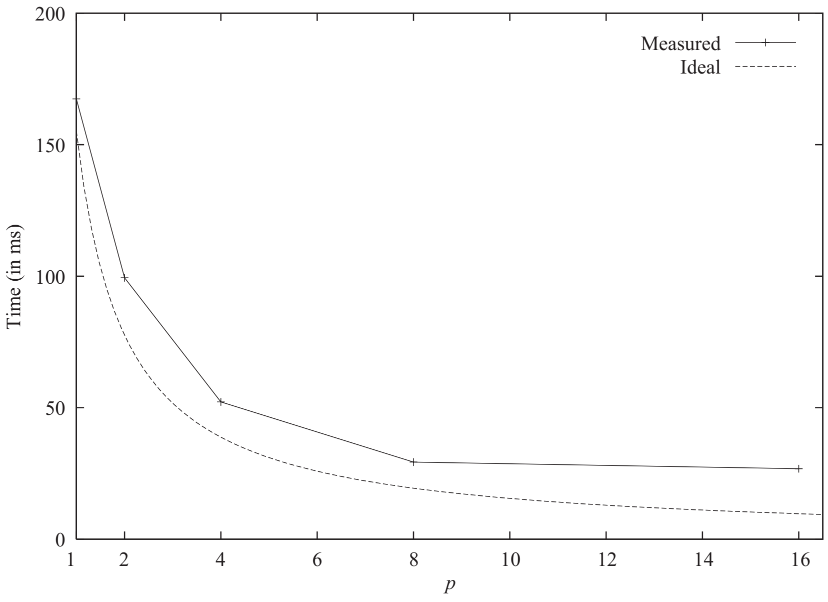


FIG. 3.7. Time T_p of actual parallel FFT of length 262 144.

an achievement of the same order as becoming the Dutch national champion in alpine skiing. The latter achievement is put into the right perspective if you know that the Netherlands is a very flat country, which does not have skiers of international fame. Similarly, the speedup achieved on a parallel computer can only be put into the right perspective if you know the time of a good sequential program. A parallel program run on one processor will always execute superfluous operations, and for $n = 262\,144$ this overhead amounts to about 8%. (Sometimes, it is unavoidable to measure speedup vs. the parallel program with $p = 1$, for instance if it is too much work to develop a sequential version, or if one adheres to the purist principle of maintaining a single program source: 'a sequential program is a parallel program run on one processor'. In such a situation, speedups should be reported with a clear warning about the reference version.) Figure 3.8 gives the speedup for $n = 65\,536,\ 262\,144$. The largest speedup obtained is about 5.95. This way of presenting timings of parallel programs gives much insight: it exposes good as well as bad scaling behaviour and it also allows easy comparison between measurements for different lengths.

In principle, $0 \leq S_p(n) \leq p$ should hold, because p processors cannot be more than p times faster than one processor. This is indeed true for our measured speedups. Practice, however, may have its surprises: in certain situations, a **superlinear speedup** of more than p can be observed. This phenomenon is often the result of cache effects, due to the fact that in the parallel case each processor has less data to handle than in the sequential case, so that a larger

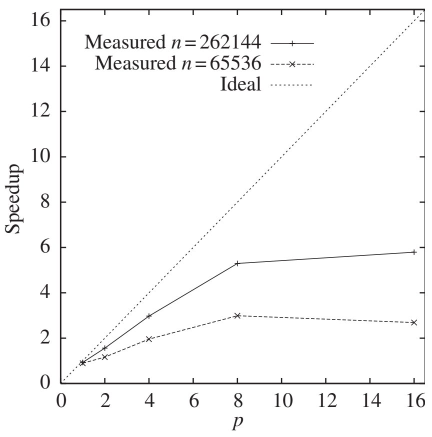


FIG. 3.8. Speedup $S_p(n)$ of parallel FFT.

part of the computation can be carried out using data that are already in cache, thus yielding fewer cache misses and a higher computing rate. Occurrence of superlinear speedup in a set of experimental results should be a warning to be cautious when interpreting results, even for results that are not superlinear themselves. In our case, it is also likely that we benefit from cache effects. Still, one may argue that the ability to use many caches simultaneously is a true benefit of parallel computing.

The **efficiency** $E_p(n)$ gives the fraction of the total computing power that is usefully employed. It is defined by

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_{\text{seq}}(n)}{pT_p(n)}. \tag{3.63}$$

In general, $0 \leq E_p(n) \leq 1$, with the same caveats as before. Figure 3.9 gives the efficiency for $n = 65\,536,\ 262\,144$.

Another measure is the **normalized cost** $C_p(n)$, which is just the time of the parallel program divided by the time that would be taken by a perfectly parallelized version of the sequential program. This cost is defined by

$$C_p(n) = \frac{T_p(n)}{T_{\text{seq}}(n)/p}. \tag{3.64}$$

Note that $C_p(n) = 1/E_p(n)$, which explains why this cost is sometimes called the **inefficiency**. Figure 3.10 gives the cost of the FFT program for $n = 65\,536,\ 262\,144$. The difference between the normalized cost and the ideal value of 1 is the parallel **overhead**, which usually consists of load imbalance, communication time, and synchronization time. A breakdown of the overhead into its main parts can be obtained by performing additional measurements, or theoretically by predictions based on the BSP model.

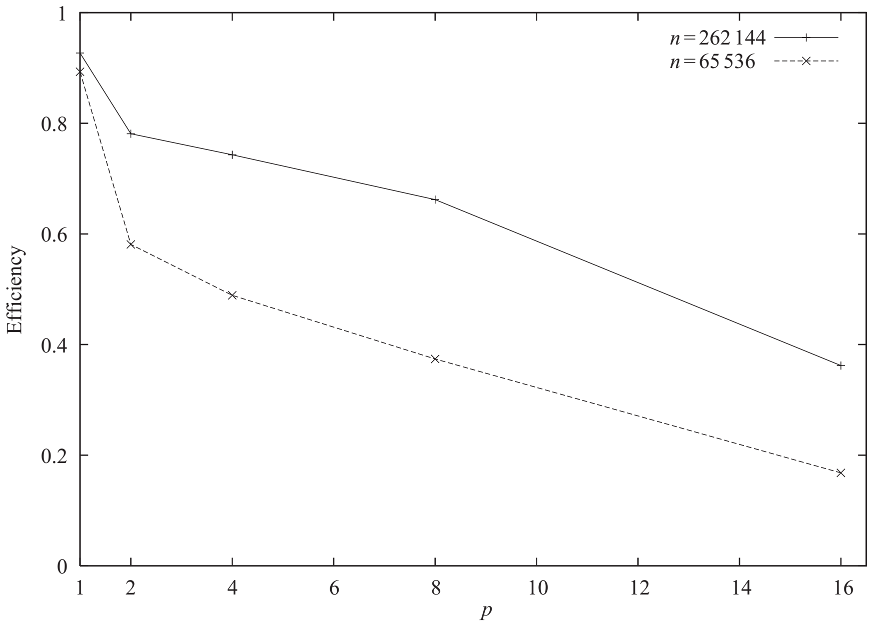


FIG. 3.9. Measured efficiency $E_p(n)$ of parallel FFT. The ideal value is 1.

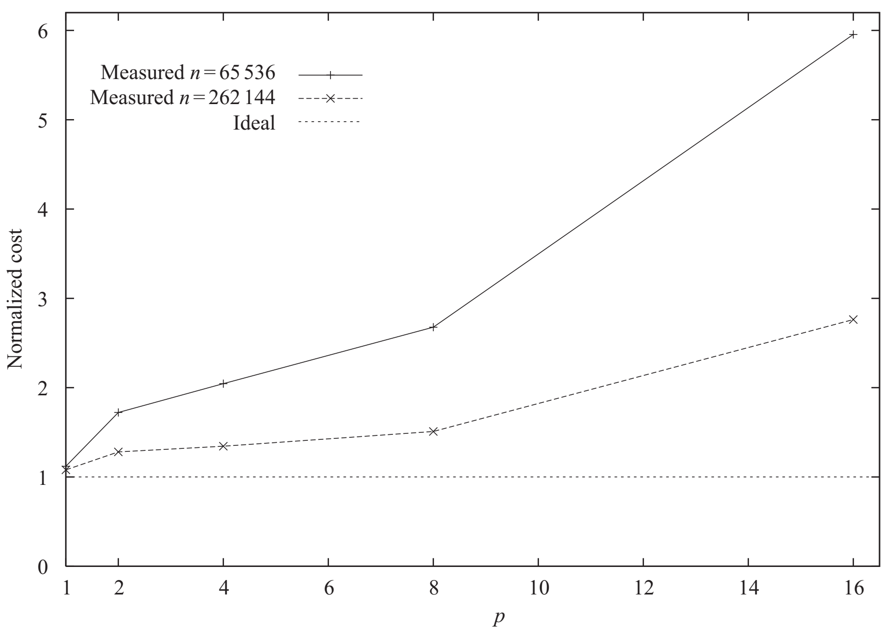


FIG. 3.10. Normalized cost $C_p(n)$ of parallel FFT.

TABLE 3.5. Breakdown of predicted execution time (in ms) of the parallel FFT program for $n = 262\,144$

p	T_{comp}	T_{comm}	T_{sync}	T_{FFT} (pred.)	T_p (meas.)
1	82.78	0.00	0.00	82.78	167.4
2	41.39	68.99	0.05	110.43	99.4
4	20.70	45.53	0.13	66.36	52.2
8	10.35	28.97	0.35	39.67	29.3
16	5.17	14.03	0.98	20.18	26.8

For comparison, the measured time is also given.

Table 3.5 predicts the computation, communication, and synchronization time of the FFT for $n = 262\,144$, based on the costs $5n \log_2 n + 0g + l$ for $p = 1$ and $(5n \log_2 n)/p + 2ng/p + 3l$ for $p > 1$, cf. eqn (3.40).

The first observation we can make from the table is that it is difficult to predict the computation time correctly. The program takes about twice the time predicted for $p = 1$. This is because our benchmark measured a computing rate of 285 Mflop/s for a small DAXPY, which fits into the primary cache, but for an FFT of length 262 144, a single processor has to handle an array of over 4 Mbyte for the complex vector $\mathbf{x}$ alone, thus exceeding the primary cache and filling half the secondary cache. The measured rate for $p = 1$ is about 144 Mflop/s. Thus, the prediction considerably underestimates the actual computing time. For larger p, the misprediction becomes less severe.

From the table, we also learn that the FFT is such a data-intensive computation that one permutation of the data vector already has a serious impact on the total execution time of a parallel FFT program. The prediction overestimates the communication time, because it is based on a pessimistic g-value, whereas the actual parallel FFT was optimized to send data in large packets. The prediction also overestimates the synchronization time, because for $p = 1$ we counted l (for a computation superstep) but in reality there is no synchronization, and for $p > 1$ we counted $3l$ (for two computation supersteps and one communication superstep) where there is only one synchronization. This does not matter much, because the synchronization time is insignificant. The overall prediction matches the measurement reasonably well, except for $p = 1$, and the reason is that the errors made in estimating computation time and communication time cancel each other to a large extent. Of course, it is possible to predict better, for example, by measuring the computing rate for this particular application and using that rate instead of r, and perhaps

TABLE 3.6. Computing rate $R_p(n)$ (in Mflop/s) of sequential and parallel FFT on p processors of a Silicon Graphics Origin 3800

p		Length n			
		4096	16 384	65 536	262 144
1	(seq)	220	197	202	155
1	(par)	193	179	180	144
2		239	240	234	243
4		397	375	395	462
8		216	591	607	824
16		30	107	545	900

even using a rate that depends on the local vector length. The communication prediction can be improved by measuring optimistic g-values.

Table 3.6 shows the computing rate $R_p(n)$ of all processors together for this application, defined by

$$R_p(n) = \frac{5n \log_2 n}{T_p(n)}, \tag{3.65}$$

where we take the standard flop count $5n \log_2 n$ as basis (as is customary for all FFT counts, even for highly optimized FFTs that perform fewer flops). The flop rate is useful in comparing results for different problem sizes and also for different applications. Furthermore, it tells us how far we are from the advertised peak performance. It is a sobering thought that we need at least four processors to exceed the top computing rate of 285 Mflop/s measured for an in-cache DAXPY operation on a single processor. Thus, instead of parallelizing, it may be preferable to make our sequential program cache-friendly. If we still need more speed, we turn to parallelism and make our parallel program cache-friendly. (Exercise 5 tells you how to do this.) Making a parallel program cache-friendly will decrease running time and hence increase the computing rate, but paradoxically it will also decrease the speedup and the efficiency, because the communication part remains the same while the computing part is made faster in both the parallel program and the sequential reference program. Use of a parallel computer for the one-dimensional FFT can therefore only be justified for very large problems. But were not parallel computers made exactly for that purpose?

3.8 Bibliographic notes

3.8.1 *Sequential FFT algorithms*

The basic idea of the FFT was discovered already in 1805 by (who else?) Gauss [74]. It has been rediscovered several times: by Danielson and Lanczos [51] in 1942 and by Cooley and Tukey [45] in 1965. Because Cooley and Tukey's rediscovery took place in the age of digital computers, their FFT algorithm found immediate widespread use in computer programs. As a result, the FFT became connected to their name. Cooley [44] tells the whole story of the discovery in a paper *How the FFT gained acceptance.* He states that one reason for the widespread use of the FFT is the decision made at IBM, the employer of Cooley at that time, to put the FFT algorithm in the public domain and not to try to obtain a patent on this algorithm. In concluding his paper, Cooley recommends not to publish papers in neoclassic Latin (as Gauss did). Heideman, Johnson, and Burrus [92] dug up the prehistory of the FFT and wrote a historical account on the FFT from Gauss to modern times. An FFT bibliography from 1995 by Sorensen, Burrus, and Heideman [165] contains over 3400 entries. Some entrance points into the vast FFT literature can be found below.

A large body of work such as the work done on FFTs inevitably contains much duplication. Identical FFT algorithms have appeared in completely different formulations. Van Loan in his book *Computational Frameworks for the Fast Fourier Transform* [187] has rendered us the great service of providing a unified treatment of many different FFT algorithms. This treatment is based on factorizing the Fourier matrix and using powerful matrix notations such as the Kronecker product. The book contains a wealth of material and it is the first place to look for suitable variants of the FFT. We have adopted this framework where possible, and in particular we have used it in describing sequential FFTs and GFFTs, and the local computations of parallel FFTs. A different look at FFT algorithms, ordered and unordered, DIF and DIT, sequential and parallel, is given by Chu and George in their book *Inside the FFT Black Box* [43]. Their book is aimed at a computing science and engineering audience; it presents FFT methods in the form of highly detailed algorithms and not in the more abstract form of matrix factorizations.

For those who want to learn more about using the DFT, but care less about how it is actually computed, the book by Briggs and Henson [33] is a good source. It discusses the various forms of the DFT and their relation with continuous Fourier transforms, two-dimensional DFTs, applications such as the reconstruction of images from projections, and related transforms. Bracewell [31] gives a detailed discussion of the continuous Fourier transform; a good understanding of the continuous case is necessary for explaining the results of a discrete transform. The book by Bracewell includes a series of pictures of functions and their continuous Fourier transforms and also a biography of Fourier.

Sequential implementations of various FFTs can be found in *Numerical Recipes in C: The Art of Scientific Computing* by Press, Teukolsky, Vetterling, and Flannery [157]. This book devotes two chapters to FFTs and their application. Programs included are, among others, complex-valued FFT, real-valued FFT, fast sine transform, fast cosine transform, multidimensional FFT, out-of-core FFT, and convolution.

The fastest Fourier transform in the West (FFTW) package by Frigo and Johnson [73] is an extremely fast sequential program. The speed of FFTW comes from the use of **codelets**, straight inline code without loops, in core parts of the software and from the program's ability to adapt itself to the hardware used. Instead of using flop counts or user-provided knowledge about the hardware (such as cache sizes) to optimize performance, the program carries out a set of FFT timings on the hardware, resulting in a **computation plan** for each FFT length. Using actual timings is better than counting flops, since most of the execution time of an FFT is spent in moving data around between registers, caches, and main memory, and not in floating-point operations. A plan is used in all runs of the FFT of the corresponding length. An optimal plan is chosen by considering several possible plans: an FFT of length $n = 128$ can be split into two FFTs of length 64 by a so-called radix-2 approach (used in this chapter) but it can also be split into four FFTs of length 32 by a radix-4 approach, and so on. Optimal plans for smaller FFTs are used in determining plans for larger FFTs. Thus, the plans are computed bottom-up, starting with the smallest lengths. The large number of possibilities to be considered is reduced by **dynamic programming**, that is, the use of optimal solutions to subproblems when searching for an optimal solution to a larger problem. (If a solution to a problem is optimal, its solutions of subproblems must also be optimal; otherwise, the overall solution could have been improved.) FFTW is recursive, with the advantage that sufficiently small subproblems fit completely in the cache. The program can also handle FFT lengths that are not a power of two. A parallel version of FFTW exists; it uses the block distribution on input and has two or three communication supersteps, depending on whether the output must be redistributed into blocks.

Johnson *et al.* [114] describe the signal processing language (SPL), a Lisp-like programming language which can be used to implement factorization formulae directly for transform matrices (such as Fourier and Walsh–Hadamard matrices) arising in digital signal processing applications. Many different factorization formulae for the same matrix F_n can be generated automatically using mathematical transformation rules such as the properties of Kronecker products and the radix-m splitting formula,

$$F_{mn} = (F_m \otimes I_n)T_{m,mn}(I_m \otimes F_n)S_{m,mn}, \tag{3.66}$$

which is the basis for the Cooley–Tukey approach [45]. Here, the twiddle matrix $T_{m,N}$ is an $N \times N$ diagonal matrix with $(T_{m,N})_{j(N/m)+k,j(N/m)+k} = \omega_N^{jk}$

for $0 \le j < m$ and $0 \le k < N/m$. Furthermore, $S_{m,N}$ is the **Mod-m sort matrix**, the $N \times N$ permutation matrix defined by

$$S_{m,N}\mathbf{x} = \begin{bmatrix} x(0\colon m\colon N-1) \\ x(1\colon m\colon N-1) \\ \vdots \\ x(m-1\colon m\colon N-1) \end{bmatrix}, \tag{3.67}$$

which has as inverse $S_{m,N}^{-1} = S_{N/m,N}$. Note that $T_{2,N} = \mathrm{diag}(I_{N/2}, \Omega_{N/2})$ and $S_{2,N} = S_N$. The SPL compiler translates each factorization formula into a Fortran program. In the same spirit as FFTW, an extensive search is carried out over the space of factorization formulae and compiler techniques. A Kronecker-product property used by SPL which is important for a parallel context is: for every $m \times m$ matrix A and $n \times n$ matrix B,

$$A \otimes B = S_{m,mn}(I_n \otimes A)S_{n,mn}(I_m \otimes B). \tag{3.68}$$

As a special case, we have

$$F_m \otimes I_n = S_{m,mn}(I_n \otimes F_m)S_{n,mn}. \tag{3.69}$$

3.8.2 *Parallel FFT algorithms with* $\log_2 p$ *or more supersteps*

Cooley and Tukey [45] already observed that all butterflies of one stage can be computed in parallel. The first parallel FFT algorithm was published by Pease [155] in 1968, a few years after the Cooley–Tukey paper. Pease presents a matrix decomposition of the Fourier matrix in which he uses the Kronecker-product notation. Each of the $\log_2 n$ stages of his algorithm requires a so-called **perfect shuffle** permutation $S_{n/2,n} = S_n^{-1}$; this would make an actual implementation on a general-purpose computer expensive. The algorithm, however, was aimed at implementation in special-purpose hardware, with specialized circuitry for butterflies, shuffles, and twiddles.

Fox *et al.* [71] present a parallel recursive algorithm based on the block distribution, which decomposes the FFT into a sequence of n/p calls to FFTs of length p and combines the results. The decomposition and combination parts of the algorithm are carried out without communication. Each FFT of length p, however, requires much communication since there is only one vector component per processor. Another disadvantage, which is true of every straightforward implementation of a recursive algorithm, is that the smaller tasks of the recursion are executed one after the other, where in principle they could have been done in parallel.

The nonrecursive parallel FFT algorithms proposed in the earlier literature typically perform $\log_2(n/p)$ stages locally without communication and $\log_2 p$ stages nonlocally involving both communication and computation, thus requiring a total of $\mathcal{O}(\log_2 p)$ communication supersteps. These algorithms were mostly designed targeting the hypercube architecture, where each processor $s = (b_{q-1} \cdots b_0)_2$ is connected to the $q = \log_2 p$ processors that differ exactly one bit with s in their processor number.

Examples of algorithms from this category are discussed by Van Loan [187, Algorithm 3.5.3], by Dubey, Zubair, and Grosch [62], who present a variant that can use every input and output distribution from the block-cyclic family, and by Gupta and Kumar [87] (see also [82]), who describe the so-called **binary exchange** algorithm. Gupta and Kumar analyse the scalability of this algorithm by using the isoefficiency function $f_E(p)$, which expresses how fast the amount of work of a problem must grow with p to maintain a constant efficiency E.

Another example is an algorithm for the hypercube architecture given by Swarztrauber [172] which is based on index-digit permutations [72]: each permutation τ on the set of m bits $\{0, \ldots, m-1\}$ induces an **index-digit permutation** which moves the index $j = (b_{m-1} \cdots b_1 b_0)_2$ into $j' = (b_{\tau(m-1)} \cdots b_{\tau(1)} b_{\tau(0)})_2$. Using the block distribution for $n = 2^m$ and $p = 2^q$, an index is split into the processor number $s = (b_{m-1} \cdots b_{m-q})_2$ and the local index $\mathsf{j} = (b_{m-q-1} \cdots b_1 b_0)_2$. An ***i*-cycle** is an index-digit permutation where τ is a swap of the **pivot** bit $m-q-1$ with another bit r. Swarztrauber notes that no communication is needed if $r \leq m-q-1$; otherwise, the i-cycle permutation requires communication, but it still has the advantage of moving data in large chunks, namely blocks of size $n/(2p)$. Every index-digit permutation can be carried out as a sequence of i-cycles. Every butterfly operation of an FFT combines pairs (j, j') that differ in exactly one bit. An i-cycle can be used to make this bit local.

3.8.3 *Parallel FFT algorithms with $\mathcal{O}(1)$ supersteps*

We can make parallel FFT algorithms more efficient if we manage to combine many butterfly stages into one superstep (or pair of supersteps); this way, they require less communication and synchronization. Under mild assumptions, such as $p \leq \sqrt{n}$, this leads to algorithms with only a small constant number of supersteps. The BSP algorithm presented in this chapter falls into this category.

The original BSP paper by Valiant [178] discusses briefly the BSP cost of an algorithm by Papadimitriou and Yannakakis [153] which achieves a total cost within a constant factor of optimal if $g = \mathcal{O}(\log_2(n/p))$ and $l = \mathcal{O}((n/p)\log_2(n/p))$; for such a ratio n/p, our algorithm also achieves optimality, cf. eqn (3.39). For the common special case $p \leq \sqrt{n}$, this method has been implemented in an unordered DIF FFT by Culler *et al.* [47], demonstrating the application of the LogP model. The computation starts with a cyclic distribution and then switches to a block distribution. The authors avoid contention during the redistribution by scheduling the communications carefully and by inserting additional global synchronizations to enforce strict adherence to the communication schedule. (A good BSPlib system would do this automatically.)

Gupta *et al.* [88] use data redistributions to implement parallel FFTs. For the unordered Cooley–Tukey algorithm, they start with the block distribution, finish with the cyclic distribution, and use block-cyclic distributions in between. If a bit reversal must be performed and the output must be distributed in the same way as the input, this requires three communication supersteps for $p \le \sqrt{n}$. The authors also modify the ordered Stockham algorithm so they can start and finish with the cyclic distribution, and perform only one communication superstep for $p \le \sqrt{n}$. Thus they achieve the same minimal communication cost as Algorithms 3.3 and 3.5. Experimental results on an Intel iPSC/860 hypercube show that the modified Stockham algorithm outperforms all other implemented algorithms.

McColl [135] presents a detailed BSP algorithm for an ordered FFT, which uses the block distribution on input and output. The algorithm starts with an explicit bit-reversal permutation and it finishes with a redistribution from cyclic to block distribution. Thus, for $p > 1$ the algorithm needs at least three communication supersteps. Except for the extra communication at the start and finish, the algorithm of McColl is quite similar to our algorithm. His algorithm stores and communicates the original index of each vector component together with its numerical value. This facilitates the description of the algorithm, but the resulting communication should be removed in an implementation because in principle the original indices can be computed by every processor. Furthermore, the exposition is simplified by the assumption that $m - q$ is a divisor of m. This implies that $p = 1$ or $p \ge \sqrt{n}$; it is easy to generalize the algorithm so that it can handle the most common case $1 < p < \sqrt{n}$ as well.

The algorithm presented in this chapter is largely based on work by Inda and Bisseling [111]. This work introduces the group-cyclic distribution and formulates redistributions as permutations of the data vector. For example, changing the distribution from block to cyclic has the same effect as keeping the distribution constant but performing an explicit permutation $S_{p,n}$. (To be more precise, the processor and local index of the original data element x_j are the same in both cases.) For $p \le \sqrt{n}$, the factorization is written as

$$F_n = S_{p,n}^{-1} A S_{p,n} (R_p \otimes I_{n/p})(I_p \otimes F_{n/p}) S_{p,n}, \tag{3.70}$$

where A is a block-diagonal matrix with blocks of size n/p,

$$A = S_{p,n}(I_1 \otimes B_n) \cdots (I_{p/2} \otimes B_{2n/p}) S_{p,n}^{-1}. \tag{3.71}$$

This factorization needs three permutations if the input and output are distributed by blocks, but it needs only one permutation if these distributions are cyclic. Thus the cyclic I/O distribution is best.

Another related algorithm is the **transpose** algorithm, which calculates a one-dimensional FFT of size mn by storing the vector $\mathbf{x}$ as a two-dimensional matrix of size $m \times n$. Component x_j is stored as matrix element $X(j_0, j_1)$, where $j = j_0 n + j_1$ with $0 \le j_0 < m$ and $0 \le j_1 < n$. This algorithm is based on the observation that in the first part of an unordered FFT components within the same matrix row are combined, whereas in the second part components within the same column are combined. The matrix can be transposed between the two parts of the algorithm, so that all butterflies can be done within rows. In the parallel case, each processor can then handle one or more rows. The only communication needed is in the matrix transposition and the bit reversal. For a description and experimental results, see, for example, Gupta and Kumar [87]. This approach works for $p \leq \min(m, n)$. Otherwise, the transpose algorithm must be generalized to a higher dimension. A detailed description of such a generalization can be found in the book by Grama and coworkers [82]. Note that this algorithm is similar to the BSP algorithm presented here, except that its bit-reversal permutation requires communication.

The two-dimensional view of a one-dimensional FFT can be carried one step further by formulating the algorithm such that it uses explicit shorter-length FFTs on the rows or columns of the matrix storing the data vector. Van Loan [187, Section 3.3.1] calls the corresponding approach the **four-step framework** and the **six-step framework**. This approach is based on a mixed-radix method due to Agarwal and Cooley [2] (who developed it for the purpose of vectorization). The six-step framework is equivalent to a factorization into six factors,

$$F_{mn} = S_{m,mn}(I_n \otimes F_m)S_{n,mn}T_{m,mn}(I_m \otimes F_n)S_{m,mn}. \tag{3.72}$$

This factorization follows immediately from eqns (3.66) and (3.69). In a parallel algorithm based on eqn (3.72), with the block distribution used throughout, the only communication occurs in the permutations $S_{m,mn}$ and $S_{n,mn}$. The four-step framework is equivalent to the factorization

$$F_{mn} = (F_m \otimes I_n)S_{m,mn}T_{n,mn}(F_n \otimes I_m), \tag{3.73}$$

which also follows from eqns (3.66) and (3.69). Note that now the shorter-length Fourier transforms need strided access to the data vector, with stride n for $F_m \otimes I_n$. (In a parallel implementation, such access is local if a cyclic distribution is used, provided $p \leq \min(m, n)$.)

One advantage of the four-step and six-step frameworks is that the shorter-length FFTs may fit into the cache of computers that cannot accommodate an FFT of full length. This may result in much higher speeds on cache-sensitive computers. The use of genuine FFTs in this approach makes it possible to call fast system-specific FFTs in an implementation. A disadvantage is that the multiplication by the twiddle factors takes an additional $6n$ flops. In a parallel implementation, the communication is nicely isolated in the permutations.

Hegland [91] applies the four-step framework twice to generate a factorization of F_{mnm} with m maximal, treating FFTs of length m recursively by the same method. He presents efficient algorithms for multiple FFTs and multidimensional FFTs on vector and parallel computers. In his implementation, the key to efficiency is a large vector length of the inner loops of the computation.

Edelman, McCorquodale, and Toledo [68] present an approximate parallel FFT algorithm aimed at reducing communication from three data permutations (as in the six-step framework) to one permutation, at the expense of an increase in computation. As computation rates are expected to grow faster than communication rates, future FFTs are likely to be communication-bound, making such an approach worthwhile. The authors argue that speedups in this scenario will be modest, but that using parallel FFTs will be justified on other grounds, for instance because the data do not fit in the memory of a single processor, or because the FFT is part of a larger application with good overall speedup.

3.8.4 *Applications*

Applications of the FFT are ubiquitous. Here, we mention only a few with emphasis on parallel applications. Barros and Kauranne [13] and Foster and Worley [70] parallelize the spectral transform method for solving partial differential equations on a sphere, aimed at global weather and climate modelling. The spectral transform for a two-dimensional latitude/longitude grid consists of a DFT along each latitude (i.e. in the east–west direction) and a discrete Legendre transform (DLT) along each longitude (i.e. in the north–south direction). Barros and Kauranne [13] redistribute the data between the DFT and the DLT, so that these computations can be done sequentially. For instance, each processor performs several sequential FFTs. The main advantages of this approach are simplicity, isolation of the communication parts (thus creating bulk), and reproducibility: the order of the computations is exactly the same as in a sequential computation and it does not depend on the number of processors used. (Weather forecasts being that fragile, they must at least be reproducible in different runs of the same program. No numerical butterfly effects please!) Foster and Worley [70] also investigate the alternative approach of using parallel one-dimensional FFTs and Legendre transforms as a basis for the spectral transform.

Nagy and O'Leary [143] present a method for restoring blurred images taken by the Hubble Space Telescope. The method uses a preconditioned conjugate gradient solver with fast matrix-vector multiplications carried out by using FFTs. This is possible because the matrices involved are **Toeplitz matrices**, that is, they have constant diagonals: $a_{ij} = \alpha_{i-j}$, for all i, j.

The FFT is the computational workhorse in grid methods for quantum molecular dynamics, where the time-dependent Schrödinger equation is solved numerically on a multidimensional grid, see a review by Kosloff [123] and a comparison of different time propagation schemes by Leforestier *et al.* [127]. In each time step, a potential energy operator is multiplied by a wavefunction, which is a local (i.e. point-wise) operation in the spatial domain. This means that every possible distribution including the cyclic one can be used in a parallel implementation. The kinetic energy operator is local in the transformed domain, that is, in momentum space. The transformation between the two domains is done efficiently using the FFT. Here, the cyclic distribution as used in our parallel FFT is applicable in both domains. In one dimension, the FFT thus has only one communication superstep.

Haynes and Côté [90] parallelize a three-dimensional FFT for use in an electronic structure calculation based on solving the time-independent Schrödinger equation. The grids involved are relatively small compared with those used in other FFT applications, with a typical size of only $128\times128\times128$ grid points. As a consequence, reducing the communication is of prime importance. In momentum space, the grid is cyclically distributed in each dimension to be split. In the spatial domain, the distribution is by blocks. The number of communication supersteps is $\log_2 p$. An advantage of the cyclic distribution in momentum space is a better load balance: momentum is limited by an energy cut-off, which means that all array components outside a sphere in momentum space are zero; in the cyclic distribution, all processors have an approximately equal part of this sphere.

Zoldi *et al.* [195] solve the nonlinear Schrödinger equation in a study on increasing the transmission capacity of optical fibre lines. They apply a parallel one-dimensional FFT based on the four-step framework. They exploit the fact that the Fourier transform is carried out in both directions, avoiding unnecessary permutations and combining operations from the forward FFT and the inverse FFT on the same block of data. This results in better cache use and even causes a speedup of the parallel program with $p = 1$ compared with the original sequential program.

3.9 Exercises

1. The recursive FFT, Algorithm 3.1, splits vectors repeatedly into two vectors of half the original length. For this reason, we call it a radix-2 FFT. We can generalize the splitting method by allowing to split the vectors into r parts of equal length. This leads to a radix-r algorithm.

(a) How many flops are actually needed for the computation of $F_4\mathbf{x}$, where $\mathbf{x}$ is a vector of length four? Where does the gain in this specific case come from compared to the $5n\log_2 n$ flops of an FFT of arbitrary length n?

(b) Let n be a power of four. Derive a sequential recursive radix-4 FFT algorithm. Analyse the computation time and compare the number of flops with that of a radix-2 algorithm. Is the new algorithm faster?

(c) Let n be a power of four. Formulate a sequential nonrecursive radix-4 algorithm. Invent an appropriate name for the new starting permutation. Can you modify the algorithm to handle all powers of two, for example by ending with a radix-2 stage if needed?

(d) Let n be a power of two. Implement your nonrecursive radix-4 algorithm in a sequential function `fft4`. Compare its performance with a sequential radix-2 function based on functions used in `bspfft`. Explain your results. Is the difference in performance only due to a difference in flop count?

(e) Modify `fft4` for use in a parallel program, replacing the unordered `ufft` in `bspfft`. The new function, `ufft4`, should have exactly the same input and output specification as `ufft`. Note: take special care that the ordering is correct before the butterflies start flying.

2. $(*)$ Let f be a T-periodic smooth function. The Fourier coefficients c_k of f are given by eqn (3.2).

(a) Let d_k be the kth Fourier coefficient of the derivative f'. Prove that $d_k = 2\pi i k c_k / T$. How can you use this relation to differentiate f?

(b) The Fourier coefficients c_k, $k \in \mathbf{Z}$, can be obtained in approximation by eqn (3.3). The quality of the approximation depends on how well the trapezoidal rule estimates the integral of the function $f(t)e^{-2\pi ikt/T}$ on the subintervals $[t_j, t_{j+1}]$. The average of the two function values in the endpoints t_j and t_{j+1} is a good approximation of the integral on $[t_j, t_{j+1}]$ if the function $f(t)e^{-2\pi ikt/T}$ changes slowly on the subinterval. On the other hand, if the function changes quickly, the approximation is poor. Why is the approximation meaningless for $|k| > n/2$? For which value of k is the approximation best? In answering these questions, you may assume that f itself behaves well and is more or less constant on each subinterval $[t_j, t_{j+1}]$.

(c) Let n be even. How can you compute (in approximation) the set of coefficients $c_{-n/2+1}, \ldots, c_{n/2}$ by using a DFT? Hint: relate the y_k's of eqn (3.4) to the approximated c_k's.

(d) Write a parallel program for differentiation of a function f sampled in $n = 2^m$ points. The program should use a forward FFT to compute the Fourier coefficients c_k, $-n/2 + 1 \le k \le n/2$, then compute the corresponding d_k's, and finally perform an inverse FFT to obtain the derivative f' in the original sample points. Use the cyclic distribution. Test the accuracy of your program by comparing your results with analytical results, using a suitable test function. Take for instance the Gaussian function $f(t) = \mathrm{e}^{-\alpha(t-\beta)^2}$, and place it in the middle of your sampling interval $[0, T]$ by choosing $\beta = T/2$.

3. $(*)$ The **two-dimensional discrete Fourier transform** (2D DFT) of an $n_0 \times n_1$ matrix X is defined as the $n_0 \times n_1$ matrix Y given by

$$Y(k_0, k_1) = \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} X(j_0, j_1) \omega_{n_0}^{j_0 k_0} \omega_{n_1}^{j_1 k_1}, \tag{3.74}$$

for $0 \leq k_0 < n_0$ and $0 \leq k_1 < n_1$. We can rewrite this as

$$Y(k_0, k_1) = \sum_{j_0=0}^{n_0-1} \left(\sum_{j_1=0}^{n_1-1} X(j_0, j_1)\omega_{n_1}^{j_1 k_1} \right) \omega_{n_0}^{j_0 k_0}, \qquad (3.75)$$

showing that the 2D DFT is equivalent to a set of n_0 1D DFTs of length n_1, each in the row direction of the matrix, followed by a set of n_1 1D DFTs of length n_0, each in the column direction.

(a) Write a function `bspfft2d` that performs a 2D FFT, assuming that X is distributed by the $M \times N$ cyclic distribution with $1 \leq M < n_0$ and $1 \leq N < n_1$, and with M, N powers of two. The result Y must be in the same distribution as X. Use the function `bspfft` to perform parallel 1D FFTs, but modify it to perform several FFTs together in an efficient manner. In particular, avoid unnecessary synchronizations.
(b) As an alternative, write a function `bspfft2d_transpose` that first performs sequential 1D FFTs on the rows of X, assuming that X is distributed by the cyclic row distribution, then transposes the matrix, performs sequential 1D FFTs on its rows, and finally transposes back to return to the original distribution.
(c) Let $n_0 \geq n_1$. For each function, how many processors can you use? Compare the theoretical cost of the two implemented algorithms. In particular, which algorithm is better in the important case $p \leq \sqrt{n_0}$? Hint: you can choose M, N freely within the constraint $p = MN$; use this freedom well.
(d) Compare the performance of the two functions experimentally.
(e) Optimize the best of the two functions.

4. (∗) The **three-dimensional discrete Fourier transform** (3D DFT) of an $n_0 \times n_1 \times n_2$ array X is defined as the $n_0 \times n_1 \times n_2$ array Y given by

$$Y(k_0, k_1, k_2) = \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} X(j_0, j_1, j_2)\omega_{n_0}^{j_0 k_0}\omega_{n_1}^{j_1 k_1}\omega_{n_2}^{j_2 k_2}, \qquad (3.76)$$

for $0 \leq k_d < n_d$, $d = 0, 1, 2$.

(a) Write a function `bspfft3d`, similar to `bspfft2d` from Exercise 3, that performs a 3D FFT, assuming that X is distributed by the $M_0 \times M_1 \times M_2$ cyclic distribution, where M_d is a power of two with $1 \leq M_d < n_d$, for $d = 0, 1, 2$. The result Y must be in the same distribution as X.
(b) Explain why each communication superstep of the parallel 3D FFT algorithm has the same cost.
(c) In the case $n_0 = n_1 = n_2 = n$, how do you choose the M_d? Hint: for $p \leq \sqrt{n}$ you need only one communication superstep; for $p \leq n$ only two.

(d) For arbitrary n_0, n_1, n_2, what is the maximum number of processors you can use? Write a function that determines, for a given p, the triple (M_0, M_1, M_2) with $M_0 M_1 M_2 = p$ that causes the least number of communication supersteps.

(e) Test your parallel 3D FFT and check its performance. Is the theoretically optimal triple indeed optimal?

5. $(*)$ The parallel FFT algorithm of this chapter uses two levels of memory: local memory with short access time and remote memory with longer access time. Another situation where such a two-level memory hierarchy exists is in sequential computers with a cache. The cache is a fast memory of limited size whereas the random access memory (RAM) is a much larger but slower memory.

A bulk synchronous parallel computation by a number of virtual processors can be simulated on one real, physical processor by executing the work of one computation superstep for all virtual processors in turn. If possible, the work of one virtual processor in a computation superstep is carried out within the cache. The switch between different virtual processors working on the same computation superstep causes **page faults**, that is, movements of a memory page between the cache and the RAM. A communication superstep can be viewed as a set of assignments that causes page faults.

(a) Implement a fast sequential FFT algorithm for cache-based computers by adapting the function `bspfft` to that particular situation. Replace the `bsp_put` statements by buffered assignments. What happens with the `bsp_sync`s?

(b) Compare your program with a sequential program that does not exploit the cache. Use a cache-based computer in your tests. How much do you gain?

(c) Find the optimal number of virtual processors, p_{virtual}, for several values of n. Relate your results to the cache size of the computer.

(d) Develop a parallel program based on a three-level memory hierarchy, namely cache, local RAM, and remote RAM.

(e) Test the parallel program. Use p_{virtual} virtual processors for each physical processor.

6. $(**)$ Let $\mathbf{x}$ be a real vector of length n. We could compute the Fourier transform $\mathbf{y} = F_n\mathbf{x}$, which is complex, by using a straightforward complex FFT. Still, we may hope to do better and accelerate the computation by exploiting the fact that $\mathbf{x}$ is real.

(a) Show that $y_{n-k} = \overline{y_k}$, for $k = 1, \ldots, n-1$. This implies that the output of the FFT is completely determined by $y_0, \ldots, y_{n/2}$. The remaining $n/2 - 1$ components of $\mathbf{y}$ can be obtained cheaply by complex conjugation, so that they need not be stored. Also show that y_0 and $y_{n/2}$

are real. It is customary to pack these two reals on output into one complex number, stored at position 0 in the $\mathbf{y}$-array.

(b) We can preprocess the input data by packing the real vector $\mathbf{x}$ of length n as a complex vector $\mathbf{x}'$ of length $n/2$ defined by $x'_j = x_{2j} + ix_{2j+1}$, for $0 \leq j < n/2$. The packing operation is for free in our FFT data structure, which stores a complex number as two adjacent reals. It turns out that if we perform a complex FFT of length $n/2$ on the conjugate of $\mathbf{x}'$, yielding $\mathbf{y}' = F_n\overline{\mathbf{x}'}$, we can retrieve the desired vector $\mathbf{y}$ by

$$y_k = y'_k - \tfrac{1}{2}(1 - i\omega_n^k)(y'_k - \overline{y'_{n/2-k}}), \tag{3.77}$$

for $0 \leq k \leq n/2$. Prove by substitution that the postprocessing by (3.77) is correct. The variable $y'_{n/2}$ appearing in the right-hand side for $k = 0$ and $k = n/2$ is defined by $y'_{n/2} = y'_0$, so that the DFT definition (3.4) also holds for $k = n/2$.

(c) Formulate a sequential algorithm for a real FFT based on the procedure above and count its number of flops. Take care to perform the postprocessing efficiently by computing pairs $(y_k, y_{n/2-k})$ together, using only one complex multiplication. Store constants such as $\frac{1}{2}(1 - i\omega_n^k)$ in a table.

(d) Design and implement a parallel algorithm. You may assume that p is so small that only one communication superstep is needed in the complex FFT. Start the complex FFT with the cyclic distribution of $\mathbf{x}'$ (which is equivalent to the cyclic distribution of component pairs in $\mathbf{x}$). Finish the complex FFT with the zig-zag cyclic distribution shown in Fig. 3.11(b); this distribution was proposed for use in FFT-based transforms in [112]. The **zig-zag cyclic distribution** of a vector $\mathbf{x}$

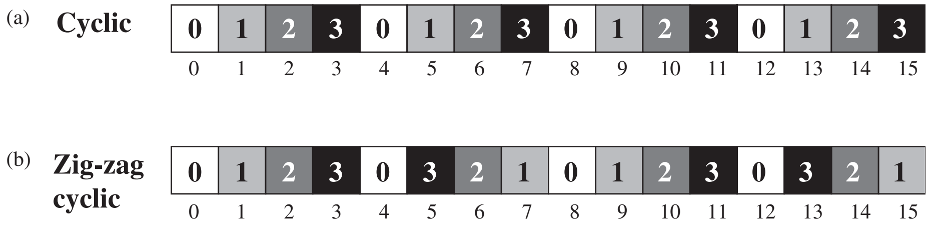


FIG. 3.11. Distribution of a vector of size 16 over four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. (a) Cyclic distribution; (b) zig-zag cyclic distribution.

of length n over p processors is defined by the mapping

$$x_j \longmapsto \begin{cases} P(j \bmod p) & \text{if } j \bmod 2p < p \\ P((-j) \bmod p) & \text{otherwise,} \end{cases} \qquad \text{for } 0 \le j < n. \tag{3.78}$$

The local index of x_j on the processor that owns it is $\mathtt{j} = j \operatorname{div} p$, just as in the cyclic case. What is the main advantage of the zig-zag cyclic distribution? Which operations are local in this distribution? Take care in designing your algorithm: a subtle point is to decide where exactly to communicate and move to the zig-zag cyclic distribution.

(e) Modify the algorithm so that it starts with the zig-zag cyclic distribution of $\mathbf{x}'$, instead of the cyclic distribution.

(f) For which p is your algorithm valid? What is the BSP cost of your algorithm?

(g) Design and implement an inverse real FFT algorithm by inverting each of the main phases of your real FFT and performing the phases in reverse order.

7. (∗∗) The **discrete cosine transform** (DCT) can be defined in several ways. One version, which is often used in image processing and data compression, is as follows. Let $\mathbf{x}$ be a real vector of length n. The DCT of $\mathbf{x}$ is the real vector $\mathbf{y}$ of length n given by

$$y_k = \sum_{j=0}^{n-1} x_j \cos \frac{\pi k(j+1/2)}{n}, \quad \text{for } 0 \le k < n. \tag{3.79}$$

(a) In our pursuit of a **fast cosine transform** (FCT), we try to pack the vector $\mathbf{x}$ in a suitable form into another vector $\mathbf{x}'$; then compute the Fourier transform $\mathbf{y}' = F_n\mathbf{x}'$; and hope to be able to massage $\mathbf{y}'$ into $\mathbf{y}$. By way of miracle, we can indeed succeed if we define

$$\begin{aligned} x'_j &= x_{2j}, \\ x'_{n-1-j} &= x_{2j+1}, \quad \text{for } 0 \le j < n/2. \end{aligned} \tag{3.80}$$

We can retrieve $\mathbf{y}$ by

$$\begin{aligned} y_k &= \mathrm{Re}(\omega_{4n}^k y'_k), \\ y_{n-k} &= -\mathrm{Im}(\omega_{4n}^k y'_k), \quad \text{for } 0 \le k \le n/2. \end{aligned} \tag{3.81}$$

(For $k = 0$, we do not compute the undefined value y_{n-k}.) Prove by substitution that this method, due to Narasimha and Peterson [144], is correct.

(b) Formulate a sequential FCT algorithm based on the procedure above and count its number of flops. Since $\mathbf{x}'$ is real, it is best to perform the Fourier transform by using the real FFT of Exercise 6.

(c) Describe how you would perform a parallel FCT. Which distributions do you use for input and output? Here, they need not be the same. Motivate your choices. Give your distributions a meaningful name. Hint: try to avoid redistribution as much as possible. You may assume that p is so small that only one communication superstep is needed in the complex FFT used by the real FFT.

(d) Analyse the cost of your parallel algorithm, implement it, and test the resulting program. How does the computing time scale with n and p?

