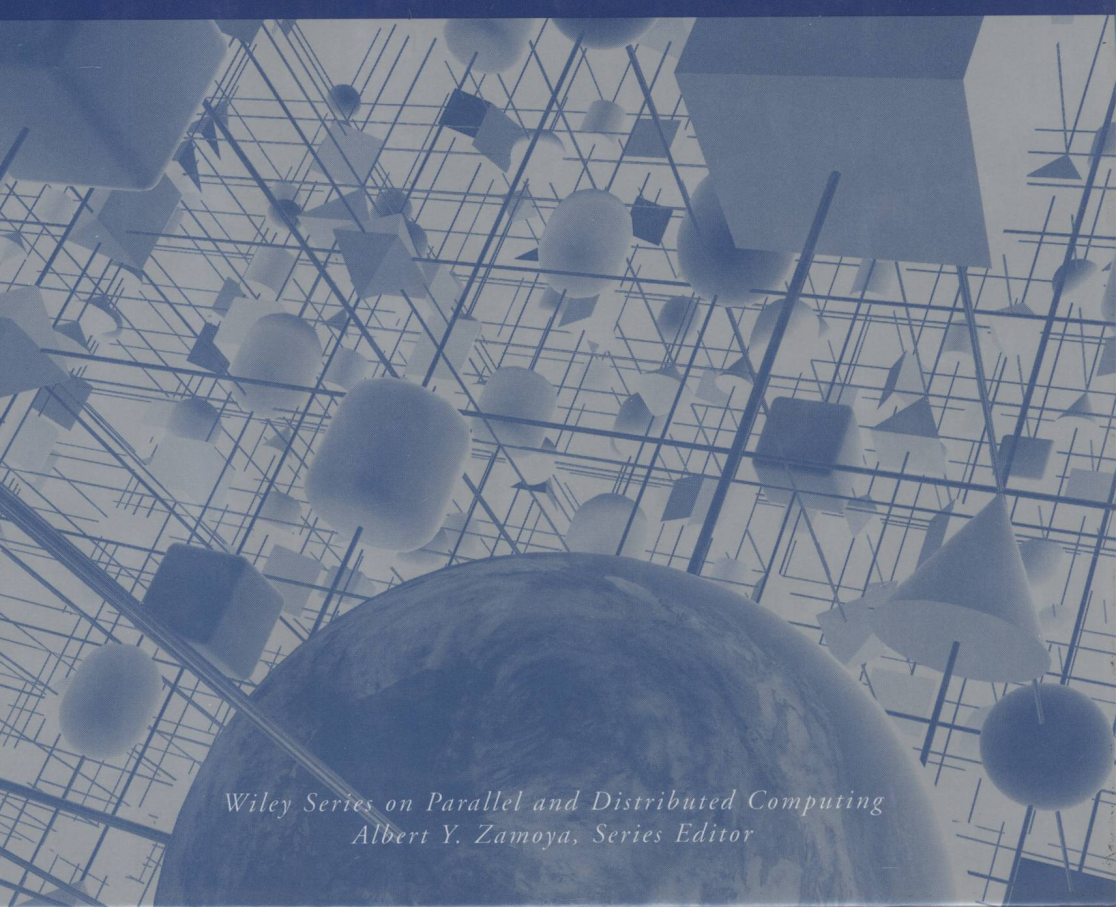
Parallel Computing on Heterogeneous Networks

Alexey L. Lastovetsky



Wiley Series on Parallel and Distributed Computing
Albert Y. Zamoya, Series Editor

PARALLEL COMPUTING ON HETEROGENEOUS NETWORKS

Alexey Lastovetsky
University College, Dublin, Ireland

WILEY-INTERSCIENCE

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-22982-2

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

PARALLEL COMPUTING ON HETEROGENEOUS NETWORKS

WILEY SERIES ON PARALLEL AND DISTRIBUTED COMPUTING

Editor: Albert Y. Zomaya

A complete list of titles in this series appears at the end of this volume.

