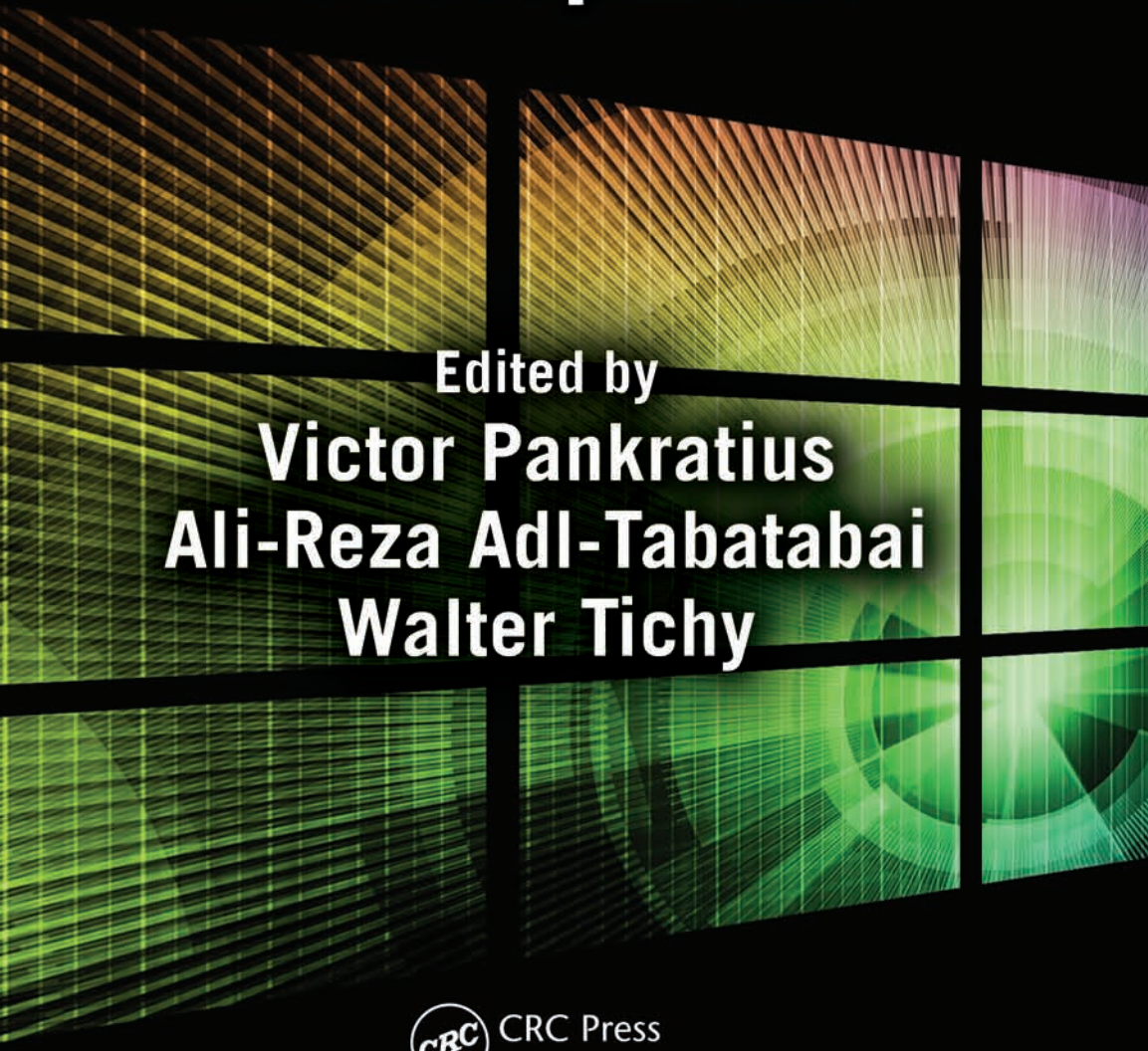
Chapman & Hall/CRC
Computational Science Series

Fundamentals of Multicore Software Development



Edited by
Victor Pankratius
Ali-Reza Adl-Tabatabai
Walter Tichy

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Fundamentals of Multicore Software Development

Chapman & Hall/CRC
Computational Science Series

SERIES EDITOR

Horst Simon
Deputy Director
Lawrence Berkeley National Laboratory
Berkeley, California, U.S.A.