(e) Refresh your trigonometry and formula manipulation skills by proving that the inverse DCT is given by

$$x_l = \frac{y_0}{n} + \frac{2}{n}\sum_{k=1}^{n-1} y_k \cos\frac{\pi k(l+1/2)}{n}, \quad \text{for } 0 \le l < n. \tag{3.82}$$

Hint: Use the formula $\cos\alpha\cos\beta = [\cos(\alpha+\beta)+\cos(\alpha-\beta)]/2$ to remove products of cosines. Also, derive and use a simple expression for $\sum_{k=1}^{n-1}\cos(\pi kr/n)$, for integer r. How would you compute the inverse DCT in parallel?

8. (**) Today, wavelet transforms are a popular alternative to Fourier transforms: wavelets are commonly used in areas such as image processing and compression; for instance, they form the basis for the JPEG 2000 image-compression and MPEG-4 video-compression standards. An interesting application by the FBI is in the storage and retrieval of fingerprints. Wavelets are suitable for dealing with short-lived or transient signals, and for analysing images with highly localized features. Probably the best-known wavelet is the wavelet of order four (DAUB4) proposed by Ingrid Daubechies [52]. A nice introduction to the DAUB4 wavelet transform is given in *Numerical Recipes in C* [157,section 13.10]. It is defined as follows. For $k \ge 4$, k even, let W_k denote the $k \times k$ matrix given by

$$W_k = \begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & \\
 & & & & & & \ddots & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & & c_3 & -c_2 & c_1 & -c_0 \\
c_2 & c_3 & & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2
\end{bmatrix},$$

where the coefficients are given by $c_0 = (1+\sqrt{3})/(4\sqrt{2})$, $c_1 = (3+\sqrt{3})/(4\sqrt{2})$, $c_2 = (3-\sqrt{3})/(4\sqrt{2})$, and $c_3 = (1-\sqrt{3})/(4\sqrt{2})$. For this choice of wavelet, the discrete wavelet transform (DWT) of an input vector $\mathbf{x}$ of length $n = 2^m$

can be obtained by: first multiplying $\mathbf{x}$ with W_n and then moving the even components of the result to the front, that is, multiplying the result with S_n; repeating this procedure on the first half of the current vector, using $W_{n/2}$ and $S_{n/2}$; and so on. The algorithm terminates after multiplication by W_4 and S_4. We denote the result, obtained after $m-1$ stages, by $W\mathbf{x}$.

(a) Factorize W in terms of matrices W_k, S_k, and I_k of suitable size, similar to the sequential factorization of the Fourier matrix F_n. You can use the notation $\text{diag}(A_0, \ldots, A_r)$, which stands for a block-diagonal matrix with blocks $A_0, \ldots, A_r$ on the diagonal.

(b) How many flops are needed to compute $W\mathbf{x}$? How does this scale compared with the FFT? What does this mean for communication in a parallel algorithm?

(c) Formulate a sequential algorithm that computes $W\mathbf{x}$ in place, without performing permutations. The output data will become available in scrambled order. To unscramble the data in one final permutation, where would we have to move the output value at location $28 = (0011100)_2$ for length $n = 128$? And at $j = (b_{m-1} \cdots b_0)_2$ for arbitrary length n? Usually, there is no need to unscramble.

(d) Choose a data distribution that enables the development of an efficient parallel in-place DWT algorithm. This distribution must be used on input and as long as possible during the algorithm.

(e) Formulate a parallel DWT algorithm. Hint: avoid communicating data at every stage of your algorithm. Instead, be greedy and compute what you can from several stages without communicating in between. Then communicate and finish the stages you started. Furthermore, find a sensible way of finishing the whole algorithm.

(f) Analyse the BSP cost of the parallel DWT algorithm.

(g) Compare the characteristics of your DWT algorithm to those of the parallel FFT, Algorithm 3.5. What are the essential similarities and differences?

(h) Implement and test your algorithm.

(i) Prove that the matrix W_n is orthogonal (i.e. $W_n^{\mathrm{T}} W_n = I_n$) by showing that the rows are mutually orthogonal and have unit norm. Thus W is orthogonal and $W^{-1} = W^{\mathrm{T}}$. Extend your program so that it can also compute the inverse DWT.

(j) Take a picture of a beloved person or animal, translate it into a matrix A, where each element represents a pixel, and perform a 2D DWT by carrying out 1D DWTs over the rows, followed by 1D DWTs over the columns. Choose a threshold value $\tau > 0$ and set all matrix elements a_{jk} with $|a_{jk}| \leq \tau$ to zero. What would the compression factor be if we would store A as a sparse matrix by keeping only the nonzero values a_{jk} together with their index pairs (j, k)? What does your beloved one look like after an inverse 2D DWT? You may vary τ.

9. (∗∗) The **discrete convolution** of two vectors $\mathbf{u}$ and $\mathbf{v}$ of length n is defined as the vector $\mathbf{u} * \mathbf{v}$ of length n with

$$(\mathbf{u} * \mathbf{v})_k = \sum_{j=0}^{n-1} u_j v_{k-j}, \quad \text{for } 0 \leq k < n, \tag{3.83}$$

where v_j for $j < 0$ is defined by periodic extension with period n, $v_j = v_{j+n}$.

(a) Prove the convolution theorem:

$$(F_n(\mathbf{u} * \mathbf{v}))_k = (F_n\mathbf{u})_k(F_n\mathbf{v})_k, \quad \text{for } 0 \leq k < n. \tag{3.84}$$

How can we use this to perform a fast convolution? An important application of convolutions is high-precision arithmetic, used for instance in the record computation of π in zillions of decimals. This will become clear in the following.

(b) A large nonnegative integer can be stored as a sequence of coefficients, which represents its expansion in a radix-r digit system,

$$x = (x_{m-1} \cdots x_1 x_0)_r = \sum_{k=0}^{m-1} x_k r^k, \tag{3.85}$$

where $0 \leq x_k < r$, for all k. We can view the integer x as a vector $\mathbf{x}$ of length m with integer components. Show that we can use a convolution to compute the product xy of two integers x and y. Hint: pad the vectors of length m with at least m zeros, giving the vector $\mathbf{u} = (x_0, \ldots, x_{m-1}, 0, \ldots, 0)^{\mathrm{T}}$ of length $n \geq 2m$, and a similar vector $\mathbf{v}$. Note that $\mathbf{u}*\mathbf{v}$ represents xy in the form (3.85), but its components may be larger than or equal to r. The components can be reduced to values between 0 and $r-1$ by performing carry-add operations, similar to those used in ordinary decimal arithmetic.

(c) Use a parallel FFT to multiply two large integers in parallel. Choose $r = 256$, so that an expansion coefficient can be stored in a byte, that is, a `char`. Choose a suitable distribution for the FFTs and the carry-adds, not necessarily the same distribution. On input, each coefficient x_k is stored as a complex number, and on output the result must be rounded to the nearest integer. Check the maximum rounding error incurred during the convolution, to see whether the result components are sufficiently close to the nearest integer, with a solid safety margin.

(d) Check the correctness of the result by comparing with an ordinary (sequential) large-integer multiplication program, based on the $\mathcal{O}(m^2)$ algorithm you learnt at primary school. Sequentially, which m is the break-even point between the two multiplication methods?

(e) Check that

```
1026395928297411057720541965739916759007
16567808038066803341933521790711307779
*
1066034883801684548209272203600128786792
07958575989291522270608237193062808643
=
1094173864157052742180970732204035761200
3732945449205990913842131476349984288934
7847179972578912673324976257528997818337
97076537244027146743531593354333897,
```

which has 155 decimal digits and is called RSA-155. The reverse computation, finding the two (prime) factors of RSA-155 was posed as a cryptanalytic challenge by RSA Security; its solution took a total of 35 CPU years using 300 workstations and PCs in 1999 by Cavallar *et al.* [39]. (The next challenge, factoring a 174 decimal-digit number, carries an award of \$10 000.)

(f) Develop other parallel functions for fast operations on large integers, such as addition and subtraction, using the block distribution.

(g) The **Newton–Raphson** method for finding a zero of a function f, that is, an x with $f(x) = 0$, computes successively better approximations

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}. \tag{3.86}$$

Apply this method with $f(x) = 1/x - a$ and $f(x) = 1/x^2 - a$ to compute $1/a$ and $1/\sqrt{a}$, respectively, with high precision for a given fixed real a using your parallel functions. Choose a suitable representation of a as a finite sequence of bytes. Pay special attention to termination of the Newton–Raphson iterations.

(h) At the time of writing, the world record in π computation is held by Kanada, Ushiro, Kuroda, Kudoh, and nine co-workers who obtained about 1241 100 000 000 decimal digits of π in December 2002 running a 64-processor Hitachi SR8000 supercomputer with peak speed of 2 Tflop/s for 600 h. They improved the previous record by Kanada and Takahashi who got 206 158 430 000 digits right in 1999 using all 128 processors of a Hitachi SR8000 parallel computer with peak speed of 1 Tflop/s. That run used the Gauss–Legendre method proposed by Brent [32] and Salamin [161], which works as follows. Define sequences

$a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ by $a_0 = \sqrt{2}$, $b_0 = 1$, $a_{k+1} = (a_k + b_k)/2$ is the **arithmetic mean** of the pair (a_k, b_k), and $b_{k+1} = \sqrt{a_k b_k}$ is its **geometric mean**, for $k \geq 0$. Let $c_k = 2^k(a_k^2 - b_k^2)$. Define the sequence $d_0, d_1, d_2, \ldots$ by $d_0 = 1$ and $d_{k+1} = d_k - c_{k+1}$. Then $\lim_{k\to\infty} 2a_k^2/d_k = \pi$, with fast convergence: the number of digits of π produced doubles at every iteration. Use your own parallel functions to compute as many decimal digits of π as you can. You need an efficient conversion from binary to decimal digits to produce human-readable output.

4

SPARSE MATRIX–VECTOR MULTIPLICATION

This chapter gently leads you into the world of irregular algorithms, through the example of multiplying a sparse matrix with a vector. The sparsity pattern of the matrix may be irregular, but fortunately it does not change during the multiplication, and the multiplication may be repeated many times with the same matrix. This justifies putting a lot of effort in finding a good data distribution for a parallel multiplication. We are able to analyse certain special types of matrices fully, such as random sparse matrices and Laplacian matrices, and of course we can also do this for dense matrices. For the first time, we encounter a useful non-Cartesian matrix distribution, which we call the Mondriaan distribution, and we study an algorithm for finding such a distribution for a general sparse matrix. The program of this chapter demonstrates the use of the bulk synchronous message passing primitives from BSPlib, which were designed to facilitate irregular computations; the discussion of these primitives completes the presentation of the whole BSPlib standard. After having read this chapter, you are able to design and implement parallel iterative solvers for linear systems and eigensystems, and to build higher-level solvers on top of them, such as solvers for nonlinear systems, partial differential equations, and linear programming problems.

4.1 The problem

Sparse matrices are matrices that are, well, sparsely populated by nonzero elements. The vast majority of their elements are zero. This is in contrast to **dense** matrices, which have mostly nonzero elements. The borderline between sparse and dense may be hard to draw, but it is usually clear whether a given matrix is sparse or dense.

For an $n \times n$ sparse matrix A, we denote the number of nonzeros by

$$nz(A) = |\{a_{ij} : 0 \leq i, j < n \wedge a_{ij} \neq 0\}|, \tag{4.1}$$

the average number of nonzeros per row or column by

$$c(A) = \frac{nz(A)}{n}, \tag{4.2}$$

and the **density** by

$$d(A) = \frac{nz(A)}{n^2}. \tag{4.3}$$

In this terminology, a matrix is sparse if $nz(A) \ll n^2$, or equivalently $c(A) \ll n$, or $d(A) \ll 1$. We drop the A and write nz, c, and d in cases where this does not cause confusion. An example of a small sparse matrix is given by Fig. 4.1.

Sparse matrix algorithms are much more difficult to analyse than their dense matrix counterparts. Still, we can get some grip on their time and memory requirements by analysing them in terms of n and c, while making simplifying assumptions. For instance, we may assume that c remains constant during a computation, or even that each matrix row has a fixed number of nonzeros c.

In practical applications, sparse matrices are the rule rather than the exception. A sparse matrix arises in every situation where each variable from a large set of variables is connected to only a few others. For example, in a computation scheme for the heat equation discretized on a two-dimensional grid, the temperature at a grid point may be related to the temperatures at the neighbouring grid points to the north, east, south, and west. This can be expressed by a sparse linear system involving a sparse matrix with $c = 5$.

The problem studied in this chapter is the multiplication of a sparse square matrix A and a dense vector $\mathbf{v}$, yielding a dense vector $\mathbf{u}$,

$$\mathbf{u} := A\mathbf{v}. \tag{4.4}$$

The size of A is $n \times n$ and the length of the vectors is n. The components of $\mathbf{u}$ are defined by

$$u_i = \sum_{j=0}^{n-1} a_{ij} v_j, \quad \text{for } 0 \le i < n. \tag{4.5}$$

Of course, we can exploit the sparsity of A by summing only those terms for which $a_{ij} \neq 0$.

Sparse matrix–vector multiplication is almost trivial as a sequential problem, but it is surprisingly rich as a parallel problem. Different sparsity patterns of A lead to a wide variety of communication patterns during a parallel computation. The main task then is to keep this communication within bounds.

Sparse matrix–vector multiplication is important in a range of computations, most notably in the iterative solution of linear systems and eigensystems. Iterative solution methods start with an initial guess $\mathbf{x}^0$ of the solution and then successively improve it by finding better approximations $\mathbf{x}^k$, $k = 1, 2, \ldots$, until convergence within a prescribed error tolerance. Examples of such methods are the conjugate gradient method for solving symmetric positive definite sparse linear systems $A\mathbf{x} = \mathbf{b}$ and the Lanczos method for solving symmetric sparse eigensystems $A\mathbf{x} = \lambda\mathbf{x}$; for an introduction, see [79]. The attractive property of sparse matrix–vector multiplication as the core of these solvers is that the matrix does not change, and in particular that it remains sparse throughout the computation. This is in contrast to methods such as sparse LU decomposition that create **fill-in**, that is, new nonzeros. In

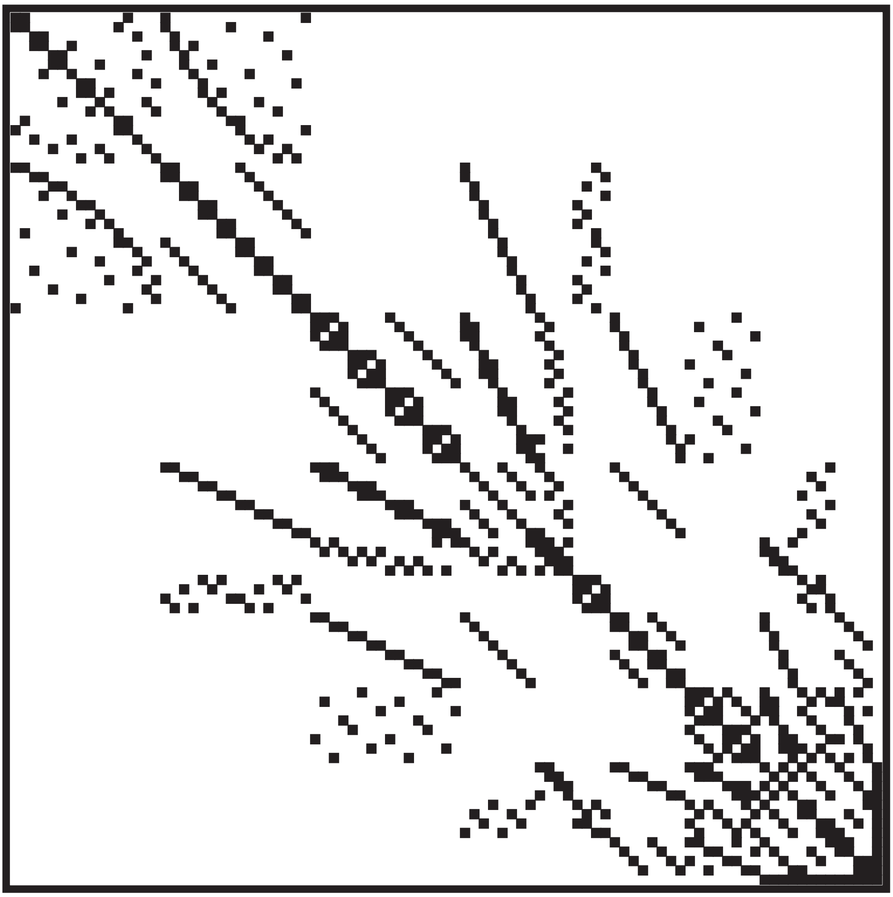


FIG. 4.1. Sparse matrix `cage6` with $n = 93$, $nz = 785$, $c = 8.4$, and $d = 9.1\%$, generated in a DNA electrophoresis study by van Heukelum, Barkema, and Bisseling [186]. Black squares denote nonzero elements; white squares denote zeros. This transition matrix represents the movement of a DNA polymer in a gel under the influence of an electric field. Matrix element a_{ij} represents the probability that a polymer in state j moves to a state i. The matrix has the name `cage6` because the model used is the cage model and the polymer modelled contains six monomers. The sparsity pattern of this matrix is symmetric (i.e. $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$), but the matrix is unsymmetric (since in general $a_{ij} \neq a_{ji}$). In this application, the eigensystem $A\mathbf{x} = \mathbf{x}$ is solved by the power method, which computes $A\mathbf{x}, A^2\mathbf{x}, A^3\mathbf{x}, \ldots$, until convergence. Solution component x_i represents the frequency of state i in the steady-state situation.

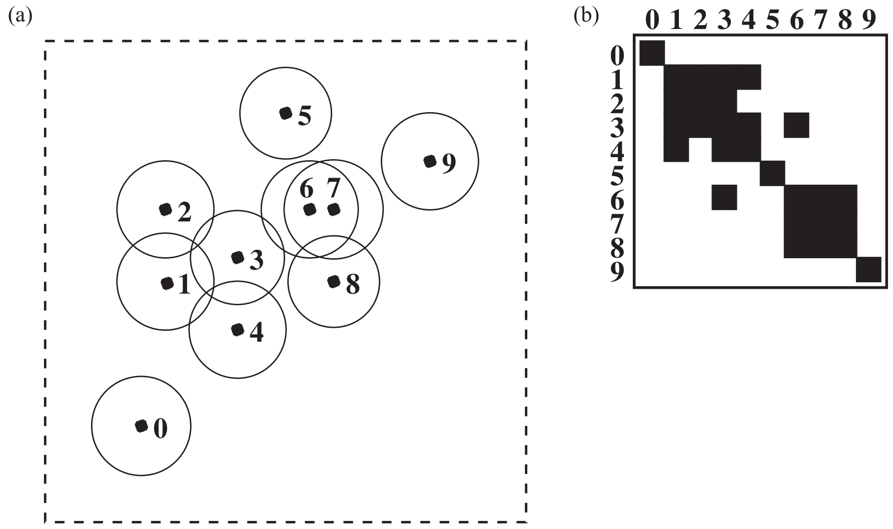


FIG. 4.2. (a) Two-dimensional molecular dynamics domain of size 1.0×1.0 with ten particles. Each circle denotes the interaction region of a particle, and is defined by a cut-off radius $r_c = 0.1$. (b) The corresponding 10×10 sparse matrix F. If the circles of particles i and j overlap in (a), these particles interact and nonzeros f_{ij} and f_{ji} appear in (b).

principle, iterative methods can solve larger systems, but convergence is not guaranteed.

The study of sparse matrix–vector multiplication also yields more insight into other areas of scientific computation. In a molecular dynamics simulation, the interaction between particles i and j can be described by a force f_{ij}. For short-range interactions, this force is zero if the particles are far apart. This implies that the force matrix F is sparse. The computation of the new positions of particles moving under two-particle forces is similar to the multiplication of a vector by F. A two-dimensional particle domain and the corresponding matrix are shown in Fig. 4.2.

Algorithm 4.1 is a sequential sparse matrix–vector multiplication algorithm. The '**for all**' statement of the algorithm must be interpreted such that all index pairs involved are handled in some arbitrary sequential order. Tests such as '$a_{ij} \neq 0$' need never be performed in an actual implementation, since only the nonzeros of A are stored in the data structure used. The formulation '$a_{ij} \neq 0$' is a simple notational device for expressing sparsity without having to specify the details of a data structure. This allows us to formulate sparse matrix algorithms that are data-structure independent. The algorithm costs $2cn$ flops.

Algorithm 4.1. Sequential sparse matrix–vector multiplication.

```
input:  A: sparse $n \times n$ matrix,
        $\mathbf{v}$ : dense vector of length $n$.
output: $\mathbf{u}$ : dense vector of length $n$, $\mathbf{u} = A\mathbf{v}$.

for $i := 0$ to $n - 1$ do
        $u_i := 0$;
for all $(i, j) : 0 \leq i, j < n \wedge a_{ij} \neq 0$ do
        $u_i := u_i + a_{ij}v_j$;
```

4.2 Sparse matrices and their data structures

The main advantage of exploiting sparsity is a reduction in memory usage (zeros are not stored) and computation time (operations with zeros are skipped or simplified). There is, however, a price to be paid, because sparse matrix algorithms are more complicated than their dense equivalents. Developing and implementing sparse algorithms costs more human time and effort. Furthermore, sparse matrix computations have a larger integer overhead associated with each floating-point operation. This implies that sparse matrix algorithms are less efficient than dense algorithms on sparse matrices with density close to one.

In this section, we discuss a few basic concepts from sparse matrix computations. For an extensive coverage of this field, see the books by Duff, Erisman, and Reid [63] and Zlatev [194]. To illustrate a fundamental sparse technique, we first study the addition of two sparse vectors. Assume that we have to add an input vector $\mathbf{y}$ of length n to an input vector $\mathbf{x}$ of length n, overwriting $\mathbf{x}$, that is, we have to perform $\mathbf{x} := \mathbf{x} + \mathbf{y}$. The vectors are sparse, which means that $x_i = 0$ for most i and $y_i = 0$ for most i. We denote the number of nonzeros of $\mathbf{x}$ by c_x and that of $\mathbf{y}$ by c_y.

To store $\mathbf{x}$ as a sparse vector, we only need $2c_x$ memory cells: for each nonzero, we store its index i and numerical value x_i as a pair (i, x_i). Since storing $\mathbf{x}$ as a dense vector requires n cells, we save memory if $c_x < n/2$. There is no need to order the pairs. In fact, ordering is sometimes disadvantageous in sparse matrix computations, for instance when the benefit from ordering the input is small but the obligation to order the output is costly. In the data structure, the vector $\mathbf{x}$ is stored as an array x of c_x nonzeros. For a nonzero x_i stored in position j, $0 \leq j < c_x$, it holds that $x[j].i = i$ and $x[j].a = x_i$.

The computation of the new component x_i requires a floating-point addition only if both $x_i \neq 0$ and $y_i \neq 0$. The case $x_i = 0$ and $y_i \neq 0$ does not require a flop because the addition reduces to an assignment $x_i := y_i$. For $y_i = 0$, nothing needs to be done.

Example 4.1 Vectors $\mathbf{x}$, $\mathbf{y}$ have length $n = 8$; their number of nonzeros is $c_x = 3$ and $c_y = 4$, respectively. Let $\mathbf{z} = \mathbf{x} + \mathbf{y}$. The sparse data structure for $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ is:

$x[j].a =$	2	5	1
$x[j].i =$	5	3	7

$y[j].a =$	1	4	1	4
$y[j].i =$	6	3	5	2

$z[j].a =$	3	9	1	1	4
$z[j].i =$	5	3	7	6	2

Now, I suggest you pause for a moment to think about how you would add two vectors $\mathbf{x}$ and $\mathbf{y}$ that are stored in the data structure described above. When you are done contemplating this, you may realize that the main problem is to find the matching pairs (i, x_i) and (i, y_i) without incurring excessive costs; this precludes for instance sorting. The magic trick is to use an auxiliary array of length n that has been initialized already. We can use this array for instance to register the location of the components of the vector $\mathbf{y}$ in its sparse data structure, so that for a given i we can directly find the location $j = loc[i]$ where y_i is stored. A value $loc[i] = -1$ denotes that y_i is not stored in the data structure, implying that $y_i = 0$. After the computation, loc must be left behind in the same state as before the computation, that is, with every array element set to -1. For each nonzero y_i, the addition method modifies the component x_i if it is already nonzero and otherwise creates a new nonzero in the data structure of $\mathbf{x}$. Algorithm 4.2 gives the details of the method.

This sparse vector addition algorithm is more complicated than the straightforward dense algorithm, but it has the advantage that the computation time is only proportional to the sum of the input lengths. The total number of operations is $\mathcal{O}(c_x + c_y)$, since there are $c_x + 2c_y$ loop iterations, each with a small constant number of operations. The number of flops equals the number of nonzeros in the intersection of the sparsity patterns of $\mathbf{x}$ and $\mathbf{y}$. The initialization of array loc costs n operations, and this cost will dominate that of the algorithm itself if only one vector addition has to be performed. Fortunately, loc can be reused in subsequent vector additions, because each modified array element is reset to -1. For example, if we add two $n \times n$ matrices row by row, we can amortize the initialization cost over n vector additions. The relative cost of initialization then becomes insignificant.

The addition algorithm does not check its output for **accidental zeros**, that is, elements that are numerically zero but still present as a nonzero pair $(i, 0)$ in the data structure. Such accidental zeros are created for instance when a nonzero $y_i = -x_i$ is added to a nonzero x_i and the resulting zero is retained in the data structure. Furthermore, accidental zeros can propagate: if y_i is an accidental zero included in the data structure of $\mathbf{y}$, and $x_i = 0$ is not in the data structure of $\mathbf{x}$, then Algorithm 4.2 will insert the accidental

Algorithm 4.2. Addition of two sparse vectors.

```
input:      $\mathbf{x}$ : sparse vector with $c_x \geq 0$ nonzeros, $\mathbf{x} = \mathbf{x}_0$,
            $\mathbf{y}$ : sparse vector with $c_y \geq 0$ nonzeros,
            $loc$ : dense vector of length $n$, $loc[i] = -1$, for $0 \leq i < n$.
output:     $\mathbf{x} = \mathbf{x}_0 + \mathbf{y}$,
            $loc[i] = -1$, for $0 \leq i < n$.

{ Register location of nonzeros of $\mathbf{y}$}
for $j := 0$ to $c_y - 1$ do
        $loc[y[j].i] := j$;

{ Add matching nonzeros of $\mathbf{x}$ and $\mathbf{y}$ }
for $j := 0$ to $c_x - 1$ do
        $i := x[j].i$;
        if $loc[i] \neq -1$ then
                $x[j].a := x[j].a + y[loc[i]].a$;
                $loc[i] := -1$;

{ Append remaining nonzeros of $\mathbf{y}$ to $\mathbf{x}$ }
for $j := 0$ to $c_y - 1$ do
        $i := y[j].i$;
        if $loc[i] \neq -1$ then
                $x[c_x].i := i$;
                $x[c_x].a := y[j].a$;
                $c_x := c_x + 1$;
                $loc[i] := -1$;
```

zero into the data structure of $\mathbf{x}$. Still, testing all operations in a sparse matrix algorithm for zero results is more expensive than computing with a few additional nonzeros, so accidental zeros are usually kept. Another reason for keeping accidental zeros is that removing them would make the output data structure dependent on the numerical values of the input and not on their sparsity pattern alone. This may cause problems for certain computations, for example, if the same program is executed repeatedly for a matrix with different numerical values but the same sparsity pattern and if knowledge obtained from the first program run is used to speed up subsequent runs. (Often, the first run of a sparse matrix program uses a dynamic data structure but subsequent runs use a simplified static data structure based on the sparsity patterns encountered in the first run.) In our terminology, we ignore accidental zeros and we just assume that they do not exist.

Sparse matrices can be stored using many different data structures; the best choice depends on the particular computation at hand. Some of the most common data structures are:

The **coordinate scheme**, or **triple scheme**. Every nonzero element a_{ij} is represented by a triple (i, j, a_{ij}), where i is the row index, j the column index, and a_{ij} the numerical value. The triples are stored in arbitrary order in an array. This data structure is easiest to understand and therefore is often used for input/output, for instance in Matrix Market [26], a WWW-based repository of sparse test matrix collections. It is also suitable for input to a parallel computer, since all information about a nonzero is contained in its triple. The triples can be sent directly and independently to the responsible processors. It is difficult, however, to perform row-wise or column-wise operations on this data structure.

Compressed row storage (CRS). Each row i of the matrix is stored as a sparse vector consisting of pairs (j, a_{ij}) representing nonzeros. In the data structure, $a[k]$ denotes the numerical value of the nonzero numbered k, and $j[k]$ its column index. Rows are stored consecutively, in order of increasing row index. The address of the first nonzero of row i is given by $start[i]$; the number of nonzeros of row i equals $start[i+1] - start[i]$, where by convention $start[n] = nz(A)$.

Example 4.2

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 8 & 9 \end{bmatrix}, \quad n = 5, \quad nz(A) = 13.$$

The CRS data structure for A is:

$a[k] =$	3	1	4	1	5	9	2	6	5	3	5	8	9
$j[k] =$	1	4	0	1	1	2	3	0	3	4	2	3	4
$k =$	0	1	2	3	4	5	6	7	8	9	10	11	12

$start[i] =$	0	2	4	7	10	13
$i =$	0	1	2	3	4	5

The CRS data structure has the advantage that the elements of a row are stored consecutively, so that row-wise operations are easy. If we compute $\mathbf{u} := A\mathbf{v}$ by components of $\mathbf{u}$, then the nonzero elements a_{ij} needed to compute u_i are conveniently grouped together, so that the value of u_i can be kept in cache on a cache-based computer, thus speeding up the computation. Algorithm 4.3 shows a sequential sparse matrix–vector multiplication that uses CRS.

Algorithm 4.3. Sequential sparse matrix–vector multiplication for the CRS data structure.

```
input:  $A$: sparse $n \times n$ matrix,
        $\mathbf{v}$ : dense vector of length $n$.
output: $\mathbf{u}$ : dense vector of length $n$, $\mathbf{u} = A\mathbf{v}$.

for $i := 0$ to $n-1$ do
        $u[i] := 0$;
        for $k := start[i]$ to $start[i+1]-1$ do
                $u[i] := u[i] + a[k] \cdot v[j[k]]$;
```

Compressed column storage (CCS). Similar to CRS, but with columns instead of rows. This is the data structure employed by the Harwell–Boeing collection [64], now called the Rutherford–Boeing collection [65], the first sparse test matrix collection that found widespread use. The motivation for building such a collection is that researchers testing different algorithms on different machines should at least be able to use a common set of test problems, in particular in the sparse matrix field where rigid analysis is rare and where heuristics reign.

Incremental compressed row storage (ICRS) [124]. A variant of CRS, where the location (i, j) of a nonzero a_{ij} is encoded as a one-dimensional index $i \cdot n + j$, and the difference with the one-dimensional index of the previous nonzero is stored (except for the first nonzero, where the one-dimensional index itself is stored). The nonzeros within a row are ordered by increasing column index, so that the one-dimensional indices form a monotonically increasing sequence. Thus, their differences are positive integers, which are called the **increments** and are stored in an array *inc*. A dummy nonzero is added at the end, representing a dummy location $(n, 0)$; this is useful in addressing the *inc* array. Note that the increments are less than $2n$ if the matrix does not have empty rows. If there are many empty rows, increments can become large (but at most n^2), so we must make sure that each increment fits in a data word.

Example 4.3 The matrix A is the same as in Example 4.2. The ICRS data structure for A is given by the arrays a and *inc* from:

$a[k] =$	3	1	4	1	5	9	2	6	5	3	5	8	9	0
$j[k] =$	1	4	0	1	1	2	3	0	3	4	2	3	4	0
$i[k] \cdot n + j[k] =$	1	4	5	6	11	12	13	15	18	19	22	23	24	25
$inc[k] =$	1	3	1	1	5	1	1	2	3	1	3	1	1	1
$k =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13

The ICRS data structure has been used in Parallel Templates [124], a parallel version of the Templates package for iterative solution of sparse linear

Algorithm 4.4. Sequential sparse matrix–vector multiplication for the ICRS data structure.

```
input:  $A$: sparse $n \times n$ matrix,
        $\mathbf{v}$ : dense vector of length $n$.
output: $\mathbf{u}$ : dense vector of length $n$, $\mathbf{u} = A\mathbf{v}$.

$j := inc[0]$;
$k := 0$;
for $i := 0$ to $n - 1$ do
        $u[i] := 0$;
        while $j < n$ do
                $u[i] := u[i] + a[k] \cdot v[j]$;
                $k := k + 1$;
                $j := j + inc[k]$;
        $j := j - n$;
```

systems [11]. ICRS does not need the *start* array and its implementation in C was found to be somewhat faster than that of CRS because the increments translate well into the pointer arithmetic of the C language. Algorithm 4.4 shows a sequential sparse matrix–vector multiplication that uses ICRS. Note that the algorithm avoids the indirect addressing of the vector $\mathbf{v}$ in the CRS data structure, replacing access to $v[j[k]]$ by access to $v[j]$.

Jagged diagonal storage (JDS). The matrix A is permuted into a matrix PA by ordering the rows by decreasing number of nonzeros. The first jagged diagonal is formed by taking the first nonzero element of every row in PA. If the matrix does not have empty rows, the length of the first jagged diagonal is n. The second jagged diagonal is formed by taking the second nonzero of every row. The length may now be less than n. This process is continued, until all c_0 jagged diagonals have been formed, where c_0 is the number of nonzeros of row 0 in PA. As in CRS, for each element the numerical value and column index are stored. The main advantage of JDS is the large average length of the jagged diagonals (of order n) that occurs if the number of nonzeros per row does not vary too much. In that case, the sparse matrix–vector multiplication can be done using efficient operations on long vectors.

Gustavson's data structure [89]. This data structure combines CRS and CCS, except that it stores the numerical values only for the rows. It provides row-wise and column-wise access to the matrix, which is useful for sparse LU decomposition.

The **two-dimensional doubly linked list**. Each nonzero is represented by a tuple, which includes i, j, a_{ij}, and links to a next and a previous nonzero in the same row and column. The elements within a row or column need not be ordered. This data structure gives maximum flexibility: row-wise and column-wise access are easy and elements can be inserted and deleted in $\mathcal{O}(1)$ operations. Therefore it is applicable in dynamic computations where the matrix changes. The two-dimensional doubly linked list was proposed as the best data structure for parallel sparse LU decomposition with pivoting [183], where frequently rows or columns have to move from one set of processors to another. (A two-dimensional singly linked list for sparse linear system solving was already presented by Knuth in 1968 in the first edition of [122].) A disadvantage of linked list data structures is the amount of storage needed, for instance seven memory cells per nonzero for the doubly linked case, which is much more than the two cells per nonzero for CRS. A severe disadvantage is that following the links causes arbitrary jumps in the computer memory, thus often incurring cache misses.

Matrix-free storage. In certain applications it may be too costly or unnecessary to store the matrix explicitly. Instead, each matrix element is recomputed every time it is needed. In certain situations this may enable the solution of huge problems that otherwise could not have been solved.

4.3 Parallel algorithm

How to distribute a sparse matrix over the processors of a parallel computer? Should we first build a data structure and then distribute it, or should we start from the distribution? The first approach constructs a global data structure and then distributes its components; a parallelizing compiler would take this road. Unfortunately, this requires global collaboration between the processors, even for basic operations such as insertion of a new nonzero. For example, if the nonzeros of a matrix row are linked in a list with each nonzero pointing to the next, then the predecessor and the successor in the list and the nonzero to be inserted may reside on three different processors. This means that three processors have to communicate to adjust their link information.

The alternative approach distributes the sparse matrix first, that is, distributes its nonzeros over the processors, assigning a subset of the nonzeros to each processor. The subsets are disjoint, and together they contain all nonzeros. (This means that the subsets form a **partitioning** of the nonzero set.) Each subset can be viewed as a smaller sparse submatrix containing exactly those rows and columns that have nonzeros in the subset and exactly those nonzeros that are part of the subset. Submatrices can then be stored using a familiar sequential sparse matrix data structure. This approach has the virtue of simplicity: it keeps basic operations such as insertion and deletion local. When a new nonzero is inserted, the distribution scheme first determines which processor is responsible, and then this processor inserts the nonzero in its local data structure without communicating. Because of this, our motto is: distribute first, represent later.

We have already encountered Cartesian matrix distributions (see Section 2.3), which are suitable for dense LU decomposition and many other matrix computations, sparse as well as dense. Here, however, we would like to stay as general as possible and remove the restriction of Cartesianity. This will give us, in principle, more possibilities to find a good distribution. We do not fear the added complexity of non-Cartesian distributions because the sparse matrix–vector multiplication is a relatively simple problem, and because the matrix does not change in this computation (unlike for instance in LU decomposition). Our general scheme maps nonzeros to processors by

$$a_{ij} \longmapsto P(\phi(i,j)), \quad \text{for } 0 \le i,j < n \quad \text{and} \quad a_{ij} \neq 0, \tag{4.6}$$

where $0 \le \phi(i,j) < p$. Zero elements of the matrix are not assigned to processors. For notational convenience, we define ϕ also for zeros: $\phi(i,j) = -1$ if $a_{ij} = 0$. Note that we use a one-dimensional processor numbering for our general matrix distribution. For the moment, we do not specify ϕ further. Obviously, it is desirable that the nonzeros of the matrix are evenly spread over the processors. Figure 4.3 shows a non-Cartesian matrix distribution. (Note that the distribution in the figure is indeed not Cartesian, since a Cartesian distribution over two processors must be either a row distribution or a column distribution, and clearly neither is the case.)

Each nonzero a_{ij} is used only once in the matrix–vector multiplication. Furthermore, usually $nz(A) \gg n$ holds, so that there are many more nonzeros than vector components. For these reasons, we perform the computation of $a_{ij} \cdot v_j$ on the processor that possesses the nonzero element. Thus, we bring the vector component to the matrix element, and not the other way round. We add products $a_{ij}v_j$ belonging to the same row i; the resulting sum is the **local contribution** to u_i, which is sent to the owner of u_i. This means that we do not have to communicate elements of A, but only components of $\mathbf{v}$ and contributions to components of $\mathbf{u}$.

How to distribute the input and output vectors of a sparse matrix–vector multiplication? In most iterative linear system solvers and eigensystem solvers, the same vector is repeatedly multiplied by a matrix A, with a few vector operations interspersed. These vector operations are mainly DAXPYs (additions of a scalar times a vector to a vector) and inner product computations. In such a situation, it is most natural to distribute all vectors in the same way, and in particular the input and output vectors of the matrix–vector multiplication, thus requiring $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$.

Another common situation is that the multiplication by A is followed by a multiplication by A^{T}. This happens for instance when a vector has to be multiplied by a matrix $B = A^{\mathrm{T}}A$, where B itself is not explicitly stored but only its factor A. The output vector of the multiplication by A is thus the input vector of the multiplication by A^{T}, so that we do not need to revert immediately to the same distribution. In this situation, we can use two different distributions, so that $\text{distr}(\mathbf{u}) \neq \text{distr}(\mathbf{v})$ is allowed.

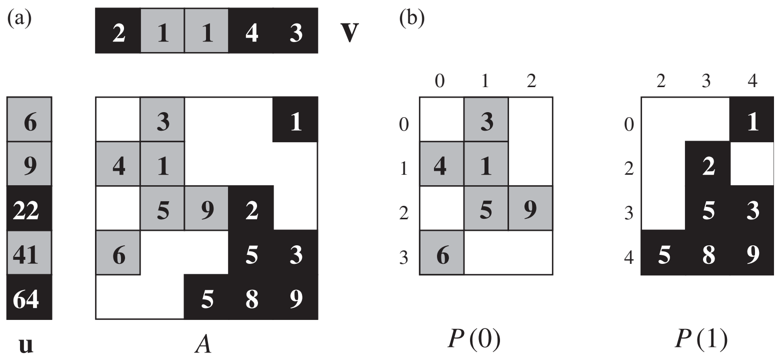


FIG. 4.3. (a) Distribution of a 5×5 sparse matrix A and vectors $\mathbf{u}$ and $\mathbf{v}$ of length five over two processors. The matrix nonzeros and vector components of processor $P(0)$ are shown as grey cells; those of $P(1)$ as black cells. The numbers in the cells denote the numerical values a_{ij}. The matrix is the same as in Example 4.2. Vector component u_i is shown to the left of the matrix row that produces it; vector component v_j is shown above the matrix column that needs it. (b) The local matrix part of the processors. Processor $P(0)$ has six nonzeros; its row index set is $I_0 = \{0, 1, 2, 3\}$ and its column index set $J_0 = \{0, 1, 2\}$. Processor $P(1)$ has seven nonzeros; $I_1 = \{0, 2, 3, 4\}$ and $J_1 = \{2, 3, 4\}$.

For a vector $\mathbf{u}$ of length n, we map components to processors by

$$u_i \longmapsto P(\phi_{\mathbf{u}}(i)), \quad \text{for } 0 \le i < n, \tag{4.7}$$

where $0 \le \phi_{\mathbf{u}}(i) < p$. To stay as general as possible, we assume that we have two mappings, $\phi_{\mathbf{u}}$ and $\phi_{\mathbf{v}}$, describing the vector distributions; these can be different, as happens in Fig. 4.3. Often, it is desirable that vector components are evenly spread over the processors, to balance the work load of vector operations in other parts of an application.

It becomes straightforward to derive a parallel algorithm, once we have chosen the distribution of the matrix and the vectors and once we have decided to compute the products $a_{ij}v_j$ on the processor that contains a_{ij}. Let us focus first on the main computation, which is a local sparse matrix–vector multiplication. Processor $P(s)$ multiplies each local nonzero element a_{ij} by v_j and adds the result into a local partial sum,

$$u_{is} = \sum_{0 \le j < n,\ \phi(i,j) = s} a_{ij} v_j, \tag{4.8}$$

for all i with $0 \le i < n$. As in the sequential case, only terms for which $a_{ij} \neq 0$ are summed. Furthermore, only the local partial sums u_{is} for which the set $\{j : 0 \le j < n \wedge \ \phi(i,j) = s\}$ is nonempty are computed. (The other partial

Algorithm 4.5. Parallel sparse matrix–vector multiplication for $P(s)$.

```
input:  $A$: sparse $n \times n$ matrix, $\mathrm{distr}(A) = \phi$,
        $\mathbf{v}$ : dense vector of length $n$, $\mathrm{distr}(\mathbf{v}) = \phi_{\mathbf{v}}$.
output: $\mathbf{u}$ : dense vector of length $n$, $\mathbf{u} = A\mathbf{v}$, $\mathrm{distr}(\mathbf{u}) = \phi_{\mathbf{u}}$.

$I_s = \{i : 0 \leq i < n \wedge (\exists j : 0 \leq j < n \wedge \phi(i,j) = s)\}$
$J_s = \{j : 0 \leq j < n \wedge (\exists i : 0 \leq i < n \wedge \phi(i,j) = s)\}$

(0)     { Fanout }
        for all $j \in J_s$ do
            get $v_j$ from $P(\phi_{\mathbf{v}}(j))$;

(1)     { Local sparse matrix–vector multiplication }
        for all $i \in I_s$ do
            $u_{is} := 0$;
            for all $j : 0 \leq j < n \wedge \phi(i,j) = s$ do
                $u_{is} := u_{is} + a_{ij}v_j$;

(2)     { Fanin }
        for all $i \in I_s$ do
            put $u_{is}$ in $P(\phi_{\mathbf{u}}(i))$;

(3)     { Summation of nonzero partial sums }
        for all $i : 0 \leq i < n \wedge \phi_{\mathbf{u}}(i) = s$ do
            $u_i := 0$;
            for all $t : 0 \leq t < p \wedge u_{it} \neq 0$ do
                $u_i := u_i + u_{it}$;
```

sums are zero.) If there are less than p nonzeros in row i, then certainly one or more processors will have an empty row part. For $c \ll p$, this will happen in many rows. To exploit this, we introduce the index set I_s of the rows that are locally nonempty in processor $P(s)$. We compute u_{is} if and only if $i \in I_s$. An example of row index sets is depicted in Fig. 4.3(b). Superstep (1) of Algorithm 4.5 is the resulting local matrix–vector multiplication.

A suitable sparse data structure must be chosen to implement superstep (1). Since we formulate our algorithm by rows, row-based sparse data structures such as CRS and ICRS are a good choice, see Section 4.2. The data structure should, however, only include nonempty local rows, to avoid unacceptable overhead for very sparse matrices. To achieve this, we can number the nonempty local rows from 0 to $|I_s| - 1$. The corresponding indices

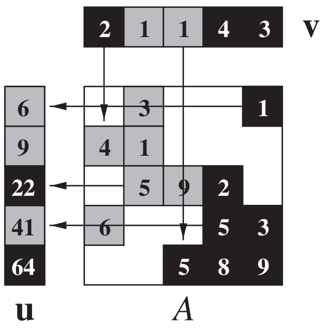


FIG. 4.4. Communication during sparse matrix–vector multiplication. The matrix is the same as in Fig. 4.3. Vertical arrows denote communication of components v_j: v_0 must be sent from its owner $P(1)$ to $P(0)$, which owns the nonzeros $a_{10} = 4$ and $a_{30} = 6$; v_2 must be sent from $P(0)$ to $P(1)$; v_1, v_3, v_4 need not be sent. Horizontal arrows denote communication of partial sums u_{is}: $P(1)$ sends its contribution $u_{01} = 3$ to $P(0)$; $P(0)$ sends $u_{20} = 14$ to $P(1)$; and $P(1)$ sends $u_{31} = 29$ to $P(0)$; u_1 and u_4 are computed locally, without contribution from the other processor. The total communication volume is $V = 5$ data words.

$\mathtt{i}$ are the **local indices**. The original **global indices** from the set I_s are stored in increasing order in an array *rowindex* of length $|I_s|$. For $0 \leq \mathtt{i} < |I_s|$, the global row index is $i = \mathit{rowindex}[\mathtt{i}]$. If for instance CRS is used, the address of the first local nonzero of row i is $\mathit{start}[\mathtt{i}]$ and the number of local nonzeros of row i is $\mathit{start}[\mathtt{i}+1] - \mathit{start}[\mathtt{i}]$.

The vector component v_j needed for the computation of $a_{ij}v_j$ must be obtained before the start of superstep (1). This is done in communication superstep (0), see the vertical arrows in Fig. 4.4. The processor that has to receive v_j knows from its local sparsity pattern that it needs this component. On the other hand, the processor that has to send the value is not aware of the needs of the receiver. This implies that the receiver should be the initiator of the communication, that is, we should use a 'get' primitive. Here, we encounter an important difference between dense and sparse algorithms. In dense algorithms, the communication patterns are predictable and thus known to every processor, so that we can formulate communication supersteps exclusively in terms of 'put' primitives. In sparse algorithms, this is often not the case, and we have to use 'get' primitives as well.