To my wife Gulnara, my daughters Olga and Oksana, and my parents Leonid and Lyudmila.

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my colleagues and friends Alexey Kalinov, Ilya Ledovskih, Dmitry Arapov, and Mikhail Posypkin for the happy days when we were working together on the mpC programming language. Their skills, devotion, and creativity created the basis for its successful implementation. I am very grateful to Victor Ivannikov for his persistent support of both the mpC project and myself in good and bad times. I am also very grateful to Ted Lewis without whose support and contribution the mpC project would not be possible at all. My special thanks are to Hesham El-Rewini and Albert Zomaya for their positive and encouraging attitude to the idea of this book. I also wish to thank Ravi Reddi for his comments and valuable contribution in the material presented in Sections 8.2, 9.1.1.3, and 9.1.1.4.

CONTENTS

Introduction

The current situation with parallel programming resembles computer programming before the appearance of personal computers. Computing was concentrated in special computer centers, and computer programs were written by nerds. Soon after PCs appeared, computer programming became available to millions of ordinary people. The result of the change can be clearly seen now.

Similarly parallel computing is now concentrating mainly in supercomputer centers established around specialized high-performance parallel computers or clusters of workstations, and only highly trained people write parallel programs for the computer systems. At the same time, local networks of computers have become personal supercomputers available to millions of ordinary people. They only need appropriate programming languages and tools to write fast and portable parallel applications for the networks. The release of the huge performance potential currently hidden in networks of computers might have even a more significant impact on science and technology than the invention of more powerful processors and supercomputers.

The intent of this book is to introduce into the area of parallel computing on common local networks of computers NoCs.

Nowadays NoCs are a common and widespread parallel architecture. In general, a NoC comprises PCs, workstations, servers, and sometimes supercomputers interconnected via mixed communication equipment. Traditional parallel software was developed for homogeneous multiprocessors. It tries to distribute computations evenly over available processors and therefore cannot utilize the performance potential of this heterogeneous architecture. A good parallel program for a NoC should distribute computations and communications over the NoC unevenly, taking into account actual performances of both processors and communication links. This book mainly introduces in parallel programming for heterogeneous, in other words, NoCs in heterogeneous parallel programming.

To present both basic and advanced concepts of heterogeneous parallel programming, the book extensively uses the mpC language. This is a high-level

Parallel Computing on Heterogeneous Networks, by Alexey Lastovetsky
ISBN 0-471-22982-2

language aimed at programming portable parallel computations on NoCs. The design of this language allows easy expression in portable form of a wide range of heterogeneous parallel algorithms. The introduction to the mpC language and the accompanying programming model and language constructs serves also to introduce the area of heterogeneous parallel programming. A representative series of mpC programs was carefully selected to illustrate all of the presented concepts. All of the programs can be compiled and executed on a local network of workstations or even on a single workstation with the freely available mpC programming system installed. While the basic concepts are illustrated by very simple programs, a representative set of real-life problems and their portable parallel solutions on NoCs are also included. The problems studied here involve linear algebra, modeling of oil extraction, integration, *N*-body applications, data mining, business applications, and distributed software testing.

It is important for any book to clearly define from the beginning basic terms, especially if the terms are in common use but are understood differently. This is particularly true in parallel computing on distributed memory architectures, where a specific case is parallel computing on heterogeneous networks. Indeed, it is easy to confuse parallel computing on distributed memory architectures with distributed computing, especially high-performance distributed computing. In a distributed memory computer system both parallel and distributed applications are nothing more than a number of processes running in parallel on different computing nodes and interacting via message passing. They both can use the same communication protocols (e.g., TCP/IP) and the same basic software (e.g., sockets).