AIMS AND SCOPE

This series aims to capture new developments and applications in the field of computational science through the publication of a broad range of textbooks, reference works, and handbooks. Books in this series will provide introductory as well as advanced material on mathematical, statistical, and computational methods and techniques, and will present researchers with the latest theories and experimentation. The scope of the series includes, but is not limited to, titles in the areas of scientific computing, parallel and distributed computing, high performance computing, grid computing, cluster computing, heterogeneous computing, quantum computing, and their applications in scientific disciplines such as astrophysics, aeronautics, biology, chemistry, climate modeling, combustion, cosmology, earthquake prediction, imaging, materials, neuroscience, oil exploration, and weather forecasting.

PUBLISHED TITLES

PETASCALE COMPUTING: ALGORITHMS AND APPLICATIONS
Edited by David A. Bader

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING
Edited by Michael Alexander and William Gardner

GRID COMPUTING: TECHNIQUES AND APPLICATIONS
Barry Wilkinson

INTRODUCTION TO CONCURRENCY IN PROGRAMMING LANGUAGES
Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen

INTRODUCTION TO SCHEDULING
Yves Robert and Frédéric Vivien

SCIENTIFIC DATA MANAGEMENT: CHALLENGES, TECHNOLOGY, AND DEPLOYMENT
Edited by Arie Shoshani and Doron Rotem

INTRODUCTION TO THE SIMULATION OF DYNAMICS USING SIMULINK®
Michael A. Gray

INTRODUCTION TO HIGH PERFORMANCE COMPUTING FOR SCIENTISTS AND ENGINEERS, **Georg Hager and Gerhard Wellein**

PERFORMANCE TUNING OF SCIENTIFIC APPLICATIONS, **Edited by David Bailey, Robert Lucas, and Samuel Williams**

HIGH PERFORMANCE COMPUTING: PROGRAMMING AND APPLICATIONS
John Levesque with Gene Wagenbreth

PEER-TO-PEER COMPUTING: APPLICATIONS, ARCHITECTURE, PROTOCOLS, AND CHALLENGES
Yu-Kwong Ricky Kwok

FUNDAMENTALS OF MULTICORE SOFTWARE DEVELOPMENT
Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter Tichy

Fundamentals of Multicore Software Development

Edited by
Victor Pankratius
Ali-Reza Adl-Tabatabai
Walter Tichy

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20111031

International Standard Book Number-13: 978-1-4398-1274-7 (eBook - PDF)

Contents

Foreword

Parallel computing is almost as old as computing itself, but until quite recently it had been interesting only to a small cadre of aficionados. Today, the evolution of technology has elevated it to central importance. Many "true believers" including me were certain it was going to be vital to the mainstream of computing eventually. I thought its advent was imminent in 1980. What stalled its arrival was truly spectacular improvements in processor performance, due partly to ever-faster clocks and partly to instruction-level parallelism. This combination of ideas was abetted by increasingly abundant and inexpensive transistors and was responsible for postponing the need to do anything about the von Neumann bottleneck—the requirement that the operations in a program must appear to execute in linear order—until now.

What some of us discovered about parallel computing during the backwater period of the 1980s formed a foundation for our current understanding. We knew that operations that are independent of each other can be performed in parallel, and so dependence became a key target for compiler analysis. Variables, formerly a benign concept thanks to the von Neumann bottleneck, became a major concern for some of us because of the additional constraints needed to make variables work. Heterodox programming models such as synchronous languages and functional programming were proposed to mitigate antidependences and data races. There was a proliferation of parallel languages and compiler technologies aimed at the known world of applications, and very few computational problems escaped the interest and zeal of those of us who wanted to try to run everything in parallel.

Soon thereafter, though, the worlds of parallel servers and high-performance computing emerged, with architectures derived from interconnected workstation or personal computer processors. The old idea of totally rethinking the programming model for parallel systems took a backseat to pragmatism, and existing languages like Fortran and C were augmented for parallelism and pressed into service. Parallel programming got the reputation of being difficult, even becoming a point of pride for those who did it for a living. Nevertheless, this body of practice became parallel computing in its current form and dominated most people's thinking about the subject until recently.

With the general realization that the von Neumann bottleneck has arrived at last, interest in parallelism has exploded. New insights are emerging as the whole field of computing engages with the challenges. For example, we have an emerging understanding of many common patterns in parallel algorithms and can talk about parallel programming from a new point of view. We understand that a transaction, i.e., an isolated atomic update of the set of variables that comprise the domain of an invariant, is the key abstraction needed to maintain the commutativity of variable updates.