Component v_j has to be obtained only once by every processor that needs it, even if it is used repeatedly for different local nonzeros a_{ij} in the same matrix column j. If column j contains at least one local nonzero, then v_j must be obtained; otherwise, v_j is not needed. Therefore, it is convenient to define the index set J_s of the locally nonempty columns, similar to the row index set I_s. We get v_j if and only if $j \in J_s$. This gives superstep (0) of

Algorithm 4.5. We call superstep (0) the **fanout**, because vector components fan out from their initial location. The set J_s can be represented by an array *colindex* of length $|J_s|$, similar to *rowindex*. An example of column index sets is also depicted in Fig. 4.3(b). We consider the arrays *rowindex* and *colindex* to be part of the data structure for the local sparse matrix.

The partial sum u_{is} must be contributed to u_i if the local row i is nonempty, that is, if $i \in I_s$. In that case, we call u_{is} nonzero, even if accidental cancellation of terms $a_{ij}v_j$ has occurred. Each nonzero partial sum u_{is} should be sent to the processor that possesses u_i. Note that in this case the sender has the knowledge about the existence of a nonzero partial sum, so that we have to use a 'put' primitive. The resulting communication superstep is superstep (2), which we call the **fanin**, see the horizontal arrows in Fig. 4.4.

Finally, the processor responsible for u_i computes its value by adding the previously received nonzero contributions u_{it}, $0 \leq t < p$ with $t \neq s$, and the local contribution u_{is}. This is superstep (3).

'*Wat kost het?*' is an often-used Dutch phrase meaning 'How much does it cost?'. Unfortunately, the answer here is, 'It depends', because the cost of Algorithm 4.5 depends on the matrix A and the chosen distributions $\phi, \phi_{\mathbf{v}}, \phi_{\mathbf{u}}$. Assume that the matrix nonzeros are evenly spread over the processors, each processor having cn/p nonzeros. Assume that the vector components are also evenly spread over the processors, each processor having n/p components. Under these two load balancing assumptions, we can obtain an upper bound on the cost. This bound may be far too pessimistic, since distributions may exist that reduce the communication cost by a large factor.

The cost of superstep (0) is as follows. In the worst case, $P(s)$ must receive all n components v_j except for the n/p locally available components. Therefore, in the worst case $h_{\mathrm{r}} = n - n/p$; also, $h_{\mathrm{s}} = n - n/p$, because the n/p local vector components must be sent to the other $p-1$ processors. The cost is $T_{(0)} = (1 - 1/p)ng + l$. The cost of superstep (1) is $T_{(1)} = 2cn/p + l$, since two flops are needed for each local nonzero. The cost of superstep (2) is $T_{(2)} = (1-1/p)ng + l$, similar to $T_{(0)}$. The cost of superstep (3) is $T_{(3)} = n + l$, because each of the n/p local vector components is computed by adding at most p partial sums. The total cost of the algorithm is thus bounded by

$$T_{\mathrm{MV}} \leq \frac{2cn}{p} + n + 2\left(1 - \frac{1}{p}\right)ng + 4l. \tag{4.9}$$

Examining the upper bound (4.9), we see that the computation cost dominates if $2cn/p > 2ng$, that is, if $c > pg$. In that (rare) case, a distribution is already efficient if it only satisfies the two load balancing assumptions. Note that it is the number of nonzeros per row c and not the density d that directly determines the efficiency. Here, and in many other cases, we see that the parameter c is the most useful one to characterize the sparsity of a matrix.

The synchronization cost of $4l$ is usually insignificant and it does not grow with the problem size.

To achieve efficiency for smaller c, we can use a Cartesian distribution and exploit its two-dimensional nature, see Section 4.4, or we can refine the general distribution scheme using an automatic procedure to detect the underlying structure of the matrix, see Section 4.5. We can also exploit known properties of specific classes of sparse matrices, such as random sparse matrices, see Section 4.7, and Laplacian matrices, see Section 4.8.

In our model, the cost of a computation is the BSP cost. A closely related metric is the communication volume V, which is the total number of data words sent. The volume depends on ϕ, $\phi_{\mathbf{u}}$, and $\phi_{\mathbf{v}}$. For a given matrix distribution ϕ, a lower bound V_ϕ on V can be obtained by counting for each vector component u_i the number of processors p_i that has a nonzero a_{ij} in matrix row i and similarly for each vector component v_j the number of processors q_j that has a nonzero a_{ij} in matrix column j. The lower bound equals

$$V_\phi = \sum_{0\le i<n,\ p_i\ge 1} (p_i - 1) \quad + \sum_{0\le j<n,\ q_j\ge 1} (q_j - 1), \tag{4.10}$$

because every processor that has a nonzero in a matrix row i must send a value u_{is} in superstep (2), except perhaps one processor (the owner of u_i), and similarly every processor that has a nonzero in a matrix column j must receive v_j in superstep (0), except perhaps one processor (the owner of v_j). An upper bound is $V_\phi + 2n$, because in the worst case all n components u_i are owned by processors that do not have a nonzero in row i, and similar for the components v_j. Therefore,

$$V_\phi \le V \le V_\phi + 2n. \tag{4.11}$$

We can achieve $V = V_\phi$ by choosing the vector distribution after the matrix distribution, taking care that u_i is assigned to one of the processors that owns a nonzero a_{ij} in row i, and similar for v_i. We can always do this if u_i and v_i can be assigned independently; if, however, they must be assigned to the same processor, and $a_{ii} = 0$, then achieving the lower bound may be impossible. If $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$ must be satisfied, we can achieve

$$V = V_\phi + |\{i : 0 \le i < n \wedge a_{ii} = 0\}|, \tag{4.12}$$

and hence certainly $V \le V_\phi + n$. In the example of Fig. 4.4, the communication volume can be reduced from $V = 5$ to $V_\phi = 4$ if we would assign v_0 to $P(0)$ instead of $P(1)$.

4.4 Cartesian distribution

For sparse matrices, a Cartesian distribution is defined in the same way as for dense matrices, by mapping an element a_{ij} (whether it is zero or not) to a processor $P(\phi_0(i), \phi_1(j))$ with $0 \leq \phi_0(i) < M$, $0 \leq \phi_1(j) < N$, and $p = MN$. We can fit this assignment in our general scheme by identifying one-dimensional and two-dimensional processor numbers. An example is the natural column-wise identification

$$P(s,t) \equiv P(s+tM), \quad \text{for } 0 \leq s < M \quad \text{and} \quad 0 \leq t < N, \tag{4.13}$$

which can also be written as

$$P(s) \equiv P(s \bmod M, s \text{ div } M), \quad \text{for } 0 \leq s < p. \tag{4.14}$$

Thus we map nonzeros a_{ij} to processors $P(\phi(i,j))$ with

$$\phi(i,j) = \phi_0(i) + \phi_1(j)M, \quad \text{for } 0 \leq i,j < n \quad \text{and} \quad a_{ij} \neq 0. \tag{4.15}$$

Again we define $\phi(i,j) = -1$ if $a_{ij} = 0$. In this section, we examine only Cartesian distributions, and use both one-dimensional and two-dimensional processor numberings, choosing the numbering that is most convenient in the given situation.

Cartesian distributions have the same advantage for sparse matrices as for dense matrices: row-wise operations require communication only within processor rows and column-wise operations only within processor columns, and this restricts the amount of communication. For sparse matrix–vector multiplication this means that a vector component v_j has to be sent to at most M processors, and a vector component u_i is computed using contributions received from at most N processors. Another advantage is simplicity: Cartesian distributions partition the matrix orthogonally into rectangular submatrices (with rows and columns that are not necessarily consecutive in the original matrix). In general, non-Cartesian distributions create arbitrarily shaped matrix parts, see Fig. 4.3. We can view the local part of a processor $P(s)$ as a submatrix $\{a_{ij} : i \in I_s \wedge j \in J_s\}$, but in the non-Cartesian case the local submatrices may overlap. For instance, in Fig. 4.3(b), we note that column 2 has three overlapping elements. In the Cartesian case, however, the local submatrix of a processor $P(s)$ equals its Cartesian submatrix $\{a_{ij} : \phi_0(i) = s \bmod M \wedge \phi_1(j) = s \text{ div } M\}$ without the empty rows and columns. Since the Cartesian submatrices are disjoint, the local submatrices are also disjoint. Figure 4.5 shows a Cartesian distribution of the matrix `cage6`.

To reduce communication, the matrix distribution and the vector distribution should match. The vector component v_j is needed only by processors that possess an $a_{ij} \neq 0$ with $0 \leq i < n$, and these processors are contained in processor column $P(*, \phi_1(j))$. Assigning component v_j to one of the processors in that processor column reduces the upper bound on the communication,

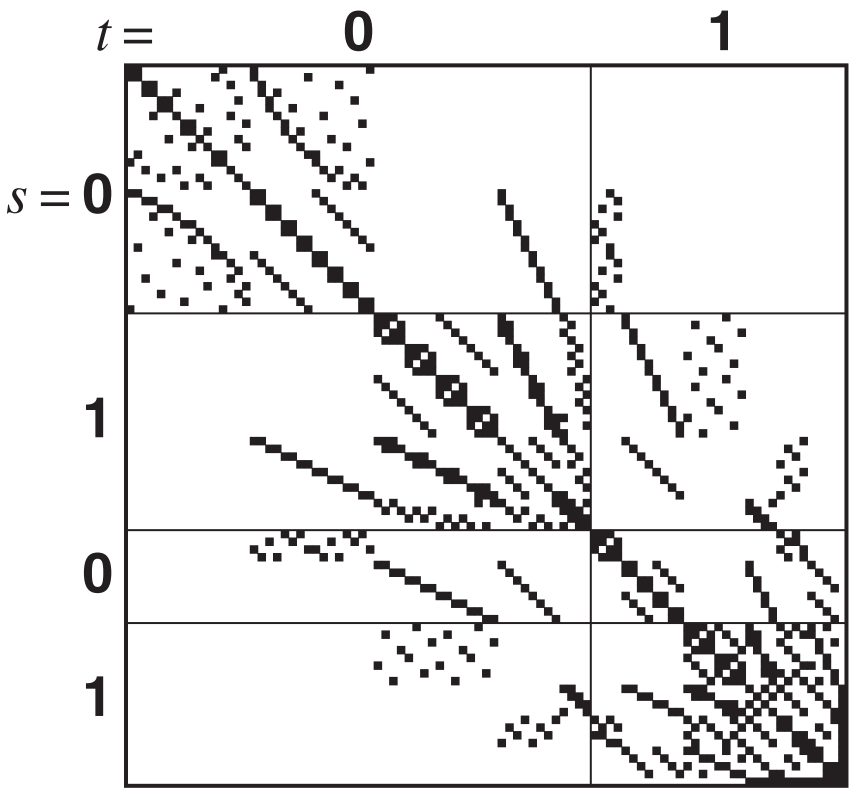


FIG. 4.5. Sparse matrix `cage6` with $n = 93$ and $nz = 785$ distributed in a Cartesian manner over four processors with $M = N = 2$; the matrix is the same as the one shown in Fig. 4.1. Black squares denote nonzero elements; white squares denote zeros. Lines denote processor boundaries. The processor row of a matrix element a_{ij} is denoted by $s = \phi_0(i)$, and the processor column by $t = \phi_1(j)$. The distribution has been determined visually, by trying to create blocks of rows and columns that fit the sparsity pattern of the matrix. Note that the matrix diagonal is assigned in blocks to the four processors, in the order $P(0) \equiv P(0,0)$, $P(1) \equiv P(1,0)$, $P(2) \equiv P(0,1)$, and $P(3) \equiv P(1,1)$. The number of nonzeros of the processors is 216, 236, 76, 257, respectively. The number of diagonal elements is 32, 28, 12, 21, respectively; these are all nonzero. Assume that the vectors $\mathbf{u}$ and $\mathbf{v}$ are distributed in the same way as the matrix diagonal. In that case, 64 (out of 93) components of $\mathbf{v}$ must be communicated in superstep (0) of the sparse matrix–vector multiplication, and 72 contributions to $\mathbf{u}$ in superstep (2). For example, components $v_0, \ldots, v_{15}$ are only needed locally. The total communication volume is $V = 136$ and the BSP cost is $24g + 2 \cdot 257 + 28g + 28 \cdot 2 + 4l = 570 + 52g + 4l$. Try to verify these numbers by clever counting!

because then v_j has to be sent to at most $M-1$ processors, instead of M. This decrease in upper bound may hardly seem worthwhile, especially for large M, but the lower bound also decreases, from one to zero, and this is crucial. If v_j were assigned to a different processor column, it would always have to be communicated (assuming that matrix column j is nonempty). If component v_j is assigned to processor column $P(*, \phi_1(j))$, then there is a chance that it is needed only by its own processor, so that no communication is required. A judicious choice of distribution may enhance this effect. This is the main reason why for Cartesian matrix distributions we impose the constraint that v_j resides in $P(*, \phi_1(j))$. If we are free to choose the distribution of $\mathbf{v}$, we assign v_j to one of the owners of nonzeros in matrix column j, thereby satisfying the constraint.

We impose a similar constraint on the output vector $\mathbf{u}$. To compute u_i, we need contributions from the processors that compute the products $a_{ij}v_j$, for $0 \leq j < n$ and $a_{ij} \neq 0$. These processors are all contained in processor row $P(\phi_0(i), *)$. Therefore, we assign u_i to a processor in that processor row. The number of contributions sent for u_i is thus at most $N-1$. If we are free to choose the distribution of $\mathbf{u}$, we assign u_i to one of the owners of nonzeros in matrix row i.

If the requirement $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$ must be satisfied, our constraints on $\mathbf{u}$ and $\mathbf{v}$ imply that u_i and v_i must be assigned to $P(\phi_0(i), \phi_1(i))$, which is the processor that owns the diagonal element a_{ii} in the Cartesian distribution of A. In that case,

$$\phi_{\mathbf{u}}(i) = \phi_{\mathbf{v}}(i) = \phi_0(i) + \phi_1(i)M, \quad \text{for } 0 \leq i < n. \tag{4.16}$$

As a result, for a fixed M and N, the choice of a Cartesian matrix distribution determines the vector distribution. The reverse is also true: an arbitrary matrix element a_{ij} is assigned to the processor row that possesses the vector component u_i and to the processor column that possesses u_j.

The following trivial but powerful theorem states that the amount of communication can be restricted based on a suitable distribution of the vectors.

Theorem 4.4 *Let A be a sparse $n \times n$ matrix and $\mathbf{u}, \mathbf{v}$ vectors of length n. Assume that: (i) the distribution of A is Cartesian,* $\mathrm{distr}(A) = (\phi_0, \phi_1)$*; (ii) the distribution of $\mathbf{u}$ is such that u_i resides in $P(\phi_0(i), *)$, for all i; (iii) the distribution of $\mathbf{v}$ is such that v_j resides in $P(*, \phi_1(j))$, for all j. Then: if u_i and v_j are assigned to the same processor, the matrix element a_{ij} is also assigned to that processor and does not cause communication.*

Proof Component u_i is assigned to a processor $P(\phi_0(i), t)$ and component v_j to a processor $P(s, \phi_1(j))$. Since this is the same processor, it follows that $(s, t) = (\phi_0(i), \phi_1(j))$, so that this processor also owns matrix element a_{ij}. $\square$

Example 4.5 Let A be the $n \times n$ tridiagonal matrix defined by

$$A = \begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & & \ddots & & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -2 \end{bmatrix}.$$

This matrix represents a Laplacian operator on a one-dimensional grid of n points. (Section 4.8 treats Laplacian operators on multidimensional grids.) The nonzeros are the elements on the main diagonal and on the diagonals immediately above and below it, so that $a_{ij} \neq 0$ if and only if $i - j = 0, \pm 1$. Assume that we have to find a suitable Cartesian matrix distribution (ϕ_0, ϕ_1) and a single distribution for the input and output vectors. Theorem 4.4 says that here it is best to assign u_i and u_j to the same processor if $i = j \pm 1$. Therefore, a suitable vector distribution over p processors is the block distribution,

$$u_i \longmapsto P\left(i \text{ div } \left\lceil \frac{n}{p} \right\rceil\right), \quad \text{for } 0 \leq i < n. \tag{4.17}$$

Communication takes place only on the boundary of a block. Each processor has to send and receive at most two components v_j, and to send and receive at most two contributions to a component u_i. The vector distribution completely determines the matrix distribution, except for the choice of M and N.

For $n = 12$ and $M = N = 2$, the matrix distribution corresponding to the block distribution of the vectors is

$$\text{distr}(A) = \begin{bmatrix} 0 & 0 & & & & & & & & & & \\ 0 & 0 & 0 & & & & & & & & & \\ & 0 & 0 & 0 & & & & & & & & \\ & & 1 & 1 & 1 & & & & & & & \\ & & & 1 & 1 & 1 & & & & & & \\ & & & & 1 & 1 & \mathbf{3} & & & & & \\ & & & & & \mathbf{0} & 2 & 2 & & & & \\ & & & & & & 2 & 2 & 2 & & & \\ & & & & & & & 2 & 2 & 2 & & \\ & & & & & & & & 3 & 3 & 3 & \\ & & & & & & & & & 3 & 3 & 3 \\ & & & & & & & & & & 3 & 3 \end{bmatrix}.$$

Position (i, j) of distr(A) gives the one-dimensional identity of the processor that possesses matrix element a_{ij}. The matrix distribution is obtained by first distributing the matrix diagonal in the same way as the vectors, and then translating the corresponding one-dimensional processor numbers into two-dimensional numbers by $P(0) \equiv P(0,0)$, $P(1) \equiv P(1,0)$, $P(2) \equiv P(0,1)$, and $P(3) \equiv P(1,1)$. After that, the off-diagonal nonzeros are assigned to processors, for instance a_{56} is in the same processor row as a_{55}, which is owned by $P(1) = P(1,0)$, and it is in the same processor column as a_{66}, which is owned by $P(2) = P(0,1)$. Thus a_{56} is owned by $P(1,1) = P(3)$. Note that this distribution differs only slightly from the row distribution defined by $M = 4$, $N = 1$; the only elements distributed differently are a_{56} and a_{65} (marked in boldface), which in the row distribution are assigned to $P(1)$ and $P(2)$, respectively.

The cost of Algorithm 4.5 for a Cartesian distribution depends on the matrix A and the chosen distribution. Because of our additional assumptions, we can improve the upper bound (4.9). As before, we assume a good spread of the matrix elements and vector components over the processors, but now we also assume a good spread of the matrix rows over the processor rows and the matrix columns over the processor columns. In superstep (0), $P(s,t)$ must receive at most all components v_j with $\phi_1(j) = t$, except for the n/p locally available components. Therefore, in the worst case $h_{\mathrm{r}} = n/N - n/p = (M-1)n/p = h_{\mathrm{s}}$. The cost is at most $T_{(0)} = (M-1)ng/p + l$. As before, $T_{(1)} = 2cn/p + l$. The cost of superstep (2) is at most $T_{(2)} = (N-1)ng/p + l$, similar to $T_{(0)}$. The cost of superstep (3) is at most $T_{(3)} = Nn/p + l = n/M + l$, because each of the n/p local vector components is computed by adding at most N partial sums. The total cost of the algorithm is thus bounded by

$$T_{\mathrm{MV},\ M\times N} \leq \frac{2cn}{p} + \frac{n}{M} + \frac{M+N-2}{p} ng + 4l. \tag{4.18}$$

The communication term of the upper bound (4.18) is minimal for $M = N = \sqrt{p}$. For that choice, the bound reduces to

$$T_{\mathrm{MV},\ \sqrt{p}\times\sqrt{p}} \leq \frac{2cn}{p} + \frac{n}{\sqrt{p}} + 2\left(\frac{1}{\sqrt{p}} - \frac{1}{p}\right) ng + 4l. \tag{4.19}$$

Examining the upper bound (4.19), we see that the computation cost dominates if $2cn/p > 2ng/\sqrt{p}$, that is, if $c > \sqrt{p}g$. This is an improvement of the critical c-value by a factor $\sqrt{p}$ compared with the value for the general upper bound (4.9).

Dense matrices can be considered as the extreme limit of sparse matrices. Analysing the dense case is easier and it can give us insight into the sparse case as well. Let us therefore examine an $n \times n$ dense matrix A. Assume we have to use the same distribution for the input and output vectors, which therefore must be the distribution of the matrix diagonal. Assume for simplicity that n is a multiple of p and p is a square.

In our study of dense LU decomposition, see Chapter 2, we have extolled the virtues of the square cyclic distribution for parallel linear algebra. One may ask whether this is also a good distribution for dense matrix–vector multiplication by Algorithm 4.5. Unfortunately, the answer is negative. The reason is that element a_{ii} from the matrix diagonal is assigned to processor $P(i \bmod \sqrt{p}, i \bmod \sqrt{p})$, so that the matrix diagonal is assigned to the **diagonal processors**, that is, the processors $P(s,s)$, $0 \leq s < \sqrt{p}$. This implies that only $\sqrt{p}$ out of p processors have part of the matrix diagonal and hence of the vectors, so that the load balancing assumption for vector components is not satisfied. Diagonal processors have to send out $\sqrt{p}-1$ copies of $n/\sqrt{p}$ vector components, so that $h_{\rm s} = n - n/\sqrt{p}$ in superstep (0) and h is $\sqrt{p}$ times larger than the h of a well-balanced distribution. The total cost for a dense matrix with the square cyclic distribution becomes

$$T_{\text{MV, dense, } \sqrt{p}\times\sqrt{p} \text{ cyclic}} = \frac{2n^2}{p} + n + 2\left(1 - \frac{1}{\sqrt{p}}\right) ng + 4l. \tag{4.20}$$

The communication cost for this unbalanced distribution is a factor $\sqrt{p}$ higher than the upper bound (4.19) with $c = n$ for balanced distributions. The total communication volume is $V_\phi = 2(\sqrt{p}-1)n$. For dense matrices with the square cyclic distribution, the communication imbalance can be reduced by changing the algorithm and using two-phase broadcasting for the fanout and a similar technique, two-phase combining, for the fanin. This, however, does not solve the problem of vector imbalance in other parts of the application, and of course it would be better to use a good distribution in the first place, instead of redistributing the data during the fanout or fanin.

The communication balance can be improved by choosing a distribution that spreads the vectors and hence the matrix diagonal evenly, for example choosing the one-dimensional distribution $\phi_{\mathbf{u}}(i) = \phi_{\mathbf{v}}(i) = i \bmod p$ and using the 1D–2D identification (4.14). We still have the freedom to choose M and N, where $MN = p$. For the choice $M = p$ and $N = 1$, this gives $\phi_0(i) = i \bmod p$ and $\phi_1(j) = 0$, which is the same as the cyclic row distribution. It is easy to see that now the cost is

$$T_{\text{MV, dense, } p\times 1 \text{ cyclic}} = \frac{2n^2}{p} + \left(1 - \frac{1}{p}\right) ng + 2l. \tag{4.21}$$

This distribution disposes of the fanin and the summation of partial sums, since each matrix row is completely contained in one processor. Therefore, the last two supersteps are empty and can be deleted. Still, this is a bad distribution, since the gain by fewer synchronizations is lost by the much more expensive fanout: each processor has to send n/p vector components to all other processors. The communication volume is large: $V_\phi = (p-1)n$.

For the choice $M = N = \sqrt{p}$, we obtain $\phi_0(i) = (i \bmod p) \bmod \sqrt{p} = i \bmod \sqrt{p}$ and $\phi_1(j) = (j \bmod p) \operatorname{div} \sqrt{p}$. The cost of the fanout and fanin

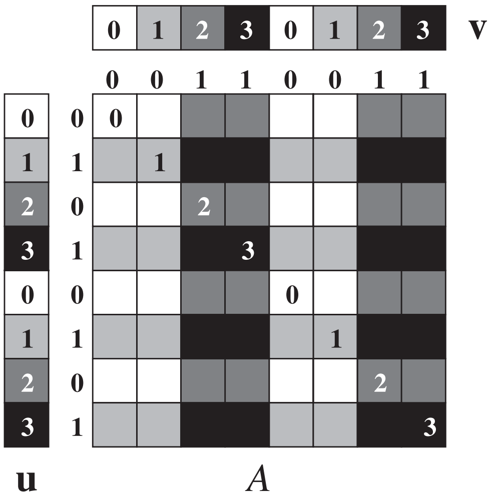


FIG. 4.6. Dense 8×8 matrix distributed over four processors using a square Cartesian distribution based on a cyclic distribution of the matrix diagonal. The vectors are distributed in the same way as the matrix diagonal. The processors are shown by greyshades; the one-dimensional processor numbering is shown as numbers in the cells of the matrix diagonal and the vectors. The two-dimensional numbering is represented by the numbering of the processor rows and columns shown along the borders of the matrix. The 1D–2D correspondence is $P(0) \equiv P(0,0)$, $P(1) \equiv P(1,0)$, $P(2) \equiv P(0,1)$, and $P(3) \equiv P(1,1)$.

for this distribution are given by $h_{\rm s} = h_{\rm r} = (\sqrt{p} - 1)n/p$, so that

$$T_{\rm MV,\ dense} = \frac{2n^2}{p} + \frac{n}{\sqrt{p}} + 2\left(\frac{1}{\sqrt{p}} - \frac{1}{p}\right) ng + 4l, \qquad (4.22)$$

which equals the upper bound (4.19) for $c = n$. We see that this distribution is much better than the square cyclic distribution and the cyclic row distribution. The communication volume is $V_\phi = 2(\sqrt{p}-1)n$. Figure 4.6 illustrates this data distribution.

We may conclude that the case of dense matrices is a good example where Cartesian matrix partitioning is useful in deriving an optimal distribution, a square Cartesian distribution based on a cyclic distribution of the matrix diagonal. (Strictly speaking, we did not give an optimality proof; it seems, however, that this distribution is hard to beat.)

4.5 Mondriaan distribution for general sparse matrices

Sparse matrices usually have a special structure in their sparsity pattern. Sometimes this structure is known, but more often it has to be detected by an automatic procedure. The aim of this section is to present an algorithm that finds the underlying structure of an arbitrary sparse matrix and exploits this structure to generate a good distribution for sparse matrix–vector multiplication. We name the resulting distribution **Mondriaan distribution**, honouring the Dutch painter Piet Mondriaan (1872–1944) who is most famous for his compositions of brightly coloured rectangles.

Suppose we have to find a matrix and vector distribution for a sparse matrix A where we have the freedom to distribute the input and output vectors independently. Thus, we can first concentrate on the matrix distribution problem and try to find a matrix distribution ϕ that minimizes the communication volume V_ϕ, defined by eqn (4.10), while balancing the computational work. We define $A_s = \{(i,j) : 0 \leq i,j < n \wedge \phi(i,j) = s\}$ as the set of index pairs corresponding to the nonzeros of $P(s)$, $0 \leq s < p$. Thus, $A_0, \ldots, A_{p-1}$ forms a ***p*-way partitioning** of $A = \{(i,j) : 0 \leq i,j < n \wedge a_{ij} \neq 0\}$. (For the purpose of partitioning, we identify a nonzero with its index pair and a sparse matrix with its set of index pairs.) We use the notation $V(A_0, \ldots, A_{p-1}) = V_\phi$ to express the explicit dependence of the communication volume on the partitioning.

The figure on the cover of this book (see also Plate 1) shows a 4-way partitioning of the 60×60 matrix `prime60`, defined by $a_{ij} = 1$ if $i \bmod j = 0$ or $j \bmod i = 0$, and $a_{ij} = 0$ otherwise, for $i,j = 1, \ldots, n$. It is easy to recognize rows and columns with an index that is prime. (To establish this direct connection, we must start counting the indices of `prime60` at one, as an exception to the rule of always starting at zero.)

For mutually disjoint subsets of nonzeros $A_0, \ldots, A_{k-1}$, where $k \geq 1$, not necessarily with $A_0 \cup \cdots \cup A_{k-1} = A$, we define the communication volume $V(A_0, \ldots, A_{k-1})$ as the volume of the sparse matrix–vector multiplication for the matrix $A_0 \cup \cdots \cup A_{k-1}$ where subset A_s is assigned to $P(s)$, $0 \leq s < k$. For $k = p$, this reduces to the original definition. When we split a subset of a k-way partitioning of A, we obtain a $(k+1)$-way partitioning. The following theorem says that the new communication volume equals the old volume plus the volume incurred by splitting the subset as a separate problem, ignoring all other subsets.

Theorem 4.6 *(Vastenhouw and Bisseling [188]) Let A be a sparse $n \times n$ matrix and let $A_0, \ldots, A_k \subset A$ be mutually disjoint subsets of nonzeros, where $k \geq 1$. Then*

$$V(A_0, \ldots, A_k) = V(A_0, \ldots, A_{k-2}, A_{k-1} \cup A_k) + V(A_{k-1}, A_k). \qquad (4.23)$$

Proof The number of processors that contributes to a vector component u_i depends on the partitioning assumed, $p_i = p_i(A_0, \ldots, A_{k-1})$ for the k-way partitioning $A_0, \ldots, A_{k-1}$, and similarly $q_j = q_j(A_0, \ldots, A_{k-1})$.

Let $p_i' = \max(p_i - 1, 0)$ and $q_j' = \max(q_j - 1, 0)$. We are done if we prove

$$p_i'(A_0, \ldots, A_k) = p_i'(A_0, \ldots, A_{k-2}, A_{k-1} \cup A_k) + p_i'(A_{k-1}, A_k), \qquad (4.24)$$

for $0 \le i < n$, and a similar equality for q_j', because the result then follows from summing over $i = 0, \ldots, n-1$ for p_i' and $j = 0, \ldots, n-1$ for q_j'. We only prove the equality for the p_i', and do this by distinguishing two cases. If row i has a nonzero in A_{k-1} or A_k, then $p_i' = p_i - 1$ in all three terms of eqn (4.24). Thus,

$$\begin{aligned}
&p_i'(A_0, \ldots, A_{k-2}, A_{k-1} \cup A_k) + p_i'(A_{k-1}, A_k) \\
&\quad = p_i(A_0, \ldots, A_{k-2}, A_{k-1} \cup A_k) - 1 + p_i(A_{k-1}, A_k) - 1 \\
&\quad = p_i(A_0, \ldots, A_{k-2}) + 1 - 1 + p_i(A_{k-1}, A_k) - 1 \\
&\quad = p_i(A_0, \ldots, A_{k-2}) + p_i(A_{k-1}, A_k) - 1 \\
&\quad = p_i(A_0, \ldots, A_k) - 1 \\
&\quad = p_i'(A_0, \ldots, A_k). \qquad (4.25)
\end{aligned}$$

If row i has no nonzero in A_{k-1} or A_k, then

$$\begin{aligned}
&p_i'(A_0, \ldots, A_{k-2}, A_{k-1} \cup A_k) + p_i'(A_{k-1}, A_k) \\
&\quad = p_i'(A_0, \ldots, A_{k-2}) + 0 \\
&\quad = p_i'(A_0, \ldots, A_k). \qquad (4.26)
\end{aligned}$$

□

The proof of Theorem 4.6 shows that the theorem also holds for the communication volume of the fanout and fanin separately. The theorem implies that we only have to look at the subset we want to split when trying to optimize the split, and not at the effect such a split has on communication for other subsets.

The theorem helps us to achieve our goal of minimizing the communication volume. Of course, we must at the same time also consider the load balance of the computation; otherwise, the problem is easily solved: assign all nonzeros to the same processor *et voilà*, no communication whatsoever! We specify the allowed load imbalance by a parameter $\epsilon > 0$, requiring that the p-way partitioning of the nonzeros satisfies the computational balance constraint

$$\max_{0 \le s < p} nz(A_s) \le (1 + \epsilon) \frac{nz(A)}{p}. \qquad (4.27)$$

We do not allow $\epsilon = 0$, because obtaining a perfect load balance is often impossible, and when it is possible, the demand for perfect balance does not leave much room for minimizing communication. Besides, nobody is perfect and epsilons are never zero.

The load imbalance achieved for the matrix `prime60` shown on the cover of this book is $\epsilon' \approx 2.2\%$: the matrix has 462 nonzeros, and the number of nonzeros per processor is 115, 118, 115, 114, for $P(0)$ (red), $P(1)$ (black), $P(2)$ (yellow), and $P(3)$ (blue), respectively. (We denote the achieved imbalance by ϵ', to distinguish it from the allowed imbalance ϵ.) The partitioning has been obtained by a vertical split into blocks of consecutive columns, followed by two independent horizontal splits into blocks of consecutive rows. The splits were optimized for load balance only. (Requiring splits to yield consecutive blocks is an unnecessary and harmful restriction, but it leads to nice and easily comprehensible pictures.) The communication volume for this partitioning is 120, which is bad because it is the highest value possible for the given split directions.

The best choice of the imbalance parameter ϵ is machine-dependent and can be found by using the BSP model. Suppose we have obtained a matrix distribution with volume V that satisfies the constraint (4.27). Assuming that the subsequent vector partitioning does a good job, balancing the communication well and thus achieving a communication cost of Vg/p, we have a BSP cost of $2(1+\epsilon')nz(A)/p + Vg/p + 4l$. To get a good trade-off between computation imbalance and communication, the corresponding overhead terms should be about equal, that is, $\epsilon' \approx Vg/(2nz(A))$. If this is not the case, we can increase or decrease ϵ and obtain a lower BSP cost. We cannot determine ϵ beforehand, because we cannot predict exactly how its choice affects V.

How to split a given subset? Without loss of generality, we may assume that the subset is A itself. In principle, we can assign every individual nonzero to one of the two available processors. The number of possible 2-way partitionings, however, is huge, namely $2^{nz(A)-1}$. (We saved a factor of two by using symmetry: we can swap the two processors without changing the volume of the partitioning.) Trying all partitionings and choosing the best is usually impossible, even for modest problem sizes. In the small example of Fig. 4.3, we already have $2^{12} = 4096$ possibilities (one of which is shown). Thus, our only hope is to develop a **heuristic** method, that is, a method that gives an approximate solution, hopefully close to the optimum and computed within reasonable time. A good start is to try to restrict the search space, for example, by assigning complete columns to processors; the number of possibilities then decreases to 2^{n-1}. In the example, we now have $2^4 = 16$ possibilities. In general, the number of possibilities is still large, and heuristics are still needed, but the problem is now more manageable. One reason is that bookkeeping is simpler for n columns than for $nz(A)$ nonzeros. Furthermore, a major advantage of assigning complete columns is that the split does not generate communication in the fanout. Thus we decide to perform each split by complete columns, or, alternatively, by complete rows. We can express the splitting by the assignment

$$(A_0, A_1) := \mathit{split}(A, \mathit{dir}, \epsilon), \tag{4.28}$$

where *dir* $\in$ {row, col} is the splitting direction and ϵ the allowed load imbalance. Since we do not know ahead, which of the two splitting directions is best, we try both and choose the direction with the lowest communication volume.

Example 4.7 Let

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 8 & 9 \end{bmatrix}.$$

For $\epsilon = 0.1$, the maximum number of nonzeros per processor must be seven and the minimum six. For a column split, this implies that one processor must have two columns with three nonzeros and the other processor the remaining columns. A solution that minimizes the communication volume V is to assign columns 0, 1, 2 to $P(0)$ and columns 3, 4 to $P(1)$. This gives $V = 4$. For a row split, assigning rows 0, 1, 3 to $P(0)$ and rows 2, 4 to $P(1)$ is optimal, giving $V = 3$. Can you find a better solution if you are allowed to assign nonzeros individually to processors?

The function *split* can be applied repeatedly, giving a method for partitioning a matrix into several parts. The method can be formulated concisely as a recursive computation. For convenience, we assume that $p = 2^q$, but the method can be adapted to handle other values of p as well. (This would also require generalizing the splitting function, so that for instance in the first split for $p = 3$, it can produce two subsets with a nonzero ratio of about $2:1$.) The recursive method should work for a rectangular input matrix, since the submatrices involved may be rectangular (even though the initial matrix is square). Because of the splitting into sets of complete columns or rows, we can view the resulting p-way partitioning as a splitting into p mutually disjoint submatrices (not necessarily with consecutive rows and columns): we start with a complete matrix, split it into two submatrices, split each submatrix, giving four submatrices, and so on. The number of times the original submatrix must be split to reach a given submatrix is called the **recursion level** of the submatrix. The level of the original matrix is 0. The final result for processor $P(s)$ is a submatrix defined by an index set $\bar{I}_s \times \bar{J}_s$. This index set is different from the index set $I_s \times J_s$ of pairs (i, j) with $i \in I_s$ and $j \in J_s$ defined in Algorithm 4.5, because the submatrices $\bar{I}_s \times \bar{J}_s$ may contain empty rows and columns; removing these gives $I_s \times J_s$. Thus we have

$$A_s \subset I_s \times J_s \subset \bar{I}_s \times \bar{J}_s, \quad \text{for } 0 \le s < p. \tag{4.29}$$

Furthermore, all the resulting submatrices are mutually disjoint, that is,

$$(\bar{I}_s \times \bar{J}_s) \cap (\bar{I}_t \times \bar{J}_t) = \emptyset, \quad \text{for } 0 \le s, t < p \text{ with } s \neq t, \tag{4.30}$$

and together they comprise the original matrix,

$$\bigcup_{s=0}^{p-1} (\bar{I}_s \times \bar{J}_s) = \{0,\ldots,n-1\} \times \{0,\ldots,n-1\}. \tag{4.31}$$

To achieve a final load imbalance of at most ϵ, we must take care that the maximum number of nonzeros per matrix part grows slowly enough with the recursion level. If the growth factor at each level is $1+\delta$, then the overall growth factor is $(1+\delta)^q \approx 1+q\delta$ in a first-order approximation. This motivates our choice of starting with $\delta = \epsilon/q$. After the first split, a new situation arises. One part has at least half the nonzeros, and the other part at most half. Assume that the matrix parts, or subsets, are B_0 and B_1. Subset B_s, $s = 0, 1$, has $nz(B_s)$ nonzeros and will be partitioned over $p/2$ processors with a load imbalance parameter ϵ_s. Equating the maximum number of nonzeros per processor specified for the remainder of the partitioning process to the maximum specified at the beginning,

$$(1+\epsilon_s)\frac{nz(B_s)}{p/2} = (1+\epsilon)\frac{nz(A)}{p}, \tag{4.32}$$

gives the value ϵ_s to be used in the remainder. In this way, the allowed load imbalance is dynamically adjusted during the partitioning. A matrix part that has fewer nonzeros than the average will have a larger ϵ in the remainder, giving more freedom to minimize communication for that part. The resulting algorithm is given as Algorithm 4.6.

Figure 4.7 presents a **global view** of the sparse matrix `prime60` and the corresponding input and output vectors distributed over four processors by the Mondriaan package [188], version 1.0. The matrix distribution program of this package is an implementation of Algorithm 4.6. The allowed load imbalance specified by the user is $\epsilon = 3\%$; the imbalance achieved by the program is $\epsilon' \approx 1.3\%$, since the maximum number of nonzeros per processor is 117 and the average is 462/4=115.5. The communication volume of the fanout is 51 and that of the fanin is 47, so that $V = 98$. Note that rows $i = 11$, 17, 19, 23, 25, 31, 37, 41, 43, 47, 53, 55, 59 (in the exceptional numbering starting from one) are completely owned by one processor and hence do not cause communication in the fanin; vector component u_i is owned by the same processor. Not surprisingly, all except two of these row numbers are prime. The distribution is better than the distribution on the book cover, both with respect to computation and with respect to communication. The matrix `prime60` is symmetric, and although the Mondriaan package has an option to produce symmetric partitionings, we did not use it for our example.

Figure 4.8 presents a **local view**, or **processor view**, of `prime60` and the corresponding input and output vectors. For processor $P(s)$, $s = 0, 1, 2, 3$, the local submatrix $\bar{I}_s \times \bar{J}_s$ is shown. The size of this submatrix is 29×26 for

Algorithm 4.6. Recursive matrix partitioning.

```
input:          $A$: $m \times n$ sparse matrix,
                $p$: number of processors, $p = 2^q$ with $q \geq 0$,
                $\epsilon$ = allowed load imbalance, $\epsilon > 0$.
output:         $(A_0, \ldots, A_{p-1})$: $p$-way partitioning of $A$,
                satisfying $\max_{0 \leq s < p} nz(A_s) \leq (1 + \epsilon)\frac{nz(A)}{p}$.
function call:  $(A_0, \ldots, A_{p-1}) :=$ MatrixPartition$(A, p, \epsilon)$.

if $p > 1$ then
        $maxnz := (1 + \epsilon)\frac{nz(A)}{p}$;
        $(B_0^{\rm row}, B_1^{\rm row}) := split(A, {\rm row}, \frac{\epsilon}{q})$;
        $(B_0^{\rm col}, B_1^{\rm col}) := split(A, {\rm col}, \frac{\epsilon}{q})$;
        if $V(B_0^{\rm row}, B_1^{\rm row}) \leq V(B_0^{\rm col}, B_1^{\rm col})$ then
                $(B_0, B_1) := (B_0^{\rm row}, B_1^{\rm row})$;
        else    $(B_0, B_1) := (B_0^{\rm col}, B_1^{\rm col})$;
        $\epsilon_0 := \frac{maxnz}{nz(B_0)} \cdot \frac{p}{2} - 1$;
        $\epsilon_1 := \frac{maxnz}{nz(B_1)} \cdot \frac{p}{2} - 1$;
        $(A_0, \ldots, A_{p/2-1}) :=$ MatrixPartition$(B_0, \frac{p}{2}, \epsilon_0)$;
        $(A_{p/2}, \ldots, A_{p-1}) :=$ MatrixPartition$(B_1, \frac{p}{2}, \epsilon_1)$;
else $A_0 := A$;
```

$P(0)$ (red), 29×34 for $P(1)$ (black), 31×31 for $P(2)$ (yellow), and 31×29 for $P(3)$ (blue). Together, the submatrices fit in the space of the original matrix. The global indices of a submatrix are not consecutive in the original matrix, but scattered. For instance, $\bar{I}_0 = \{2, 3, 4, 5, 11, 12, 14, \ldots, 52, 53, 55, 56, 57\}$, cf. Fig. 4.7. Note that $\bar{I}_0 \times \bar{J}_0 = I_0 \times J_0$ and $\bar{I}_3 \times \bar{J}_3 = I_3 \times J_3$, but that $\bar{I}_1 \times \bar{J}_1$ has six empty rows and nine empty columns, giving a size of 23×25 for $I_1 \times J_1$, and that $\bar{I}_2 \times \bar{J}_2$ has seven empty rows, giving a size of 24×31 for $I_2 \times J_2$. Empty rows and columns in a submatrix are the aim of a good partitioner, because they do not incur communication. An empty row is created by a column split in which all nonzeros of a row are assigned to the same processor, leaving the other processor empty-handed. The partitioning directions chosen for `prime60` were first splitting in the row direction, and then twice, independently, in the column direction.

The high-level recursive matrix partitioning algorithm does not specify the inner workings of the *split* function. To find a good split, we need a bipartitioning method based on the exact communication volume. It is convenient to express our problem in terms of hypergraphs, as was first done for matrix partitioning problems by Çatalyürek and Aykanat [36,37]. A **hypergraph** $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of **vertices** $\mathcal{V}$ and a set of **hyperedges**, or

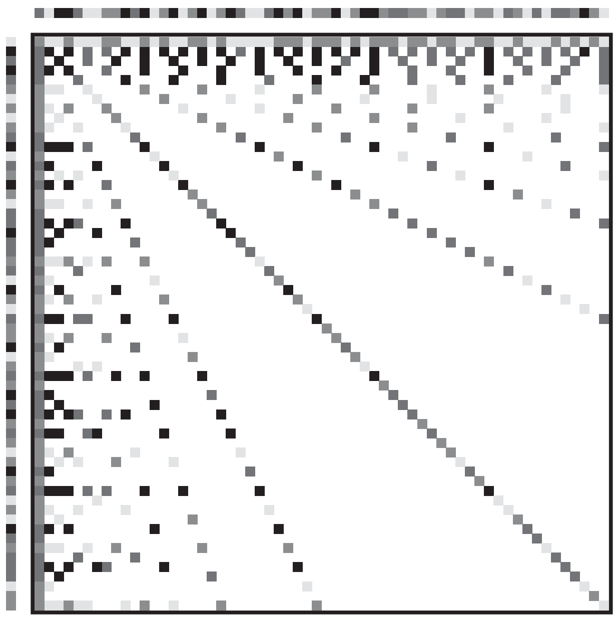


Fig. 4.7. Matrix and vector distribution for the sparse matrix `prime60`. Global view (see also Plate 2).

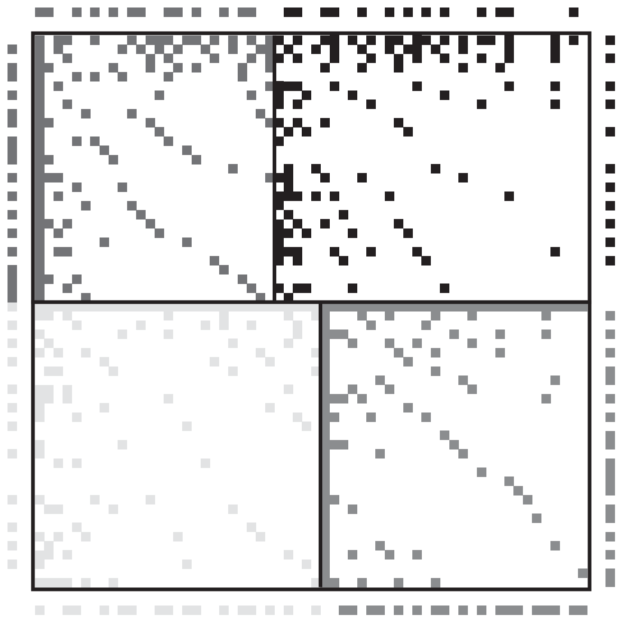


Fig. 4.8. Same matrix and distribution as in Fig. 4.7. Local view (see also Plate 3).

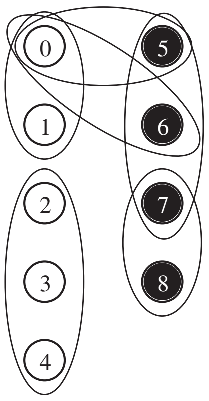


FIG. 4.9. Hypergraph with nine vertices and six nets. Each circle represents a vertex. Each oval curve enclosing a set of vertices represents a net. The vertex set is $\mathcal{V} = \{0, \ldots, 8\}$ and the nets are $n_0 = \{0,1\}$, $n_1 = \{0,5\}$, $n_2 = \{0,6\}$, $n_3 = \{2,3,4\}$, $n_4 = \{5,6,7\}$, and $n_5 = \{7,8\}$. The vertices have been coloured to show a possible assignment to processors, where $P(0)$ has the white vertices and $P(1)$ the black vertices.

nets, $\mathcal{N}$, which are subsets of $\mathcal{V}$. (A hypergraph is a generalization of an **undirected graph** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{E}$ is a set of **undirected edges**, which are unordered pairs (i, j) with $i, j \in \mathcal{V}$. Thus $(i, j) = (j, i)$ and can be identified with a subset $\{i, j\}$ of two elements.) Figure 4.9 illustrates the definition of a hypergraph.