The key difference between parallel computing technologies and distributed computing technologies lies in the main goal of each of the technologies. The main goal of distributed computing technologies is to make software components, inherently located on different computers, work together. The main goal of parallel computing technologies is to speed up the solution of a single problem on the available computer hardware. Correspondingly, in the case of parallel computing, the partition of an application into a number of distributed components located on different computers is just a way to speed up its execution on the distributed memory computer system; it is not an intrinsic feature of the application nor of the problem that the application solves. The book presents the technology of heterogeneous parallel computing proceeding from this basic understanding of the main goal of parallel computing technologies.

Thus the target computer hardware is the material basis of any parallel computing technology. Correspondingly the evolution of computer hardware is followed by the evolution of parallel computing technologies. In general, the resulting trajectory of computer hardware is aimed at higher performance, that is, at the ability to compute faster and store more data. There exist two ways to make computer systems execute the same volume of computations

faster: reduce the time of execution of a single instruction (i.e., to increase the processor clock rate), and increase the number of instructions executed in parallel. The first way is determined by the level of microelectronic technology, and it has some natural limits conditioned by universal physical constants such as the velocity of light. Nowadays the clock rates have reached the magnitude of gigacycles per second. But it is not the higher clock rate that distinguishes high-performance computer systems. Rather, at any stage of the development of microprocessor technology, this index is approximately the same for most manufactured microprocessors. The high-performance computer systems can handle a higher parallelism of computations. Therefore high-performance computer systems are always of parallel architecture.

Part I provides an overview of the evolution of parallel computer architectures in relation to the evolution of parallel programming models and tools. The starting-point of all parallel architectures is the serial scalar processor. Main architectural milestones include vector and superscalar processors, a shared memory multiprocessor, a distributed memory multiprocessor, and a common heterogeneous network of computers. The parallel architectures represent the logic of a parallel architecture development rather than its chronology. They represent the main stream of architectural ideas that have proved their viability and effectiveness and made a major impact on the real-life hardware. Compared to a historical approach, the logical approach gives a concise and conceptually clear picture of the evolution of parallel architectures, throwing off a good deal of secondary, nonviable, or simply erroneous architectural decisions, and separating more strictly different architectural concepts often mixed in real hardware. In the series of parallel architectures, each next architecture contains the preceding one as a particular case, and provides more parallelism and, hence, more performance potential.

For each of the listed parallel architectures, its intrinsic model of parallel program is presented and followed by outline of programming tools implementing the model. The models represent all main paradigms of parallel programming. Apart from optimizing compilers for traditional serial programming languages, the programming tools outlined include parallel libraries and parallel programming languages. The book does not pretend to cover all aspects of parallel programming. Many important topics such as debugging of parallel applications and maintenance of fault tolerance of parallel computations are beyond the scope of this book. The book focuses on basic parallel programming models and their implementation by the most popular parallel programming tools.

In Chapter 2 vector and superscalar processors are presented. The architectures provide instruction-level parallelism, which is best exploited by applications with intensive operations on arrays. Such applications can be written in a serial programming language, such as C or Fortran 77, and complied by dedicated optimizing compilers performing some specific loop optimizations. Array libraries allow the programmers to avoid the use of dedicated compil-

ers performing sophisticated optimizations. Instead, the programmers express operations on arrays directly, using calls to carefully implemented subroutines implementing the array operations. Parallel languages, such as Fortran 90 or C[ ], combine advantages of the first and second approaches. They allow the programmer explicitly express operations on arrays, and they therefore do not need to use sophisticated algorithms to recognize parallelized loops. They are able to perform global optimization of combined array operations. Last, unlike existing array libraries, they support general-purpose programming.