Language innovations in C++, Microsoft .NET, and Java have been introduced to support task-oriented as well as thread-oriented programming. Heterogeneous parallel architectures with both GPUs and CPUs can deliver extremely high performance for parallel programs that are able to exploit them.

Now that most of the field is engaged, progress has been exciting. But there is much left to do. This book paints a great picture of where we are, and gives more than an inkling of where we may go next. As we gain broader, more general experience with parallel computing based on the foundation presented here, we can be sure that we are helping to rewrite the next chapter—probably the most significant one—in the amazing history of computing.

Burton J. Smith
Seattle, Washington

Editors

Dr. Victor Pankratius heads the Multicore Software Engineering investigator group at the Karlsruhe Institute of Technology, Karlsruhe, Germany. He also serves as the elected chairman of the Software Engineering for Parallel Systems (SEPARS) international working group. Dr. Pankratius' current research concentrates on how to make parallel programming easier and covers a range of research topics, including auto-tuning, language design, debugging, and empirical studies. He can be contacted at http://www.victorpankratius.com

Ali-Reza Adl-Tabatabai is the director of the Programming Systems Laboratory and a senior principal engineer at Intel Labs. Ali leads a group of researchers developing new programming technologies and their hardware support for future Intel architectures. His group currently develops new language features for heterogeneous parallelism, compiler and runtime support for parallelism, parallel programming tools, binary translation, and hardware support for all these. Ali holds 37 patents and has published over 40 papers in leading conferences and journals. He received his PhD in computer science from Carnegie Mellon University, Pittsburgh, Pennsylvania, and his BSc in computer science and engineering from the University of California, Los Angeles, California.

Walter F. Tichy received his PhD at Carnegie-Mellon University, Pittsburgh, Pennsylvania, in1980, with one of the first dissertations on software architecture. His 1979 paper on the subject received the SIGSOFT Most Influential Paper Award in 1992. In 1980, he joined Purdue University, where he developed the revision control system (RCS), a version management system that is the basis of CVSand has been in worldwide use since the early 1980s. After a year at an AI-startup in Pittsburgh, he returned to his native Germany in 1986, where he was appointed chair of programming systems at the University Karlsruhe (now Karlsruhe Institute of Technology).

He is also a director of FZI, a technology transfer institute.

Professor Tichy's research interests include software engineering and parallel computing. He has pioneered a number of new software tools, such as smart recompilation, analysis of software project repositories, graph editors with automatic layout, automatic configuration inference, and language-aware differencing and merging. He has courted controversy by insisting that software researchers need to test their claims with empirical studies rather than rely on intuition or argumentation. He has conducted controlled experiments testing the influence of type-checking, inheritance depth, design patterns, testing methods, and agile methods on programmer productivity.

Professor Tichy has worked with a number of parallel machines, beginning with C.mmp in the 1970s. In the 1990s, he and his students developed Parastation, a communication and management software for computer clusters that made it to the Top500 list of the world's fastest computers several times (rank 10 as of June 2009). Now that multicore chips make parallel computing available to everyone, he is researching tools and methods to simplify the engineering of general-purpose, parallel software. Race detection, auto-tuning, and high-level languages for expressing parallelism are some of his current research efforts.

Contributors

Ali-Reza Adl-Tabatabai
Intel Corporation
Santa Clara, California

David I. August
Princeton University
Princeton, New Jersey

Judith Bishop
Microsoft Research
Redmond, Washington

Hans Boehm
HP Labs
Palo Alto, California

Barbara Chapman
University of Houston
Houston, Texas

Pradeep Dubey
Intel Labs
Santa Clara, California

Michael Garland
NVIDIA Corporation
Santa Clara, California

Vinod Grover
NVIDIA Corporation
Santa Clara, California

Tim Harris
Microsoft Research
Cambridge, United Kingdom

Jialu Huang
Princeton University
Princeton, New Jersey

Thomas B. Jablin
Princeton University
Princeton, New Jersey

Christoph W. Kessler
Linköping University
Linköping, Sweden

Hanjun Kim
Princeton University
Princeton, New Jersey

James LaGrone
University of Houston
Houston, Texas

Thomas R. Mason
Princeton University
Princeton, New Jersey

Tim Mattson
Intel Corporation
DuPont, Washington

Victor Pankratius
Karlsruhe Institute of Technology
Karlsruhe, Germany

Prakash Prabhu
Princeton University
Princeton, New Jersey

Arun Raman
Princeton University
Princeton, New Jersey

Christoph A. Schaefer
Agilent Technologies
Waldbronn, Germany

Kevin Skadron
University of Virginia
Charlottesville, Virginia

Walter F. Tichy
Karlsruhe Institute of Technology
Karlsruhe, Germany

Barry Wilkinson
University of North Carolina, Charlotte
Charlotte, North Carolina