Our splitting problem is to assign each of the n columns of a sparse $m \times n$ matrix to either processor $P(0)$ or $P(1)$. We can identify a matrix column j with a vertex j of a hypergraph with vertex set $\mathcal{V} = \{0, \ldots, n-1\}$, so that our problem translates into assigning each vertex to a processor. We have to minimize the communication volume of the fanin: assigning the nonzeros of a row to different processors gives rise to one communication, whereas assigning them to the same processor avoids communication. We can identify a matrix row i with a net n_i, defining

$$n_i = \{j : 0 \leq j < n \wedge a_{ij} \neq 0\}, \quad \text{for } 0 \leq i < m. \tag{4.33}$$

A communication arises if the net is **cut**, that is, not all its vertices are assigned to the same processor. The total communication volume incurred by the split thus equals the number of cut nets of the hypergraph. In the assignment of Fig. 4.9, two nets are cut: n_1 and n_2.

Example 4.8 Let $\mathcal{V} = \{0, 1, 2, 3, 4\}$. Let the nets be $n_0 = \{1, 4\}$, $n_1 = \{0, 1\}$, $n_2 = \{1, 2, 3\}$, $n_3 = \{0, 3, 4\}$, and $n_4 = \{2, 3, 4\}$. Let $\mathcal{N} = \{n_0, n_1, n_2, n_3, n_4\}$. Then $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is the hypergraph that corresponds to the column partitioning problem of Example 4.7. The optimal column solution has four cut nets: n_0, n_2, n_3, n_4.

We have to assign the vertices of the hypergraph in a balanced way, so that both processors receive about the same number of nonzeros. This is best modelled by making the vertices weighted, defining the weight c_j of vertex j to be the number of nonzeros in column j,

$$c_j = |\{i : 0 \leq i < m \wedge a_{ij} \neq 0\}|, \quad \text{for } 0 \leq j < n. \tag{4.34}$$

The splitting has to satisfy

$$\sum_{j \in P(s)} c_j \leq (1+\epsilon)\frac{nz(A)}{2}, \quad \text{for } s = 0, 1. \tag{4.35}$$

Having converted our problem to a hypergraph bipartitioning problem, we can apply algorithms developed for such problems. An excellent approach is to use the **multilevel** method [34], which consists of three phases: coarsening, initial partitioning, and uncoarsening. During the **coarsening phase**, the problem is reduced in size by merging similar vertices, that is, vertices representing columns with similar sparsity patterns. A natural heuristic is to do this pairwise, halving the number of vertices at each coarsening level. The best match for column j is an unmatched column j' with maximal overlap in the sparsity pattern, that is, maximal $|\{i : 0 \leq i < m \wedge a_{ij} \neq 0 \wedge a_{ij'} \neq 0\}|$. This value can be computed as the inner product of columns j and j', taking all nonzeros to be ones. The result of the merger is a column which has a nonzero in row i if $a_{ij} \neq 0$ or $a_{ij'} \neq 0$. In Example 4.7, the best match for column 2 is column 3, since their sparsity patterns have two nonzeros in common. For the purpose of load balancing, the new column gets a weight equal to the sum of the weights of the merged columns.

The **initial partitioning** phase starts when the problem is sufficiently reduced in size, typically when a few hundred columns are left. Each column is then assigned to a processor. The simplest initial partitioning method is by random assignment, but more sophisticated methods give better results. Care must be taken to obey the load balance criterion for the weights.

The initial partitioning for the smallest problem is transferred to the larger problems during the **uncoarsening phase**, which is similar to the coarsening phase but is carried out in the reverse direction. At each level, both columns of a matched pair are assigned to the same processor as their merged column. The resulting partitioning of the larger problem is refined, for instance by trying to move columns, or vertices, to the other processor. A simple approach would be to try a move of all vertices that are part of a cut net. Note that a vertex can be part of several cut nets. Moving a vertex to the other processor may increase or decrease the number of cut nets. The **gain** of a vertex is the reduction in cut nets obtained by moving it to the other processor. The best move has the largest gain, for example, moving vertex 0 to $P(1)$ in Fig. 4.9 has a gain of 1. The gain may be zero, such as in the case of moving vertex 5 or 6 to $P(0)$. The gain may also be negative, for example, moving vertex 1, 2, 3, 4, or 8 to the other processor has a gain of -1. The worst move is that of vertex 7, since its gain is -2.

Example 4.9 The following is a column bipartitioning of an 8×8 matrix by the multilevel method. During the coarsening, columns are matched in even–odd pairs, column 0 with 1, 2 with 3, and so on. The initial partitioning assigns columns to processors. A column owned by $P(1)$ has its nonzeros marked in boldface. All other columns are owned by $P(0)$.

$$\begin{bmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & 1 \\ \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & 1 & 1 \end{bmatrix} \xrightarrow{\textit{coarsen}} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & 1 \\ 1 & 1 & \cdot & 1 \\ 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \\ \cdot & 1 & \cdot & 1 \end{bmatrix} \xrightarrow{\textit{coarsen}} \begin{bmatrix} 1 & \cdot \\ 1 & 1 \\ 1 & 1 \\ 1 & \cdot \\ \cdot & 1 \\ \cdot & 1 \\ \cdot & 1 \\ 1 & 1 \end{bmatrix} \xrightarrow{\textit{partition}}$$

$$\begin{bmatrix} 1 & \cdot \\ 1 & \mathbf{1} \\ 1 & \mathbf{1} \\ 1 & \cdot \\ \cdot & \mathbf{1} \\ \cdot & \mathbf{1} \\ \cdot & \mathbf{1} \\ 1 & \mathbf{1} \end{bmatrix} \xrightarrow{\textit{uncoarsen}} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \mathbf{1} & \mathbf{1} \\ 1 & 1 & \cdot & \mathbf{1} \\ 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \mathbf{1} & \cdot \\ \cdot & \cdot & \mathbf{1} & \cdot \\ \cdot & \cdot & \cdot & \mathbf{1} \\ \cdot & 1 & \cdot & \mathbf{1} \end{bmatrix} \xrightarrow{\textit{refine}} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \mathbf{1} & \mathbf{1} \\ 1 & 1 & \cdot & \mathbf{1} \\ 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \mathbf{1} & \cdot \\ \cdot & \cdot & \mathbf{1} & \cdot \\ \cdot & \cdot & \cdot & \mathbf{1} \\ \cdot & 1 & \cdot & \mathbf{1} \end{bmatrix} \xrightarrow{\textit{uncoarsen}}$$

$$\begin{bmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & \mathbf{1} & \cdot & \mathbf{1} & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \mathbf{1} \\ \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \mathbf{1} & \mathbf{1} \end{bmatrix} \xrightarrow{\textit{refine}} \begin{bmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \mathbf{1} & \cdot & \cdot & \cdot & \mathbf{1} & \cdot & \mathbf{1} & \cdot \\ \mathbf{1} & 1 & 1 & 1 & \cdot & \cdot & \cdot & \mathbf{1} \\ \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{1} & \mathbf{1} \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \mathbf{1} & \mathbf{1} \end{bmatrix}$$

The communication volume after the initial partitioning is $V = 3$, and is caused by rows 1, 2, and 7. The first refinement does not change the partitioning, because no move of a single column reduces the communication and because all such moves cause a large imbalance in the computation. The second refinement moves column 0 to $P(1)$, giving a partitioning with $V = 2$ as the final result.

The Kernighan–Lin algorithm [120], originally developed for graph bipartitioning, can be applied to hypergraph bipartitioning. It can be viewed as a method for improving a given bipartitioning. The algorithm consists of several passes. Fiduccia and Mattheyses [69] give an efficient implementation of the Kernighan–Lin algorithm based on a priority-queue data structure for which one pass costs $\mathcal{O}(nz(A) + n)$. In a pass, all vertices are first marked as movable and their gain is computed. The vertex with the largest gain among the movable vertices is moved, provided this does not violate the load balance constraint. The vertex is then marked as nonmovable for the remainder of the current pass; this is to guarantee termination of the pass. The gains of the other vertices are then updated and a new move is determined. This process is continued until no more moves can be carried out. The best partitioning encountered during the pass (not necessarily the final partitioning) is saved and used as starting point for the next pass. Note that moves with negative gain are allowed and that they occur when no moves with positive or zero gain are available. A move with negative gain may still be advantageous, for instance if it is followed by a move of **adjacent** vertices (i.e. vertices that share a net with the moved vertex).

The Kernighan–Lin algorithm can be used in the initial partitioning and uncoarsening phases of the multilevel method. In the initial partitioning, the algorithm can be applied several times, each time to improve a different random assignment of the vertices to the processors. The best result is chosen. In the uncoarsening phase, the algorithm is commonly applied only once, and often with the movable vertices restricted to those that are part of a cut net. This cheaper variant is called the boundary Kernighan–Lin algorithm. Its use is motivated by the larger problem sizes involved and by the limited purpose of the uncoarsening phase, which is to refine a partitioning, and not to compute a completely new one.

4.6 Vector distribution

The matrix distribution algorithm of the previous section should lead to a matrix–vector multiplication with low communication volume and a good computational load balance. What remains to be done is to partition the input and output vectors such that the communication is balanced as well. In other words, given a matrix distribution ϕ, we have to determine a vector distribution $\phi_{\mathbf{v}}$ that minimizes the value h of the fanout and that satisfies $j \in J_{\phi_{\mathbf{v}}(j)}$, for $0 \leq j < n$. This constraint says that the processor $P(s) = P(\phi_{\mathbf{v}}(j))$ that obtains v_j must own a nonzero in matrix column j, that is, $j \in J_s$. We also have to find a vector distribution $\phi_{\mathbf{u}}$ that minimizes the value h of the fanin and that satisfies the constraint $i \in I_{\phi_{\mathbf{u}}(i)}$, for $0 \leq i < n$. These are two independent vector distribution problems, except if the requirement distr($\mathbf{u}$) = distr($\mathbf{v}$) must be satisfied; we assume that this is not the case.

Figure 4.7 gives a possible vector distribution in the global view. The vectors in this figure are depicted in the familiar way, cf. Fig. 4.4. Note that the two constraints mentioned above are satisfied: one of the nonzeros in each matrix column has the colour of the corresponding component v_j, and one of the nonzeros in each row has the colour of the corresponding component u_i. Figure 4.8 gives the same vector distribution, but now in the local view. The local components of the vector $\mathbf{u}$ are placed to the left of the local submatrix (for $P(0)$ and $P(2)$) or to the right (for $P(1)$ and $P(3)$), just outside the matrix boundary, whereas the local components of the vector $\mathbf{v}$ are placed above the local submatrix (for $P(0)$ and $P(1)$) or below it (for $P(2)$ and $P(3)$). For processor $P(0)$ (red), this gives a familiar picture, with the difference that now only the local part is shown.

A vector component v_j is depicted above or below the corresponding column of the submatrix if it is owned locally. Otherwise, the corresponding space in the picture of the vector distribution remains empty. For each resulting hole, a component v_j must be received, unless the corresponding local column is empty. Thus, it is easy to count the receives of the fanout by using this picture: 13, 12, 15, 11 components are received by $P(0)$, $P(1)$, $P(2)$, $P(3)$, respectively. The number of sends in the fanin can be counted similarly. Because of the way the matrix has been split, communication occurs in the fanin only between $P(0)$ and $P(1)$, and between $P(2)$ and $P(3)$. This makes it easy to count the number of receives in the fanin: a send for $P(0)$ implies a receive for $P(1)$, and vice versa; similarly, for $P(2)$ and $P(3)$. For the fanout, it is more difficult to count the sends, because $P(0)$ can now send a component v_j either to $P(2)$ or $P(3)$ (but not to $P(1)$). To determine the number of sends we need to count them in Fig. 4.7. Table 4.1 summarizes the statistics of the given vector distribution. The table shows that the communication cost

TABLE 4.1. Components of vectors $\mathbf{u}$ and $\mathbf{v}$ owned and communicated by the different processors for the matrix `prime60` in the distribution of Figures 4.7 and 4.8

	u			**v**		
s	$N(s)$	$h_{\rm s}(s)$	$h_{\rm r}(s)$	$N(s)$	$h_{\rm s}(s)$	$h_{\rm r}(s)$
0	18	11	12	13	13	13
1	11	12	11	13	13	12
2	12	12	12	16	14	15
3	19	12	12	18	11	11

The number of components owned by processor $P(s)$ is $N(s)$; the number sent is $h_{\rm s}(s)$; the number received is $h_{\rm r}(s)$

of the fanout is $15g$ and the cost of the fanin $12g$. The total communication cost of $27g$ is thus only slightly above the average $Vg/p = 98g/4 = 24.5g$, which means that the communication is well-balanced. Note that the number of vector components is less well-balanced, but this does not influence the cost of the matrix–vector multiplication. (It influences the cost of other operations though, such as the vector operations accompanying the matrix–vector multiplication in an iterative solver.)

The two vector distribution problems are similar; it is easy to see that we can solve the problem of finding a good distribution $\phi_{\mathbf{u}}$ given $\phi = \phi_A$ by finding a good distribution $\phi_{\mathbf{v}}$ given $\phi = \phi_{A^{\mathrm{T}}}$. This is because the nonzero pattern of row i of A is the same as the nonzero pattern of column i of A^{T}, so that a partial sum u_{is} is sent from $P(s)$ to $P(t)$ in the multiplication by A if and only if a vector component v_i is sent from $P(t)$ to $P(s)$ in the multiplication by A^{T}. Therefore, we only treat the problem for $\phi_{\mathbf{v}}$ and hence only consider the communication in the fanout.

Let us assume without loss of generality that we have a vector distribution problem with $q_j \geq 2$, for all j. Columns with $q_j = 0$ or $q_j = 1$ do not cause communication and hence may be omitted from the problem formulation. (A good matrix distribution method will give rise to many columns with $q_j = 1$.) Without loss of generality we may also assume that the columns are ordered by increasing q_j; this can be achieved by renumbering. Then the h-values for the fanout are

$$h_{\mathrm{s}}(s) = \sum_{0\leq j<n,\ \phi_{\mathbf{v}}(j)=s} (q_j - 1), \quad \text{for } 0 \leq s < p, \tag{4.36}$$

and

$$h_{\mathrm{r}}(s) = |\{j : j \in J_s \wedge \phi_{\mathbf{v}}(j) \neq s\}|, \quad \text{for } 0 \leq s < p. \tag{4.37}$$

Me first! Consider what would happen if a processor $P(s)$ becomes utterly egoistic and tries to minimize its own $h(s) = \max(h_{\mathrm{s}}(s), h_{\mathrm{r}}(s))$ without consideration for others. To minimize $h_{\mathrm{r}}(s)$, it just has to maximize the number of components v_j with $j \in J_s$ that it owns. To minimize $h_{\mathrm{s}}(s)$, it has to minimize the total weight of these components, where we define the weight of v_j as $q_j - 1$. An optimal strategy would thus be to start with $h_{\mathrm{s}}(s) = 0$ and $h_{\mathrm{r}}(s) = |J_s|$ and grab the components in increasing order (and hence increasing weight), adjusting $h_{\mathrm{s}}(s)$ and $h_{\mathrm{r}}(s)$ to account for each newly owned component. The processor grabs components as long as $h_{\mathrm{s}}(s) \leq h_{\mathrm{r}}(s)$, the new component included. We denote the resulting value of $h_{\mathrm{s}}(s)$ by $\hat{h}_{\mathrm{s}}(s)$, the resulting value of $h_{\mathrm{r}}(s)$ by $\hat{h}_{\mathrm{r}}(s)$, and that of $h(s)$ by $\hat{h}(s)$. Thus,

$$\hat{h}_{\mathrm{s}}(s) \leq \hat{h}_{\mathrm{r}}(s) = \hat{h}(s), \quad \text{for } 0 \leq s < p. \tag{4.38}$$

The value $\hat{h}(s)$ is indeed optimal for an egoistic $P(s)$, because stopping earlier would result in a higher $h_{\mathrm{r}}(s)$ and hence a higher $h(s)$ and because stopping later would not improve matters either: if for instance $P(s)$ would grab one component more, then $h_{\mathrm{s}}(s) > h_{\mathrm{r}}(s)$ so that $h(s) = h_{\mathrm{s}}(s) \geq h_{\mathrm{r}}(s) + 1 = \hat{h}_{\mathrm{r}}(s) = \hat{h}(s)$. The value $\hat{h}(s)$ is a local lower bound on the actual value that can be achieved in the fanout,

$$\hat{h}(s) \leq h(s), \quad \text{for } 0 \leq s < p. \tag{4.39}$$

Example 4.10 The following table gives the input of a vector distribution problem. If a processor $P(s)$ owns a nonzero in matrix column j, this is denoted by a 1 in the corresponding location; if it does not own such a nonzero, this is denoted by a dot. This problem could for instance be the result of a matrix partitioning for $p = 4$ with all splits in the row direction. (We can view the input itself as a sparse $p \times n$ matrix.)

$s = 0$	1	·	1	·	1	1	1	1
1	1	1	·	1	1	1	1	·
2	·	1	·	·	·	1	1	1
3	·	·	1	1	1	·	·	1
$q_j =$	2	2	2	2	3	3	3	3
$j =$	0	1	2	3	4	5	6	7

Processor $P(0)$ wants v_0 and v_2, so that $\hat{h}_{\mathrm{s}}(0) = 2$, $\hat{h}_{\mathrm{r}}(0) = 4$, and $\hat{h}(0) = 4$; $P(1)$ wants v_0, v_1, and v_3, so that $\hat{h}(1) = 3$; $P(2)$ wants v_1, giving $\hat{h}(2) = 3$; and $P(3)$ wants v_2 and v_3, giving $\hat{h}(3) = 2$. The fanout will cost at least $4g$.

More in general, we can compute a lower bound $\hat{h}(J, ns_0, nr_0)$ for a given index set $J \subset J_s$ and a given initial number of sends ns_0 and receives nr_0. We denote the corresponding send and receive values by $\hat{h}_{\mathrm{s}}(J, ns_0, nr_0)$ and $\hat{h}_{\mathrm{r}}(J, ns_0, nr_0)$. The initial communications may be due to columns outside J. This bound is computed by the same method, but starting with the values $h_{\mathrm{s}}(s) = ns_0$ and $h_{\mathrm{r}}(s) = nr_0 + |J|$. Note that $\hat{h}(s) = \hat{h}(J_s, 0, 0)$. The generalization of eqn (4.38) is

$$\hat{h}_{\mathrm{s}}(J, ns_0, nr_0) \leq \hat{h}_{\mathrm{r}}(J, ns_0, nr_0) = \hat{h}(J, ns_0, nr_0). \tag{4.40}$$

Think about the others! Every processor would be happy to own the lighter components and would rather leave the heavier components to the others. Since every component v_j will have to be owned by exactly one processor, we must devise a mechanism to resolve conflicting desires. A reasonable heuristic seems to be to give preference to the processor that faces the toughest future, that is, the processor with the highest value $\hat{h}(s)$. Our aim in the vector distribution algorithm is to minimize the highest $h(s)$, because $(\max_{0 \leq s < p} h(s)) \cdot g$ is the communication cost of the fanout.

Algorithm 4.7 is the vector distribution algorithm based on the local-bound heuristic; it has been proposed by Meesen and Bisseling [136]. The algorithm

Algorithm 4.7. Local-bound based vector partitioning.

```
input:  $\phi = \mathrm{distr}(A)$, matrix distribution over $p$ processors, $p \geq 1$,
        where $A$ is an $n \times n$ sparse matrix.
output: $\phi_{\mathbf{v}} = \mathrm{distr}(\mathbf{v})$: vector distribution over $p$ processors,
        satisfying $j \in J_{\phi_{\mathbf{v}}(j)}$, for $0 \leq j < n$,
        where $J_s = \{j : 0 \leq j < n \wedge (\exists i : 0 \leq i < n \wedge \phi(i,j) = s)\}$,
        for $0 \leq s < p$.

for $s := 0$ to $p-1$ do
        $L_s := J_s$;
        $h_\mathrm{s}(s) := 0$;
        $h_\mathrm{r}(s) := 0$;
        if $h_\mathrm{s}(s) < \hat{h}_\mathrm{s}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$ then
                active$(s)$ := $true$;
        else    active$(s)$ := $false$;

while $(\exists s : 0 \leq s < p \wedge \mathrm{active}(s))$ do
        $s_{\max} := \mathrm{argmax}(\hat{h}_\mathrm{r}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s)) : 0 \leq s < p \wedge \mathrm{active}(s))$;
        $j := \min(L_{s_{\max}})$;
        $\phi_{\mathbf{v}}(j) = s_{\max}$;
        $h_\mathrm{s}(s_{\max}) := h_\mathrm{s}(s_{\max}) + q_j - 1$;
        for all $s : 0 \leq s < p \wedge s \neq s_{\max} \wedge j \in J_s$ do
                $h_\mathrm{r}(s) := h_\mathrm{r}(s) + 1$;
        for all $s : 0 \leq s < p \wedge j \in J_s$ do
                $L_s := L_s \backslash \{j\}$;
                if $h_\mathrm{s}(s) = \hat{h}_\mathrm{s}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$ then
                        active$(s)$ := $false$;
```

successively assigns components v_j; the set L_s is the index set of components that may still be assigned to $P(s)$. The number of sends caused by the assignments is registered as $h_\mathrm{s}(s)$ and the number of receives as $h_\mathrm{r}(s)$. The processor with the highest local lower bound $\hat{h}_\mathrm{r}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$ becomes the happy owner of the lightest component available. The values $\hat{h}_\mathrm{r}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$ and $\hat{h}_\mathrm{s}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$ may change after every assignment. A processor will not accept components any more from the moment it knows it has achieved its optimum, which happens when $h_\mathrm{s}(s) = \hat{h}_\mathrm{s}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$. (Note that $ns_0 \leq \hat{h}_\mathrm{s}(J, ns_0, nr_0)$, so that trivially $h_\mathrm{s}(s) \leq \hat{h}_\mathrm{s}(L_s, h_\mathrm{s}(s), h_\mathrm{r}(s))$.) Accepting additional components would raise its final $h(s)$. This egoistic approach holds for every processor, and not only for the one with the highest current bound. (Accepting more components for altruistic reasons may be well-intended, but is still a bad idea because the components thus accepted may be more useful

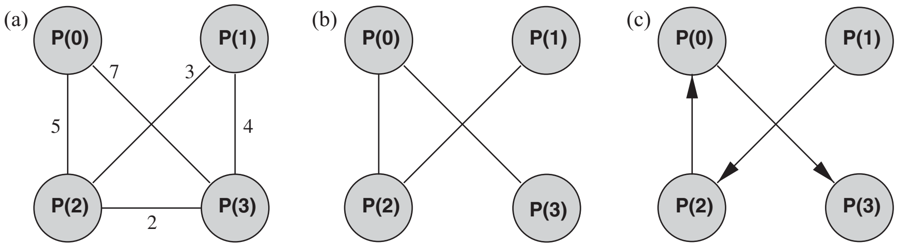


Fig. 4.10. Transforming the undirected communication graph of a matrix distribution into a directed graph. (a) The original undirected communication graph with multiple edges shown as edge weights; (b) the undirected graph after removal of all pairs of edges; (c) the final directed graph. As a result, $P(0)$ has to send two values to $P(2)$ and receive three values from $P(2)$; it has to send four values to $P(3)$ and receive three.

to other processors.) The algorithm terminates when no processor is willing to accept components any more.

After termination, a small fraction (usually much less than 10%) of the vector components remains unowned. These are the heavier components. We can assign the remaining components in a greedy fashion, each time assigning a component v_j to the processor $P(s)$ for which this would result in the lowest new value $h(s)$.

A special case occurs if the matrix partitioning has the property $q_j \leq 2$, for all j. This case can be solved to optimality by an algorithm that has as input the undirected communication graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defined by a vertex set $\mathcal{V} = \{0, \ldots, p-1\}$ representing the processors and an edge set $\mathcal{E}$ representing matrix columns shared by a pair of processors, where an edge (s, t) implies communication from $P(s)$ to $P(t)$ or vice versa. Multiple edges between the same pair of vertices are possible, see Fig. 4.10(a). The algorithm first removes all multiple edges in pairs: if matrix column j and j' are both shared by $P(s)$ and $P(t)$, then v_j is assigned to $P(s)$ and $v_{j'}$ to $P(t)$. This gives rise to one send and one receive for both processors, balancing their communication obligations. The undirected graph that remains has at most one edge between each pair of vertices, see Fig. 4.10(b).

The algorithm now picks an arbitrary vertex with odd degree as the starting point for a path. The **degree** of a vertex is the number of edges connected to it, for example, the degree of vertex 0 in Fig. 4.10(b) is two. A **path** is a sequence of vertices that are connected by edges. Edges along the path are transformed into directed edges in a new, directed graph, see Fig. 4.10(c). (In a **directed graph**, each edge has a direction. Thus an edge is an ordered pair (s, t), which differs from (t, s).) The direction of the created edge is the same as that of the path. The path ends when a vertex is reached that has no more undirected edges connected to it. This procedure is repeated until no more odd-degree vertices are present. It is easy to see that our procedure cannot change the degree of a vertex from even to odd. Finally, the same procedure is carried out starting at even-degree vertices.

Once all undirected edges have been transformed into directed edges, we have obtained a directed graph, which determines the owner of every remaining vector component: component v_j corresponding to a directed edge (s, t) is assigned to $P(s)$, causing a communication from $P(s)$ to $P(t)$. The resulting vector distribution has minimal communication cost; for a proof of optimality, see [136]. The vector distribution shown in Fig. 4.7 has been determined this way. The matrix `prime60` indeed has the property $p_i \leq 2$ for all i and $q_j \leq 2$ for all j, as a consequence of the different splitting directions of the matrix, that is, first horizontal, then vertical.

4.7 Random sparse matrices

A **random sparse matrix** A can be obtained by determining randomly and independently for each matrix element a_{ij} whether it is zero or nonzero. If the probability of creating a nonzero is d and hence that of creating a zero is $1 - d$, the matrix has an expected density $d(A) = d$ and an expected number of nonzeros $nz(A) = dn^2$. This definition of randomness only concerns the sparsity pattern, and not the numerical values of the nonzeros.

Historically, the first sparse matrix algorithms were tested using random sparse matrices. Later, one realized that these matrices constitute a very particular class and that many sparse matrices from practical applications fall outside this class. This led to the development of the Harwell–Boeing collection [64,65] of sparse matrices, now called the Rutherford–Boeing collection.

If nothing is known about a given sparse matrix A, except for its size $n \times n$ and its sparsity pattern, and if no structure is discernible, then a first approximation is to consider A as a random sparse matrix with density $d = nz(A)/n^2$. Still, it is best to call such a sparse matrix **unstructured**, and not random sparse, because random sparse matrices have a very special property: every subset of the matrix elements, chosen independently from the sparsity pattern, has an expected fraction d of nonzeros. This property provides us with a powerful tool for analysing algorithms involving random sparse matrices and finding distributions for them.

The question whether a given sparse matrix such as the one shown in Fig. 4.11 is random is tricky and just as hard to answer as the question whether a given random number generator generates a true sequence of random numbers. If the random number generator passes a battery of tests, then for all practical purposes the answer is positive. The same pragmatic approach can be taken for random sparse matrices. One test could be to split the matrix into four submatrices of equal size, and check whether each has about $dn^2/4$ nonzeros, within a certain tolerance given by probability theory. In this section, we do not have to answer the tricky question, since we

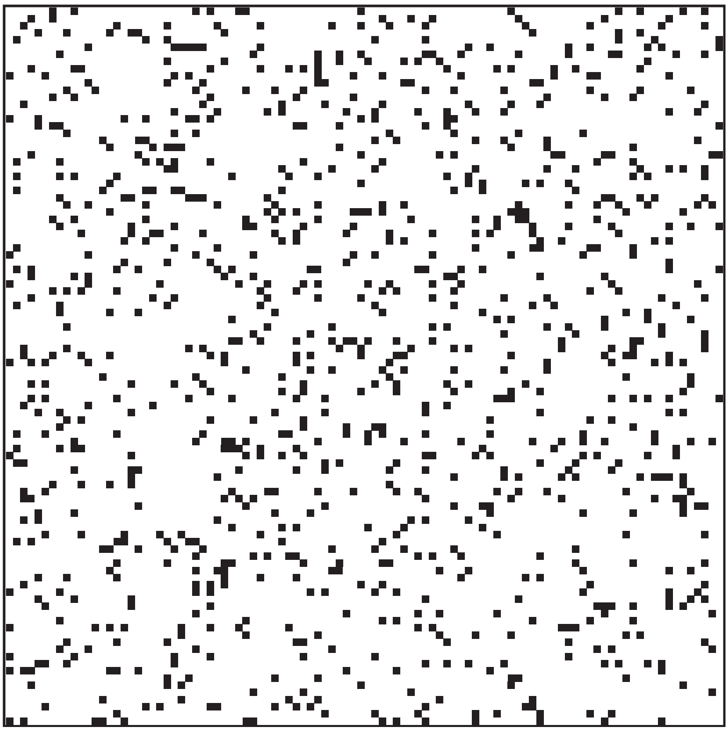


FIG. 4.11. Sparse matrix `random100` with $n = 100$, $nz = 1000$, $c = 10$, and $d = 0.1$, interactively generated at the Matrix Market Deli [26], see `http://math.nist.gov/MatrixMarket/deli/Random/`.

assume that the sparse matrix is random by construction. (We have faith in the random number generator we use, `ran2` from [157]. One of its characteristics is a period of more than 2×10^{18}, meaning that it will not repeat itself for a very long time.)

Now, let us study parallel matrix–vector multiplication for random sparse matrices. Suppose we have constructed a random sparse matrix A by drawing for each index pair (i, j) a random number $r_{ij} \in [0, 1]$, doing this independently and **uniformly** (i.e. with each outcome equally likely), and then creating a nonzero a_{ij} if $r_{ij} < d$. Furthermore, suppose that we have distributed A over the p processors of a parallel computer in a manner that is independent of the sparsity pattern, by assigning an equal number of elements (whether zero or nonzero) to each processor. For simplicity, assume that $n \bmod p = 0$. Therefore, each processor has n^2/p elements. Examples of such a distribution are the square block distribution and the cyclic row distribution.

First, we investigate the effect of such a fixed, pattern-independent distribution scheme on the spread of the nonzeros, and hence on the load balance in the main computation part of Algorithm 4.5, the local matrix–vector multiplication (1). The load balance can be estimated by using probability theory. The problem here is to determine the expected maximum, taken over all processors, of the local number of nonzeros. We cannot solve this problem exactly, but we can still obtain a useful bound on the probability of the maximum exceeding a certain value, by applying a theorem of Chernoff, which is often used in the analysis of randomized algorithms. A proof and further details and applications can be found in [142].

Theorem 4.11 *(Chernoff [40]). Let $0 < d < 1$. Let $X_0, X_2, \ldots, X_{m-1}$ be independent Bernoulli trials with outcome 0 or 1, such that $\mathbf{Pr}[X_k = 1] = d$, for $0 \leq k < m$. Let $X = \sum_{k=0}^{m-1} X_k$ and $\mu = md$. Then for every $\epsilon > 0$,*

$$\mathbf{Pr}[X > (1+\epsilon)\mu] < \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu}. \tag{4.41}$$

If we flip a biased coin which produces heads with probability d, then the Chernoff bound tells us how small the probability is of getting $\epsilon\mu$ more heads than the expected average μ. The bound for $\epsilon = 1$ tells us that the probability of getting more than twice the expected number of heads is less than $(\mathrm{e}/4)^{\mu} \approx (0.68)^{md}$. Often, we apply the bound for smaller values of ϵ.

In the case of a random sparse matrix distributed over p processors, every processor has $m = n^2/p$ elements, each being nonzero with a probability d. The expected number of nonzeros per processor is $\mu = dn^2/p$. Let E_s be the event that processor $P(s)$ has more than $(1+\epsilon)\mu$ nonzeros and E the event that at least one processor has more than $(1+\epsilon)\mu$ nonzeros, that is, $E = \cup_{s=0}^{p-1} E_s$. The probability that at least one event from a set of events happens is less than or equal to the sum of the separate probabilities of the events, so that $\mathbf{Pr}[E] \leq \sum_{s=0}^{p-1} \mathbf{Pr}[E_s]$. Because all events have the same probability $\mathbf{Pr}[E_0]$, this yields $\mathbf{Pr}[E] \leq p\mathbf{Pr}[E_0]$. Since each nonzero causes two flops in superstep (1), we get as a result

$$\mathbf{Pr}\left[T_{(1)} > \frac{2(1+\epsilon)dn^2}{p}\right] < p\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{dn^2/p}. \tag{4.42}$$

The bound for $\epsilon = 1$ tells us that the extra time caused by load imbalance exceeds the ideal time of the computation itself with probability less than $p(0.68)^{dn^2/p}$. Figure 4.12 plots the function $F(\epsilon)$ defined as the right-hand side of eqn (4.42) against the normalized computation cost $1+\epsilon$, for $n = 1000$, $p = 100$, and three different choices of d. The normalized computation cost of superstep (1) is the computation cost in flops divided by the cost of a perfectly parallelized computation. The figure shows for instance that for $d = 0.01$ the expected normalized cost is at most 1.5; this is because the probability of exceeding 1.5 is almost zero.

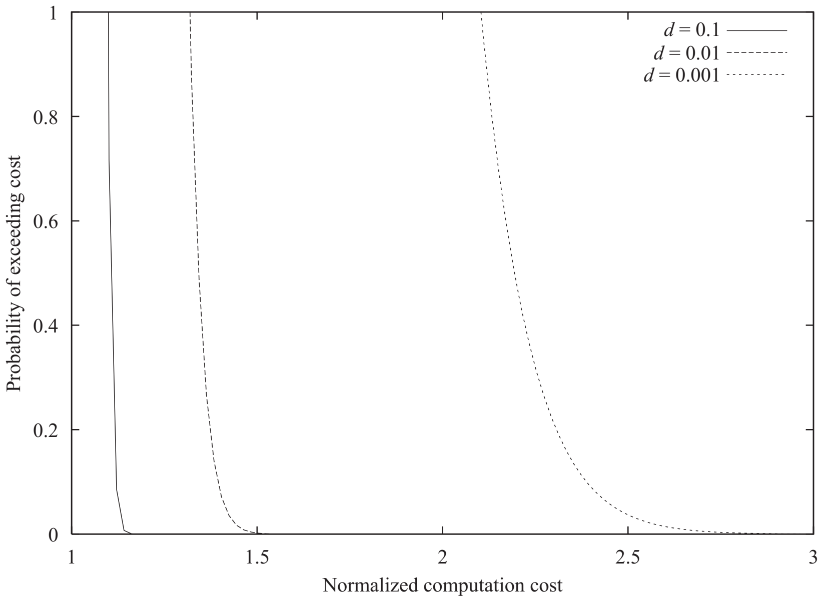


FIG. 4.12. Chernoff bound on the probability that a given normalized computation cost is exceeded, for a random sparse matrix of size $n = 1000$ and density d distributed over $p = 100$ processors.

The expected normalized computation cost for given n, p, and d can be estimated more accurately by performing a simulation experiment. In this experiment, a set of random sparse matrices is created, each matrix is distributed by a fixed scheme that is independent of the sparsity pattern (e.g. by a square block distribution), and its maximum local number of nonzeros is determined. The average over the whole set of matrices is an estimate of the expected maximum number of nonzeros; dividing the average by dn^2/p gives an estimate of the expected normalized cost. For the matrices of Fig. 4.12, the average normalized computation costs are: 1.076 for $d = 0.1$; 1.258 for $d = 0.01$; and 1.876 for $d = 0.001$. These values were obtained by creating 10 000 matrices.

Figure 4.13 shows a histogram of the probability distribution for $d = 0.01$, obtained by creating 100 000 matrices. The most frequent result in our simulation is a maximum local nonzero count of 124; this occurred 9288 times. Translated into normalized costs and probabilities, this means that the normalized cost of 1.24 has the highest probability, namely 9.3%. For comparison, the figure also shows the derivative of the function $1-F(\epsilon)$. This derivative can be interpreted as a probability density function corresponding to the Chernoff bound. The derivative has been scaled, multiplying it by a factor $\Delta\epsilon = 0.01$, to make the function values comparable with the histogram values. The bars of the histogram are far below the line representing the scaled derivative, meaning that our Chernoff bound is rather pessimistic.

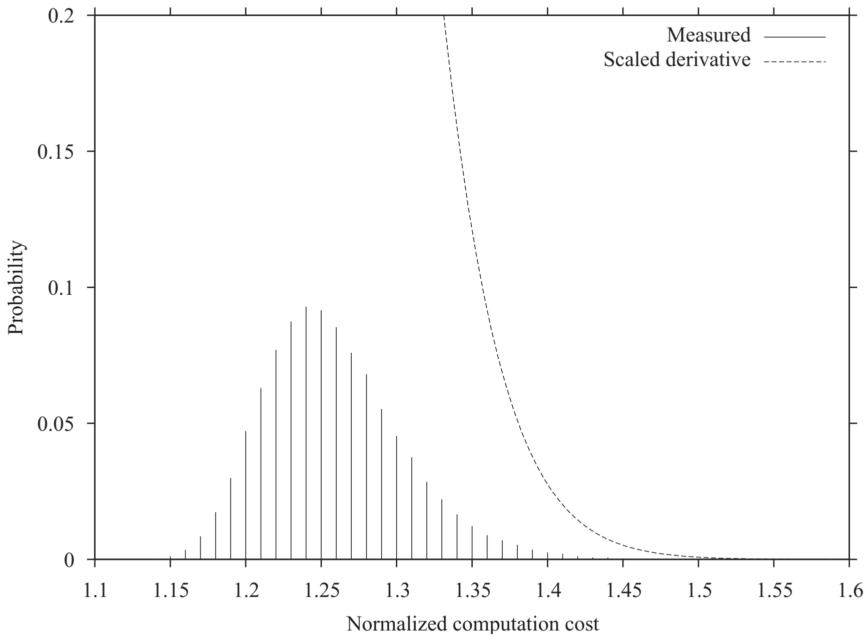


FIG. 4.13. Histogram of the probability distribution of the normalized computation cost, for a random sparse matrix of size $n = 1000$ and density $d = 0.01$ distributed over $p = 100$ processors. Also shown is the function $-F'(\epsilon)/100$, the scaled derivative of the Chernoff bound.

Based on the above, we may expect that distributing a random sparse matrix independently of its sparsity pattern spreads the computation well; we can quantify this expectation using the Chernoff bound. The same quality of load balance is expected for every distribution scheme with an equal number of matrix elements assigned to the processors. For the communication, however, the choice of distribution scheme makes a difference. The communication volume for a dense matrix is an upper bound on the volume for a sparse matrix distributed by the same fixed, pattern-independent distribution scheme. For a random sparse matrix with a high density, the communication obligations will be the same as for a dense matrix. Therefore, the best we can do to find a good fixed distribution scheme for random sparse matrices, is to apply methods for reducing communication in the dense case. A good choice is a square Cartesian distribution based on a cyclic distribution of the matrix diagonal, cf. Fig. 4.6, where each processor has an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix. A suitable corresponding choice of vector distribution is to distribute the vector $\mathbf{u}$ and $\mathbf{v}$ in the same cyclic way as the matrix diagonal. (This does not guarantee though that the owner of a vector component v_j also owns a nonzero in column j.)

To obtain the communication cost of Algorithm 4.5 for this distribution, we first examine superstep (0). Vector component v_j is needed only by processors in $P(*, \phi_1(j))$. A processor $P(s, \phi_1(j))$ does not need the component v_j if all $n/\sqrt{p}$ elements in the local part of matrix column j are zero; this event has probability $(1-d)^{n/\sqrt{p}}$. The probability that $P(s, \phi_1(j))$ needs v_j is $1-(1-d)^{n/\sqrt{p}}$. Since $\sqrt{p}-1$ processors each have to receive v_j with this probability, the expected number of receives for component v_j is $(\sqrt{p}-1)(1-(1-d)^{n/\sqrt{p}})$. The owner of v_j does not have to receive it. The expected communication volume of the fanout is therefore $n(\sqrt{p}-1)(1-(1-d)^{n/\sqrt{p}})$. Since no processor is preferred, the h-relation is expected to be balanced, so that the expected communication cost of superstep (0) is

$$T_{(0)} = \left(\frac{1}{\sqrt{p}} - \frac{1}{p}\right)(1-(1-d)^{n/\sqrt{p}})ng. \tag{4.43}$$

Communication superstep (2) is similar to (0), with the operation of sending vector components replaced by the operation of receiving partial sums. The communication cost of superstep (2) is

$$T_{(2)} = T_{(0)}. \tag{4.44}$$

Superstep (3) adds the nonzero partial sums, both those just received and those present locally. This costs

$$T_{(3)} = \frac{n}{\sqrt{p}}(1-(1-d)^{n/\sqrt{p}}). \tag{4.45}$$

If $g \gg 1$, which is often the case, then $T_{(3)} \ll T_{(2)}$, so that the cost of superstep (3) can be neglected. Finally, the synchronization cost of the whole algorithm is $4l$.

For our example of $n = 1000$ and $p = 100$, the matrix with highest density, $d = 0.1$, is expected to cause a communication cost of $179.995g$, which is close to the cost of $180g$ for a completely dense matrix. The corresponding expected normalized communication cost is $(T_{(0)} + T_{(2)})/(2dn^2/p) \approx 0.09g$. This means that we need a parallel computer with $g \leq 11$ ro run our algorithm with more than 50% efficiency.

For matrices with very low density, the local part of a matrix column is unlikely to have more than one nonzero. Every nonzero will thus incur a communication. In that case a row distribution is better than a square matrix distribution, because this saves the communications of the fanin. For our example of $n = 1000$ and $p = 100$, the matrix with lowest density, $d = 0.001$, is expected to cause a normalized communication cost of $0.86g$ for a square matrix distribution and $0.49g$ for the cyclic row distribution.

One way of improving the performance is by tailoring the distribution used to the sparsity pattern of the random sparse matrix. Figure 4.14 shows a tailored distribution produced by the Mondriaan package for the matrix `random100`. The figure gives a local view of the matrix, showing the

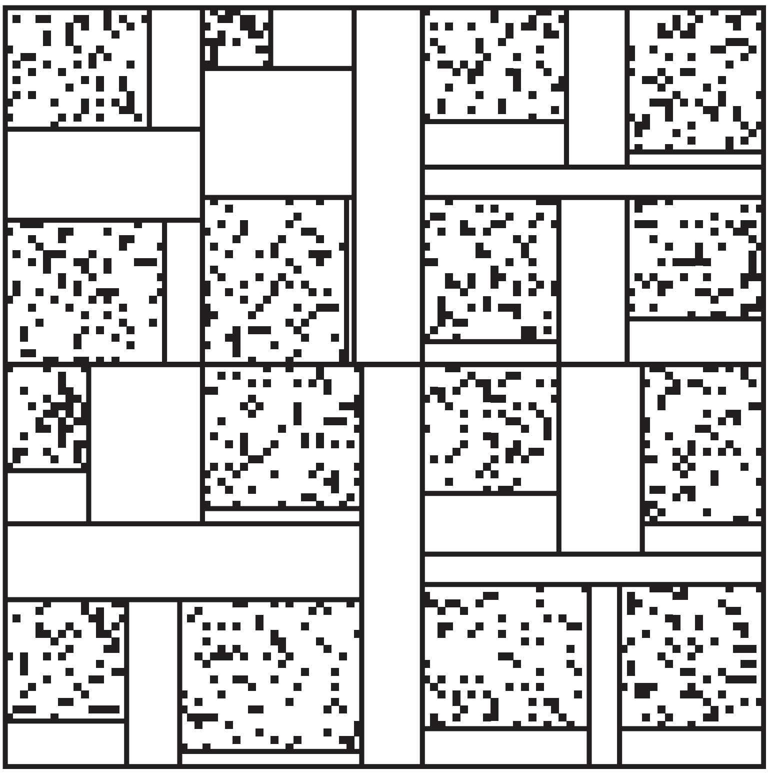


FIG. 4.14. Local view of sparse matrix `random100` from Fig. 4.11 with $n = 100$, $nz = 1000$, and $d = 0.1$, distributed by the Mondriaan package over $p = 16$ processors. The allowed imbalance is $\epsilon = 20\%$; the achieved imbalance is $\epsilon' = 18.4\%$. The maximum number of nonzeros per processor is 74; the average is 62.5; and the minimum is 25. The communication volume is $V = 367$. The first split is in the row direction; the next two splits are in the column direction. The empty row and column parts created by the splits are collected in empty rectangles.

submatrix $I_s \times J_s$ for each processor $P(s)$, $0 \leq s < 16$. (This is slightly different from the local view given in Fig. 4.8, which shows the submatrix $\bar{I}_s \times \bar{J}_s$, where empty row and column parts are included.) Note that the relatively large allowed load imbalance helps reduce the communication, resulting in large empty rectangles. Note that each submatrix $I_s \times J_s$ is the result of two row splits and two column splits, but in different order for different processors.

Table 4.2 compares the theoretical communication volume for the best pattern-independent distribution scheme, the square Cartesian distribution based on a cyclic distribution of the matrix diagonal, with the volume for

TABLE 4.2. Communication volume for a random sparse matrix of size $n = 1000$ and density $d = 0.01$ distributed over p processors, using a pattern-independent Cartesian distribution and a pattern-dependent distribution produced by the Mondriaan package

p	ϵ (in %)	V (Cartesian)	V (Mondriaan)
2	0.8	993	814
4	2.1	1987	1565
8	4.0	3750	2585
16	7.1	5514	3482
32	11.8	7764	4388

the distribution produced by the Mondriaan package (version 1.0, run with default parameters), averaged over a set of 100 random sparse matrices. The volume for the Cartesian distribution is based on the cost formula eqn (4.43), generalized to handle nearly square distributions as well, such as the 8×4 distribution for $p = 32$. For ease of comparison, the value of ϵ specified as input to Mondriaan equals the expected load imbalance for the Cartesian distribution. The achieved imbalance ϵ' is somewhat below that value. It is clear from the table that the Mondriaan distribution causes less communication, demonstrating that on average the package succeeds in tailoring a better distribution to the sparsity pattern. For $p = 32$, we gain about 45%.

The vector distribution corresponding to the Cartesian distribution satisfies $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$, which is an advantage if this requirement must be met. The volume for the Mondriaan distribution given in Table 4.2 may increase in case of such a requirement, but at most by n, see eqn (4.12). For $p \geq 8$, the Mondriaan distribution is guaranteed to be superior in this case.

We may conclude from these results that the parallel multiplication of a random sparse matrix and a vector is a difficult problem, most likely leading to much communication. Only low values of g or high nonzero densities can make this operation efficient. Using the fixed, pattern-independent Cartesian distribution scheme based on a cyclic distribution of the matrix diagonal already brings us close to the best distribution we can achieve. The load balance of this distribution is expected to be good in most cases, as the numerical simulation and the Chernoff bound show. The distribution can be improved by tailoring the distribution to the sparsity pattern, for example, by using the Mondriaan package, but the improvement is modest.