In Chapter 3 the shared memory multiprocessor architecture is shown to provide a higher level of parallelism than the vector and superscalar architectures via multiple parallel streams of instructions. Nevertheless, the SMP architecture is not scalable. The speedup provided by this architecture is limited by the bandwidth of the memory bus. Multithreading is the primary programming model for the SMP architecture. Serial languages, such as C and Fortran 77, may be used in concert with optimizing compilers to write efficient programs for SMP computers. Unfortunately, only a limited and simple class of multithreaded algorithms can be implemented in an efficient and portable way by this approach. Thread libraries directly implement the multithreading paradigm and allow the programmers to explicitly write efficient multithreaded programs independent of optimizing compilers. Pthreads are standard for Unix platforms supporting thus efficiently portable parallel programming Unix SMP computers. Thread libraries are powerful tools supporting both parallel and distributed computing. The general programming model underlying the thread libraries is universal and seen too powerful, complicated, and error-prone for parallel programming. OpenMP is a high-level parallel extension of Fortran, C, and C++, providing a simplified multithreaded programming model based on the master/slave design strategy, and aimed specifically at parallel computing on SMP architectures. OpenMP significantly facilitates writing parallel mutlithreaded applications.

In Chapter 4 the distributed memory mutltiprocessor architecture, also known as the MPP architecture, is introduced. It provides much more parallelism than the SMP architecture. Moreover, unlike all other parallel architectures, the MPP architecture is scalable. It means that the speed increase provided by this architecture is potentially infinite. This is due to the absence of principal bottlenecks, such as might limit the number of efficiently interacting processors. Message passing is the dominant programming model for the MPP architecture. As the MPP architecture is farther away from the serial scalar architecture than the vector, superscalar, and even SMP architectures, it is very difficult to automatically generate an efficient message-passing code for the serial source code written in C or Fortran 77. In fact, optimizing C or Fortran 77 compilers for MPPs would involve solving the problem of automatic synthesis of an efficient message-passing program using the source serial code as a specification of its functional semantics. This problem is still a challenge for researchers. Therefore no industrial optimizing C or Fortran 77 compiler for the MPP architecture is now available. Basic programming tools

for MPPs are message-passing libraries and high-level parallel languages. Message-passing libraries directly implement the message-passing paradigm and allow the programmers to explicitly write efficient parallel programs for MPPs. MPI is a standard message-passing interface supporting efficiently portable parallel programming MPPs. Unlike the other popular message-passing library, PVM, MPI supports modular parallel programming and hence can be used for development of parallel libraries. MPI is a powerful programming tool for implementing a wide range of parallel algorithms on MPPs in highly efficient and portable message-passing applications. Scientific programmers, who find the explicit message passing provided by MPI tedious and error-prone, can use data parallel programming languages, mainly HPF, to write programs for MPPs. When programming in HPF, the programmer specifies the strategy for parallelization and data partitioning at a higher level of abstraction, based on the single-threaded data parallel model with a global name space. The tedious low-level details of translating from an abstract global name space to the local memories of individual processors and the management of explicit interprocessor communication are left to the compiler. Data parallel programs are easy to write and debug. However, the data parallel programming model allows the programmer to express only a limited class of parallel algorithms. HPF 2.0 addresses the problem by extending purely data-parallel HPF 1.1 with some task parallel features. The resulting multi-paradigm language is more complicated and not as easy to use as pure data parallel languages. Data parallel languages (i.e., HPF) are difficult to compile. Therefore it is hard to get top performance via data parallel programming. The efficiency of data parallel programs strongly depends on the quality of the compiler.

In Chapter 5 we analyze challenges associated with parallel programming for common networks of computers (NoCs) that are, unlike dedicated parallel computer systems, inherently heterogeneous and unreliable. This analysis results in description of main features of an ideal parallel program running on a NoC. Such a program distributes computations and communications unevenly across processors and communications links during the execution of the code of the program. The distribution may be different for different NoCs and for different executions of the program on the same NoC, depending on the work load of its elements. The program keeps running even if some resources in the executing network fail. In the case of resource failure, it is able to reconfigure itself and resume computations from some point in the past. The program takes into account differences in machine arithmetic on different computers and avoids erroneous behaviour of the program that might be caused by the differences.

Part II is the core of the book and presents the mpC parallel programming language.

In Chapter 6 a basic subset of the mpC language is described. It addresses some primary challenges of heterogeneous parallel computing, focusing on uneven distribution of computations in heterogeneous parallel algorithms, and