Yun Zhang
Princeton University
Princeton, New Jersey

Chapter 1

Introduction

Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter F. Tichy

Contents

1.1 Where We Are Today

Multicore chips are about to dramatically change software development. They are already everywhere; in fact, it is difficult to find PCs with a single, main processor. As of this writing, laptops come equipped with two to eight cores. Even smartphones and tablets contain multicore chips. Intel produces chips with 48 cores, Tilera with 100, and Nvidia's graphical processor chips provide several hundred execution units. For major chip manufacturers, multicore has already passed single core in terms of volume shipment. The question for software developers is what to do with this embarrassment of riches.

Ignoring multicore is not an option. One of the reasons is that single processor performance is going to increase only marginally in the future; it might even decrease for lowering energy consumption. Thus, the habit of waiting for the next processor generation to increase application performance no longer works. Future increases of computing power will come from parallelism, and software developers need to embrace parallel programming rather than resist it.

Why did this happen? The current sea change from sequential to parallel processing is driven by the confluence of three events. The first event is the end of exponential growth in single processor performance. This event is caused by our inability to increase clock frequencies without increasing power dissipation. In the past, higher clock speeds could be compensated by lower supply voltages. Since this is no longer

possible, increasing clock speeds would exceed the few hundred watts per chip that can practically be dissipated in mass-market computers as well as the power available in battery-operated mobile devices.

The second event is that parallelism internal to the architecture of a processor has reached a point of diminishing returns. Deeper pipelines, instruction-level parallelism, and speculative execution appear to offer no opportunity to significantly improve performance.

The third event is really a continuing trend: Moore's law projecting an exponential growth in the number of transistors per chip continues to hold. The 2009 *International Technology Roadmap for Semiconductors* (http://www.itrs.net/Links/2009ITRS/Home2009.htm) expects this growth to continue for another 10 years; beyond that, fundamental limits of CMOS scaling may slow growth.

The net result is that hardware designers are using the additional transistors to provide additional cores, while keeping clock rates constant. Some of the extra processors may even be specialized, for example, for encryption, video processing, or graphics. Specialized processors are advantageous in that they provide more performance per watt than general-purpose CPUs. Not only will programmers have to deal with parallelism, but also with heterogeneous instruction sets on a single chip.

1.2 How This Book Helps

This book provides an overview of the current programming choices for multicores, written by the leading experts in the field. Since programmers are the ones that will put the power of multicores to work, it is important to understand the various options and choose the best one for the software at hand.

What are the current choices? All mainstream programming languages (C++, Java, .NET, OpenMP) provide threads and synchronization primitives; these are all covered in this book. Parallel programming patterns such as pipelines and thread pools are built upon these primitives; the book discusses Intel's Threading Building Blocks, Microsoft's Task Parallel Library, and Microsoft's PLINQ, a parallel query language.

Graphics processing units (GPUs) require specialized languages; CUDA is the example chosen here. Combined with a chapter on IBM's Cell, the reader can get an understanding of how heterogeneous multicores are programmed.

As to future choices, additional chapters introduce automatic extraction of parallelism, auto-tuning, and transactional memory. The final chapter provides a survey of future applications of multicores, such as recognition, mining, and synthesis.

Most of today's multicore platforms are shared memory systems. A particular topic is conspicuously absent: distributed computing with message passing. Future multicores may well change from shared memory to distributed memory, in which case additional programming techniques will be needed.

1.3 Audience

This book targets students, researchers, and practitioners interested in parallel programming, as well as instructors of courses in parallelism. The authors present the basics of the various parallel programming models in use today, plus an overview of emerging technologies. The emphasis is on software; hardware is only covered to the extent that software developers need to know.