4.8 Laplacian matrices

In many applications, a physical domain exists that can be distributed naturally by assigning a contiguous subdomain to every processor. Communication is then only needed for exchanging information across the subdomain boundaries. Often, the domain is structured as a multidimensional rectangular grid, where grid points interact only with a set of immediate neighbours. In the two-dimensional case, this set could for instance contain the neighbours to the north, east, south, and west. One example of a grid application is the Ising model used to study ferromagnetism, in particular phase transitions and critical temperatures. Each grid point in this model represents a particle with positive or negative spin; neighbours tend to prefer identical spins. (For more on the Ising model, see [145].) Another example is the heat equation, where the value at a grid point represents the temperature at the corresponding location. The heat equation can be solved iteratively by a relaxation procedure that computes the new temperature at a point using the old temperature of that point and its neighbours.

An important operation in the solution of the two-dimensional heat equation is the application of the two-dimensional **Laplacian operator** to the grid, computing

$$\Delta_{i,j} = x_{i-1,j} + x_{i+1,j} + x_{i,j+1} + x_{i,j-1} - 4x_{i,j}, \quad \text{for } 0 \le i,j < k, \tag{4.46}$$

where $x_{i,j}$ denotes the temperature at grid point (i,j). The difference $x_{i+1,j} - x_{i,j}$ approximates the derivative of the temperature in the i-direction, and the difference $(x_{i+1,j} - x_{i,j}) - (x_{i,j} - x_{i-1,j}) = x_{i-1,j} + x_{i+1,j} - 2x_{i,j}$ approximates the second derivative. By convention, we assume that $x_{i,j} = 0$ outside the $k \times k$ grid; in practice, we just ignore zero terms in the right-hand side of eqn (4.46).

We can view the $k \times k$ array of values $x_{i,j}$ as a one-dimensional vector $\mathbf{v}$ of length $n = k^2$ by the identification

$$v_{i+jk} \equiv x_{i,j}, \quad \text{for } 0 \le i,j < k, \tag{4.47}$$

and similarly we can identify the $\Delta_{i,j}$ with a vector $\mathbf{u}$.

Example 4.12 Consider the 3×3 grid shown in Fig. 4.15. Equation (4.46) now becomes $\mathbf{u} = A\mathbf{v}$, where

$$A = \begin{bmatrix} -4 & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & -4 & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & -4 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & -4 & 1 & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & -4 & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & 1 & -4 & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & -4 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & -4 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & -4 \end{bmatrix} = \begin{bmatrix} B & I_3 & 0 \\ I_3 & B & I_3 \\ 0 & I_3 & B \end{bmatrix}.$$

The matrix A is pentadiagonal because vector components (representing grid points) are only connected to components at distance ± 1 or ± 3. The holes in

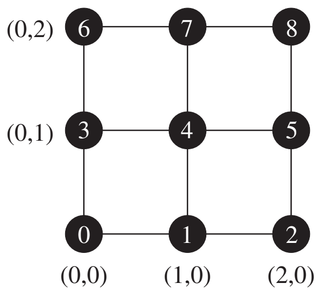


FIG. 4.15. A 3×3 grid. For each grid point (i,j), the index $i+3j$ of the corresponding vector component is shown.

the subdiagonal and superdiagonal occur because points (i,j) with $i = 0,2$ do not have a neighbour at distance -1 and $+1$, respectively. As a result, the matrix has a block-tridiagonal structure with 3×3 blocks B on the main diagonal and 3×3 identity matrices just below and above it.

In general, it is best to view application of the Laplacian operator as an operation on the physical domain. This domain view has the advantage that it naturally leads to the use of a regular data structure for storing the data. Occasionally, however, it may also be beneficial to view the Laplacian operation as a matrix operation, so that we can apply our knowledge about sparse matrix–vector multiplication and gain from insights obtained in that context.

Let us try to find a good distribution for the $k\times k$ grid. We adopt the domain view, and not the matrix view, and therefore we must assign each grid point to a processor. The resulting distribution of the grid should uniquely determine the distribution of the matrix and the vectors in the corresponding matrix–vector multiplication. We assign the values $x_{i,j}$ and $\Delta_{i,j}$ to the owner of grid point (i,j), and this translates into $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$. It is easiest to use a row distribution for the matrix and assign row $i+jk$ to the same processor as vector component u_{i+jk} and hence grid point (i,j). (For low-dimensional Laplacian matrices, using a square matrix distribution does not give much advantage over a row distribution.) The resulting sparse matrix–vector multiplication algorithm has two supersteps, the fanout and the local matrix–vector multiplication. If a neighbouring point is on a different processor, its value must be obtained during the fanout. The computation time assuming an equal spread of grid points is $5k^2/p$, since eqn (4.46) gives rise to five flops per grid point. Note that in this specific application we use the flop count corresponding to the domain view, and not the more general count of two flops per matrix nonzero used in other sections of this chapter; the latter count yields ten flops per grid point. Furthermore, we do not have to store the matrix elements explicitly, that is, we may use matrix-free storage, cf. Section 4.2.

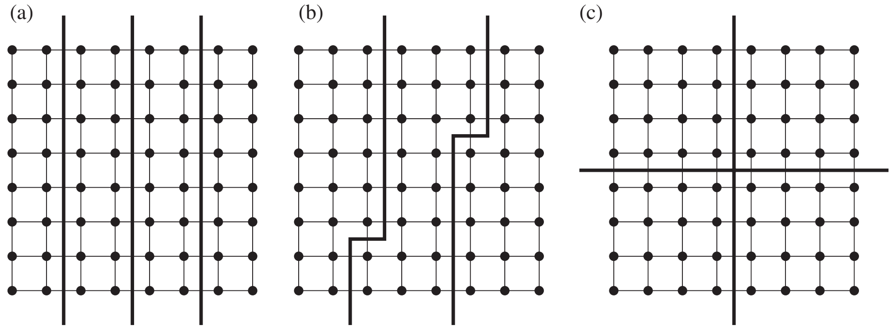


FIG. 4.16. Distribution of an 8×8 grid. (a) by strips for $p = 4$ processors; (b) by strips with border corrections for $p = 3$; (c) by square blocks for $p = 4$.

The simplest distribution of the grid can be obtained by dividing it into p strips, each of size $k \times k/p$, assuming that $k \bmod p = 0$, see Fig. 4.16(a). The advantage is, indeed, simplicity and the fact that communication is only needed in the east–west direction, between a grid point (i, j) on the eastern border of a strip and its neighbour $(i+1, j)$, or between a point on the western border and its neighbour $(i-1, j)$. The northern and southern neighbours of a grid point are on the same processor as the point itself and hence do not cause communication. The main disadvantage will immediately be recognized by every Norwegian or Chilean: relatively long borders. Each processor, except the first and the last, has to send and receive $2k$ boundary values. Therefore,

$$T_{\text{comm, strips}} = 2kg. \tag{4.48}$$

A related disadvantage is that $p \le k$ should hold, because otherwise processors would be idle. For $p \le k$, load balance may still be a serious problem: if $k \bmod p \ne 0$, some processors will contain one extra column of k points. This problem can be solved by border corrections, jumping one point to the east or west somewhere along the border, as shown in Fig. 4.16(b).

A better distribution can be obtained by dividing the grid into p square blocks, each of size $k/\sqrt{p} \times k/\sqrt{p}$, where we assume that p is a square number and that $k \bmod \sqrt{p} = 0$, see Fig. 4.16(c). The borders are smaller now. In the special case $p = 4$, each processor has to receive k neighbouring values, send $k-2$ values to one destination processor, and send a corner value to two destination processors, thus sending a total of k values. The communication is reduced by a factor of two compared with the strip distribution. In the general case $p > 4$, processors not on the boundary are busiest communicating, having to send and receive $4k/\sqrt{p}$ values, so that

$$T_{\text{comm, squares}} = \frac{4k}{\sqrt{p}} g. \tag{4.49}$$

This is a factor $\sqrt{p}/2$ less than for division into strips. The resulting communication-to-computation ratio is

$$\frac{T_{\text{comm, squares}}}{T_{\text{comp, squares}}} = \frac{4k/\sqrt{p}}{5k^2/p} g = \frac{4\sqrt{p}}{5k} g. \tag{4.50}$$

The communication-to-computation ratio is often called the **surface-to-volume** ratio. This term originates in the three-dimensional case, where the volume of a domain represents the amount of computation of a processor and the surface represents the communication with other processors.

Not only are square blocks better with respect to communication, they are also better with respect to computation: in case of load imbalance, the surplus of the busiest processor is at most $2\lceil k/\sqrt{p}\rceil - 1$ grid points, instead of k.

It may seem that the best we can do is to distribute the grid by square blocks. This intuitive belief may even be stronger if you happen to use a square processor network and are used, in the old ways, to exploiting network proximity to optimize communication. In the BSP model, however, there is no particular advantage in using such regular schemes. Therefore, we can freely try other shapes for the area allocated to one processor. Consider what the computer scientist would call the **digital diamond**, and the mathematician the **closed l_1-sphere**, defined by

$$B_r(c_0, c_1) = \{(i,j) \in \mathbf{Z}^2 : \ |i - c_0| + |j - c_1| \le r\}, \tag{4.51}$$

for integer radius $r \ge 0$ and centre $\mathbf{c} = (c_0, c_1) \in \mathbf{Z}^2$. This is the set of points with Manhattan distance at most r to the central point $\mathbf{c}$, see Fig. 4.17. The number of points of $B_r(\mathbf{c})$ is $1+3+5+\cdots+(2r-1)+(2r+1)+(2r-1)+\cdots+1 = 2r^2+2r+1$. The number of neighbouring points is $4r+4$. If $B_r(\mathbf{c})$ represents a

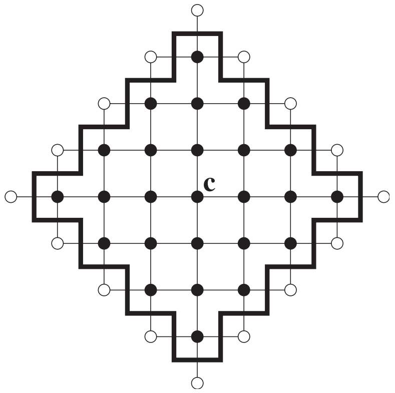


FIG. 4.17. Digital diamond of radius $r = 3$ centred at $\mathbf{c}$. Points inside the diamond are shown in black; neighbouring points are shown in white.

set of grid points allocated to one processor, then the fanout involves receiving $4r+4$ values. Just on the basis of receives, we may conclude that this processor has a communication-to-computation ratio

$$\frac{T_{\text{comm, diamonds}}}{T_{\text{comp, diamonds}}} = \frac{4r+4}{5(2r^2+2r+1)}g \approx \frac{2}{5r}g \approx \frac{2\sqrt{2p}}{5k}g, \tag{4.52}$$

for large enough r, where we use the approximation $r \approx k/\sqrt{2p}$, obtained by assuming that the processor has its fair share $2r^2+2r+1 = k^2/p$ of the grid points. The resulting asymptotic ratio is a factor $\sqrt{2}$ lower than for square blocks, cf. eqn (4.50). This reduction is caused by using each received value twice. Diamonds are a parallel computing scientist's best friend.

The gain of using diamonds can only be realized if the outgoing traffic is balanced with the incoming traffic, that is, if the number of sends $h_{\rm s}(s)$ of processor $P(s)$ is the same as the number of receives $h_{\rm r}(s) = 4r+4$. The number of sends of a processor depends on which processors own the neighbouring points. Each of the $4r$ border points of a diamond has to be sent to at least one processor and at most two processors, except corner points, which may have to be sent to three processors. Therefore, $4r \leq h_{\rm s}(s) \leq 8r+4$.

To find a distribution that balances the sends and the receives, we try to fit the diamonds in a regular pattern. Consider first the infinite lattice $\mathbf{Z}^2$; to make mathematicians cringe we view it as a $k \times k$ grid with $k = \infty$; to make matters worse we let the grid start at $(-\infty, -\infty)$. We try to partition this $\infty \times \infty$ grid over an infinite number of processors using diamonds. It turns out that we can do this by placing the diamonds in a periodic fashion at centres $\mathbf{c} = \lambda\mathbf{a} + \mu\mathbf{b}$, $\lambda, \mu \in \mathbf{Z}$, where $\mathbf{a} = (r, r+1)$ and $\mathbf{b} = (-r-1, r)$. Part of this infinite partitioning is shown in Fig. 4.18. The centres of the diamonds form a lattice defined by the orthogonal basis vectors $\mathbf{a}, \mathbf{b}$. We leave it to the mathematically inclined reader to verify that the diamonds $B_r(\lambda\mathbf{a} + \mu\mathbf{b})$ are indeed mutually disjoint and that they fill the whole domain. It is easy to see that each processor sends $h_{\rm s}(s) = 4r+4$ values. This distribution of an infinite grid over an infinite number of processors achieves the favourable ratio of eqn (4.52).

Practical computational grids and processor sets are, alas, finite and therefore the region of interest must be covered using a finite number of diamonds. Sometimes, the shape of the region allows us to use diamonds directly without too much waste. In other situations, many points from the covering diamonds fall outside the region of interest. These points are then discarded and the remaining pieces of diamonds are combined, assigning several pieces to one processor, such that each processor obtains about the same number of grid points.

Figure 4.18 shows a 10×10 grid partitioned into one complete and seven incomplete diamonds by using the infinite diamond partitioning. This can

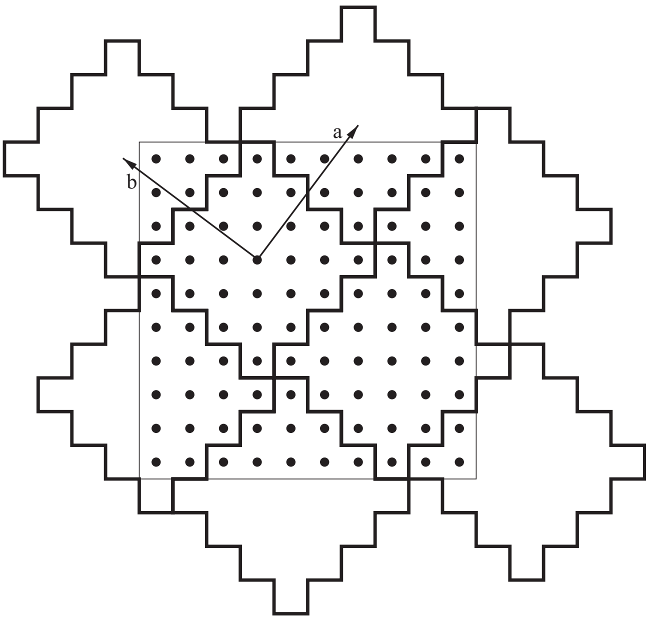


FIG. 4.18. Distribution of a 10×10 grid by digital diamonds of radius $r = 3$. Each complete diamond has 25 grid points. Pieces of incomplete diamonds can be combined, assigning them to the same processor.

be transformed into a partitioning over four processors by assigning the two incomplete diamonds near the lower-left corner of the picture to $P(0)$; the two incomplete diamonds near the lower-right corner to $P(1)$; the complete diamond to $P(2)$; and the remaining pieces to $P(3)$. This assigns 24, 27, 25, 24 grid points to the respective processors. The processor with the complete diamond has to communicate most: $h_{\rm s}(2) = 15$ and $h_{\rm r}(2) = 14$. Thus the communication cost is close to the maximum $4r + 4 = 16$ for diamonds of radius $r = 3$. It is possible to improve this assignment, for instance by border corrections. (What is the best assignment you can find?) In general, to find an optimal assignment of pieces to processors is a hard optimization problem, and perhaps the best approach is to use a heuristic, for instance assigning pieces to processors in order of decreasing size. In a few cases, we are lucky and we can fit all the pieces together creating complete diamonds. An example is the 25×25 grid, which can be covered exactly with 25 diamonds of radius $r = 3$. (For the curious, a picture can be found in [22].)

The main disadvantage of using diamonds is that it is much more complicated than using blocks. To make the partitioning method based on diamonds practical, it must be simplified. Fortunately, this can be done by a small modification, discarding one layer of points from the north-eastern and

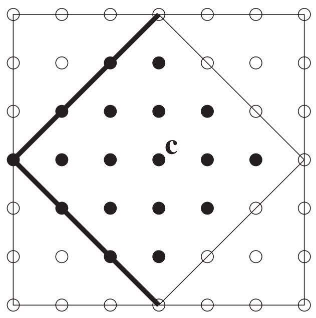


FIG. 4.19. Basic cell of radius $r = 3$ assigned to a processor. The cell has 18 grid points, shown in black. Grid points outside the cell are shown in white. Grid points on the thick lines are included; those on the thin lines are excluded. The cell contains 13 grid points that are closer to the centre than to a corner of the enclosing square and it contains five points at equal distance. The cell has 14 neighbouring grid points.

south-eastern border of the diamond, as shown in Fig. 4.19. For $r = 3$, the number of points decreases from 25 (see Fig. 4.17) to 18. The resulting set of points can be seen as the set of points closer to the centre than to the corner of the enclosing large square, and such a set is called a **Voronoi cell**. Ties are broken by assigning border points to the interior for the western borders, and to the exterior for the eastern borders. Only one corner is assigned to the interior. This way, we obtain a basic cell that can be repeated and used to fill the whole space. We can place the diamonds in a periodic fashion at centres $\mathbf{c} = \lambda\mathbf{a} + \mu\mathbf{b}$, $\lambda, \mu \in \mathbf{Z}$, where $\mathbf{a} = (r, r)$ and $\mathbf{b} = (-r, r)$.

Figure 4.20 shows how the basic cell is used to distribute a 12×12 grid over eight processors. Five cells are complete (black, green, pink, gold, and brown); the dark blue processor has points near the left and right border of the grid; the light blue processor has points near the top and bottom; and the red processor has points near the four corners. The pieces have been combined by treating the grid boundaries as periodic. Each processor has 18 grid points. The communication volume is 104 and $h_{\rm s} = h_{\rm r} = 14$; the BSP cost of the computation is $90 + 14g + 2l$.

A different way of partitioning a $k \times k$ grid is to translate it into the corresponding $k^2 \times k^2$ matrix and vectors of length k^2, then let the Mondriaan package find a good data distribution for the corresponding sparse matrix–vector multiplication, and translate this back into a grid distribution. As before, we impose $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ for the vectors and use a row distribution for the matrix. Figure 4.21 shows the result for an allowed load imbalance $\epsilon = 10\%$. Note that Mondriaan (the package!) prefers stepwise borders, similar to the borders of a digital diamond. The communication volume is $V = 85$, which is lower than for the regular distribution of Fig. 4.20. The reason is

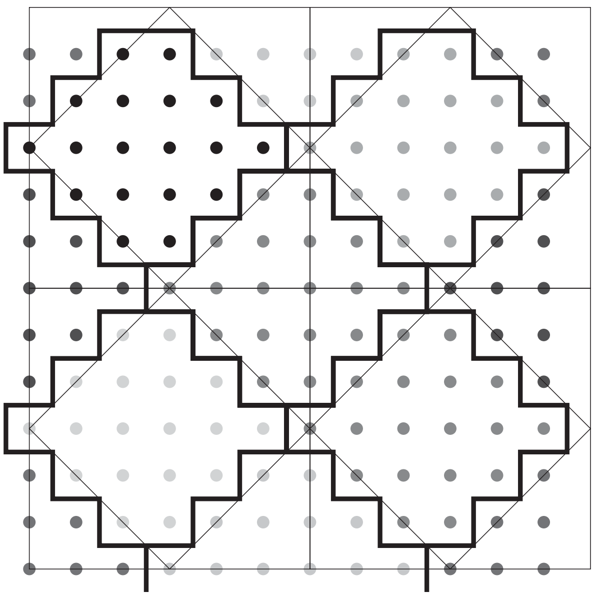


FIG. 4.20. Distribution of a 12×12 grid over eight processors obtained by using the basic cell from Fig. 4.19 (see also Plate 4).

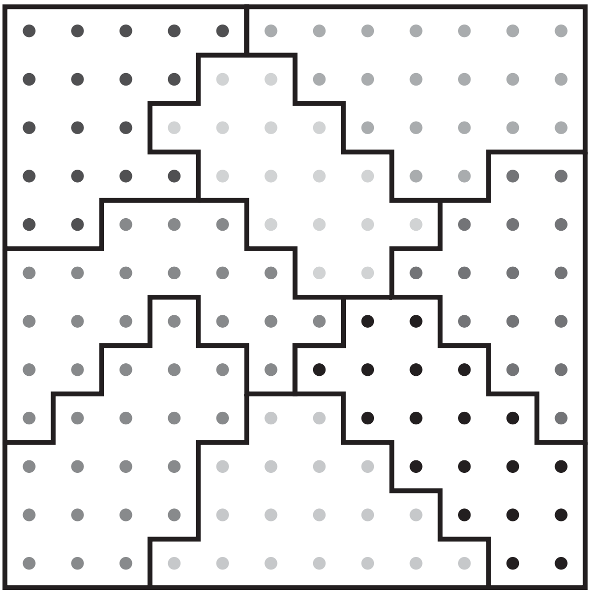


FIG. 4.21. Distribution of a 12×12 grid over eight processors produced by the Mondriaan package (see also Plate 5).

that Mondriaan manages to make each processor subdomain connected, thus reducing the communication compared with the regular case where disconnected subdomains occur. The achieved load imbalance in the corresponding matrix–vector multiplication is $\epsilon' = 8.3\%$. The maximum number of vector components per processor is 20; the average is 18. We had to allow such a load imbalance, because otherwise Mondriaan could not succeed in keeping the communication low. (For $\epsilon = 3\%$, the result is perfect load balance, $\epsilon' = 0$, but disconnected subdomains and a high volume, $V = 349$.) The BSP cost of the computation is $91 + 16g + 2l$, where we have taken into account that points on the grid boundary have fewer flops associated with them. Running Mondriaan takes much more time than performing one Laplacian operation on the grid, but the effort of finding a good distribution has to be spent only once, and the cost can be amortized over many parallel computations on the same grid.

The crucial property of our matrix–vector multiplication algorithm applied to grid computation is that a value needed by another processor is sent only once, even if it is used several times. Our cost model reflects this, thus encouraging reuse of communicated data. If a partitioning method based on this cost model is applied to a regular grid, then the result will be diamond-like shapes of the subdomain, as shown in Fig. 4.21.

The prime application of diamond-shaped partitioning will most likely be in three dimensions, where the number of grid points at the boundary of a subdomain is relatively large compared to the number of interior points. If a processor has a cubic block of $N = k^3/p$ points, about $6k^2/p^{2/3} = 6N^{2/3}$ are boundary points; in the two-dimensional case this is only $4N^{1/2}$. For example, if a processor has a $10 \times 10 \times 10$ block, 488 points are on the processor boundary. In three dimensions, communication is important. Based on the surface-to-volume ratio of a three-dimensional digital diamond, we can expect a reduction by a factor $\sqrt{3} \approx 1.73$ in communication cost, which is certainly worthwhile. The reduction achieved in reality depends on whether we manage to fill three-dimensional space with shapes that closely resemble digital diamonds.

The basic cell that suits our purpose is a truncated octahedron, shown in Fig. 4.22. The surface parts have been carefully assigned to the interior or exterior, so that the whole space can be filled with nonoverlapping copies of the cell, that is, with no point in space belonging to more than one cell. This has been achieved by a fair assignment of the faces, edges, and vertices of the cell. For instance, the front square is included in the cell but the back square is not. A cell of radius r is enclosed by a cube with edge length $2r$. We can fill space with such cubes, and place copies of the basic cell at the corners and centres of these cubes. As a result, cells are centred at points $(\lambda r, \mu r, \nu r)$, with λ, μ, ν three even integers, or three odd integers. This set of centre points is called the **body-centred cubic** (BCC) lattice. Each centre point represents a processor. As in the two-dimensional case, the basic cell is a Voronoi cell,

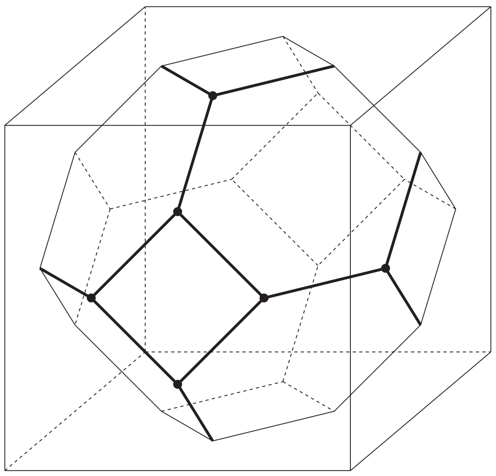


FIG. 4.22. Basic three-dimensional cell assigned to a processor. The cell is defined as the set of grid points that fall within a truncated octahedron. The boundaries of this truncated octahedron are included/excluded as follows. Included are the four hexagons and three squares visible at the front (which are enclosed by solid lines), the twelve edges shown as thick solid lines, and the six vertices marked in black. The other faces, edges, and vertices are excluded. The enclosing cube is shown for reference only. Neighbouring cells are centred at the eight corners of the cube and the six centres of neighbouring cubes.

since grid points of the cell are closer to its centre than to the centres of other cells (with fair tie breaking).

The number of grid points of the basic cell is $4r^3$. A careful count shows that the number of points on the surface is $9r^2 + 2$ and that the number of sends and receives is $9r^2+6r+2$. The resulting communication-to-computation ratio is

$$\frac{T_{\text{comm, truncated octahedron}}}{T_{\text{comp, truncated octahedron}}} = \frac{9r^2+6r+2}{7\cdot 4r^3}g \approx \frac{9}{28r}g \approx \frac{9(4p)^{1/3}}{28k}g, \qquad (4.53)$$

which is better by a factor of 1.68 than the ratio $6p^{1/3}/(7k)$ for blocks. In an actual implementation, it may be most convenient to use a cubic array as local data structure, with an extra layer of points along each border, and to fill this array only partially. This enables the use of a regular array structure while still reducing the communication.

Table 4.3 shows the communication cost for various distributions of a two-dimensional grid with $k = 1024$ and a three-dimensional grid with $k = 128$. For the two-dimensional Laplacian, the ideal case for the rectangular block distribution occurs for $p = q^2$, that is, for $p = 4, 16, 64$, since the local subdomains then become square blocks. For $p = 2q^2$, that is, for $p = 2, 8, 32, 128$, the blocks become rectangles with an aspect ratio $2 : 1$. In contrast, the ideal case for the diamond distribution is $p = 2q^2$. To handle the nonideal case $p = q^2$ as well, the diamond distribution is generalized by stretching the basic cell in one direction, giving a communication cost of $4kg/\sqrt{p}$. The table shows that the diamond distribution is better than the rectangular block distribution for $p = 2q^2$ and performs the same for $p = q^2$, except in the special case of small p, where the boundaries of the grid play a prominent role. For the three-dimensional Laplacian, only the ideal case for the diamond distribution, $p = 2q^3$, is shown. (The other cases $p = q^3$ and $p = 4q^3$ are more difficult to treat, requiring a generalization of the diamond distribution based on stretching the basic cell. An application developer having 32 or 64 processors at his disposal might be motivated to implement this kind of generalization.) We observe a reduction by a factor of 1.71 for $p = 16$ compared with the block distribution. Asymptotically, for large radius r, the reduction factor is $16/9 \approx 1.78$ in the case $p = 2q^3$.

For comparison, Table 4.3 also presents results obtained by using the Mondriaan package (version 1.0) to produce a row distribution of the Laplacian matrix and a corresponding distribution of the grid. For the Mondriaan distribution, the allowed load imbalance for the corresponding matrix is $\epsilon = 10\%$. The communication cost given is the average over 100 runs of the Mondriaan program, each time with a different seed of the random number generator used. In three dimensions, the Mondriaan distribution is better than blocks and for large local subdomains (such as for $p = 16$) it comes close to the performance of diamonds.

TABLE 4.3. Communication cost (in g) for a Laplacian operation on a grid, using distributions based on rectangular blocks and diamond cells, and a distribution produced by the Mondriaan package

Grid	p	Rectangular	Diamond	Mondriaan
1024×1024	2	1024	2046	1024
	4	1024	2048	1240
	8	1280	1026	1378
	16	1024	1024	1044
	32	768	514	766
	64	512	512	548
	128	384	258	395
$64 \times 64 \times 64$	16	4096	2402	2836
	128	1024	626	829

4.9 Remainder of BSPlib: example function `bspmv`

The function `bspmv` is an implementation of Algorithm 4.5 for sparse matrix–vector multiplication. It can handle every possible data distribution for the matrix and vectors. Before executing the algorithm, each processor builds its own local data structure for representing the local part of the sparse matrix. The local nonempty rows are numbered $\texttt{i} = 0, \ldots, \texttt{nrows} - 1$, where $\texttt{nrows} = |I_s|$. The global index of the row with local index `i` is given by $i = \texttt{rowindex}[\texttt{i}]$. Similarly, the global index of the column with local index `j` is given by $j = \texttt{colindex}[\texttt{j}]$, for $0 \leq \texttt{j} < \texttt{ncols}$. (The local indices `i` and `j` of the matrix data structure are distinct from those of the vectors.) The nonzeros are stored in order of increasing local row index `i`. The nonzeros of each local row are stored consecutively in increasing order of local column index `j`, using the ICRS data structure. The `k`th nonzero is stored as a pair $(\texttt{a}[\texttt{k}], \texttt{inc}[\texttt{k}])$, where $\texttt{a}[\texttt{k}]$ is the numerical value of the nonzero and $\texttt{inc}[\texttt{k}]$ the increment in the local column index.

This data structure is convenient for use in repeated sparse matrix–vector multiplication. Building the structure, however, requires quite a bit of preprocessing on input. An outline of the input preprocessing is as follows. Each triple (i, j, a_{ij}) is read from an input file and sent to the responsible processor, as determined by the matrix distribution of the file. The local triples are then sorted by increasing global column index, which enables conversion to local column indices. During the conversion, the global indices are registered in `colindex`. The triples are sorted again, this time by global row index, taking care that the original mutual precedences are maintained between triples from the same matrix row. The global row indices are then converted to local ones and the array `rowindex` is initialized.

The nonzeros must be sorted with care. Sequentially, the $nz(A)$ nonzeros of a sparse matrix A can be sorted by row in time $\mathcal{O}(nz(A)+n)$, simply by counting the number of nonzeros in each row during one pass through the nonzeros, allocating exactly the right amount of memory space for each row, and filling the space in a second pass. In parallel, it is more difficult to sort efficiently, because the range of possible global indices remains $0, \ldots, n-1$, while the number of local nonzeros decreases to $nz(A)/p$. Clearly, such $\mathcal{O}(nz(A)/p+n)$ behaviour for a straightforward sort by row index is nonscalable and hence unacceptable. Fortunately, a radix sort (see [46]) with radix $r = \sqrt{n}$ will do the job. This method first sorts the triples by using $i \bmod r$ as a key, and then in a second pass sorts the indices by using $i \operatorname{div} r$ as a key, maintaining the original mutual precedences between triples that have the same key for the second pass. The total time and memory needed is about $\mathcal{O}(nz(A)/p + n/r + r)$, which is minimal for the choice $r = \sqrt{n}$. We choose the radix to be a power of two close to $\sqrt{n}$, because of the cheaper modular arithmetic for powers of two. This sorting procedure is scalable in time and memory by our definition in Section 3.5, because $\sqrt{n} = \mathcal{O}(n/p+p)$. The driver program `bspmv_test` (not printed here because of its length, but included in BSPedupack) implements the complete input phase, for matrix as well as vectors.

The relation between the vector variables of Algorithm 4.5 and the functions `bspmv` and `bspmv_init` is as follows. Vector component v_j corresponds to a local component `v[k]` in $P(\phi_{\mathbf{v}}(j))$, where $j =$ `vindex[k]`. All the needed vector components v_j, whether obtained from other processors or already present locally, are written into a local array `vloc`, which has the same local indices as the matrix columns; `vloc[j]` stores a copy of v_j, where $j =$ `colindex[j]`. This copy is obtained using a `bsp_get`, because the receiver knows it needs the value. The processor from which to get the value v_j has processor number $\phi_{\mathbf{v}}(j) =$ `srcprocv[j]` and this number is stored beforehand by the initialization function `bspmv_init`. This way, the source processor needs to be determined only once and its processor number can be used without additional cost in repeated application of the matrix–vector multiplication. We also need to determine the location in the source processor where v_j resides. This location is stored as the local index `srcindv[j]`.

The partial sum $u_{it} =$ `sum` is computed by pointer magic, to be explained later on, and this sum is immediately sent to the processor $P(\phi_{\mathbf{u}}(i))$ that computes u_i. A convenient way of sending this value is by using the `bsp_send` primitive, which is the core primitive of **bulk synchronous message passing**, a new style of communication introduced in this section. The function `bspmv` is the first occasion where we see the five important bulk synchronous message passing primitives of BSPlib in action. (An additional high-performance primitive is presented in Exercise 10.) The `bsp_send` primitive allows us to send data to a given processor without specifying the location where the data is to be stored. One can view `bsp_send` as a `bsp_put` with a wildcard for the destination address. In all other aspects, `bsp_send` acts like `bsp_put`; in particular, it performs a one-sided communication since it does not require any activity by the receiver in the same superstep. (In the next superstep, however, the receiver must do something if it wants to use the received data, see below.) The `bsp_send` primitive is quite unlike traditional message passing primitives, which require coordinated action of a sender and a receiver.

The reason for the existence of `bsp_send` is nicely illustrated by `Superstep 2` of `bspmv`, which employs `bsp_send` to send a nonzero partial sum u_{it} to processor $P(\phi_{\mathbf{u}}(i))$. The information whether a nonzero partial sum for a certain row exists is only available at the sender. As a consequence, a sending processor does not know what the other processors send. Furthermore, processors do not know what they will receive. If we were to use `bsp_put` statements, we would have to specify a destination address. One method of doing this is by having each receiving processor reserve memory space to store p partial sums u_{it} for each of its vector components u_i. If this is done, the processor that computes a partial sum u_{it} can write it directly into the memory cell reserved for it on $P(\phi_{\mathbf{u}}(i))$. Unfortunately, the amount of reserved local memory, of the order $p \cdot n/p = n$ cells, is p times larger than the memory needed for the vectors and a large part of this memory may never be used for writing nonzero partial sums. Furthermore, this method also requires $\mathcal{O}(n)$ computing time, since all memory cells must be inspected. Thus, this method is nonscalable both in time and memory. An alternative is a rather clumsy method that may be termed the 'three-superstep' approach. In the first superstep, each processor tells each of the other processors how many partial sums it is going to send. In the second superstep, each receiving processor reserves exactly the required amount of space for each of the senders, and tells them the address from which they can start writing. Finally, in the third superstep, the partial sums are put as pairs (i, u_{it}). Fortunately, we can organize the communication in a more efficient and more elegant way by using the `bsp_send` primitive instead of `bsp_put`. This is done in the function `bspmv`. Any one writing programs with irregular communication patterns will be grateful for the existence of `bsp_send`!

The `bsp_send` primitive sends a message which consists of a **tag** and a **payload**. The tag is used to identify the message; the payload contains the actual data. The use of the `bsp_send` primitive is illustrated by the top part of Fig. 4.23. In our case, the tag is an index corresponding to i and the payload is the partial sum u_{it}. The syntax is

```
bsp_send(pid, tag, source, nbytes);
```

Here, `int pid` is the identity of the destination processor; `void *tag` is a pointer to the tag; `void *source` is a pointer to the source memory from

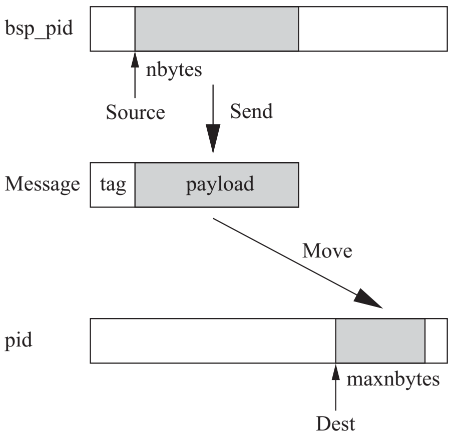


FIG. 4.23. Send operation from BSPlib. The `bsp_send` operation copies `nbytes` of data from the local processor `bsp_pid` into a message, adds a tag, and sends this message to the specified destination processor `pid`. Here, the pointer `source` points to the start of the data to be copied. In the next superstep, the `bsp_move` operation writes at most `maxnbytes` from the message into the memory area specified by the pointer `dest`.

which the data to be sent are read; `int nbytes` is the number of bytes to be sent. In our case, the number of the destination processor is available as $\phi_{\mathtt{u}}(i) =$ `destprocu[i]`, which has been initialized beforehand by the function `bspmv_init`. It is important to choose a tag that enables the receiver to handle the payload easily. Here, the receiver needs to know to which vector component the partial sum belongs. We could have used the global index i as a tag, but then this index would have to be translated on receipt into the local index `i` used to access `u`. Instead, we use the local index directly. Note that in this case the tag need not identify the source processor, since its number is irrelevant.

The message to be sent using the `bsp_send` primitive is first stored by the system in a local send buffer. (This implies that the tag and source variable can be reused immediately.) The message is then sent and stored in a buffer on the receiving processor. The send and receive buffers are invisible to the user (but there is a way of emptying the receive buffer, as you may guess).

Some time after the message has been sent, it becomes available on the receiving processor. In line with the philosophy of the BSP model, this happens at the end of the current superstep. In the next superstep, the messages can be read; reading messages means moving them from the receive buffer into the desired destination memory. At the end of the next superstep all remaining unmoved messages will be lost. This is to save buffer memory and to force the user into the right habit of cleaning his desk at the end of the day. (As said before, the BSP model and its implementation BSPlib are quite paternalistic. They often force you to do the right thing, for lack of alternatives.) The syntax of the move primitive is

```
bsp_move(dest, maxnbytes);
```

Here, `void *dest` is a pointer to the destination memory where the data are written; `int maxnbytes` is an upper bound on the number of bytes of the payload that is to be written. This is useful if only part of the payload needs to be retrieved. The use of the `bsp_move` primitive is illustrated by the bottom part of Fig. 4.23.

In our case, the payload of a message is one double, which is written in its entirety into `sum`, so that `maxnbytes` = `SZDBL`.

The header information of a message consists of the tag and the length of the payload. This information can be retrieved by the statement

```
bsp_get_tag(status, tag);
```

Here, `int *status` is a pointer to the status, which equals -1 if the buffer is empty; otherwise, it equals the length of the payload in bytes. Furthermore, `void *tag` is a pointer to the memory where the tag is written. The status information can be used to decide whether there is an unread message, and if so, how much space to allocate for it. In our case, we know that each payload has the same fixed length `SZDBL`.

We could have used the status in the termination criterion of the loop in `Superstep 3`, to determine whether we have handled all partial sums. Instead, we choose to use the enquiry primitive

```
bsp_qsize(nmessages, nbytes);
```

Here, `int *nmessages` is a pointer to the total number of messages received in the preceding superstep, and `int *nbytes` is a pointer to the total number of bytes received. In our program, we only use `bsp_qsize` to determine the number of iterations of the loop, that is, the number of partial sums received. In general, the `bsp_qsize` primitive is useful for allocating the right amount of memory for storing the received messages. Here, we do not need to allocate memory, because we process and discard the messages immediately after we read them. The name `bsp_qsize` derives from the fact that we can view the receive buffer as a queue: messages wait patiently in line until they are processed.

In our program, the tag is an integer, but in general it can be of any type. The size of the tag in bytes is set by

```
bsp_set_tagsize(tagsz);
```

On input, `int *tagsz` points to the desired tag size. As a result, the system uses the desired tag size for all messages to be sent by `bsp_send`. The function `bsp_set_tagsize` takes effect at the start of the next superstep. All processors must call the function with the same tag size. As a side effect, the contents of `tagsz` will be modified, so that on output it contains the previous tag size of the system. This is a way of preserving the old value, which can be useful if an initial global state of the system must be restored later.

In one superstep, an arbitrary number of communication operations can be performed, using either `bsp_put`, `bsp_get`, or `bsp_send` primitives, and they can be mixed freely. The only practical limitation is imposed by the amount of buffer memory available. The BSP model and BSPlib do not favour any particular type of communication, so that it is up to the user to choose the most convenient primitive in a given situation.

The local matrix–vector multiplication in `Superstep 2` is an implementation of Algorithm 4.4 for the local data structure, modified to handle a rectangular `nrows` $\times$ `ncols` matrix. The inner loop of the multiplication has been optimized by using pointer arithmetic. For once deviating from our declared principles, we sacrifice readability here because this loop is expected to account for a large proportion of the computing time spent, and because pointer arithmetic is the *raison d'être* of the ICRS data structure. The statement

```
*psum += (*pa) * (*pvloc);
```

is a translation of

```
sum += a[k] * vloc[j];
```

We move through the array `a` by incrementing `pa` (i.e. the pointer to `a`) and do the same for the `inc` array. Instead of using an index `j` to access `vloc`, we use a pointer `pvloc`; after a nonzero has been processed, this pointer is moved $*$`pinc` $=$ `inc`[`k`] places forward.

The initialization function `bspmv_init` first reveals the owner of each global index i, storing its number in a temporary array `tmpproc` that can be queried by all processors. As a result, every processor can find answers to questions such as: who is the owner of u_i and where is this component parked? For scalability, the temporary array is itself distributed, and this is done by the cyclic distribution. In addition, the local index on the owning processor is stored in an array `tmpind`. The temporary arrays are then queried to initialize the local lists used for steering the communications. For example, vector component v_j with $j =$ `jglob` $=$ `colindex`[`j`], which is needed for local matrix column `j`, can be obtained from the processor whose number is stored in array `tmpprocv` on processor $P(j \bmod p)$, in location j div p. A suitable mental picture is that of a collection of notice boards: every processor first announces the availability of its vector components on the appropriate notice boards and then reads the announcements that concern the components it needs. We finish `bspmv_init` by deregistering and freeing memory. Note that we deregister memory in `Superstep 3` but deallocate it only in `Superstep 4`. This is because deregistration takes effect only at the end of `Superstep 3`; allocated memory must still exist at the time of actual deregistration.

The main purpose of the program `bspmv` is to explain the bulk synchronous message passing primitives. It is possible to optimize the program further, by performing a more extensive initialization, so that all data for the same destination can be sent together in one block. This can even be done using puts! (Such optimization is the subject of Exercise 10.)

The program text is:

```
#include "bspedupack.h"

void bspmv(int p, int s, int n, int nz, int nrows, int ncols,
           double *a, int *inc,
           int *srcprocv, int *srcindv, int *destprocu,
           int *destindu, int nv, int nu, double *v, double *u){

    /* This function multiplies a sparse matrix A with a
       dense vector v, giving a dense vector u=Av.
       A is n by n, and u,v are vectors of length n.
       A, u, and v are distributed arbitrarily on input.
       They are all accessed using local indices, but the local
       matrix indices may differ from the local vector indices.
       The local matrix nonzeros are stored in an incremental
       compressed row storage (ICRS) data structure defined by
       nz, nrows, ncols, a, inc.
```

```
   All rows and columns in the local data structure are
   nonempty.

   p is the number of processors.
   s is the processor number, 0 <= s < p.
   n is the global size of the matrix A.
   nz is the number of local nonzeros.
   nrows is the number of local rows.
   ncols is the number of local columns.

   a[k] is the numerical value of the k'th local nonzero of
        the sparse matrix A, 0 <= k < nz.
   inc[k] is the increment in the local column index of the
          k'th local nonzero, compared to the column index
          of the (k-1)th nonzero, if this nonzero is in the
          same row; otherwise, ncols is added to the
          difference. By convention, the column index of the
          -1'th nonzero is 0.

   srcprocv[j] is the source processor of the component in v
          corresponding to the local column j, 0 <= j < ncols.
   srcindv[j] is the local index on the source processor
          of the component in v corresponding to the local
          column j.
   destprocu[i] is the destination processor of the partial sum
              corresponding to the local row i, 0 <= i < nrows.
   destindu[i] is the local index in the vector u on the
               destination processor corresponding to the
               local row i.

   nv is the number of local components of the input vector v.
   nu is the number of local components of the output vector u.
   v[k] is the k'th local component of v, 0 <= k < nv.
   u[k] is the k'th local component of u, 0 <= k < nu.
*/

int i, j, k, tagsz, status, nsums, nbytes, *pinc;
double sum, *psum, *pa, *vloc, *pvloc, *pvloc_end;

/****** Superstep 0. Initialize and register ******/
for(i=0; i<nu; i++)
    u[i]= 0.0;
vloc= vecallocd(ncols);
bsp_push_reg(v,nv*SZDBL);
tagsz= SZINT;
bsp_set_tagsize(&tagsz);
bsp_sync();

/****** Superstep 1. Fanout ******/
for(j=0; j<ncols; j++)
    bsp_get(srcprocv[j],v,srcindv[j]*SZDBL,&vloc[j],SZDBL);
bsp_sync();
```

```
    /****** Superstep 2. Local matrix--vector multiplication
        and fanin */
    psum= &sum;
    pa= a;
    pinc= inc;
    pvloc= vloc;
    pvloc_end= pvloc + ncols;

    pvloc += *pinc;
    for(i=0; i<nrows; i++){
        *psum= 0.0;
        while (pvloc<pvloc_end){
            *psum += (*pa) * (*pvloc);
            pa++;
            pinc++;
            pvloc += *pinc;
        }
        bsp_send(destprocu[i],&destindu[i],psum,SZDBL);
        pvloc -= ncols;
    }
    bsp_sync();

    /****** Superstep 3. Summation of nonzero partial sums ******/
    bsp_qsize(&nsums,&nbytes);
    bsp_get_tag(&status,&i);
    for(k=0; k<nsums; k++){
        /* status != -1, but its value is not used */
        bsp_move(&sum,SZDBL);
        u[i] += sum;
        bsp_get_tag(&status,&i);
    }

    bsp_pop_reg(v);
    vecfreed(vloc);

} /* end bspmv */

int nloc(int p, int s, int n){
    /* Compute number of local components of processor s for vector
       of length n distributed cyclically over p processors. */

    return  (n+p-s-1)/p ;

} /* end nloc */

void bspmv_init(int p, int s, int n, int nrows, int ncols,
                int nv, int nu, int *rowindex, int *colindex,
                int *vindex, int *uindex, int *srcprocv,
                int *srcindv, int *destprocu, int *destindu){
```

```
/* This function initializes the communication data structure
   needed for multiplying a sparse matrix A with a dense
   vector v, giving a dense vector u=Av.

   Input: the arrays rowindex, colindex, vindex, uindex,
   containing the global indices corresponding to the local
   indices of the matrix and the vectors.
   Output: initialized arrays srcprocv, srcindv, destprocu,
   destindu containing the processor number and the local
   index on the remote processor of vector components
   corresponding to local matrix columns and rows.

   p, s, n, nrows, ncols, nv, nu are the same as
   in bspmv.

   rowindex[i] is the global index of the local row
   i, 0 <= i < nrows.
   colindex[j] is the global index of the local column
   j, 0 <= j < ncols.
   vindex[j] is the global index of the local v-component
   j, 0 <= j < nv.
   uindex[i] is the global index of the local u-component
   i, 0 <= i < nu.

   srcprocv, srcindv, destprocu, destindu are the same as in bspmv.
*/

int nloc(int p, int s, int n);
int np, i, j, iglob, jglob, *tmpprocv, *tmpindv, *tmpprocu,
    *tmpindu;

/****** Superstep 0. Allocate and register temporary arrays */
np= nloc(p,s,n);
tmpprocv=vecalloci(np); bsp_push_reg(tmpprocv,np*SZINT);
tmpindv=vecalloci(np);  bsp_push_reg(tmpindv,np*SZINT);
tmpprocu=vecalloci(np); bsp_push_reg(tmpprocu,np*SZINT);
tmpindu=vecalloci(np);  bsp_push_reg(tmpindu,np*SZINT);
bsp_sync();

/****** Superstep 1. Write into temporary arrays ******/
for(j=0; j<nv; j++){
    jglob= vindex[j];
    /* Use the cyclic distribution */
    bsp_put(jglob%p,&s,tmpprocv,(jglob/p)*SZINT,SZINT);
    bsp_put(jglob%p,&j,tmpindv, (jglob/p)*SZINT,SZINT);
}
for(i=0; i<nu; i++){
    iglob= uindex[i];
    bsp_put(iglob%p,&s,tmpprocu,(iglob/p)*SZINT,SZINT);
    bsp_put(iglob%p,&i,tmpindu, (iglob/p)*SZINT,SZINT);
}
bsp_sync();
```

```
    /****** Superstep 2. Read from temporary arrays ******/
    for(j=0; j<ncols; j++){
        jglob= colindex[j];
        bsp_get(jglob%p,tmpprocv,(jglob/p)*SZINT,&srcprocv[j],SZINT);
        bsp_get(jglob%p,tmpindv, (jglob/p)*SZINT,&srcindv[j], SZINT);
    }
    for(i=0; i<nrows; i++){
        iglob= rowindex[i];
        bsp_get(iglob%p,tmpprocu,(iglob/p)*SZINT,&destprocu[i],SZINT);
        bsp_get(iglob%p,tmpindu, (iglob/p)*SZINT,&destindu[i], SZINT);
    }
    bsp_sync();

    /****** Superstep 3. Deregister temporary arrays ******/
    bsp_pop_reg(tmpindu); bsp_pop_reg(tmpprocu);
    bsp_pop_reg(tmpindv); bsp_pop_reg(tmpprocv);
    bsp_sync();

    /****** Superstep 4. Free temporary arrays ******/
    vecfreei(tmpindu); vecfreei(tmpprocu);
    vecfreei(tmpindv); vecfreei(tmpprocv);

} /* end bspmv_init */
```

4.10 Experimental results on a Beowulf cluster

Nowadays, a Beowulf cluster is a powerful and relatively cheap alternative to the traditional supercomputer. A Beowulf cluster consists of several PCs connected by communication switches, see Fig. 1.11. In this section, we use a cluster of 32 IBM x330 nodes, located at the Physics Department of Utrecht University and part of DAS-2, the 200-node Distributed ASCI Supercomputer built by five collaborating Dutch universities for research into parallel and Grid computing. (A cluster of the DAS-2 was the machine Romein and Bal [158] used to solve the African game of Awari.) Each node of the cluster contains two Pentium-III processors with 1 GHz clock speed, 1 Gbyte of memory, a local disk, and interfaces to Fast Ethernet and Myrinet. The nodes within the cluster are connected by a Myrinet-2000 communication network. The operating system is Linux and the compiler used in our experiments is version 3.2.2 of the GNU C compiler.

The 32-node, 64-processor Beowulf cluster has been turned into a BSP computer with $p = 64$ by using a preliminary version of the BSPlib implementation by Takken [173], the Panda BSP library, which runs on top of the Panda portability layer [159], version 4.0. For $p \leq 32$, one processor per node is used; for $p = 64$, two processors per node. The BSP parameters of the cluster measured by using `bspbench` are given in Table 4.4. Note that g remains more or less constant as a function of p, and that l grows linearly. The values of g and l are about ten times higher than the values for the Origin

TABLE 4.4. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Myrinet-based Beowulf cluster running the Panda BSP library. All times are in flop units ($r = 323$ Mflop/s)

p	g	l	$T_{\mathrm{comm}}(0)$
1	1337	7188	6767
2	1400	100 743	102 932
4	1401	226 131	255 307
8	1190	440 742	462 828
16	1106	835 196	833 095
32	1711	1 350 775	1 463 009
64	2485	2 410 096	2 730 173

TABLE 4.5. Test set of sparse matrices

Matrix	n	nz	Origin
`random1k`	1000	9779	Random sparse matrix
`random20k`	20 000	99 601	Random sparse matrix
`amorph20k`	20 000	100 000	Amorphous silicon [169]
`prime20k`	20 000	382 354	Prime number matrix
`lhr34`	35 152	764 014	Light hydrocarbon recovery [192]
`nasasrb`	54 870	1 366 097	Shuttle rocket booster
`bcsstk32`	44 609	2 014 701	Automobile chassis
`cage12`	130 228	2 032 536	DNA electrophoresis [186]

3800 supercomputer given in Table 4.3, and the computing rate is about the same. The version of the Panda BSP library used has not been fully optimized yet. For instance, if a processor puts data into itself, this can be done quickly by a memory copy via a buffer, instead of letting the underlying Panda communication system discover, at much higher cost, that the destination is local. The lack of this feature is revealed by the relatively high value of g for $p = 1$. In the experiments described below, the program `bspmv` has been modified to avoid sending data from a processor to itself. (In principle, as users we should refuse to perform such low-level optimizations. The BSP system should do this for us, since the main advantage of BSP is that it enables such communication optimizations.)

Table 4.5 shows the set of sparse matrices used in our experiments. The set consists of: `random1k` and `random20k`, which represent the random sparse matrices discussed in Section 4.7; `amorph20k`, which was created by converting a model of 20 000 silicon atoms, each having four connections with other atoms, to a sparse matrix, see Fig. 4.2 and Exercise 7; `prime20k`, which extends the matrix `prime60` from the cover of this book to size $n = 20\,000$;

`lhr34`, which represents the simulation of a chemical process; `nasasrb` and `bcsstk32`, which represent three-dimensional structural engineering problems from NASA and Boeing, respectively; and `cage12`, one of the larger matrices from the DNA electrophoresis series, see Fig. 4.1. The original matrix `nasasrb` is symmetric, but we take only the lower triangular part as our test matrix. The matrices have been ordered by increasing number of nonzeros. The largest four matrices can be obtained from the University of Florida collection [53]. The size of the matrices in the test set is considered medium by current standards (but `bcsstk32` was the largest matrix in the original Harwell–Boeing collection [64]).

The matrices of the test set and the corresponding input and output vectors were partitioned by the Mondriaan package (version 1.0) for the purpose of parallel sparse matrix–vector multiplication. The resulting BSP costs are given in Table 4.6. Note that every nonzero is counted as two flops, so that the cost for $p = 1$ equals $2nz(A)$. Mondriaan was run with 3% load imbalance allowed, with input and output vectors distributed independently, and with all parameters set to their default values. The synchronization cost is not shown, since it is always $4l$, except in a few cases: for $p = 1$, the cost is l; for $p = 2$, it is $2l$; and for $p = 4$, it is $2l$ in the case of the matrices `amorph20`, `nasasrb`, and `bcsstk32`, that is, the matrices with underlying three-dimensional structure. Still, synchronization cost can be important: since $l \approx 100\,000$ for $p = 2$, matrices must have at least 100 000 nonzeros to make parallel multiplication worthwhile on this machine. Note that the matrices with three-dimensional structure have much lower communication cost than the random sparse matrices.

The time measured on our DAS-2 cluster for parallel sparse matrix–vector multiplication using the function `bspmv` from Section 4.9 is given in Table 4.7. The measured time always includes the synchronization overhead of two communication supersteps, since `bspmv` does not take advantage from a possibly empty fanin or fanout. We note that, as predicted, the matrix `random1k` is too small to observe a speedup. Instead, the total time grows with p in the same way as the synchronization cost grows. It is interesting to compare the two matrices `random20k` and `amorph20k`, which have the same size and number of nonzeros (and are too small to expect any speedup). The matrix `amorph20k` has a much lower communication cost, see Table 4.6, and this results in considerably faster execution, see Table 4.7. The largest four matrices display modest speedups, as expected for $g \approx 1000$. The matrix `nasasrb` shows a speedup on moving from $p = 1$ to $p = 2$, whereas `cage12` shows a slowdown, which agrees with the theoretical prediction.

It is quite common to finish research papers about parallel computing with a remark 'it has been shown that for large problem sizes the algorithm scales well.' If only all problems were large! More important than showing good speedups by enlarging problem sizes until the experimenter is happy, is gaining an understanding of what happens for various problem sizes, small as well as large.

TABLE 4.6. Computation and communication cost for sparse matrix–vector multiplication

p	`random1k`	`random20k`	`amorph20k`	`prime20k`
1	19 558	199 202	200 000	764 708
2	$10\,048 + 408g$	$102\,586 + 5073g$	$100\,940 + 847g$	$393\,520 + 4275g$
4	$5028 + 392g$	$51\,292 + 4663g$	$51\,490 + 862g$	$196\,908 + 5534g$
8	$2512 + 456g$	$25\,642 + 3452g$	$25\,742 + 1059g$	$98\,454 + 4030g$
16	$1256 + 227g$	$12\,820 + 2152g$	$12\,872 + 530g$	$49\,226 + 3148g$
32	$626 + 224g$	$6408 + 1478g$	$6434 + 371g$	$24\,612 + 2620g$
64	$312 + 132g$	$3202 + 1007g$	$3216 + 267g$	$12\,304 + 2235g$

p	`lhr34`	`nasasrb`	`bcsstk32`	`cage12`
1	1 528 028	2 732 194	4 029 402	4 065 072
2	$782\,408 + 157g$	$1\ 378\,616 + 147g$	$2\,070\,816 + 630g$	$2\,093\,480 + 10\,389g$
4	$388\,100 + 945g$	$703\,308 + 294g$	$1\,036\,678 + 786g$	$1\,046\,748 + 15\,923g$
8	$196\,724 + 457g$	$351\,746 + 759g$	$518\,676 + 842g$	$523\,376 + 16\,543g$
16	$98\,364 + 501g$	$175\,876 + 733g$	$259\,390 + 1163g$	$261\,684 + 9\,984g$
32	$49\,160 + 516g$	$87\,938 + 585g$	$129\,692 + 917g$	$130\,842 + 6658g$
64	$24\,588 + 470g$	$43\,966 + 531g$	$64\,836 + 724g$	$65\,420 + 5385g$

TABLE 4.7. Measured execution time (in ms) for sparse matrix–vector multiplication

p		random 1k	random 20k	amorph 20k	prime 20k	lhr34	nasasrb	bcsstk32	cage12
1	(seq)	0.2	9	7	18	30	52	71	92
1	(par)	0.3	10	8	19	31	53	72	96
2		5.1	73	13	56	26	41	59	205
4		5.2	57	16	77	22	26	39	228
8		6.9	48	15	50	14	21	25	226
16		9.1	32	11	46	13	18	24	128
32		14.8	28	17	37	18	21	23	87
64		27.4	36	29	45	29	32	34	73

Qualitatively, the BSP cost can be used to explain the timing results, or predict them. Quantitatively, the agreement is less than perfect. The reader can easily check this by substituting the measured values of g and l into the cost expressions of Table 4.6. The advantage of presenting BSP costs for sparse matrices as shown in Table 4.6 over presenting raw timings as is done in Table 4.7 is the longevity of the results: in 20 years from now, when all present supercomputers will rest in peace, when I shall be older and hopefully wiser, the results expressed as BSP costs can still be used to predict execution time on a state-of-the-art parallel computer.

4.11 Bibliographic notes

4.11.1 *Sparse matrix computations*

The book *Direct methods for sparse matrices* by Duff, Erisman, and Reid [63], is a good starting point for a study of sequential sparse matrix computations of the direct type. Direct methods such as sparse LU decomposition for unsymmetric matrices and sparse Cholesky factorization for symmetric matrices are based on the Gaussian elimination method for solving linear systems. The unsymmetric case is characterized by the interplay between numerical stability and sparsity; in the symmetric case, it is often possible to separate these concerns by performing symbolic, sparsity-related computations before the actual numerical factorization. The book has a practical flavour, discussing in detail important issues such as sparse data structures, reuse of information obtained during previous runs of a solver, and heuristics for finding good pivot elements that preserve both the numerical stability and the sparsity. The book also introduces graph-theoretic concepts commonly used in the sparse matrix field. Another book on sparse matrix computations, written by Zlatev [194], tries to combine direct and iterative methods, for instance by first performing a sparse LU decomposition, dropping small matrix elements a_{ij} with $|a_{ij}| \leq \tau$, where τ is the **drop tolerance**, and then applying an iterative method to improve the solution. Zlatev also presents a parallel implementation for a shared-memory parallel computer.

A recent book which pays much attention to sparse matrix computations is *Numerical Linear Algebra for High-Performance Computers* by Dongarra, Duff, Sorensen, and van der Vorst [61]. The book treats direct and iterative solvers for sparse linear systems and eigensystems in much detail, and it also discusses parallel aspects. One chapter is devoted to preconditioning, solving a system $K^{-1}A\mathbf{x} = K^{-1}\mathbf{b}$ instead of $A\mathbf{x} = \mathbf{b}$, where the $n \times n$ matrix K is the **preconditioner**, which must be a good approximation to A for which $K\mathbf{x} = \mathbf{b}$ is easy to solve. Finding good parallel preconditioners is an active area of research. Duff and van der Vorst [67] review recent developments in the parallel solution of linear systems by direct and iterative methods. The authors note that the two types of method cannot be clearly distinguished, since many linear solvers have elements of both.

The Templates project aims at providing precise descriptions in **template** form of the most important iterative solvers for linear systems and eigensystems, by giving a general algorithm with sufficient detail to enable customized implementation for specific problems. Templates for linear systems [11] contains a host of iterative linear system solvers, including the conjugate gradient (CG) [99], generalized minimal residual (GMRES) [160], and bi-conjugate gradient stabilized (Bi-CGSTAB) [184] methods. It is important to have a choice of solvers, since no single iterative method can solve all problems efficiently. Implementations of the complete templates for linear systems are available in C++, Fortran 77, and MATLAB. Most modern iterative methods build a **Krylov subspace**

$$K_m = \mathrm{span}\{\mathbf{r}, A\mathbf{r}, A^2\mathbf{r}, \ldots, A^{m-1}\mathbf{r}\}, \tag{4.54}$$

where $\mathbf{r} = \mathbf{b} - A\mathbf{x}^0$ is the **residual** of the initial solution $\mathbf{x}^0$. Sparse matrix–vector multiplication is the main building block of Krylov subspace methods. A recent book on these methods has been written by van der Vorst [185]. Templates for algebraic eigensystems [8] treats the more difficult problem of solving eigensystems $A\mathbf{x} = \lambda\mathbf{x}$ and generalized eigensystems $A\mathbf{x} = \lambda B\mathbf{x}$, where A, B are square matrices, and it also treats the singular value decomposition $A = U\Sigma V^{\mathrm{T}}$, where U and V are orthogonal matrices and Σ is a diagonal matrix. An important eigenvalue solver discussed is the Lanczos method [125]. The expert knowledge of the authors has been captured in a decision tree which helps choosing the most suitable eigensystem solver for the application at hand. A complete implementation of all eigensystem templates does not exist, but the book [8] gives many references to software sources.

The Sparse Basic Linear Algebra Subprograms (Sparse BLAS) [66] have been formulated recently as a standard interface for operations involving sparse matrices or vectors. The most important primitive is `BLAS_dusmv`

('Double precision Unstructured Sparse Matrix–Vector multiplication'), which computes $\mathbf{y} := \alpha A\mathbf{x} + \mathbf{y}$. The sparse BLAS have been designed in an object-oriented fashion to make them independent of the data structure used in an implementation. Reference to a sparse matrix is made through a **handle**, which is in fact an integer that represents the matrix. As a result, user programs are not cluttered by details of the data structure. In an iterative solver using the sparse BLAS, a matrix will typically be created, filled with nonzero entries, used repeatedly, and then destroyed. One way of filling a sparse matrix is by dense subblocks, which occur in many applications. For each block, the location, row and column indices, and numerical values are stored. This can be used to great advantage by suitable data structures, saving memory space and yielding high computing rates that approach peak performance. The sparse BLAS are available for the languages C, Fortran 77, and Fortran 90.

The performance of sparse matrix algorithms achieved in practice depends to a large extent on the problem solved and hence it is important to use realistic test problems. The Harwell–Boeing collection was the first widely available set of sparse test matrices originating in real applications. Release 1, the version from 1989 described in [64], contained almost 300 matrices with the largest matrix `bcsstk32` of size $n = 44\,609$ possessing $1\,029\,655$ stored nonzeros. (For this symmetric matrix, only the nonzeros below or on the main diagonal were stored.) The Harwell–Boeing input format is based on the CCS data structure, see Section 4.2. The original distribution medium of the 110-Mbyte collection was a set of three 9-track tapes of 2400 feet length and 1600 bits-per-inch (bpi) density. Using the CCS format and not the triple scheme saved at least one tape! The original Harwell–Boeing collection has evolved into the Rutherford–Boeing collection [65], available online through the Matrix Market repository [26]. Matrices from this repository are available in two formats: Rutherford–Boeing (based on CCS) and Matrix Market (based on the triple scheme). Matrix Market can be searched by size, number of nonzeros, shape (square or rectangular), and symmetry of the matrices. Useful statistics and pictures are provided for each matrix. Another large online collection is maintained and continually expanded by Tim Davis at the University of Florida [53]. (The `cage` matrices [186] used in this chapter can be obtained from there.) The currently largest matrix of the Florida collection, `cage15`, has $n = 5\,154\,859$ and $nz = 99\,199\,551$.

4.11.2 *Parallel sparse matrix–vector multiplication algorithms*

Already in 1988, Fox and collaborators [71,section 21–3.4] presented a parallel algorithm for dense matrix–vector multiplication that distributes the matrix in both dimensions. They assume that a block-distributed copy of the complete input vector is available in every processor row. Their algorithm starts with a local matrix–vector multiplication, then performs a so-called **fold** operation which gathers and adds partial sums $u_{it}, 0 \leq t < N$, into a sum u_i, spreading the responsibility for computing the sums u_i of a processor row over all its processors, and finally broadcasts the sums within their processor row. Thus, the output vector becomes available in the same format as the input vector but with the role of processor rows and columns reversed. If the input and output vector are needed in exactly the same distribution, the broadcast must be preceded by a vector transposition.

The parallel sparse matrix–vector multiplication algorithm described in this chapter, Algorithm 4.5, is based on previous work by Bisseling and McColl [19,21,22]. The Cartesian version of the algorithm was first presented in [19] as part of a parallel implementation of GMRES, an iterative solver for square unsymmetric linear systems. Bisseling [19] outlines the advantages of using a two-dimensional Cartesian distribution and distributing the vectors in the same way as the matrix diagonal and suggests to use as a fixed matrix-independent distribution the **square block/cyclic distribution**, defined by assigning matrix element a_{ij} to processor $P(i \text{ div } (n/\sqrt{p}), j \bmod \sqrt{p})$. The cost analysis of the algorithm in [19] and the implementation, however, are closely tied to a square mesh communication network with store-and-forward routing. This means for instance that a partial sum sent from processor $P(s, t_0)$ to processor $P(s, t_1)$ has to be transferred through all intermediate processors $P(s, t)$, $t_0 < t < t_1$. Experiments performed on a network of 400 transputers for a subset of unsymmetric sparse matrices from the Harwell–Boeing collection [64] give disappointing speedups for the matrix–vector multiplication part of the GMRES solver, due to the limitations of the communication network.

Bisseling and McColl [21,22] transfer the sparse matrix–vector multiplication algorithm from [19] to the BSP context. The matrix distribution is Cartesian and the vectors are distributed in the same way as the matrix diagonal. Now, the algorithm benefits from the complete communication network provided by the BSP architecture. Architecture-independent time analysis becomes possible because of the BSP cost function. This leads the authors to a theoretical and experimental study of scientific computing applications such as molecular dynamics, partial differential equation solving on multidimensional grids, and linear programming, all interpreted as an instance of sparse matrix–vector multiplication. This work shows that the block/cyclic distribution is an optimal fixed Cartesian distribution for unstructured sparse matrices; also optimal is a Cartesian matrix distribution based on a balanced random distribution of the matrix diagonal. Bisseling and McColl propose to use digital diamonds for the Laplacian operator on a square grid. They perform numerical experiments using MLIB, a library of matrix generators and BSP cost analysers specifically developed for the investigation of sparse matrix–vector multiplication.

Ogielski and Aiello [148] present a four-superstep parallel algorithm for multiplication of a sparse rectangular matrix and a vector. The algorithm exploits sparsity in the computation, but not in the communication. The input and output vector are distributed differently. The matrix is distributed in a Cartesian manner by first randomly permuting the rows and columns (independently from each other) and then using an $M \times N$ cyclic distribution. A probabilistic analysis shows that the randomization leads to good expected load balance in the computation, for every matrix with a limited number of nonzeros per row and column. (This is a more general result than the load balance result given in Section 4.7, which only applies to random sparse matrices.) The vectors used in the multiplication $\mathbf{u} := A\mathbf{v}$ are first permuted in correspondence with the matrix and then component v_i is assigned to $P(s \text{ div } N, s \bmod N)$ and u_i to $P(s \bmod M, s \text{ div } M)$, where $s = i \bmod p$. Experimental results on a 16384-processor MasPar machine show better load balance than theoretically expected.

Lewis and van de Geijn [128] present several algorithms for sparse matrix–vector multiplication on a parallel computer with a mesh or hypercube communication network. Their final algorithm for $\mathbf{u} := A\mathbf{v}$ distributes the matrix and the vectors by assigning a_{ij} to processor $P((i \text{ div } (n/p)) \bmod M, j \text{ div } (n/N))$ and u_i and v_i to $P((i \text{ div } (n/p)) \bmod M, i \text{ div } (n/N))$. This data distribution fits in the scheme of Section 4.4: the matrix distribution is Cartesian and the input and output vectors are distributed in the same way as the matrix diagonal. The vector distribution uses blocks of size n/p. The resulting data distribution is similar to that of Fig. 4.5. The sparsity of the matrix is exploited in the computation, but not in the communication. Experimental results are given for the random sparse matrix with $n = 14\,000$ and $nz = 18\,531\,044$ from the conjugate gradient solver of the NAS benchmark. Such experiments were also carried out by Hendrickson, Leland, and Plimpton [96], using a sparse algorithm similar to the dense algorithm of Fox *et al.* [71]. Hendrickson and co-workers overlap computation and communication and use a special processor numbering to reduce the communication cost of the vector transposition on a hypercube computer. Hendrickson and Plimpton [97] apply ideas from this matrix–vector multiplication algorithm to compute the operation of a dense $n \times n$ force matrix on n particles in a molecular dynamics simulation.

4.11.3 *Parallel iterative solvers for linear systems*

The Parallel Iterative Methods (PIM) package by da Cunha and Hopkins [50] contains a set of Fortran 77 subroutines implementing iterative solvers. The user of the package has to supply the matrix–vector multiplication, inner product computations, and preconditioners. This makes the package independent of data distribution and data structure, at the expense of extra effort by the user. All vectors must be distributed in the same way. PIM can run on top of MPI and PVM.

The Aztec package by Tuminaro, Shadid, and Hutchinson [175] is a complete package of iterative solvers that provides efficient subroutines for sparse matrix–vector multiplication and preconditioning; see also [162] for the initial design. The data distribution of Aztec is a row distribution for the matrix and a corresponding distribution for the vectors. The package contains many tools to help the user initialize an iterative solver, such as a tool for detecting which vector components must be obtained during the fanout. In Aztec, a processor has three kinds of vector components: internal components that can be updated without communication; border components that belong to the processor but need components from other processors for an update; and external components, which are the components needed from other processors. The components of **u** and **v** are renumbered in the order internal, border, external, where the external components that must be obtained from the same processor are numbered consecutively. The local submatrix is reordered correspondingly. Two local data structures are supported: a variant of CRS with special treatment of the matrix diagonal and a block variant, which can handle dense subblocks, thus increasing the computing rate for certain problems by a factor of five.

Parallel Templates by Koster [124] is a parallel, object-oriented implementation in C++ of the complete linear system templates [11]. It can handle every possible matrix and vector distribution, including the Mondriaan distribution, and it can run on top of BSPlib and MPI-1. The high-level approach of the package and the easy reuse of its building blocks makes adding new parallel iterative solvers a quick exercise. This work introduces the ICRS data structure discussed in Section 4.2.

4.11.4 *Partitioning methods*

The multilevel partitioning method has been proposed by Bui and Jones [34] and improved by Hendrickson and Leland [95]. Hendrickson and Leland present a multilevel scheme for partitioning a sparse undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ among the processors of a parallel computer, where the aim is to obtain subsets of vertices of roughly equal size and with a minimum number of **cut edges**, that is, edges that connect pairs of vertices in different subsets (and hence on different processors). In the case of a square symmetric matrix, the data distribution problem for sparse matrix–vector multiplication can be converted to a graph partitioning problem by identifying matrix row i with vertex i and matrix nonzero a_{ij}, $i < j$, with an edge $(i, j) \in \mathcal{E}$. (Self-edges (i, i) are not created.) The graph is partitioned and the processor that obtains a vertex i in the graph partitioning becomes the owner of matrix row i and vector components u_i, v_i. Hendrickson and Leland [95] use a spectral method [156] for the initial partitioning, based on solving an eigensystem for the Laplacian matrix connected to the graph. Their method allows initial partitioning into two subsets, but also into four or eight, which may sometimes be better. The authors implemented their partitioning algorithms in a package called Chaco.

Karypis and Kumar [118] investigate the three phases of multilevel graph partitioning, proposing new heuristics for each phase, based on extensive experiments. For the coarsening phase, they propose to match each vertex with the neighbouring vertex connected by the heaviest edge. For the initial partitioning, they propose to grow a partition greedily by highest gain, starting from an arbitrary vertex, until half the total vertex weight is included in the partition. For the uncoarsening, they propose a boundary Kernighan–Lin algorithm. The authors implemented the graph partitioner in a package called METIS. Karypis and Kumar [119] also developed a parallel multilevel algorithm that performs the partitioning itself on a parallel computer, with p processors computing a p-way partitioning. The algorithm uses a **graph colouring**, that is, a colouring of the vertices such that neighbouring vertices have different colours. To avoid conflicts between processors when matching vertices in the coarsening phase, each coarsening step is organized by colour, trying to find a match for vertices of one colour first, then for those of another colour, and so on. This parallel algorithm has been implemented in the ParMETIS package.

Walshaw and Cross [190] present a parallel multilevel graph partitioner called PJostle, which is aimed at two-dimensional and three-dimensional irregular grids. PJostle and ParMETIS both start with the vertices already distributed over the processors in some manner and finish with a better distribution. If the input partitioning is good, the computation of a new partitioning is faster; the quality of the output partitioning is not affected by the input partitioning. This means that these packages can be used for dynamic repartitioning of grids, for instance in an adaptive grid computation. The main difference between the two packages is that PJostle actually moves vertices between subdomains when trying to improve the partitioning, whereas ParMETIS keeps vertices on the original processor, but registers the new owner. Experiments in [190] show that PJostle produces a better partitioning, with about 10% less cut edges, but that ParMETIS is three times faster.

Bilderback [18] studies the communication load balance of the data distributions produced by five graph partitioning programs: Chaco, METIS, ParMETIS, PARTY, and Jostle. He observes that the difference between the communication load of the busiest processor and that of the least busy one, expressed in edge cuts, is considerable for all of these programs, indicating that there is substantial room for improvement.

Hendrickson [93] argues that the standard approach to sparse matrix partitioning by using graph partitioners such as Chaco and METIS is flawed because it optimizes the wrong cost function and because it is unnecessarily limited to square symmetric matrices. In his view, the emperor wears little more than his underwear. The standard approach minimizes the number of nonzeros that induce communication, but not necessarily the number of communication operations themselves. Thus the cost function does not take into account that if there are two nonzeros a_{ij} and $a_{i'j}$ on the same processor, the value v_j need not be sent twice to that processor. (Note that our Algorithm 4.5 obeys the old rule *ne bis in idem*, because it sends v_j only once to the same processor, as a consequence of using the index set J_s.) Furthermore, Hendrickson states that the cost function of the standard approach only considers communication volume and not the imbalance of the communication load nor the startup costs of sending a message. (Note that the BSP cost function is based on the maximum communication load of a processor, which naturally encourages communication balancing. The BSP model does not ignore startup costs, but lumps them together into one parameter l; BSP implementations such as BSPlib reduce startup costs by combining messages to the same destination in the same superstep. The user minimizes startup costs by minimizing the number of synchronizations.)

Çatalyürek and Aykanat [36,37] model the total communication volume of sparse matrix–vector multiplication correctly by using hypergraphs. They present a multilevel hypergraph partitioning algorithm that minimizes the true communication volume. The algorithm has been implemented in a package called PaToH (Partitioning Tool for Hypergraphs). Experimental results show that PaToH reduces the communication volume by 30–40% compared with graph-based partitioners. PaToH is about four times faster than the hypergraph version of METIS, hMETIS, while it produces partitionings of about the same quality. The partitioning algorithm in [36,37] is one-dimensional since all splits are carried out in the same direction, yielding a row or column distribution for the matrix with a corresponding vector distribution. PaToH can produce p-way partitionings where p need not be a power of two. Çatalyürek and Aykanat [38] also present a fine-grained approach to sparse matrix–vector multiplication, where nonzeros are assigned individually to processors. Each nonzero becomes a vertex in the problem hypergraph and each row and column becomes a net. The result is a matrix partitioning into disjoint sets A_s, $0 \leq s < p$, not necessarily corresponding to disjoint submatrices $I_s \times J_s$. This method is slower and needs more memory than the one-dimensional approach, but the resulting partitioning is excellent; the communication volume is almost halved. Hypergraph partitioning is commonly used in the design of electronic circuits and much improvement is due to work in that field. A hypergraph partitioner developed for circuit design is MLpart [35].

The two-dimensional Mondriaan matrix distribution method described in Section 4.5 is due to Vastenhouw and Bisseling [188] and has been implemented in version 1.0 of the Mondriaan package. The method used to split a matrix into two submatrices is based on the one-dimensional multilevel method for hypergraph bipartitioning by Çatalyürek and Aykanat [36,37]. The Mondriaan package can handle rectangular matrices as well as square matrices, and allows the user to impose the condition $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$. The package also has an option to exploit symmetry by assigning a_{ij} and a_{ji} to the same processor. The vector distribution methods described in Section 4.6 are due to Meesen and Bisseling [136]; these methods improve on the method described in [188] and will be included in the next major release of the Mondriaan package.

4.12 Exercises

1. Let A be a dense $m \times n$ matrix distributed by an $M \times N$ block distribution. Find a suitable distribution for the input and output vector of the dense matrix–vector multiplication $\mathbf{u} := A\mathbf{v}$; the input and output distributions can be chosen independently. Determine the BSP cost of the corresponding matrix–vector multiplication. What is the optimal ratio N/M and the BSP cost for this ratio?

2. Find a distribution of a 12×12 grid for a BSP computer with $p = 8$, $g = 10$, and $l = 50$, such that the BSP cost of executing a two-dimensional Laplacian operator is as low as possible. For the computation, we count five flops for an interior point, four flops for a boundary point that is not a corner point, and three flops for a corner point. Your distribution should be better than that of Fig. 4.20, which has a BSP cost of $90 + 14g + 2l = 330$ flops on this computer.

3. Modify the benchmarking program `bspbench` from Chapter 1 by changing the central `bsp_put` statement into a `bsp_send` and adding a corresponding `bsp_move` to the next superstep. Choose suitable sizes for tag and payload. Run the modified program for various values of p and measure the values of g and l. Compare the results with those of the original program. If your communication pattern allows you to choose between using `bsp_put` and `bsp_send`, which primitive would you choose? Why?

4. (∗) An $n \times n$ matrix A is **banded** with **upper bandwidth** b_U and **lower bandwidth** b_L if $a_{ij} = 0$ for $i < j - b_U$ and $i > j + b_L$. Let $b_L = b_U = b$. The matrix A has a band of $2b + 1$ nonzero diagonals and hence it is sparse if b is small. Consider the multiplication of a banded matrix A and a vector $\mathbf{v}$ by Algorithm 4.5 using the one-dimensional distribution $\phi(i) = i \text{ div } (n/p)$ for the matrix diagonal and the vectors, and a corresponding $M \times N$ Cartesian matrix distribution (ϕ_0, ϕ_1). For simplicity, assume that n is a multiple of p. Choosing M completely determines the matrix distribution. (See also Example 4.5, where $n = 12$, $b = 1$, and $p = 4$.)

(a) Let $b = 1$, which means that A is tridiagonal. Show that the communication cost for the choice $M = p$ (i.e. a row distribution of the matrix) is lower than for the choice $M = \sqrt{p}$ (i.e. a square distribution).

(b) Let $b = n - 1$, which means that A is dense. Section 4.4 shows that now the communication cost for the choice $M = \sqrt{p}$ is lower than for $M = p$. We may conclude that for small bandwidth the choice $M = p$ is better, whereas for large bandwidth the choice $M = \sqrt{p}$ is better. Which value of b is the break-even point between the two methods?

(c) Implement Algorithm 4.5 for the specific case of band matrices. Drop the constraint on n and p. Choose a suitable data structure for the matrix: use an array instead of a sparse data structure.

(d) Run your program and obtain experimental values for the break-even point of b. Compare your results with the theoretical predictions.

5. (∗) Let A be a sparse $m \times m$ matrix and B a dense $m \times n$ matrix with $m \geq n$. Consider the matrix–matrix multiplication $C = AB$.

(a) What is the time complexity of a straightforward sequential algorithm?
(b) Choose distributions for A, B, and C and formulate a corresponding parallel algorithm. Motivate your choice and discuss alternatives.
(c) Analyse the time complexity of the parallel algorithm.
(d) Implement the algorithm. Measure the execution time for various values of m, n, and p. Explain the results.

6. (∗) The CG algorithm by Hestenes and Stiefel [99] is an iterative method for solving a symmetric positive definite linear system of equations $A\mathbf{x} = \mathbf{b}$. (A matrix A is **positive definite** if $\mathbf{x}^{\mathrm{T}} A\mathbf{x} > 0$, for all $\mathbf{x} \neq 0$.) The algorithm computes a sequence of approximations $\mathbf{x}^k$, $k = 0, 1, 2 \ldots$, that converges towards the solution $\mathbf{x}$. The algorithm is usually considered converged when $\|\mathbf{r}^k\| \leq \epsilon_{\mathrm{conv}} \|\mathbf{b}\|$, where $\mathbf{r}^k = \mathbf{b} - A\mathbf{x}^k$ is the residual. One can take, for example, $\epsilon_{\mathrm{conv}} = 10^{-12}$. A sequential (nonpreconditioned) CG algorithm is given as Algorithm 4.8. For more details and a proof of convergence, see Golub and Van Loan [79].

(a) Design a parallel CG algorithm based on the sparse matrix–vector multiplication of this chapter. How do you distribute the vectors $\mathbf{x}, \mathbf{r}, \mathbf{p}, \mathbf{w}$? Motivate your design choices. Analyse the time complexity.
(b) Implement your algorithm in a function `bspcg`, which uses `bspmv` for the matrix–vector multiplication and `bspip` from Chapter 1 for inner product computations.
(c) Write a test program that first generates an $n \times n$ sparse matrix B with a random sparsity pattern and random nonzero values in the interval $[-1, 1]$ and then turns B into a symmetric matrix $A = B + B^{\mathrm{T}} + \mu I_n$. Choose the scalar μ sufficiently large to make A **strictly diagonally dominant**, that is, $|a_{ii}| > \sum_{j=0, j \neq i}^{n-1} |a_{ij}|$ for all i, and to make the diagonal elements a_{ii} positive. It can be shown that such a matrix is positive definite, see [79]. Use the Mondriaan package with suitable options to distribute the matrix and the vectors.
(d) Experiment with your program and explain the results. Try different n and p and different nonzero densities. How does the run time of `bspcg` scale with p? What is the bottleneck? Does the number of iterations needed depend on the number of processors and the distribution?

7. (∗) In a typical molecular dynamics simulation, the movement of a large number of particles is followed for a long period of time to gain insight into a physical process. For an efficient parallel simulation, it is crucial to use a

Algorithm 4.8. Sequential conjugate gradient algorithm.

```
input:  $A$: sparse $n \times n$ matrix,
        $\mathbf{b}$ : dense vector of length $n$.
output: $\mathbf{x}$ : dense vector of length $n$, such that $A\mathbf{x} \approx \mathbf{b}$.

$\mathbf{x} := \mathbf{x}^0$; { initial guess }
$k := 0$; { iteration number }
$\mathbf{r} := \mathbf{b} - A\mathbf{x}$;
$\rho := \|\mathbf{r}\|^2$;
while $\sqrt{\rho} > \epsilon_{\rm conv}\|\mathbf{b}\| \wedge k < k_{\max}$ do
        if $k = 0$ then
                $\mathbf{p} := \mathbf{r}$;
        else
                $\beta := \rho/\rho_{\rm old}$;
                $\mathbf{p} := \mathbf{r} + \beta\mathbf{p}$;
        $\mathbf{w} := A\mathbf{p}$;
        $\gamma := \mathbf{p}^{\rm T}\mathbf{w}$;
        $\alpha := \rho/\gamma$;
        $\mathbf{x} := \mathbf{x} + \alpha\mathbf{p}$;
        $\mathbf{r} := \mathbf{r} - \alpha\mathbf{w}$;
        $\rho_{\rm old} := \rho$;
        $\rho := \|\mathbf{r}\|^2$;
        $k := k + 1$;
```

good data distribution, especially in three dimensions. We can base the data distribution on a suitable geometric partitioning of space, following [169].

Consider a simulation with a three-dimensional simulation box of size $1.0 \times 1.0 \times 1.0$ containing n particles, spread homogeneously, which interact if their distance is less than a cut-off radius $r_{\rm c}$, with $r_{\rm c} \ll 1$, see Fig. 4.2. Assume that the box has periodic boundaries, meaning that a particle near a boundary interacts with particles near the opposite boundary.

(a) Design a geometric distribution of the particles for $p = 2q^3$ processors based on the truncated octahedron, see Fig. 4.22. What is the difference with the case of the Laplacian operator on a three-dimensional grid? For a given small value of $r_{\rm c}$, how many nonlocal particles are expected to interact with the n/p local particles of a processor? These neighbouring nonlocal particles form the **halo** of the local domain; their position must be obtained by communication. Give the ratio between the number of halo particles and the number of local particles, which is proportional to the communication-to-computation ratio of a parallel simulation.

(b) Implement your distribution by writing a function that computes the processor responsible for a particle at location (x, y, z).
(c) Design a scaling procedure that enables use of the distribution method for $p = 2q_0q_1q_2$ processors with q_0, q_1, q_2 arbitrary positive integers. Give the corresponding particle ratio.
(d) Test you distribution function for a large ensemble of particles located at random positions in the box. Compare the corresponding ratio with the predicted ratio.
(e) How would you distribute the particles for $p = 8$?
(f) Compare the output quality of the geometric distribution program to that of the distribution produced by running the Mondriaan package in one-dimensional mode for the matrix A defined by taking $a_{ij} \neq 0$ if and only if particles i and j interact. Use the option $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ in Mondriaan, and convert the output to a particle distribution by assigning particle i to the processor that owns u_i and v_i.

8. $(**)$ Sometimes, it is worthwhile to use an optimal splitting for the function *split* of Algorithm 4.6, even if such a splitting takes much more computation time than a multilevel splitting. An optimal splitting could be the best choice for the final phases of a p-way partitioning process, in particular for the smaller submatrices.

(a) Design and implement a sequential algorithm that splits a sparse matrix into two sets of columns satisfying the load imbalance constraint (4.27) and with guaranteed minimum communication volume in the corresponding matrix–vector multiplication. Because of the strict demand for optimality, the algorithm should be based on some form of brute-force enumeration of all possible splits with selection of the best split. Formulate the algorithm recursively, with the recursive task defined as assigning columns $j, \ldots, n-1$ to processor $P(0)$ or $P(1)$.
(b) The assignment of columns $j = 0, 1, \ldots, n-1$ to $P(0)$ or $P(1)$ can be viewed as a binary computation tree. A node of the tree at depth j corresponds to an assignment of all columns up to and including column j to processors. The root of the tree, at depth 0, corresponds to an assignment of column 0 to $P(0)$, which can be done without loss of generality. Each node at depth j with $j < n-1$ has two branches to children nodes, representing assignment of column $j+1$ to $P(0)$ and $P(1)$, respectively. Nodes at depth $n-1$ are leaves, that is, they have no children.

Accelerate the search for the best solution, that is, the best path to a leaf of the tree, by pruning part of the tree. For instance, when the imbalance exceeds ϵ after assigning a number of columns, further assignments can make matters only worse. Thus, there is no need to search the subtree corresponding to all possible assignments of the remaining columns. A subtree can also be pruned when the number of communications incurred exceeds the minimum number found so far for a complete assignment of all columns. This pruning approach makes the search algorithm a so-called **branch-and-bound** method.

(c) Try to accelerate the search further by adding heuristics to the search method, such as choosing the branch to search first by some greedy criterion, or reordering the columns at the start.

(d) Compare the quality and computing time of your optimal splitter to that of the splitter used in the Mondriaan package. What size of problems can you solve on your computer?

(e) How would you parallelize your splitting algorithm?

9. ($**$) Finding the best matrix distribution for a parallel sparse matrix–vector multiplication $\mathbf{u} := A\mathbf{v}$ is a combinatorial optimization problem. It can be solved by recursive matrix partitioning using multilevel splitting (see Section 4.5) or optimal splitting by enumeration (see Exercise 8). We can also use the general-purpose method for combinatorial optimization known as **simulated annealing**, which is based on the Metropolis algorithm [139]. This method simulates slow cooling of a liquid: the temperature is gradually lowered until the liquid freezes; at that point the molecules of the liquid loose their freedom to move and they line up to form crystals. If the liquid cools down slowly enough, the molecules have time to adapt to the changing temperature so that the final configuration will have the lowest possible energy.

(a) Implement the simulated annealing algorithm for parallel sparse matrix–vector multiplication. Start with a random distribution ϕ of the matrix A. Write a sequential function that computes the corresponding communication volume V_ϕ, defined in eqn (4.10). For the purpose of optimization, take $V_\phi g/p$ as the communication cost and ignore the synchronization cost because it is either $2l$ or $4l$. Assume that the value of g is known. Try to improve the distribution by a sequence of moves, that is, assignments of a randomly chosen nonzero a_{ij} to a randomly chosen processor. A move is accepted if the BSP cost of the new distribution is lower than that of the old one. If, however, only cost decreases were allowed, the process could easily get stuck in a local minimum, and this will not always be a global minimum. Such a process would not be able to peek over the upcoming mountain ridge to see that there lies a deeper valley ahead. To escape from local minima, the method occasionally accepts an increase in cost. This is more likely at the beginning than at the end. Suppose we use as cost function the normalized cost C, that is, the BSP cost divided by $2nz(A)/p$. A move with cost increase ΔC is accepted with probability $e^{-\Delta C/T}$, where T is the current temperature of the annealing process. Write a sequential function that decides whether to accept a move with a given (positive or negative) cost increment ΔC.

(b) Write an efficient function that computes the cost increment for a given move. Note that simply computing the cost from scratch before and after the move and taking the difference is inefficient; this approach would be too slow for use inside a simulated annealing program, where many moves must be evaluated. Take care that updating the cost for a sequence of moves yields the same result as computing the cost from scratch. Hint: keep track of the contribution of each processor to the cost of the four supersteps of the matrix–vector multiplication.

(c) Put everything together and write a complete simulated annealing program. The main loops of your program should implement a **cooling schedule**, that is, a method for changing the temperature T during the course of the computation. Start with a temperature T_0 that is much larger than every possible increment ΔC to be encountered. Try a large number of moves at the initial temperature, for instance $p{\cdot}nz(A)$ moves, and then reduce the temperature, for example, to $T_1 = 0.99T_0$, thus making cost increases less likely to be accepted. Perform another round of moves, reduce the temperature further, and so on. Finding a good cooling schedule requires some trial and error.

(d) Compare the output quality and computing time of the simulated annealing program to that of the Mondriaan package. Discuss the difference between the output distributions produced by the two programs.

10. (∗∗) The matrix–vector multiplication function `bspmv` is educational and can be optimized. The communication performance can be improved by sending data in large packets, instead of single values v_j or u_{is}, and by sending the numerical values without index information. All communication of index information can be handled by an initialization function. Such preprocessing is advantageous when `bspmv` is used repeatedly for the same matrix, which is the common situation in iterative solution methods.

(a) Write an initialization function `v_init` that computes a permutation of the local column indices `j` into the order: (i) indices corresponding to locally available vector components v_j that need not be sent; (ii) indices corresponding to locally available components v_j that must be sent; (iii) indices corresponding to components v_j that are needed but not locally available and hence must be received from other processors. To create a unique and useful ordering, the components in (i) and (ii) must be ordered by increasing global index j and those in (iii) by increasing processor number of the source processor and by increasing global index as a secondary criterion. Set pointers to the start of (i), (ii), and (iii), and to the start of each source processor within (iii). Initialize an array `colindex` (such as used in `bspmv_init`) that reflects the new ordering, giving $j =$ `colindex`[j]. Furthermore, extend the array `colindex` by adding: (iv) indices that correspond to locally available vector components v_j that are not used by a local matrix column, that is, with $j \notin J_s$; such components may need to be sent to other processors. (For a good data distribution, there will be few such components, or none.)

(b) Write a similar function `u_init` for the local row indices.

(c) Write an initialization function that permutes the rows and columns of the local matrix using `v_init` and `u_init`.

(d) The new ordering of the local indices can be used to avoid unnecessary data copying and to improve the communication. In the function `bspmv`, replace the two arrays `v` and `vloc` by a single array `V` to be used throughout the computation for storing vector components in the order given by `v_init`. Initialize `V` before the first call to `bspmv` by using `v_init`. Similarly, replace the array `u` by a new array `U` to be used for storing local partial sums u_{is} and vector components u_i in the order given by `u_init`. The values `psum` $= u_{is}$ are now stored into `U` instead of being sent immediately, to enable sending large packets. Initialize `U` to zero at the start of `bspmv`. The new index ordering will be used in a sequence of matrix–vector multiplications. Permute the output components u_i back to the original order after the last call to `bspmv`.

(e) The components v_j to be sent away must first be packed into a temporary array such that components destined for the same processor form a contiguous block. Construct a list that gives the order in which the components must be packed. Use this packing list in `bspmv` to carry out `bsp_hpput`s of blocks of numerical values, one for each destination processor. Because of the ordering of `V`, the components need not be unpacked at their destination.