1.4 Organization

- Part I: Basics of Parallel Programming (Chapters 2 and 3)
- Part II: Programming Languages for Multicore (Chapters 4 through 6)
- Part III: Programming Heterogeneous Processors (Chapters 7 and 8)
- Part IV: Emerging Technologies (Chapters 9 through 12)

1.4.1 Part I: Basics of Parallel Programming

In Chapter 2, Barry Wilkinson presents the fundamentals of multicore hardware and parallel programming that every software developer should know. It also talks about fundamental limitations of sequential computing and presents common classifications of parallel computing platforms and processor architectures. Wilkinson introduces basic notions of processes and threads that are relevant for the understanding of higher-level parallel programming in the following chapters. The chapter also explains available forms of parallelism, such as task parallelism, data parallelism, and pipeline parallelism. Wilkinson concludes with a summary of key insights.

In Chapter 3, Tim Mattson presents how the concept of design patterns can be applied to parallel programming. Design patterns provide common solutions to recurring problems and have been used successfully in mainstream object-oriented programming. Applying patterns to parallel programming helps programmers cope with the complexity by reusing strategies that were successful in the past. Mattson introduces a set of design patterns for parallel programming, which he categorizes into software structure patterns and algorithm strategy patterns. The end of the chapter surveys with work in progress in the pattern community.

1.4.2 Part II: Programming Languages for Multicore

In Chapter 4, Hans Boehm shows how C++, one of the most widely used programming languages, supports parallelism. He describes how C++ started off with platform-dependent threading libraries with ill-defined semantics. He discusses the new C++0x standard that carefully specifies the semantics of parallelism in C++ and the rationale for adding threads directly to the language specification. With these

extensions, parallel programming in C++ becomes less error prone, and C++ implementations become more robust. Boehm's chapter concludes with a comparison of the current standard to earlier standards.

In Chapter 5, Judy Bishop describes parallelism in .NET and Java. The chapter starts with a presentation of .NET. Bishop outlines particular features of the Task Parallel Library (TPL) and the Parallel Language Integrated Queries (PLINQ), a language that allows declarative queries into datasets that execute in parallel. She also presents examples on how to use parallel loops and futures. Then she discusses parallel programming in Java and constructs of the java.util.concurrent library, including thread pools, task scheduling, and concurrent collections. In an outlook, she sketches proposals for Java on fork-join parallelism and parallel array processing.

In Chapter 6, Barbara Chapman and James LaGrone overview OpenMP. The chapter starts with describing the basic concepts of how OpenMP directives parallelize programs in C, C++, and Fortran. Numerous code examples illustrate loop-level parallelism and task-level parallelism. The authors also explain the principles of how an OpenMP compiler works. The chapter ends with possible future extensions of OpenMP.

1.4.3 Part III: Programming Heterogeneous Processors

In Chapter 7, Michael Garland, Vinod Grover, and Kevin Skandron discuss scalable manycore computing with CUDA. They show how throughput-oriented computing can be implemented with GPUs that are installed in most systems along multicore CPUs. The chapter presents the basics of the GPU machine model and how to program GPUs in the CUDA language extensions for C/C++. In particular, the reader is introduced to parallel compute kernel design, synchronization, task coordination, and memory management handling. The chapter also shows detailed programming examples and gives advice for performance optimization.

In Chapter 8, Christoph Kessler discusses programming approaches for the Cell processor, which is a heterogeneous processor built into Sony's PlayStation® 3. Kessler first outlines the hardware architecture of the Cell processor. Programming approaches for the Cell are introduced based on IBM's software development kit. In particular, Kessler discusses constructs for thread coordination, DMA communication, and SIMD parallelization. The chapter also provides an overview of compilers, libraries, tools, as well as relevant algorithms for scientific computing, sorting, image processing, and signal processing. The chapter concludes with a comparison of the Cell processor and GPUs.

1.4.4 Part IV: Emerging Technologies

In Chapter 9, David I. August, Jialu Huang, Thomas B. Jablin, Hanjun Kim, Thomas R. Mason, Prakash Prabhu, Arun Raman, and Yun Zhang introduce techniques for automatic extraction of parallelism from sequential code. The chapter thoroughly describes dependence analysis as a central building block for automatic parallelization. Numerous examples are used to explain compiler auto-parallelization techniques,

such as automatic loop parallelization, speculation, and pipelining techniques. The chapter concludes by discussing the role of auto-parallelization techniques and future developments.