(f) The partial sums u_{is} can be sent in blocks directly from `U`, one block for each destination processor. Use `bsp_send` with a suitable tag for this operation. On arrival, the data must be unpacked. Construct an unpacking list for the partial sums received. Use this list in `bspmv` to add the sums to the appropriate components in `U`.

(g) The high-performance move primitive

```
bsp_hpmove(tagptr, payloadptr);
```

saves copying time and memory by setting a pointer to the start of the tag and payload instead of moving the tag and payload explicitly out of the receive buffer. Here, `void **tagptr` is a pointer to a pointer to the tag, that is, it gives the address where the pointer to the start of the tag can be found. Similarly, `void **payloadptr` is a pointer to a pointer to the payload. The primitive returns an integer -1 if there are no messages, and otherwise the length of the payload. Modify your program to unpack the sums directly from the receive buffer by using `bsp_hpmove` instead of `bsp_get_tag` and `bsp_move`.

(h) Test the effect of these optimizations. Do you attain communication rates corresponding to optimistic g-values?

(i) Try to improve the speed of the local matrix–vector multiplication by treating the local nonzeros that do not cause communication separately. This optimization should enhance cache use on computers with a cache.

APPENDIX A

AUXILIARY BSPEDUPACK FUNCTIONS

A.1 Header file `bspedupack.h`

This header file is included in every program file of BSPedupack. It contains necessary file inclusions, useful definitions and macros, and prototypes for the memory allocation and deallocation functions.

```
/*
  ###########################################################################
  ##      BSPedupack Version 1.0                                          ##
  ##      Copyright (C) 2004 Rob H. Bisseling                             ##
  ##                                                                      ##
  ##      BSPedupack is released under the GNU GENERAL PUBLIC LICENSE     ##
  ##      Version 2, June 1991 (given in the file LICENSE)                ##
  ##                                                                      ##
  ###########################################################################
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "bsp.h"

#define SZDBL (sizeof(double))
#define SZINT (sizeof(int))
#define TRUE (1)
#define FALSE (0)
#define MAX(a,b) ((a)>(b) ? (a) : (b))
#define MIN(a,b) ((a)<(b) ? (a) : (b))

double *vecallocd(int n);
int *vecalloci(int n);
double **matallocd(int m, int n);
void vecfreed(double *pd);
void vecfreei(int *pi);
void matfreed(double **ppd);
```

A.2 Utility file `bspedupack.c`

This file contains the functions used by BSPedupack to allocate and deallocate memory dynamically for storing vectors and matrices. The functions are suited for use in a parallel program, because they can abort a parallel computation on all processors if they detect a potential memory overflow on one of the processors. If zero memory space is requested, which could happen if the local part of a distributed vector or matrix is empty, a `NULL` pointer is returned and nothing else is done.

Allocating an $m \times n$ matrix reserves a contiguous chunk of mn memory cells and prepares it for row-wise matrix access. As a result, matrix element a_{ij} can conveniently be addressed as `a[i][j]` and row i as `a[i]`.

```
#include "bspedupack.h"

/* These functions can be used to allocate and deallocate vectors and
   matrices. If not enough memory available, one processor halts them all.
*/

double *vecallocd(int n){
    /* This function allocates a vector of doubles of length n */
    double *pd;

    if (n==0){
        pd= NULL;
    } else {
        pd= (double *)malloc(n*SZDBL);
        if (pd==NULL)
            bsp_abort("vecallocd: not enough memory");
    }
    return pd;

} /* end vecallocd */

int *vecalloci(int n){
    /* This function allocates a vector of integers of length n */
    int *pi;

    if (n==0){
        pi= NULL;
    } else {
        pi= (int *)malloc(n*SZINT);
        if (pi==NULL)
            bsp_abort("vecalloci: not enough memory");
    }
    return pi;

} /* end vecalloci */

double **matallocd(int m, int n){
    /* This function allocates an m x n matrix of doubles */
    int i;
    double *pd, **ppd;

    if (m==0){
        ppd= NULL;
    } else {
        ppd= (double **)malloc(m*sizeof(double *));
        if (ppd==NULL)
            bsp_abort("matallocd: not enough memory");
        if (n==0){
            for (i=0; i<m; i++)
                ppd[i]= NULL;
        } else {
            pd= (double *)malloc(m*n*SZDBL);
            if (pd==NULL)
                bsp_abort("matallocd: not enough memory");
            ppd[0]=pd;
            for (i=1; i<m; i++)
                ppd[i]= ppd[i-1]+n;
        }
    }
    return ppd;

} /* end matallocd */

void vecfreed(double *pd){
    /* This function frees a vector of doubles */

    if (pd!=NULL)
        free(pd);

} /* end vecfreed */

void vecfreei(int *pi){
    /* This function frees a vector of integers */

    if (pi!=NULL)
        free(pi);

} /* end vecfreei */

void matfreed(double **ppd){
    /* This function frees a matrix of doubles */

    if (ppd!=NULL){
        if (ppd[0]!=NULL)
            free(ppd[0]);
        free(ppd);
    }

} /* end matfreed */
```

APPENDIX B

A QUICK REFERENCE GUIDE TO BSPLIB

Table B.1 groups the primitives of BSPlib into three classes: Single Program Multiple Data (SPMD) for creating the overall parallel structure; Direct Remote Memory Access (DRMA) for communication with puts or gets; and Bulk Synchronous Message Passing (BSMP) for communication with sends.

Functions `bsp_nprocs`, `bsp_pid`, and `bsp_hpmove` return an `int`; `bsp_time` returns a `double`; and all others return `void`. A parameter with an asterisk is a pointer; a parameter with two asterisks is a pointer to a pointer. The parameter `spmd` is a parameterless function returning `void`. The parameter `error_message` is a string. The remaining parameters are `int`s.

TABLE B.1. The 20 primitives of BSPlib [105]

Class	Primitive	Meaning	Page
SPMD	`bsp_begin(reqprocs);`	Start of parallel part	14
	`bsp_end();`	End of parallel part	15
	`bsp_init(spmd, argc, **argv);`	Initialize parallel part	15
	`bsp_nprocs();`	Number of processors	15
	`bsp_pid();`	My processor number	16
	`bsp_time();`	My elapsed time	16
	`bsp_abort(error_message);`	One processor stops all	20
	`bsp_sync();`	Synchronize globally	16
DRMA	`bsp_push_reg(*variable, nbytes);`	Register variable	19
	`bsp_pop_reg(*variable);`	Deregister variable	19
	`bsp_put(pid, *source, *dest, offset, nbytes);`	Write into remote memory	18
	`bsp_hpput(pid, *source, *dest, offset, nbytes);`	Unbuffered put	99
	`bsp_get(pid, *source, offset, *dest, nbytes);`	Read from remote memory	20
	`bsp_hpget(pid, *source, offset, *dest, nbytes);`	Unbuffered get	99
BSMP	`bsp_set_tagsize(*tagsz);`	Set new tag size	226
	`bsp_send(pid, *tag, *source, nbytes);`	Send a message	224
	`bsp_qsize(*nmessages, *nbytes);`	Number of received messages	226
	`bsp_get_tag(*status, *tag);`	Get tag of received message	225
	`bsp_move(*dest, maxnbytes);`	Store payload locally	225
	`bsp_hpmove(**tagptr, **payloadptr);`	Store by setting pointers	249

APPENDIX C

PROGRAMMING IN BSP STYLE USING MPI

Assuming you have read the chapters of this book and hence have learned how to design parallel algorithms and write parallel programs using BSPlib, this appendix quickly teaches you how to write well-structured parallel programs using the communication library MPI. For this purpose, the package MPIedupack is presented, which consists of the five programs from BSPedupack, but uses MPI instead of BSPlib, where the aim is to provide a suitable starter subset of MPI. Experimental results are given that compare the performance of programs from BSPedupack with their counterparts from MPIedupack. This appendix concludes by discussing the various ways bulk synchronous parallel style can be applied in practice in an MPI environment. After having read this appendix, you will be able to use both BSPlib and MPI to write well-structured parallel programs, and, if you decide to use MPI, to make the right choices when choosing MPI primitives from the multitude of possibilities.

C.1 The message-passing interface

The Message-Passing Interface (MPI) [137] is a standard interface for parallel programming in C and Fortran 77 that became available in 1994 and was extended by MPI-2 [138] in 1997, adding functionality in the areas of one-sided communications, dynamic process management, and parallel input/output (I/O), and adding bindings for the languages Fortran 90 and C++. MPI is widely available and has broad functionality. Most likely it has already been installed on the parallel computer you use, perhaps even in a well-optimized version provided by the hardware vendor. (In contrast, often you have to install BSPlib yourself or request this from your systems administrator.) Much parallel software has already been written in MPI, a prime example being the numerical linear algebra library ScaLAPACK [24,25,41] The availability of such a library may sometimes be a compelling reason for using MPI in a parallel application. Important public-domain implementations of MPI are MPICH [85] and LAM/MPI from Indiana University. An interesting new development is MPICH-G2 [117] for parallel programming using the Globus toolkit on the Grid, the envisioned worldwide parallel computer consisting of all the computers on the Internet.

MPI is a software interface, and not a parallel programming model. It enables programming in many different styles, including the bulk synchronous parallel style. A particular algorithm can typically be implemented in many different ways using MPI, which is the strength but also the difficulty of MPI.

MPI derives its name from the message-passing model, which is a programming model based on pairwise communication between processors, involving an active sender and an active receiver. Communication of a message (in its **blocking** form) synchronizes the sender and receiver. The cost of communicating a message of length n is typically modelled as

$$T(n) = t_{\text{startup}} + nt_{\text{word}}, \tag{C.1}$$

with a fixed startup cost t_{startup} and additional cost t_{word} per data word. Cost analysis in this model requires a detailed study of the order in which the messages are sent, their lengths, and the computations that are interleaved between the communications. In its most general form this can be expressed by a directed acyclic graph with chunks of computation as vertices and messages as directed edges.

In MPI, the archetypical primitives for the message-passing style, based on the message-passing model, are `MPI_Send` and `MPI_Recv`. An example of their use is

```
if (s==2)
    MPI_Send(x, 5, MPI_DOUBLE, 3, 0, MPI_COMM_WORLD);
if (s==3)
    MPI_Recv(y, 5, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD, &status);
```

which sends five doubles from $P(2)$ to $P(3)$, reading them from an array `x` on $P(2)$ and writing them into an array `y` on $P(3)$. Here, the integer '0' is a tag that can be used to distinguish between different messages transferred from the same source processor to the same destination processor. Furthermore, `MPI_COMM_WORLD` is the communicator consisting of all the processors. A **communicator** is a subset of processors forming a communication environment with its own processor numbering. Despite the fundamental importance of the `MPI_Send`/`MPI_Recv` pair in MPI, it is best to avoid its use if possible, as extensive use of such pairs may lead to unstructured programs that are hard to read, prove correct, or debug. Similar to the goto-statement, which was considered harmful in sequential programming by Dijkstra [56], the explicit send/receive pair can be considered harmful in parallel programming. In the parallel case, the danger of **deadlock** always exists; deadlock may occur for instance if $P(0)$ wants to send a message to $P(1)$, and $P(1)$ to $P(0)$, and both processors want to send before they receive. In our approach to using MPI, we advocate using the collective and one-sided communications of MPI where possible, and to limit the use of the send/receive pair to exceptional situations. (Note that the `goto` statement still exists in C, for good reasons, but it is hardly used any more.)

A discussion of all the MPI primitives is beyond the scope of this book, as there are almost 300 primitives (I counted 116 nondeprecated MPI-1 primitives, and 167 MPI-2 primitives). With BSPlib, we could strive for completeness, whereas with MPI, this would require a complete book by itself. We focus on the most important primitives to provide a quick entry into the MPI world. For the definitive reference, see the original MPI-1 standard [137], the most recent version of MPI-1 (currently, version 1.2) available from `http://www.mpi-forum.org`, and the MPI-2 standard [138]. A more accessible reference is the annotated standard [83,164]. For tutorial introductions, see [84,85,152].

C.2 Converting BSPedupack to MPIedupack

In this section, we create MPIedupack by converting all communication in the five BSPedupack programs from BSPlib to MPI, thereby demonstrating the various communication methods that exist in MPI. We shall comment on the suitability of each method for programming in BSP style, trying to identify a subset of MPI that can be used to program in BSP style. The first four programs are converted using only primitives from MPI-1, whereas the fifth program uses MPI-2 extensions as well.

For each program or function from BSPedupack printed in the main part of this book, we print its counterpart from MPIedupack here, but for brevity we omit the following repetitive program texts: sequential functions (such as `leastsquares` from `bspbench`); parallel functions that have not changed, except perhaps in name (the function `mpifft` is identical to `bspfft`, except that it calls the redistribution function `mpiredistr`, instead of `bspredistr`); long comments that are identical to the comments in BSPedupack; `mpiedupack.h`, which is identical to `bspedupack.h` but includes `mpi.h` instead of `bsp.h`; and `mpiedupack.c`, which is identical to `bspedupack.c` but calls `MPI_Abort` instead of `bsp_abort`.

I/O is a complicated subject in parallel programming. MPI-1 ignored the subject, leaving it up to the implementation. MPI-2 added extensive I/O functionality. We will assume for MPI that the same I/O functionality is available as for BSPlib: $P(0)$ can read, and all processors can write although the output may become multiplexed. Fortunately, this assumption often holds. For more sophisticated I/O, one should use MPI-2.

C.2.1 *Program* `mpiinprod`

The first program of BSPedupack, `bspinprod`, becomes `mpiinprod` and is presented below. The SPMD part of the new program is started by calling `MPI_Init`; note that it needs the addresses of `argc` and `argv` (and not the arguments themselves as in `bsp_init`). The SPMD part is terminated by calling `MPI_Finalize`. The standard communicator available in MPI is

MPI_COMM_WORLD; it consists of all the processors. The corresponding number of processors p can be obtained by calling MPI_Comm_size and the local processor identity s by calling MPI_Comm_rank. Globally synchronizing all the processors, the equivalent of a bsp_sync, can be done by calling MPI_Barrier for the communicator MPI_COMM_WORLD. Here, this is done before using the wall-clock timer MPI_Wtime. As in BSPlib, the program can be aborted by one processor if it encounters an error. In that case an error number is returned. Unlike bspip, the program mpiip does not ask for the number of processors to be used; it simply assumes that all available processors are used. (In BSPlib, it is easy to use less than the maximum number of processors; in MPI, this is slightly more complicated and involves creating a new communicator of smaller size.)

Collective communication requires the participation of all the processors of a communicator. An example of a collective communication is the broadcast by MPI_Bcast in the main function of one integer, n, from the **root** $P(0)$ to all other processors. Another example is the reduction operation by MPI_Allreduce in the function mpiip, which sums the double-precision local inner products inprod, leaving the result alpha on all processors. (It is also possible to perform such an operation on an array, by changing the parameter 1 to the array size, or to perform other operations, such as taking the maximum, by changing MPI_SUM to MPI_MAX.)

Note that the resulting program mpiip is shorter than the BSPlib equivalent. Using collective-communication functions built on top of the BSPlib primitives would reduce the program size for the BSP case in the same way. (Such functions are available, but they can also easily be written by the programmer herself, and tailored to the specific situation.)

Now, try to compile the program by the UNIX command

```
cc -o ip mpiinprod.c mpiedupack.c -lmpi -lm
```

and run the resulting executable program ip on four processors by the command

```
mpirun -np 4 ip
```

and see what happens. An alternative run command, with prescribed and hence portable definition of its options is

```
mpiexec -n 4 ip
```

The program text is:

```
#include "mpiedupack.h"

/*  This program computes the sum of the first n squares, for n>=0,
        sum = 1*1 + 2*2 + ... + n*n
    by computing the inner product of x=(1,2,...,n)^T and itself.
    The output should equal n*(n+1)*(2n+1)/6.
    The distribution of x is cyclic.
*/
```

```
double mpiip(int p, int s, int n, double *x, double *y){
    /* Compute inner product of vectors x and y of length n>=0 */

    int nloc(int p, int s, int n);
    double inprod, alpha;
    int i;

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++){
        inprod += x[i]*y[i];
    }
    MPI_Allreduce(&inprod,&alpha,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

    return alpha;

} /* end mpiip */

int main(int argc, char **argv){

    double mpiip(int p, int s, int n, double *x, double *y);
    int nloc(int p, int s, int n);
    double *x, alpha, time0, time1;
    int p, s, n, nl, i, iglob;

    /* sequential part */

    /* SPMD part */
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&p); /* p = number of processors */
    MPI_Comm_rank(MPI_COMM_WORLD,&s); /* s = processor number */

    if (s==0){
        printf("Please enter n:\n"); fflush(stdout);
        scanf("%d",&n);
        if(n<0)
            MPI_Abort(MPI_COMM_WORLD,-1);
    }

    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

    nl= nloc(p,s,n);
    x= vecallocd(nl);
    for (i=0; i<nl; i++){
        iglob= i*p+s;
        x[i]= iglob+1;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    time0=MPI_Wtime();

    alpha= mpiip(p,s,n,x,x);
    MPI_Barrier(MPI_COMM_WORLD);
    time1=MPI_Wtime();

    printf("Processor %d: sum of squares up to %d*%d is %.lf\n",
           s,n,n,alpha); fflush(stdout);
    if (s==0){
        printf("This took only %.6lf seconds.\n", time1-time0);
        fflush(stdout);
    }

    vecfreed(x);
    MPI_Finalize();

    /* sequential part */
    exit(0);

} /* end main */
```

C.2.2 *Program* `mpibench`

The main question regarding the conversion of the second program from BSPedupack, `bspbench`, is which communication method from MPI to benchmark. Of course, we can benchmark all methods, but that would be a very cumbersome exercise. In the BSPlib case, we opted for benchmarking a typical user program, where the user does not care about communication optimization, for example, about combining messages to the same destination, but instead relies on the BSPlib system to do this for her. When writing a program in MPI, a typical user would look first if there is a collective-communication function that can perform the job for him. This would lead to shorter program texts, and is good practice from the BSP point of view as well. Therefore, we should choose a collective communication as the operation to be benchmarked for MPI.

The BSP superstep, where every processor can communicate in principle with all the others, is reflected best by the **all-to-all** primitives from MPI, also called **total exchange** primitives. Using an all-to-all primitive gives the MPI system the best opportunities for optimization, similar to the opportunities that the superstep gives to the BSPlib system.

The primitive `MPI_Alltoall` requires that each processor send exactly the same number n of data to every processor, thus performing a full $(p-1)n$-relation. (The syntax of the primitive also requires sending n data to the processor itself. We do not count this as communication and neither should the system handle this as true communication.) A more flexible variant, the so-called **vector** variant, is the primitive `MPI_Alltoallv`, which allows a varying number of data to be sent (or even no data). This is often needed in applications, and also if we want to benchmark h-relations with h not a multiple of $p-1$. Another difference is that the `MPI_Alltoall` primitive requires the data in its send and receive arrays to be ordered by increasing processor number, whereas MPI_Alltoallv allows an arbitrary ordering. Therefore, we choose MPI_Alltoallv as the primitive to be benchmarked.

Before the use of MPI_Alltoallv, the number of sends to each processor is determined. The number of sends Nsend[s1] to a remote processor $P(s_1)$ equals $\lfloor h/(p-1)\rfloor$ or $\lceil h/(p-1)\rceil$. The number of sends to the processor itself is zero. In the same way, the number of receives is determined. The offset Offset_send[s1] is the distance, measured in units of the data type involved (doubles), from the start of the send array where the data destined for $P(s_1)$ can be found. Similarly, Offset_recv[s1] gives the location where the data received from $P(s_1)$ must be placed. Note that BSPlib expresses offset parameters in raw bytes, whereas MPI expresses them in units of the data type involved. (The MPI approach facilitates data transfer between processors with different architectures.)

Another collective-communication primitive used by the program is MPI_Gather, which gathers data from all processors in the communicator onto one processor. Here, one local double, time, of every processor is gathered onto $P(0)$. The values are gathered in order of increasing processor number and hence the time measured for $P(s_1)$ is stored in Time[s1].

```
#include "mpiedupack.h"

/*  This program measures p, r, g, and l of a BSP computer
    using MPI_Alltoallv for communication.
*/

#define NITERS 100     /* number of iterations */
#define MAXN 1024      /* maximum length of DAXPY computation */
#define MAXH 256       /* maximum h in h-relation */
#define MEGA 1000000.0

int main(int argc, char **argv){
    void leastsquares(int h0, int h1, double *t, double *g, double *l);
    int p, s, s1, iter, i, n, h,
        *Nsend, *Nrecv, *Offset_send, *Offset_recv;
    double alpha, beta, x[MAXN], y[MAXN], z[MAXN], src[MAXH], dest[MAXH],
           time0, time1, time, *Time, mintime, maxtime,
           nflops, r, g0, l0, g, l, t[MAXH+1];

    /**** Determine p ****/
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&p); /* p = number of processors */
    MPI_Comm_rank(MPI_COMM_WORLD,&s); /* s = processor number */

    Time= vecallocd(p);

    /**** Determine r ****/
    for (n=1; n <= MAXN; n *= 2){
        /* Initialize scalars and vectors */
        alpha= 1.0/3.0;
        beta= 4.0/9.0;
        for (i=0; i<n; i++)
            z[i]= y[i]= x[i]= (double)i;
        /* Measure time of 2*NITERS DAXPY operations of length n */
        time0= MPI_Wtime();

        for (iter=0; iter<NITERS; iter++){
            for (i=0; i<n; i++)
                y[i] += alpha*x[i];
            for (i=0; i<n; i++)
                z[i] -= beta*x[i];
        }
        time1= MPI_Wtime();
        time= time1-time0;
        MPI_Gather(&time,1,MPI_DOUBLE,Time,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

        /* Processor 0 determines minimum, maximum, average computing rate */
        if (s==0){
            mintime= maxtime= Time[0];
            for(s1=1; s1<p; s1++){
                mintime= MIN(mintime,Time[s1]);
                maxtime= MAX(maxtime,Time[s1]);
            }
            if (mintime>0.0){
                /* Compute r = average computing rate in flop/s */
                nflops= 4*NITERS*n;
                r= 0.0;
                for(s1=0; s1<p; s1++)
                    r += nflops/Time[s1];
                r /= p;
                printf("n= %5d min= %7.3lf max= %7.3lf av= %7.3lf Mflop/s ",
                       n, nflops/(maxtime*MEGA),nflops/(mintime*MEGA), r/MEGA);
                fflush(stdout);
                /* Output for fooling benchmark-detecting compilers */
                printf(" fool=%7.1lf\n",y[n-1]+z[n-1]);
            } else
                printf("minimum time is 0\n"); fflush(stdout);
        }
    }

    /**** Determine g and l ****/
    Nsend= vecalloci(p);
    Nrecv= vecalloci(p);
    Offset_send= vecalloci(p);
    Offset_recv= vecalloci(p);


    for (h=0; h<=MAXH; h++){
        /* Initialize communication pattern */

        for (i=0; i<h; i++)
            src[i]= (double)i;
        if (p==1){
            Nsend[0]= Nrecv[0]= h;
        } else {
            for (s1=0; s1<p; s1++)
                Nsend[s1]= h/(p-1);
            for (i=0; i < h%(p-1); i++)
                Nsend[(s+1+i)%p]++;
            Nsend[s]= 0; /* no communication with yourself */
            for (s1=0; s1<p; s1++)
                Nrecv[s1]= h/(p-1);
            for (i=0; i < h%(p-1); i++)
                Nrecv[(s-1-i+p)%p]++;
            Nrecv[s]= 0;
        }

        Offset_send[0]= Offset_recv[0]= 0;
        for(s1=1; s1<p; s1++){
            Offset_send[s1]= Offset_send[s1-1] + Nsend[s1-1];
            Offset_recv[s1]= Offset_recv[s1-1] + Nrecv[s1-1];
        }

        /* Measure time of NITERS h-relations */
        MPI_Barrier(MPI_COMM_WORLD);
        time0= MPI_Wtime();
        for (iter=0; iter<NITERS; iter++){
            MPI_Alltoallv(src, Nsend,Offset_send,MPI_DOUBLE,
                          dest,Nrecv,Offset_recv,MPI_DOUBLE,MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
        }
        time1= MPI_Wtime();
        time= time1-time0;

        /* Compute time of one h-relation */
        if (s==0){
            t[h]= (time*r)/NITERS;
            printf("Time of %5d-relation= %lf sec= %8.0lf flops\n",
                   h, time/NITERS, t[h]); fflush(stdout);
        }
    }

    if (s==0){
        printf("size of double = %d bytes\n",(int)SZDBL);
        leastsquares(0,p,t,&g0,&l0);
        printf("Range h=0 to p   : g= %.1lf, l= %.1lf\n",g0,l0);
        leastsquares(p,MAXH,t,&g,&l);
        printf("Range h=p to HMAX: g= %.1lf, l= %.1lf\n",g,l);

        printf("The bottom line for this BSP computer is:\n");
        printf("p= %d, r= %.3lf Mflop/s, g= %.1lf, l= %.1lf\n",
               p,r/MEGA,g,l);

        fflush(stdout);
    }
    vecfreei(Offset_recv);
    vecfreei(Offset_send);
    vecfreei(Nrecv);
    vecfreei(Nsend);
    vecfreed(Time);

    MPI_Finalize();

    exit(0);

} /* end main */
```

C.2.3 *Function* `mpilu`

For the LU decomposition, we have numbered the processors in two-dimensional fashion, relating one-dimensional and two-dimensional processor numbers by the standard identification $P(s,t) \equiv P(s+tM)$. We used two-dimensional processor numbers in the LU decomposition algorithm, but had to translate these into one-dimensional numbers in the actual BSPlib program. In MPI, we can carry the renumbering one step further by defining a communicator for every processor row and column, which allows us to use the processor row number s and the processor column number t directly in the communication primitives. In the function below, the new communicators are created by splitting the old communicator `MPI_COMM_WORLD` into subsets by the primitive `MPI_Comm_split`. Processors that call this primitive with the same value of s end up in the same communicator, which we call `row_comm_s`. As a result, we obtain M communicators, each corresponding to a processor row $P(s,*)$. Every processor obtains a processor number within its communicator. This number is by increasing value of the third parameter of the primitive, that is, `t`. Because of the way we split, the processor number of $P(s,t)$ within the row communicator is simply t. The column communicators are created by a similar split in the other direction. As a result, we can address processors in three different ways: by the global identity $s+tM$ within the `MPI_COMM_WORLD` communicator; by the column number t within the `row_comm_s` communicator; and by the row number s within the `col_comm_t` communicator. This allows us for instance to broadcast the pivot value, a double, from processor $P(k \bmod n, \texttt{smax})$ within its processor column, simply by using the `MPI_Bcast` primitive with `col_comm_t` as the communicator parameter, see `Superstep 1`. (For easy reference, the program supersteps retain the original numbering from BSPedupack.) Another instance where use of the new communicators is convenient is in the broadcast of `nlr-kr1` doubles from `lk` (representing a column part) within each processor row, see `Superstep 3`. The source processor of the broadcast within $P(s,*)$ is $P(s, k \bmod n)$.

Communicators can also be used to split a BSP computer into different subcomputers, such as in the decomposable BSP (D-BSP) model [55]. Care must be taken to do this in a structured way, as it is still possible for processors from different subcomputers to communicate with each other (for instance through the `MPI_COMM_WORLD` communicator). This use of communicators can be important for certain applications, for example, with a tree-like computational structure.

Because we have a wide variety of collective communications available in MPI, we should try to use one of them to determine the maximum absolute value in $A(k+1\colon n-1, k)$ and the processor that contains it. It turns out to be most convenient to perform a reduction operation over the processors in $P(*, k \bmod n)$, using `MPI_Allreduce` within one column communicator, see `Superstep 0`. The operation works on pairs consisting of a double representing the local maximum absolute value and an integer representing the index s of the corresponding processor. A pair is stored using a `struct` of a double `val` and an integer `idx`. The local input pair is `max` and the global output pair is `max_glob`; this pair becomes available on all processors in the column communicator. The operation is called `MPI_MAXLOC`, since it determines a maximum and registers the location (argmax) of the maximum. In case of equal maxima, the maximum with the lowest processor number is taken. Note that after the maximum absolute value has been determined, all processors in $P(*, k \bmod n)$ know the owner and the absolute value of the maximum, but not the corresponding pivot value; this value is broadcast in the next superstep.

The row swap of the LU decomposition can be done in many different ways using MPI. If $k \bmod M \neq r \bmod N$, the swap involves communication. In that case, the communication pattern is a swap of local row parts by N disjoint pairs of processors. Here, the communication is truly pairwise and this is a good occasion to show the point-to-point message-passing primitives in action that gave MPI its name. In the second part of `Superstep 2` the rows are swapped. Each processor $P(k \bmod\ M, t)$ sends `nlc` local values of type double to $P(r \bmod\ M, t)$, that is, $P(r \bmod\ M + tM)$ in the global one-dimensional numbering used. The messages are tagged by '1'. Note that the program text of this part is lengthy, because here we have to write lines of program text explicitly for receiving. (We became used to one-sided views of the world, but alas.) Note also the reversed order of the sends and receives for $P(k \bmod M, t)$ compared with the order for $P(r \bmod M, t)$. If we had used the same order, dreaded deadlock could occur. Whether it actually occurs is dependent on the implementation of MPI. The MPI standard allows both a buffered implementation, where processors use intermediate buffers and hence the sender is free to continue with other work after having written its message into the buffer, and an unbuffered implementation, where the sender is blocked until the communication has finished. The receiver is always blocked until the end of the communication.

The processors involved in the swapping are synchronized more than strictly necessary; the number of supersteps is actually two: one for swapping components of π, and one for swapping rows of A. In the BSPlib case, we can do the same easily within one superstep, by superstep piggybacking, that is, combining unrelated supersteps to reduce the amount of synchronization. (Using MPI, we could do this as well, if we are prepared to pack and unpack the data in a certain way, using so-called derived types. This is more complicated, and may be less efficient due to copying overhead.) An advantage of the send/receive approach is that the $p-2N$ processors not involved in the swap can race through unhindered, but they will be stopped at the next traffic light (the broadcast in `Superstep 3`).

It would be rather unnatural to coerce the row swap into the framework of a collective communication. Since the pattern is relatively simple, it cannot do much harm to use a send/receive pair here. If desired, we can reduce the program length by using a special MPI primitive for paired messages, `MPI_Sendrecv`, or, even better, by using `MPI_Sendrecv_replace`, which requires only one buffer for sending and receiving, thus saving memory. (MPI has many more variants of send and receive operations.)

```
#include "mpiedupack.h"

#define EPS 1.0e-15

void mpilu(int M, int N, int s, int t, int n, int *pi, double **a){
    /* Compute LU decomposition of n by n matrix A with partial pivoting.
       Processors are numbered in two-dimensional fashion.
       Program text for P(s,t) = processor s+t*M,
       with 0 <= s < M and 0 <= t < N.
       A is distributed according to the M by N cyclic distribution.
    */

    int nloc(int p, int s, int n);
    double *uk, *lk;
    int nlr, nlc, k, i, j, r;

    MPI_Comm row_comm_s, col_comm_t;
    MPI_Status status, status1;

    /* Create a new communicator for my processor row and column */
    MPI_Comm_split(MPI_COMM_WORLD,s,t,&row_comm_s);
    MPI_Comm_split(MPI_COMM_WORLD,t,s,&col_comm_t);

    nlr=  nloc(M,s,n); /* number of local rows */
    nlc=  nloc(N,t,n); /* number of local columns */

    uk= vecallocd(nlc);
    lk= vecallocd(nlr);

    /* Initialize permutation vector pi */
    if (t==0){
        for(i=0; i<nlr; i++)
            pi[i]= i*M+s; /* global row index */
    }
```

```
for (k=0; k<n; k++){
    int kr, kr1, kc, kc1, imax, smax, tmp;
    double absmax, pivot, atmp;
    struct {
        double val;
        int idx;
    } max, max_glob;

    /****** Superstep 0 ******/
    kr=  nloc(M,s,k); /* first local row with global index >= k */
    kr1= nloc(M,s,k+1);
    kc=  nloc(N,t,k);
    kc1= nloc(N,t,k+1);

    if (k%N==t){   /* k=kc*N+t */
        /* Search for local absolute maximum in column k of A */
        absmax= 0.0; imax= -1;
        for (i=kr; i<nlr; i++){
            if (fabs(a[i][kc])>absmax){
                absmax= fabs(a[i][kc]);
                imax= i;
            }
        }

        /* Determine value and global index of absolute maximum
           and broadcast them to P(*,t) */
        max.val= absmax;
        if (absmax>0.0){
            max.idx= imax*M+s;
        } else {
            max.idx= n; /* represents infinity */
        }
        MPI_Allreduce(&max,&max_glob,1,MPI_DOUBLE_INT,MPI_MAXLOC,col_comm_t);

        /****** Superstep 1 ******/

        /* Determine global maximum */
        r= max_glob.idx;
        pivot= 0.0;
        if (max_glob.val > EPS){
            smax= r%M;
            if (s==smax)
                pivot = a[imax][kc];
            /* Broadcast pivot value to P(*,t) */
            MPI_Bcast(&pivot,1,MPI_DOUBLE,smax,col_comm_t);

            for(i=kr; i<nlr; i++)
                a[i][kc] /= pivot;
            if (s==smax)
                a[imax][kc]= pivot; /* restore value of pivot */
        } else {
            MPI_Abort(MPI_COMM_WORLD,-6);
        }
    }

    /* Broadcast index of pivot row to P(*,*) */
    MPI_Bcast(&r,1,MPI_INT,k%N,row_comm_s);

    /****** Superstep 2 ******/
    if (t==0){
        /* Swap pi(k) and pi(r) */
        if (k%M != r%M){
            if (k%M==s){
                /* Swap pi(k) and pi(r) */
                MPI_Send(&pi[k/M],1,MPI_INT,r%M,0,MPI_COMM_WORLD);
                MPI_Recv(&pi[k/M],1,MPI_INT,r%M,0,MPI_COMM_WORLD,&status);
            }
            if (r%M==s){
                MPI_Recv(&tmp,1,MPI_INT,k%M,0,MPI_COMM_WORLD,&status);
                MPI_Send(&pi[r/M],1,MPI_INT,k%M,0,MPI_COMM_WORLD);
                pi[r/M]= tmp;
            }
        } else if (k%M==s){
            tmp= pi[k/M];
            pi[k/M]= pi[r/M];
            pi[r/M]= tmp;
        }
    }
    /* Swap rows k and r */
    if (k%M != r%M){
        if (k%M==s){
           MPI_Send(a[k/M],nlc,MPI_DOUBLE,r%M+t*M,1,MPI_COMM_WORLD);
           MPI_Recv(a[k/M],nlc,MPI_DOUBLE,r%M+t*M,1,MPI_COMM_WORLD,&status1);
        }
        if (r%M==s){
            /* abuse uk as a temporary receive buffer */
            MPI_Recv(uk,nlc,MPI_DOUBLE,k%M+t*M,1,MPI_COMM_WORLD,&status1);
            MPI_Send(a[r/M],nlc,MPI_DOUBLE,k%M+t*M,1,MPI_COMM_WORLD);
            for(j=0; j<nlc; j++)
                a[r/M][j]= uk[j];
        }
    } else if (k%M==s){
        for(j=0; j<nlc; j++){
            atmp= a[k/M][j];
            a[k/M][j]= a[r/M][j];
            a[r/M][j]= atmp;
        }
    }

    /****** Superstep 3 ******/
    if (k%N==t){
        /* Store new column k in lk */
        for(i=kr1; i<nlr; i++)
            lk[i-kr1]= a[i][kc];
    }
```

```
        if (k%M==s){
            /* Store new row k in uk */
            for(j=kc1; j<nlc; j++)
                uk[j-kc1]= a[kr][j];
        }
        MPI_Bcast(lk,nlr-kr1,MPI_DOUBLE,k%N,row_comm_s);

        /****** Superstep 4 ******/
        MPI_Bcast(uk,nlc-kc1,MPI_DOUBLE,k%M,col_comm_t);

        /****** Superstep 0 ******/
        /* Update of A */
        for(i=kr1; i<nlr; i++){
            for(j=kc1; j<nlc; j++)
                a[i][j] -= lk[i-kr1]*uk[j-kc1];
        }
    }
    vecfreed(lk);
    vecfreed(uk);

} /* end mpilu */
```

C.2.4 *Function mpifft*

The function `mpifft` differs in only one place from the function `bspfft`: it invokes a redistribution function written in MPI, instead of BSPlib. The program text of this function, `mpiredistr`, is listed below. The function creates packets of data in the same way as `bspredistr`, but it first fills a complete array `tmp` with packets, instead of putting the packets separately into a remote processor. The packets are sent to their destination by using `MPI_Alltoallv`, explained above in connection with `mpibench`. The destination processor of each packet is computed in the same way as in the BSPlib case. All packets sent have the same size, which is stored in `Nsend`, counting two doubles for each complex value. The offset is also stored in an array. Note that for $n/p \geq c_1/c_0$, the number of packets equals c_1/c_0. This means that some processors may not receive a packet from $P(s)$. In the common case of an FFT with $p \leq \sqrt{n}$, however, the function is used with $c_0 = 1$ and $c_1 = p$, so that the condition holds, the number of packets created by $P(s)$ is p, and the communication pattern is a true all-to-all; this is the motivation for using an MPI all-to-all primitive. In the common case, all packets have equal size n/p^2, and we can use `MPI_Alltoall`. For $n/p < c_1/c_0$, however, we must use the more general `MPI_Alltoallv`. An advantage of `MPI_Alltoallv` is that the packets can be stored in the receive array in every possible order, so that we do not have to unpack the data by permuting the packets.

A disadvantage of using `MPI_Alltoallv` instead of one-sided puts, is that receive information must be computed beforehand, namely the number of receives and the proper offset for each source processor. This is not needed when using `bsp_put` primitives as in `bspredistr`. Thus, `MPI_Alltoallv` is a true two-sided communication operation, albeit a collective one. For each packet, the global index of its first vector component is computed in the new distribution (with cycle c_1), and the processor `srcproc` is computed that owns this component in the old distribution (with cycle c_0). All the information about sends and receives is determined before `MPI_Alltoallv` is called.

The `MPI_Alltoallv` operation is preceded by an explicit global synchronization, which teaches us an MPI rule:

collective communications may synchronize the processors,
but you cannot rely on this.

Here, the local processor writes from `x` into `tmp`, and then all processors write back from `tmp` into `x`. It can happen that a particularly quick remote processor already starts writing into the space of `x` while the local processor is still reading from it. To prevent this, we can either use an extra temporary array, or insert a global synchronization to make sure all local writes into `tmp` have finished before `MPI_Alltoallv` starts. We choose the latter option, and it feels good. When in doubt, insert a barrier. The rule also says that synchronization *may* occur. Thus a processor cannot send a value before the collective communication, hoping that another processor receives it after the collective communication. For correctness, we have to think barriers, even if they are not there in the actual implementation.

```
#include "mpiedupack.h"

/****************** Parallel functions ********************************/

void mpiredistr(double *x, int n, int p, int s, int c0, int c1,
                char rev, int *rho_p){

    /* This function redistributes the complex vector x of length n,
       stored as pairs of reals, from group-cyclic distribution
       over p processors with cycle c0 to cycle c1, where
       c0, c1, p, n are powers of two with 1 <= c0 <= c1 <= p <= n.
       s is the processor number, 0 <= s < p.
       If rev=true, the function assumes the processor numbering
       is bit reversed on input.
       rho_p is the bit-reversal permutation of length p.
    */

    double *tmp;
    int np, j0, j2, j, jglob, ratio, size, npackets, t, offset, r,
        destproc, srcproc,
        *Nsend, *Nrecv, *Offset_send, *Offset_recv;

    np= n/p;
    ratio= c1/c0;
    size= MAX(np/ratio,1);
    npackets= np/size;
    tmp= vecallocd(2*np);
    Nsend= vecalloci(p);
    Nrecv= vecalloci(p);
    Offset_send= vecalloci(p);
    Offset_recv= vecalloci(p);

    for(t=0; t<p; t++){
        Nsend[t]= Nrecv[t]= 0;
        Offset_send[t]= Offset_recv[t]= 0;
    }

    /* Initialize sender info and copy data */
    offset= 0;
    if (rev) {
        j0= rho_p[s]%c0;
        j2= rho_p[s]/c0;
    } else {
        j0= s%c0;
        j2= s/c0;
    }
    for(j=0; j<npackets; j++){
        jglob= j2*c0*np + j*c0 + j0;
        destproc=  (jglob/(c1*np))*c1 + jglob%c1;
        Nsend[destproc]= 2*size;
        Offset_send[destproc]= offset;
        for(r=0; r<size; r++){
            tmp[offset + 2*r]=   x[2*(j+r*ratio)];
            tmp[offset + 2*r+1]= x[2*(j+r*ratio)+1];
        }
        offset += 2*size;
    }

    /* Initialize receiver info */

    offset= 0;
    j0= s%c1; /* indices for after the redistribution */
    j2= s/c1;
    for(r=0; r<npackets; r++){
        j= r*size;
        jglob= j2*c1*np + j*c1 + j0;
        srcproc=  (jglob/(c0*np))*c0 + jglob%c0;
        if (rev)
            srcproc= rho_p[srcproc];
        Nrecv[srcproc]= 2*size;
        Offset_recv[srcproc]= offset;
        offset += 2*size;
    }

    /* Necessary for safety */
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Alltoallv(tmp,Nsend,Offset_send,MPI_DOUBLE,
                  x,  Nrecv,Offset_recv,MPI_DOUBLE,MPI_COMM_WORLD);

    vecfreei(Offset_recv);
    vecfreei(Offset_send);
    vecfreei(Nrecv);
    vecfreei(Nsend);
    vecfreed(tmp);

} /* end mpiredistr */
```

C.2.5 *Function mpimv*

The final program text we discuss is different from the previous ones because it is solely based on the MPI-2 extensions for one-sided communication. In writing this program, we have tried to exploit the close correspondence between the one-sided communications in BSPlib and their counterparts in MPI-2. Six years after the MPI-2 standard has been released, partial MPI-2 implementations with reasonable functionality are starting to become available. A full public-domain implementation for many different architectures is expected to be delivered in the near future by the MPICH-2 project. The driving force in developing one-sided communications is their speed and their ease of use.

MPI-2 contains unbuffered put and get operations, which are called high-performance puts and gets in BSPlib. The motivation of the MPI-2 designers for choosing the unbuffered version is that it is easy for the user to provide the safety of buffering if this is required. In contrast, BSPlib provides both buffered and unbuffered versions; this book encourages use of the buffered version, except if the user is absolutely sure that unbuffered puts and gets are safe. The syntax of the unbuffered put primitives in BSPlib and MPI is

```
bsp_hpput(pid, src, dst, dst_offsetbytes, nbytes);

MPI_Put(src, src_n, src_type, pid, dst_offset, dst_n, dst_type, dst_win);
```

In BSPlib, data size and offsets are measured in bytes, whereas in MPI this is in units of the basic data type, `src_type` for the source array and `dst_type` for the destination array. In most cases these two types will be identical (e.g. both could be `MPI_DOUBLE`), and the source and destination sizes will thus be equal. The destination memory area in the MPI-2 case is not simply given by a pointer to memory space such as an array, but by a pointer to a window object, which will be explained below.

The syntax of the unbuffered get primitives in BSPlib and MPI is

```
bsp_hpget(pid, src, src_offsetbytes, dst, nbytes);

MPI_Get(dst, dst_n, dst_type, pid, src_offset, src_n, src_type, src_win);
```

Note the different order of the arguments, but also the great similarity between the puts and gets of BSPlib and those of MPI-2. In the fanout of `mpimv`, shown below, one double is obtained by an `MPI_Get` operation from the remote processor `srcprocv[j]`, at an offset of `srcindv[j]` doubles from the start of window `v_win`; the value is stored locally as `vloc[j]`. It is instructive to compare the statement with the corresponding one in `bspmv`.

A **window** is a preregistered and distributed memory area, consisting of local memory on every processor of a communicator. A window is created by `MPI_Win_create`, which is the equivalent of BSPlib's `bsp_push_reg`. We can consider this as the registration of the memory needed before puts or gets can be executed.

In the first call of `MPI_Win_create` in the function `mpimv`, a window of size `nv` doubles is created and the size of a double is determined to be the basic unit for expressing offsets of subsequent puts and gets into the window. All processors of the communication world participate in creating the window. The `MPI_INFO_NULL` parameter always works, but can be replaced by other parameters to give hints to the implementation for optimization. For further details, see the MPI-2 standard. The syntax of the registration and deregistration primitives in BSPlib and MPI is

```
bsp_push_reg(variable, nbytes);

MPI_Win_create(variable, nbytes, unit, info, comm, win);

bsp_pop_reg(variable);

MPI_Win_free(win);
```

Here, `win` is the window of type `MPI_Win` corresponding to the array `variable`; the integer `unit` is the unit for expressing offsets; and `comm` of type `MPI_Comm` is the communicator of the window.

A window can be used after a call to `MPI_Win_fence`, which can be thought of as a synchronization of the processors that own the window. The first parameter of `MPI_Win_fence` is again for transferring optimization hints, and can best be set to zero at the early learning stage; this is guaranteed to work. The communications initiated before a fence are guaranteed to have been completed after the fence. Thus the fence acts as a synchronization at the end of a superstep. A window is destroyed by a call to `MPI_Win_free`, which is the equivalent of BSPlib's `bsp_pop_reg`.

The `MPI_Put` primitive is illustrated by the function `mpimv_init`, which is a straightforward translation of `bspmv_init`. Four windows are created, one for each array, for example, `tmpprocv_win` representing the integer array `tmpprocv`. (It would have been possible to use one window instead, by using a four times larger array accessed with proper offsets, and thus saving some fences at each superstep. This may be more efficient, but it is perhaps also a bit clumsy and unnatural.)

The third, and final one-sided communication operation available in MPI-2, but not in BSPlib, is an accumulate operation, called `MPI_Accumulate`. It is similar to a put, but instead of putting a value into the destination location the accumulate operation adds a value into into the location, or takes a maximum, or performs another binary operation. The operation must be one of the predefined MPI reduction operations. The accumulate primitive allows targeting the same memory location by several processors. For the unbuffered put, `MPI_Put` or `bsp_hpput`, this is unsafe. The order in which the accumulate operations are carried out is not specified and may be implementation-dependent. (This resembles the variation in execution order that is caused by changing p.) In `Superstep 2` of `mpimv`, a contribution `sum`, pointed to by `psum`, is added by using the operator `MPI_SUM` into `u`[`destindu`[`i`]] on processor `destprocu`[`i`]. This operation is exactly what we need for the fanin.

BSPlib does not have a primitive for the accumulate operation. In `bspmv` we performed the fanin by sending the partial sums to their destination using the `bsp_send` primitive, retrieving them together from the system buffer in the next superstep by using the `bsp_move` primitive. This is a more general approach, less tailored to this specific situation. The `bsp_send` primitive does not have an equivalent in MPI-2. It is, however, a very convenient way of getting rid of data by sending them to another processor, not caring about how every individual message is received. This resembles the way we send messages by regular mail, with daily synchronization (if you are lucky) by the postal service. The one-sided send primitive may perhaps be a nice candidate for inclusion in a possible MPI-3 standard.

```
#include "mpiedupack.h"

void mpimv(int p, int s, int n, int nz, int nrows, int ncols,
           double *a, int *inc,
           int *srcprocv, int *srcindv, int *destprocu, int *destindu,
           int nv, int nu, double *v, double *u){

    /* This function multiplies a sparse matrix A with a
       dense vector v, giving a dense vector u=Av.
    */

    int i, j, *pinc;
    double sum, *psum, *pa, *vloc, *pvloc, *pvloc_end;

    MPI_Win v_win, u_win;

    /****** Superstep 0. Initialize and register ******/
    for(i=0; i<nu; i++)
        u[i]= 0.0;
    vloc= vecallocd(ncols);
    MPI_Win_create(v,nv*SZDBL,SZDBL,MPI_INFO_NULL,MPI_COMM_WORLD,&v_win);
    MPI_Win_create(u,nu*SZDBL,SZDBL,MPI_INFO_NULL,MPI_COMM_WORLD,&u_win);
    /****** Superstep 1. Fanout ******/
    MPI_Win_fence(0, v_win);
    for(j=0; j<ncols; j++)
        MPI_Get(&vloc[j],  1,MPI_DOUBLE,srcprocv[j],
                srcindv[j],1,MPI_DOUBLE,v_win);
    MPI_Win_fence(0, v_win);

    /****** Superstep 2. Local matrix-vector multiplication and fanin */
    MPI_Win_fence(0, u_win);
    psum= &sum;
    pa= a;
    pinc= inc;
    pvloc= vloc;
    pvloc_end= pvloc + ncols;

    pvloc += *pinc;
    for(i=0; i<nrows; i++){
        *psum= 0.0;
        while (pvloc<pvloc_end){
            *psum += (*pa) * (*pvloc);
            pa++;
            pinc++;
            pvloc += *pinc;
        }
        MPI_Accumulate(psum,1,MPI_DOUBLE,destprocu[i],destindu[i],
                           1,MPI_DOUBLE,MPI_SUM,u_win);
        pvloc -= ncols;
    }
    MPI_Win_fence(0, u_win);

    MPI_Win_free(&u_win);
    MPI_Win_free(&v_win);
    vecfreed(vloc);

} /* end mpimv */

void mpimv_init(int p, int s, int n, int nrows, int ncols,
                int nv, int nu, int *rowindex, int *colindex,
                int *vindex, int *uindex, int *srcprocv, int *srcindv,
                int *destprocu, int *destindu){

    /* This function initializes the communication data structure
       needed for multiplying a sparse matrix A with a dense vector v,
       giving a dense vector u=Av.

    */

    int nloc(int p, int s, int n);
    int np, i, j, iglob, jglob, *tmpprocv, *tmpindv, *tmpprocu, *tmpindu;

    MPI_Win tmpprocv_win, tmpindv_win, tmpprocu_win, tmpindu_win;
```

```
/****** Superstep 0. Allocate and register temporary arrays */
np= nloc(p,s,n);
tmpprocv=vecalloci(np);
tmpindv=vecalloci(np);
tmpprocu=vecalloci(np);
tmpindu=vecalloci(np);
MPI_Win_create(tmpprocv,np*SZINT,SZINT,MPI_INFO_NULL,
               MPI_COMM_WORLD,&tmpprocv_win);
MPI_Win_create(tmpindv,np*SZINT,SZINT,MPI_INFO_NULL,
               MPI_COMM_WORLD,&tmpindv_win);
MPI_Win_create(tmpprocu,np*SZINT,SZINT,MPI_INFO_NULL,
               MPI_COMM_WORLD,&tmpprocu_win);
MPI_Win_create(tmpindu,np*SZINT,SZINT,MPI_INFO_NULL,
               MPI_COMM_WORLD,&tmpindu_win);

MPI_Win_fence(0, tmpprocv_win); MPI_Win_fence(0, tmpindv_win);
MPI_Win_fence(0, tmpprocu_win); MPI_Win_fence(0, tmpindu_win);

/****** Superstep 1. Write into temporary arrays ******/
for(j=0; j<nv; j++){
    jglob= vindex[j];
    /* Use the cyclic distribution */
    MPI_Put(&s,1,MPI_INT,jglob%p,jglob/p,1,MPI_INT,tmpprocv_win);
    MPI_Put(&j,1,MPI_INT,jglob%p,jglob/p,1,MPI_INT,tmpindv_win);
}

for(i=0; i<nu; i++){
    iglob= uindex[i];
    MPI_Put(&s,1,MPI_INT,iglob%p,iglob/p,1,MPI_INT,tmpprocu_win);
    MPI_Put(&i,1,MPI_INT,iglob%p,iglob/p,1,MPI_INT,tmpindu_win);
}
MPI_Win_fence(0, tmpprocv_win); MPI_Win_fence(0, tmpindv_win);
MPI_Win_fence(0, tmpprocu_win); MPI_Win_fence(0, tmpindu_win);

/****** Superstep 2. Read from temporary arrays ******/
for(j=0; j<ncols; j++){
    jglob= colindex[j];
    MPI_Get(&srcprocv[j],1,MPI_INT,jglob%p,jglob/p,1,MPI_INT,tmpprocv_win);
    MPI_Get(&srcindv[j], 1,MPI_INT,jglob%p,jglob/p,1,MPI_INT,tmpindv_win);
}
for(i=0; i<nrows; i++){
    iglob= rowindex[i];
    MPI_Get(&destprocu[i],1,MPI_INT,iglob%p,iglob/p,1,MPI_INT,tmpprocu_win);
    MPI_Get(&destindu[i], 1,MPI_INT,iglob%p,iglob/p,1,MPI_INT,tmpindu_win);
}
MPI_Win_fence(0, tmpprocv_win); MPI_Win_fence(0, tmpindv_win);
MPI_Win_fence(0, tmpprocu_win); MPI_Win_fence(0, tmpindu_win);

/****** Superstep 3. Deregister temporary arrays ******/
MPI_Win_free(&tmpindu_win); MPI_Win_free(&tmpprocu_win);
MPI_Win_free(&tmpindv_win); MPI_Win_free(&tmpprocv_win);

/****** Superstep 4. Free temporary arrays ******/
```

```
    vecfreei(tmpindu); vecfreei(tmpprocu);
    vecfreei(tmpindv); vecfreei(tmpprocv);

} /* end mpimv_init */
```

C.3 Performance comparison on an SGI Origin 3800

To compare the performance of the programs from MPIedupack with those of BSPedupack, we performed experiments on Teras, the SGI Origin 3800 computer used in Chapter 3 to test the FFT. We ran the programs for inner product computation (`bspinprod` and `mpiinprod`), LU decomposition (`bsplu` and `mpilu`), FFT (`bspfft` and `mpifft`), and sparse matrix–vector multiplication (`bspmv` and `mpimv`), which all have the same level of optimization for the BSPlib and MPI versions. We excluded the benchmarking programs, because `bspbench` measures pessimistic g-values and `mpibench` optimistic values, making them incomparable. The results for a proper benchmark would resemble those of the FFT, because both `mpibench` and `mpifft` perform their communication by using the `MPI_Alltoallv` primitive.

All programs were compiled using the same MIPSpro C compiler, which is the native compiler of the SGI Origin 3800. The MPI-1 programs have been linked with the MPT 1.6 implementation of MPI. The MPI-2 program `mpimv` has been linked with MPT 1.8, which has just been released and which is the first version to include all three one-sided communications. Although the program `mpimv` is completely legal in MPI-2, we had to allocate the target memory of the one-sided communications by using `MPI_Alloc_mem` from MPI in `mpiedupack.c` instead of `malloc` from C. This is because of a restriction imposed by the MPT implementation. Each experiment was performed three times and the minimum of the three timings was taken, assuming this result suffered the least from other activities on the parallel computer. The problem size was deliberately chosen small, to expose the communication behaviour. (For large problems, computation would be dominant, making differences in communication performance less clearly visible.) The matrix used for the sparse matrix–vector multiplication is `amorph20k` with 100 000 nonzeros.

The timing results of our experiments are presented in Table C.1. The inner product program shows the limits of the BSP model: the amount of communication is small, one superstep performing a $(p-1)$-relation, which is dominated by the global synchronization and the overhead of the superstep. This approach is only viable for a sufficiently large amount of computation, and $n = 100\,000$ is clearly too small. The MPI version makes excellent use of the specific nature of this problem and perhaps of the hardware, leading to good scalability. The LU decomposition shows slightly better performance of BSPlib for $p = 1$ and $p = 2$, indicating lower overhead, but MPI scales better for larger p, achieving a speedup of about 10 on 16 processors. For the FFT, a similar behaviour can be observed, with a largest speedup of about 8 on 16 processors for MPI. The matrix–vector multiplication has been optimized by

TABLE C.1. Time $T_p(n)$ (in ms) of parallel programs from BSPedupack and MPIedupack on p processors of a Silicon Graphics Origin 3800

Program	n	p	BSPlib	MPI
Inner product	100 000	1	4.3	4.3
		2	4.2	2.2
		4	5.9	1.1
		8	9.1	0.6
		16	26.8	0.3
LU decomposition	1000	1	5408	6341
		2	2713	2744
		4	1590	1407
		8	1093	863
		16	1172	555
FFT	262 144	1	154	189
		2	111	107
		4	87	50
		8	41	26
		16	27	19
Matrix–vector	20 000	1	3.8	3.9
		2	11.4	2.7
		4	14.7	6.9
		8	20.8	8.4
		16	18.7	11.0

preventing a processor from sending data to itself. This yields large savings for both versions and eliminates the parallel overhead for $p = 1$. To enable a fair comparison, the buffered get operation in the fanout of the BSPlib version has been replaced by an unbuffered get; the fanin by bulk synchronous message passing remains buffered. The MPI version is completely unbuffered, as it is based on the one-sided MPI-2 primitives, which may partly explain its superior performance. The matrix–vector multiplication has not been optimized to obtain optimistic g-values, in contrast to the LU decomposition and the FFT. The test problem is too small to expect any speedup, as discussed in Section 4.10. The results of both versions can be improved considerably by further optimization.

Overall, the results show that the performance of BSPlib and MPI is comparable, but with a clear advantage for MPI. This may be explained by the fact that the MPI version used is a recent, vendor-supplied implementation, which has clearly been optimized very well. On the other hand, the BSPlib implementation (version 1.4, from 1998) is older and was actually optimized for the SGI Origin 2000, a predecessor of the Origin 3800. No adjustment was needed when installing the software, but no fine-tuning was done either.

Other experiments comparing BSPlib and MPI have been performed on different machines. For instance, the BSPlib version of the LU decomposition from ScaLAPACK by Horvitz and Bisseling [110] on the Cray T3E was found to be 10–15% faster than the original MPI version. Parallel Templates by Koster [124] contains a highly optimized version of our sparse matrix–vector multiplication, both in BSPlib and in MPI-1 (using `MPI_Alltoallv`). Koster reports close results on a T3E with a slight advantage for MPI.

C.4 Where BSP meets MPI

Almost every parallel computer these days comes equipped with an implementation of MPI-1. Sometimes, MPI-2 extensions are available as well. For many parallel computers, we can install a public-domain version of BSPlib ourselves, or have it installed by the systems administrator. Which road should we take when programming in bulk synchronous parallel style?

To answer the question, we propose four different approaches, each of which can be recommended for certain situations. The first is the purist approach used in the chapters of this book, writing our programs in BSPlib and installing BSPlib ourselves if needed. The main advantage is ease of use, and automatic enforcement of the BSP style. An important advantage is the impossibility of introducing deadlock in BSPlib programs. For some machines, an efficient implementation is available. For other machines, the BSPlib implementation on top of MPI-1 from the Oxford BSP toolset [103] and the Paderborn University BSP (PUB) library [28,30] can be used. This may be slow, but could be acceptable for development purposes. It is my hope that this book will stimulate the development of more efficient implementations of BSPlib. In particular, it should not be too difficult to implement BSPlib efficiently on top of MPI-2, basing all communications on the one-sided communications introduced by MPI-2. (Optimization by a clever BSPlib system could even lead to faster communication compared with using the underlying MPI-2 system directly.)

The programs from MPIedupack are in general shorter than those in BSPedupack, due to the use of preexisting collective-communication functions in MPI. We have learned in this book to write such functions ourselves in BSPlib, and thus we can design them for every specific purpose. It would be helpful, however, to have a common collection of efficient and well-tested functions available for everybody. A start has been made by the inclusion of so-called level-1 functions in the Oxford BSP toolset [103] and a set of collective communications in the PUB library [28,30]. I would like to encourage readers to write equivalents of MPI collective communications and to make them available under the GNU General Public License. I promise to help in this endeavour.

The second approach is the hybrid approach, writing a single program in BSP style, but expressing all communication both in MPI and BSPlib. The resulting single-source program can then be compiled conditionally. The conditionally compiled statements can for instance be complete functions, such as the redistribution function of the FFT:

```
#ifdef MPITARGET
    mpiredistr(x,n,p,s,c0,c,rev,rho_p);
#else
    bspredistr(x,n,p,s,c0,c,rev,rho_p);
#endif
```

The compilation command would then have a flag `-DMPITARGET` if compilation using MPI is required. The default would be using BSPlib. (If desired, the preference can easily be reversed.) This can also be applied at a lower level:

```
#ifdef MPITARGET
    MPI_Barrier(MPI_COMM_WORLD);
    time= MPI_Wtime();
#else
    bsp_sync();
    time= bsp_time();
#endif
```

The hybrid approach keeps one source file, and has the advantage that it allows choosing the fastest implementation available on the machine used, either BSPlib or MPI. The price to be paid is an increase in the amount of program text, but as we saw in the conversion the differences between the BSPedupack programs and the MPIedupack programs are often limited, and hence the number of extra lines of program text in a hybrid program is expected to be limited. Based on my own experience with the conversion described above, the main differences are in the I/O parts of the programs, and in communication parts that are well-isolated because of the structured approach inherent in the bulk synchronous parallel style. An additional advantage of the hybrid approach is that it encourages programming in this style also in the MPI part of programs, where the temptation of using matching send/receive pairs always lures.

The third approach is to develop program usings BSPlib, and when the need arises convert them to MPI-2. We have seen how this can be done for the programs from BSPedupack. To give an idea, it took me about a week (human processing time) to convert the whole of BSPedupack to MPI, including all driver programs, and to compile and test the resulting programs. After having read this appendix, a similar task should take you less time. The extra human time incurred by having to convert the final result to MPI is compensated for by the quicker development of the original program. (If, however, you have to develop many collective communications yourself in BSPlib, this represents an additional time investment compared with MPI.)

The fourth approach is to program directly in MPI-2, using collective communications where possible, and keeping the lessons learned from the BSP model in mind. This approach probably works best after having obtained some experience with BSPlib.

The strength of MPI is its wide availability and broad functionality. You can do almost anything in MPI, except cooking dinner. The weakness of MPI is its sheer size: the full standard [137,138] needs 550 pages, which is much more than the 34 pages of the BSPlib standard [105]. This often leads developers of system software to implementing only a subset of the MPI primitives, which harms portability. It also forces users to learn only a subset of the primitives, which makes it more difficult to read programs written by others, since different programmers will most likely choose a different subset. Every implemented MPI primitive is likely to be optimized independently, with a varying rate of success. This makes it impossible to develop a uniform cost model that realistically reflects the performance of every primitive. In contrast, the small size of BSPlib and the underlying cost model provide a better focus to the implementer and make theoretical cost analysis and cost predictions feasible.

A fundamental difference between MPI and BSPlib is that MPI provides more opportunities for optimization by the user, by allowing many different ways to tackle a given programming task, whereas BSPlib provides more opportunities for optimization by the system. For an experienced user, MPI may achieve better results than BSPlib, but for an inexperienced user this may be the reverse.

We have seen that MPI software can be used for programming in BSP style, even though it was not specifically designed for this purpose. Using collective communication wherever possible leads to supersteps and global synchronizations. Puts and gets are available in MPI-2 and can be used in the same way as BSPlib high-performance puts and gets. Still, in using MPI one would miss the imposed discipline provided by BSPlib. A small, paternalistic library such as BSPlib steers programming efforts in the right direction, unlike a large library such as MPI, which allows many different styles of programming and is more tolerant of deviations from the right path.

In this appendix, we have viewed MPI from a BSP perspective, which may be a fresh view for those readers who are already familiar with MPI. We can consider the BSP model as the theoretical cost model behind the one-sided communications of MPI-2. Even though the full MPI-2 standard is not yet available on all parallel machines, its extensions are useful and suitable to the BSP style, giving us another way of writing well-structured parallel programs.

REFERENCES

[1] Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. (1995). A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, **39**, 575–82.

[2] Agarwal, R. C. and Cooley, J. W. (1987). Vectorized mixed radix discrete Fourier transform algorithms. *Proceedings of the IEEE*, **75**, 1283–92.

[3] Aggarwal, A., Chandra, A. K., and Snir, M. (1990). Communication complexity of PRAMs. *Theoretical Computer Science*, **71**, 3–28.

[4] Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. (1997). PLAPACK: Parallel linear algebra package. In *Proceedings Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia.

[5] Alpert, R. D. and Philbin, J. F. (1997, February). cBSP: Zero-cost synchronization in a modified BSP model. Technical Report 97-054, NEC Research Institute, Princeton, NJ.

[6] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J, Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide* (3rd edn). SIAM, Philadelphia.

[7] Ashcraft, C. C. (1990, October). The distributed solution of linear systems using the torus wrap data mapping. Technical Report ECA-TR-147, Boeing Computer Services, Seattle, WA.

[8] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. (ed.) (2000). *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia.

[9] Barnett, M., Gupta, S., Payne, D. G., Shuler, L., van de Geijn, R., and Watts, J. (1994). Building a high-performance collective communication library. In *Proceedings Supercomputing 1994*, pp. 107–116. IEEE Press, Los Alamitos, CA.

[10] Barnett, M., Payne, D. G., van de Geijn, R. A., and Watts, J. (1996). Broadcasting on meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, **35**, 111–22.

[11] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia.

[12] Barriuso, R. and Knies, A. (1994, May). SHMEM user's guide revision 2.0. Technical report, Cray Research Inc., Mendota Heights, MN.

[13] Barros, S. R. M. and Kauranne, T. (1994). On the parallelization of global spectral weather models. *Parallel Computing*, **20**, 1335–56.

[14] Bauer, F. L. (2000). *Decrypted Secrets: Methods and Maxims of Cryptology* (2nd edn). Springer, Berlin.

[15] Bäumker, A., Dittrich, W., and Meyer auf der Heide, F. (1998). Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, **203**, 175–203.

[16] Bays, C. and Durham, S. D. (1976). Improving a poor random number generator. *ACM Transactions on Mathematical Software*, **2**, 59–64.

[17] Bilardi, G., Herley, K. T., Pietracaprina, A., Pucci, G., and Spirakis, P. (1996). BSP vs LogP. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 25–32. ACM Press, New York.

[18] Bilderback, M. L. (1999). Improving unstructured grid application execution times by balancing the edge-cuts among partitions. In *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing* (ed. B. Hendrickson *et al.*). SIAM, Philadelphia.

[19] Bisseling, R. H. (1993). Parallel iterative solution of sparse linear systems on a transputer network. In *Parallel Computation* (ed. A. E. Fincham and B. Ford), Volume 46 of *The Institute of Mathematics and its Applications Conference Series*, pp. 253–71. Oxford University Press, Oxford.

[20] Bisseling, R. H. (1997). Basic techniques for numerical linear algebra on bulk synchronous parallel computers. In *Workshop Numerical Analysis and its Applications 1996* (ed. L. Vulkov, J. Waśniewski, and P. Yalamov), Volume 1196 of *Lecture Notes in Computer Science*, pp. 46–57. Springer, Berlin.

[21] Bisseling, R. H. and McColl, W. F. (1993, December). Scientific computing on bulk synchronous parallel architectures. Preprint 836, Department of Mathematics, Utrecht University, Utrecht, the Netherlands.

[22] Bisseling, R. H. and McColl, W. F. (1994). Scientific computing on bulk synchronous parallel architectures. In *Technology and Foundations: Information Processing '94, Vol. 1* (ed. B. Pehrson and I. Simon), Volume 51 of *IFIP Transactions A*, pp. 509–14. Elsevier Science, Amsterdam.

[23] Bisseling, R. H. and van de Vorst, J. G. G. (1989). Parallel LU decomposition on a transputer network. In *Parallel Computing 1988* (ed. G. A. van Zee and J. G. G. van de Vorst), Volume 384 of *Lecture Notes in Computer Science*, pp. 61–77. Springer, Berlin.

[24] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997*a*). ScaLAPACK: A linear algebra library for message-passing computers. In *Proceedings Eighth SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, Philadelphia.

[25] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997*b*). *ScaLAPACK Users' Guide*. SIAM, Philadelphia.

[26] Boisvert, R. F., Pozo, R., Remington, K., Barrett, R. F., and Dongarra, J. J. (1997). Matrix Market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement* (ed. R. F. Boisvert), pp. 125–37. Chapman and Hall, London.

[27] Bongiovanni, G., Corsini, P., and Frosini, G. (1976). One-dimensional and two-dimensional generalized discrete Fourier transforms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **ASSP-24**, 97–9.

[28] Bonorden, O., Dynia, M., Gehweiler, J., and Wanka, R. (2003, July). PUB-library, release 8.1-pre, user guide and function reference. Technical report, Heinz Nixdorf Institute, Department of Computer Science, Paderborn University, Paderborn, Germany.

[29] Bonorden, O., Hüppelshäuser, N., Juurlink, B., and Rieping, I. (2000, June). The Paderborn University BSP (PUB) library on the Cray T3E. Project report, Heinz Nixdorf Institute, Department of Computer Science, Paderborn University, Paderborn, Germany.

[30] Bonorden, O., Juurlink, B., von Otte, I., and Rieping, I. (2003). The Paderborn University BSP (PUB) library. *Parallel Computing*, **29**, 187–207.

[31] Bracewell, R. N. (1999). *The Fourier Transform and its Applications* (3rd edn). McGraw-Hill Series in Electrical Engineering. McGraw-Hill, New York.

[32] Brent, R. P. (1975). Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic Computational Complexity* (ed. J. F. Traub), pp. 151–76. Academic Press, New York.

[33] Briggs, W. L. and Henson, V. E. (1995). *The DFT: An Owner's Manual for the Discrete Fourier Transform*. SIAM, Philadelphia.

[34] Bui, T. N. and Jones, C. (1993). A heuristic for reducing fill-in in sparse matrix factorization. In *Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 445–52. SIAM, Philadelphia.

[35] Caldwell, A. E., Kahng, A. B., and Markov, I. L. (2000). Improved algorithms for hypergraph bipartitioning. In *Proceedings Asia and South Pacific Design Automation Conference*, pp. 661–6. ACM Press, New York.

[36] Çatalyürek, Ü. V. and Aykanat, C. (1996). Decomposing irregularly sparse matrices for parallel matrix–vector multiplication. In *Proceedings Third International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 1996)* (ed. A. Ferreira, J. Rolim, Y. Saad, and T. Yang), Volume 1117 of *Lecture Notes in Computer Science*, pp. 75–86. Springer, Berlin.

[37] Çatalyürek, Ü. V. and Aykanat, C. (1999). Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, **10**, 673–93.

[38] Çatalyürek, Ü. V. and Aykanat, C. (2001). A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, p. 118. IEEE Press, Los Alamitos, CA.

[39] Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W., Montgomery, P. L., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, Chris, Putnam, Craig, and Zimmermann, P. (2000). Factorization of a 512-bit RSA modulus. In *Advances in Cryptology: EUROCRYPT 2000* (ed. B. Preneel), Volume 1807 of *Lecture Notes in Computer Science*, pp. 1–18. Springer, Berlin.

[40] Chernoff, H. (1952). A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, **23**, 493–507.

[41] Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W., and Whaley, R. C. (1996). The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, **5**, 173–84.

[42] Chu, E. and George, A. (1987). Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, **5**, 65–74.

[43] Chu, E. and George, A. (2000). *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms.* Computational Mathematics Series. CRC Press, Boca Raton, FL.

[44] Cooley, J. W. (1990). How the FFT gained acceptance. In *A History of Scientific Computing* (ed. S. G. Nash), pp. 133–140. ACM Press, New York.

[45] Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, **19**, 297–301.

[46] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to algorithms* (2nd edn). MIT Press, Cambridge, MA and McGraw-Hill, New York.

[47] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, **28**(7), 1–12.

[48] Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K. (1993). Parallel programming in Split-C. In *Proceedings Supercomputing 1993*, pp. 262–73. IEEE Press, Los Alamitos, CA.

[49] Culler, D. E., Karp, R. M., Patterson, D., Sahay, A., Santos, E. E., Schauser, K. E., Subramonian, R., and von Eicken, T. (1996). LogP: A practical model of parallel computation. *Communications of the ACM*, **39**(11), 78–85.

[50] da Cunha, R. D. and Hopkins, T. (1995). The Parallel Iterative Methods (PIM) package for the solution of systems of linear equations on parallel computers. *Applied Numerical Mathematics*, **19**, 33–50.

[51] Danielson, G. C. and Lanczos, C. (1942). Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. *Journal of the Franklin Institute*, **233**, 365–80, 435–52.

[52] Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, **41**, 909–96.

[53] Davis, T. A. (1994–2003). University of Florida sparse matrix collection. Online collection, `http://www.cise.ufl.edu/research/sparse/matrices`, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL.

[54] de la Torre, P. and Kruskal, C. P. (1992). Towards a single model of efficient computation in real machines. *Future Generation Computer Systems*, **8**, 395–408.

[55] de la Torre, P. and Kruskal, C. P. (1996). Submachine locality in the bulk synchronous setting. In *Euro-Par'96 Parallel Processing. Vol. 2* (ed. L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert), Volume 1124 of *Lecture Notes in Computer Science*, pp. 352–8. Springer, Berlin.

[56] Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM*, **11**, 147–8.

[57] Donaldson, S. R., Hill, J. M. D., and Skillicorn, D. B. (1999). Predictable communication on unpredictable networks: implementing BSP over TCP/IP and UDP/IP. *Concurrency: Practice and Experience*, **11**, 687–700.

[58] Dongarra, J. J. (2003, April). Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, TN. Continuously being updated at `http://www.netlib.org/benchmark/performance.ps`.

[59] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. (1990). A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, **16**, 1–17.

[60] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, **14**, 1–17.

[61] Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. (1998). *Numerical Linear Algebra for High-Performance Computers.* Software, Environments, Tools. SIAM, Philadelphia.

[62] Dubey, A., Zubair, M., and Grosch, C. E. (1994). A general purpose subroutine for fast Fourier transform on a distributed memory parallel machine. *Parallel Computing*, **20**, 1697–1710.

[63] Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct Methods for Sparse Matrices. Monographs on Numerical Analysis.* Oxford University Press, Oxford.

[64] Duff, I. S., Grimes, R. G., and Lewis, J. G. (1989). Sparse matrix test problems. *ACM Transactions on Mathematical Software*, **15**, 1–14.

[65] Duff, I. S., Grimes, R. G., and Lewis, J. G. (1997, September). The Rutherford–Boeing sparse matrix collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France.

[66] Duff, I. S., Heroux, M. A., and Pozo, R. (2002). An overview of the Sparse Basic Linear Algebra Subprograms: the new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software*, **28**, 239–67.

[67] Duff, I. S. and van der Vorst, H. A. (1999). Developments and trends in the parallel solution of linear systems. *Parallel Computing*, **25**, 1931–70.

[68] Edelman, A., McCorquodale, P., and Toledo, S. (1999). The future fast Fourier transform. *SIAM Journal on Scientific Computing*, **20**, 1094–1114.

[69] Fiduccia, C. M. and Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, pp. 175–81. IEEE Press, Los Alamitos, CA.

[70] Foster, I. T. and Worley, P. H. (1997). Parallel algorithms for the spectral transform method. *SIAM Journal on Scientific Computing*, **18**, 806–37.

[71] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. (1988). *Solving Problems on Concurrent Processors: Vol. 1, General Techniques and Regular Problems.* Prentice-Hall, Englewood Cliffs, NJ.

[72] Fraser, D. (1976). Array permutation by index-digit permutation. *Journal of the ACM*, **23**, 298–308.

[73] Frigo, M. and Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 3*, pp. 1381–4. IEEE Press, Los Alamitos, CA.

[74] Gauss, C. F. (1866). Theoria interpolationis methodo nova tractata. In *Carl Friedrich Gauss Werke, Vol. 3*, pp. 265–327. Königlichen Gesellschaft der Wissenschaften, Göttingen, Germany.

[75] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

[76] Geist, G. A., Kohl, J. A., and Papadopoulos, P. M. (1996). PVM and MPI: A comparison of features. *Calculateurs Parallèles*, **8**(2), 137–50.

[77] Geist, G. A. and Romine, C. H. (1988). LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, **9**, 639–49.

[78] Gerbessiotis, A. V. and Valiant, L. G. (1994). Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, **22**, 251–67.

[79] Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations* (3rd edn). Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD.

[80] Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., and Tsantilas, T. (1999). Portable and efficient parallel computing using the BSP model. *IEEE Transactions on Computers*, **48**, 670–89.

[81] Goudreau, M. W., Lang, K., Rao, S. B., and Tsantilas, T. (1995, June). The Green BSP library. Technical Report CS-TR-95-11, Department of Computer Science, University of Central Florida, Orlando, FL.

[82] Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing* (2nd edn). Addison-Wesley, Harlow, UK.

[83] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M. (1998). *MPI: The Complete Reference. Vol. 2, The MPI Extensions.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

[84] Gropp, W., Lusk, E., and Skjellum, A. (1999*a*). *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd edn). MIT Press, Cambridge, MA.

[85] Gropp, W., Lusk, E., and Thakur, R. (1999*b*). *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA.

[86] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, **22**, 789–828.

[87] Gupta, A. and Kumar, V. (1993). The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, **4**, 922–32.

[88] Gupta, S. K. S., Huang, C.-H., Sadayappan, P., and Johnson, R. W. (1994). Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, **4**, 477–88.

[89] Gustavson, F. G. (1972). Some basic techniques for solving sparse systems of linear equations. In *Sparse Matrices and Their Applications* (ed. D. J. Rose and R. A. Willoughby), pp. 41–52. Plenum Press, New York.

[90] Haynes, P. D. and Côté, M. (2000). Parallel fast Fourier transforms for electronic structure calculations. *Computer Physics Communications*, **130**, 130–6.

[91] Hegland, M. (1995). An implementation of multiple and multivariate Fourier transforms on vector processors. *SIAM Journal on Scientific Computing*, **16**, 271–88.

[92] Heideman, M. T., Johnson, D. H., and Burrus, C. S. (1984). Gauss and the history of the fast Fourier transform. *IEEE Acoustics, Speech, and Signal Processing Magazine*, **1**(4), 14–21.

[93] Hendrickson, B. (1998). Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proceedings Fifth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 1998)* (ed. A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng), Volume 1457 of *Lecture Notes in Computer Science*, pp. 218–25. Springer, Berlin.

[94] Hendrickson, B., Jessup, E., and Smith, C. (1999). Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM Journal on Scientific Computing*, **20**, 1132–54.

[95] Hendrickson, B. and Leland, R. (1995). A multilevel algorithm for partitioning graphs. In *Proceedings Supercomputing 1995*. ACM Press, New York.

[96] Hendrickson, B. A., Leland, R., and Plimpton, S. (1995). An efficient parallel algorithm for matrix–vector multiplication. *International Journal of High Speed Computing*, **7**, 73–88.

[97] Hendrickson, B. and Plimpton, S. (1995). Parallel many-body simulations without all-to-all communication. *Journal of Parallel and Distributed Computing*, **27**, 15–25.

[98] Hendrickson, B. A. and Womble, D. E. (1994). The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal on Scientific Computing*, **15**, 1201–26.

[99] Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, **49**, 409–36.

[100] Higham, D. J. and Higham, N. J. (2000). *MATLAB Guide*. SIAM, Philadelphia.

[101] Hill, J. M. D., Crumpton, P. I., and Burgess, D. A. (1996). Theory, practice, and a tool for BSP performance prediction. In *Euro-Par'96 Parallel Processing. Vol. 2* (ed. L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert), Volume 1124 of *Lecture Notes in Computer Science*, pp. 697–705. Springer, Berlin.

[102] Hill, J. M. D., Donaldson, S. R., and Lanfear, T. (1998*a*). Process migration and fault tolerance of BSPlib programs running on networks of workstations. In *Euro-Par'98*, Volume 1470 of *Lecture Notes in Computer Science*, pp. 80–91. Springer, Berlin.

[103] Hill, J. M. D., Donaldson, S. R., and McEwan, A. (1998*b*, September). Installation and user guide for the Oxford BSP toolset (v1.4) implementation of BSPlib. Technical report, Oxford University Computing Laboratory, Oxford, UK.

[104] Hill, J. M. D., Jarvis, S. A., Siniolakis, C. J., and Vasiliev, V. P. (1998*c*). Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *Proceedings Sixth EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pp. 286–92. IEEE Press, Los Alamitos, CA.

[105] Hill, J. M. D., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T., and Bisseling, R. H. (1998*d*). BSPlib: The BSP programming library. *Parallel Computing*, **24**, 1947–80.

[106] Hill, J. M. D. and Skillicorn, D. B. (1997/1998*a*). Lessons learned from implementing BSP. *Future Generation Computer Systems*, **13**, 327–35.

[107] Hill, J. M. D. and Skillicorn, D. B. (1998*b*). Practical barrier synchronisation. In *Proceedings Sixth EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pp. 438–44. IEEE Press, Los Alamitos, CA.

[108] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ.

[109] Hockney, R. W. (1996). *The Science of Computer Benchmarking.* SIAM, Philadelphia.

[110] Horvitz, G. and Bisseling, R. H. (1999). Designing a BSP version of ScaLAPACK. In *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing* (ed. Hendrickson, B. *et al.*). SIAM, Philadelphia.

[111] Inda, M. A. and Bisseling, R. H. (2001). A simple and efficient parallel FFT algorithm using the BSP model. *Parallel Computing*, **27**, 1847–1878.

[112] Inda, M. A., Bisseling, R. H., and Maslen, D. K. (2001). On the efficient parallel computation of Legendre transforms. *SIAM Journal on Scientific Computing*, **23**, 271–303.

[113] JáJá, J. (1992). *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA.

[114] Johnson, J., Johnson, R. W., Padua, D. A., and Xiong, J. (2001). Searching for the best FFT formulas with the SPL compiler. In *Languages and Compilers for Parallel Computing* (ed. S. P. Midkiff, J. E. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng), Volume 2017 of *Lecture Notes in Computer Science*, pp. 112–26. Springer, Berlin.

[115] Johnsson, S. L. (1987). Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, **4**, 133–72.

[116] Juurlink, B. H. H. and Wijshoff, H. A. G. (1996). Communication primitives for BSP computers. *Information Processing Letters*, **58**, 303–10.

[117] Karonis, N. T., Toonen, B., and Foster, I. (2003). MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, **63**, 551–63.

[118] Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**, 359–92.

[119] Karypis, G. and Kumar, V. (1999). Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, **41**, 278–300.

[120] Kernighan, B. W. and Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, **49**, 291–307.

[121] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language* (2nd edn). Prentice-Hall, Englewood Cliffs, NJ.

[122] Knuth, D. E. (1997). *The Art of Computer Programming, Vol. 1, Fundamental Algorithms* (3rd edn). Addison-Wesley, Reading, MA.

[123] Kosloff, R. (1996). Quantum molecular dynamics on grids. In *Dynamics of Molecules and Chemical Reactions* (ed. R. E. Wyatt and J. Z. H. Zhang), pp. 185–230. Marcel Dekker, New York.

[124] Koster, J. H. H. (2002, July). Parallel templates for numerical linear algebra, a high-performance computation library. Master's thesis, Department of Mathematics, Utrecht University, Utrecht, the Netherlands.

[125] Lanczos, C. (1950). An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, **45**, 255–82.

[126] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, **5**, 308–23.

[127] Leforestier, C., Bisseling, R. H., Cerjan, C., Feit, M. D., Friesner, R., Guldberg, A., Hammerich, A., Jolicard, G., Karrlein, W., Meyer, H.-D., Lipkin, N., Roncero, O., and Kosloff, R. (1991). A comparison of different propagation schemes for the time dependent Schrödinger equation. *Journal of Computational Physics*, **94**, 59–80.

[128] Lewis, J. G. and van de Geijn, R. A. (1993). Distributed memory matrix–vector multiplication and conjugate gradient algorithms. In *Proceedings Supercomputing 1993*, pp. 484–92. ACM Press, New York.

[129] Lewis, P. A. W., Goodman, A. S., and Miller, J. M. (1969). A pseudo-random number generator for the System/360. *IBM Systems Journal*, **8**, 136–46.

[130] Loyens, L. D. J. C. and Moonen, J. R. (1994). ILIAS, a sequential language for parallel matrix computations. In *PARLE'94, Parallel Architectures and Languages Europe* (ed. C. Halatsis, D. Maritsas, G. Phylokyprou, and S. Theodoridis), Volume 817 of *Lecture Notes in Computer Science*, pp. 250–261. Springer, Berlin.

[131] Mascagni, M. and Srinivasan, A. (2000). SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, **26**, 436–61.

[132] McColl, W. F. (1993). General purpose parallel computing. In *Lectures on Parallel Computation* (ed. A. Gibbons and P. Spirakis), Volume 4 of *Cambridge International Series on Parallel Computation*, pp. 337–91. Cambridge University Press, Cambridge, UK.

[133] McColl, W. F. (1995). Scalable computing. In *Computer Science Today: Recent Trends and Developments* (ed. J. van Leeuwen), Volume 1000 of *Lecture Notes in Computer Science*, pp. 46–61. Springer, Berlin.

[134] McColl, W. F. (1996*a*). A BSP realisation of Strassen's algorithm. In *Proceedings Third Workshop on Abstract Machine Models for Parallel and Distributed Computing* (ed. M. Kara, J. R. Davy, D. Goodeve, and J. Nash), pp. 43–6. IOS Press, Amsterdam.

[135] McColl, W. F. (1996*b*). Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, **12**, 265–72.

[136] Meesen, W. and Bisseling, R. H. (2003). Balancing communication in parallel sparse matrix–vector multiplication. Preprint, Department of Mathematics, Utrecht University, Utrecht, the Netherlands. In preparation.

[137] Message Passing Interface Forum (1994). MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High-Performance Computing*, **8**, 165–414.

[138] Message Passing Interface Forum (1998). MPI2: A message-passing interface standard. *International Journal of High Performance Computing Applications*, **12**, 1–299.

[139] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, **21**, 1087–92.

[140] Miller, R. (1993). A library for bulk synchronous parallel programming. In *General Purpose Parallel Computing*, pp. 100–108. British Computer Society Parallel Processing Specialist Group, London.

[141] Miller, R. and Reed, J. (1993). The Oxford BSP library users' guide, version 1.0. Technical report, Oxford Parallel, Oxford, UK.

[142] Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms.* Cambridge University Press, Cambridge, UK.

[143] Nagy, J. G. and O'Leary, D. P. (1998). Restoring images degraded by spatially variant blur. *SIAM Journal on Scientific Computing*, **19**, 1063–82.

[144] Narasimha, M. J. and Peterson, A. M. (1978). On the computation of the discrete cosine transform. *IEEE Transactions on Communications*, **COM-26**, 934–6.

[145] Newman, M. E. J. and Barkema, G. T. (1999). *Monte Carlo Methods in Statistical Physics.* Oxford University Press, Oxford.

[146] Nieplocha, J., Harrison, R. J., and Littlefield, R. J. (1996). Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, **10**, 169–89.

[147] Numrich, R. W. and Reid, J. (1998). Co-array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, **17**(2), 1–31.

[148] Ogielski, A. T. and Aiello, W. (1993). Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, **14**, 519–30.

[149] O'Leary, D. P. and Stewart, G. W. (1985). Data-flow algorithms for parallel matrix computations. *Communications of the ACM*, **28**, 840–53.

[150] O'Leary, D. P. and Stewart, G. W. (1986). Assignment and scheduling in parallel matrix factorization. *Linear Algebra and Its Applications*, **77**, 275–99.

[151] Oualline, S. (1993). *Practical C Programming* (2nd edn). Nutshell Handbook. O'Reilly, Sebastopol, CA.

[152] Pacheco, P. S. (1997). *Parallel Programming with MPI.* Morgan Kaufmann, San Fransisco.

[153] Papadimitriou, C. H. and Yannakakis, M. (1990). Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, **19**, 322–8.

[154] Parlett, B. N. (1971). Analysis of algorithms for reflections in bisectors. *SIAM Review*, **13**, 197–208.

[155] Pease, M. C. (1968). An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM*, **15**, 252–64.

[156] Pothen, A., Simon, H. D., and Liou, K.-P. (1990). Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, **11**, 430–52.

[157] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing* (2nd edn). Cambridge University Press, Cambridge, UK.

[158] Romein, J. W. and Bal, H. E. (2002). Awari is solved. *Journal of the International Computer Games Association*, **25**, 162–5.

[159] Rühl, T., Bal, H., Benson, G., Bhoedjang, R., and Langendoen, K. (1996). Experience with a portability layer for implementing parallel programming systems. In *Proceedings 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1477–88. CSREA Press, Athens, GA.

[160] Saad, Y. and Schultz, M. H. (1986). GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, **7**, 856–69.

[161] Salamin, E. (1976). Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, **30**, 565–70.

[162] Shadid, J. N. and Tuminaro, R. S. (1992). Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation. *Concurrency: Practice and Experience*, **4**, 481–97.

[163] Skillicorn, D. B., Hill, J. M. D., and McColl, W. F. (1997). Questions and answers about BSP. *Scientific Programming*, **6**, 249–74.

[164] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI: The Complete Reference. Vol. 1, The MPI Core* (2nd edn). Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

[165] Sorensen, H. V., Burrus, C. S., and Heideman, M. T. (1995). *Fast Fourier Transform Database*. PWS Publishing, Boston.

[166] Spirakis, P. G. (1993). PRAM models and fundamental parallel algorithmic techniques: Part II (randomized algorithms). In *Lectures on Parallel Computation* (ed. A. Gibbons and P. Spirakis), Volume 4 of *Cambridge International Series on Parallel Computation*, pp. 41–66. Cambridge University Press, Cambridge, UK.

[167] Spirakis, P. G. and Gibbons, A. (1993). PRAM models and fundamental parallel algorithmic techniques: Part I. In *Lectures on Parallel Computation* (ed. A. Gibbons and P. Spirakis), Volume 4 of *Cambridge International Series on Parallel Computation*, pp. 19–40. Cambridge University Press, Cambridge, UK.

[168] Sterling, T., Salmon, J., Becker, D. J., and Savarese, D. F. (1999). *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

[169] Stijnman, M. A., Bisseling, R. H., and Barkema, G. T. (2003). Partitioning 3D space for parallel many-particle simulations. *Computer Physics Communications*, **149**, 121–34.

[170] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, **13**, 354–6.

[171] Sunderam, V. S. (1990). PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, **2**, 315–39.

[172] Swarztrauber, P. N. (1987). Multiprocessor FFTs. *Parallel Computing*, **5**, 197–210.

[173] Takken, D. H. J. (2003, April). Implementing BSP on Myrinet. Project report, Department of Physics, Utrecht University, Utrecht, the Netherlands.

[174] Tiskin, A. (1998). The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, **196**, 109–30.

[175] Tuminaro, R. S., Shadid, J. N., and Hutchinson, S. A. (1998). Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, **10**, 229–47.

[176] Valiant, L. G. (1982). A scheme for fast parallel communication. *SIAM Journal on Computing*, **11**, 350–61.

[177] Valiant, L. G. (1989). Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence* (ed. M. Reeve and S. E. Zenith), pp. 15–22. Wiley, Chichester, UK.

[178] Valiant, L. G. (1990*a*). A bridging model for parallel computation. *Communications of the ACM*, **33**(8), 103–11.

[179] Valiant, L. G. (1990*b*). General purpose parallel architectures. In *Handbook of Theoretical Computer Science: Vol. A, Algorithms and Complexity* (ed. J. van Leeuwen), pp. 943–71. Elsevier Science, Amsterdam.

[180] van de Geijn, R. A. (1997). *Using PLAPACK.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

[181] Van de Velde, E. F. (1990). Experiments with multicomputer LU-decomposition. *Concurrency: Practice and Experience*, **2**, 1–26.

[182] van de Vorst, J. G. G. (1988). The formal development of a parallel program performing LU-decomposition. *Acta Informatica*, **26**, 1–17.

[183] van der Stappen, A. F., Bisseling, R. H., and van de Vorst, J. G. G. (1993). Parallel sparse LU decomposition on a mesh network of transputers. *SIAM Journal on Matrix Analysis and Applications*, **14**, 853–79.

[184] van der Vorst, H. A. (1992). Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, **13**, 631–44.

[185] van der Vorst, H. A. (2003). *Iterative Krylov Methods for Large Linear Systems.* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, UK.

[186] van Heukelum, A., Barkema, G. T., and Bisseling, R. H. (2002). DNA electrophoresis studied with the cage model. *Journal of Computational Physics*, **180**, 313–26.

[187] Van Loan, C. (1992). *Computational Frameworks for the Fast Fourier Transform*, Volume 10 of *Frontiers in Applied Mathematics.* SIAM, Philadelphia.

[188] Vastenhouw, B. and Bisseling, R. H. (2002, May). A two-dimensional data distribution method for parallel sparse matrix–vector multiplication. Preprint 1238, Department of Mathematics, Utrecht University, Utrecht, the Netherlands.

[189] Vishkin, U. (1993). Structural parallel algorithmics. In *Lectures on Parallel Computation* (ed. A. Gibbons and P. Spirakis), Volume 4 of *Cambridge International Series on Parallel Computation*, pp. 1–18. Cambridge University Press, Cambridge, UK.

[190] Walshaw, C. and Cross, M. (2000). Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, **26**, 1635–60.

[191] Wilkinson, B. and Allen, M. (1999). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Upper Saddle River, NJ.

[192] Zitney, S. E., Mallya, J., Davis, T. A., and Stadtherr, M. A. (1994). Multifrontal techniques for chemical process simulation on supercomputers. In *Proceedings Fifth International Symposium on Process Systems Engineering, Kyongju, Korea* (ed. E. S. Yoon), pp. 25–30. Korean Institute of Chemical Engineers, Seoul, Korea.

[193] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **IT-23**, 337–43.

[194] Zlatev, Z. (1991). *Computational Methods for General Sparse Matrices*, Volume 65 of *Mathematics and Its Applications.* Kluwer, Dordrecht, the Netherlands.

[195] Zoldi, S., Ruban, V., Zenchuk, A., and Burtsev, S. (1999, January/February). Parallel implementation of the split-step Fourier method for solving nonlinear Schrödinger systems. *SIAM News*, **32**(1), 8–9.

This page intentionally left blank

INDEX