In Chapter 10, Christoph Schaefer, Victor Pankratius, and Walter F. Tichy introduce the basics of automatic performance tuning for parallel applications. The chapter presents a classification of auto-tuning concepts and explains how to design tunable parallel applications. Various techniques are illustrated on how programmers can specify tuning-relevant information for an auto-tuner. The principles of several well-known auto-tuners are compared. An outlook on promising extensions for auto-tuners ends this chapter.

In Chapter 11, Tim Harris presents the basics of the transactional memory programming model, which uses transactions instead of locks. The chapter shows how transactional memory is used in parallel programs and how to implement it in hardware and software. Harris introduces a taxonomy that highlights the differences among existing transactional memory implementations. He also discusses performance issues and optimizations.

In Chapter 12, Pradeep Dubey elaborates on emerging applications that will benefit from multicore systems. He provides a long-term perspective on the most promising directions in the areas of recognition, mining, and synthesis, showing how such applications will be able to take advantage of multicore processors. He also outlines new opportunities for enhanced interactivity in applications and algorithmic opportunities in data-centric applications. Dubey details the implications for multicore software development based on a scalability analysis for various types of applications. The chapter concludes by highlighting the unprecedented opportunities that come with multicore.

This page intentionally left blank

Part I

Basics of Parallel Programming

This page intentionally left blank

Chapter 2

Fundamentals of Multicore Hardware and Parallel Programming

Barry Wilkinson

Contents

2.1 Introduction

In this chapter, we will describe the background to multicore processors, describe their architectures, and lay the groundwork for the remaining of the book on programming these processors. Multicore processors integrate multiple processor cores on the same integrated circuit chip (die), which are then used collectively to achieve higher overall performance. Constructing a system with multiple processors and using them collectively is a rather obvious idea for performance improvement. In fact, it became evident in the early days of computer design as a potential way of increasing the speed of computer systems. A computer system constructed with multiple processors

that are intended to operate together is called a *parallel computer* historically, and programming the processors to operate together is called *parallel programming*. Parallel computers and parallel programming have a long history. The term parallel programming is used by Gill in 1958 (Gill 1958), and his definition of parallel programming is essentially the same as today.

In this chapter, we will first explore the previous work and will start by establishing the limits for performance improvement of processors operating in parallel to satisfy ourselves that there is potential for performance improvement. Then, we will look at the different ways that a system might be constructed with multiple processors. We continue with an outline of the improvements that have occurred in the design of the processors themselves, which have led to enormous increase in the speed of individual processors. These improvements have been so dramatic that the added complexities of parallel computers have limited their use mostly to very high-performance computing in the past. Of course, with improvements in the individual processor, so parallel computers constructed with them also improve proportionately. But programming the multiple processors for collective operation is a challenge, and most demands outside scientific computing have been satisfied with single processor computers, relying on the ever-increasing performance of processors. Unfortunately, further improvements in single processor designs hit major obstacles in the early 2000s, which we will outline. These obstacles led to the multicore approach. We describe architectural designs for a multicore processor and conclude with an outline of the methods for programming multicore systems as an introduction to subsequent chapters on multicore programming.

2.2 Potential for Increased Speed

Since the objective is to use multiple processors collectively for higher performance, before we can explore the different architectures structures one might device, let us first establish the potential for increased speed. The central question is how much faster does the multiprocessor system perform over a single processor system. This can be encapsulated in the speedup factor, $S(p)$, which is defined as

$$S(p) = \frac{\text{Execution time using a single processor (with the best sequential algorithm)}}{\text{Execution time using a multiprocessor system with } p \text{ processors}} = \frac{t_s}{t_p} \tag{2.1}$$

where

t_s is the execution time on a single processor
t_p is the execution time on system with p processors

Typically, the speedup factor is used to evaluate the performance of a parallel algorithm on a parallel computer. In the comparison, we should use the best-known sequential algorithm using a single processor because the parallel algorithm is likely

not to perform as well on a single processor. The execution times could be empirical, that is, measured on real computers. It might be measured by using the Linux time command. Sometimes, one might instrument the code with routines that return wall-clock time, one at the beginning of a section of code to record the start time, and one at the end to record the end time. The elapsed time is the difference. However, this method can be inaccurate as the system usually has other processes executing concurrently in a time-shared fashion. The speedup factor might also be computed theoretically from the number of operations that the algorithms perform. The classical way of evaluating sequential algorithms is by using the time complexity notation, but it is less effective for a parallel algorithm because of uncertainties such as communication times between cooperating parallel processes.

A speedup factor of p with p processors is called *linear speedup*. Conventional wisdom is that the speedup factor should not be greater than p because if a problem is divided into p parts each executed on one processor of a p-processor system and $t_p < t_s/p$, then the same parts could be executed one after the other on a single processor system in time less than t_s. However, there are situations where the speedup factor is greater than p (*superlinear speedup*). The most notable cases are

- When the processors in the multiprocessor system have more memory (cache or main memory) than the single processor system, which provides for increased performance.
- When the multiprocessor has some special feature not present in the single processor system, such as special instructions or hardware accelerators.
- When the algorithm is nondeterministic and happens to provide the solution in one of the parallel parts very quickly, whereas the sequential solution needs to go through many parts to get to the one that has the solution.

The first two cases are not fair hardware comparisons, whereas the last certainly can happen but only with specific problems.

In 1967, Amdahl explored what the maximum speed up would be when a sequential computation is divided into parts and these parts are executed on different processors. This not comparing the best sequential algorithm with a particular parallel algorithm—it is comparing a particular computation mapped onto a single computer and mapped onto a system having multiple processors. Amdahl also assumed that a computation has sections that cannot be divided into parallel parts and these must be performed sequentially on a single processor, and other sections that can be divided equally among the available processors. The sections that cannot be divided into parallel parts would typically be an initialization section of the code and a final section of the code, but there may be several indivisible parts. For the purpose of the analysis, they are lumped together into one section that must be executed sequentially and one section that can be divided into p equal parts and executed in parallel. Let f be the fraction of the whole computation that must be executed sequentially, that is, cannot be divided into parallel parts. Hence, the fraction that can be divided into parts is $1-f$. If the whole computation executed on a single computer in time t_s, ft_s is indivisible

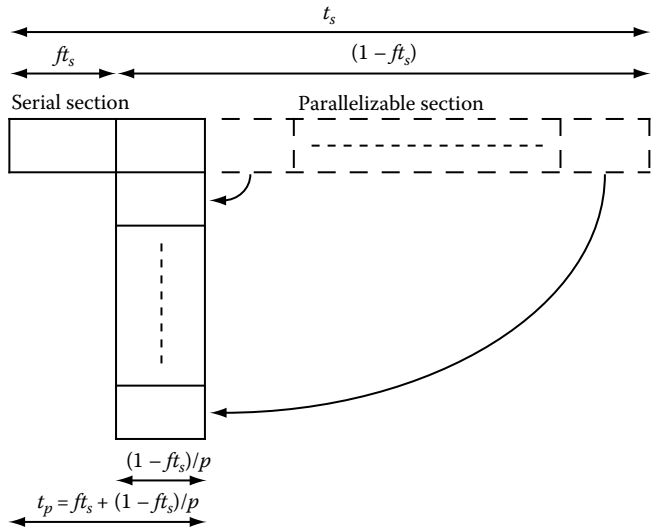


FIGURE 2.1: Amdahl's law.

and $(1 - f)t_s$ is divisible. The ideal situation is when the divisible section is divided equally among the available processors and then this section would be executed in time $(1 - f)t_s/p$ given p processors. The total execution time using p processors is then $ft_s + (1 - f)t_s/p$ as illustrated in Figure 2.1. The speedup factor is given by

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{1}{f + (1-f)/p} = \frac{p}{1 + (p-1)f} \tag{2.2}$$

This famous equation is known as Amdahl's law (Amdahl 1967). The key observation is that as p increases, $S(p)$ tends to and is limited to $1/f$ as p tends to infinity. For example, suppose the sequential part is 5% of the whole. The maximum speed up is 20 irrespective of the number of processors. This is a very discouraging result. Amdahl used this argument to support the design of ultrahigh speed single processor systems in the 1960s.

Later, Gustafson (1988) described how the conclusion of Amdahl's law might be overcome by considering the effect of increasing the problem size. He argued that when a problem is ported onto a multiprocessor system, larger problem sizes can be considered, that is, the same problem but with a larger number of data values. The starting point for Gustafson's law is the computation on the multiprocessor rather than on the single computer. In Gustafson's analysis, the parallel execution time is kept constant, which we assume to be some acceptable time for waiting for the solution. The computation on the multiprocessor is composed of a fraction that is computed sequentially, say f', and a fraction that contains parallel parts, $(1 - f')$.

This leads to Gustafson's so-called *scaled speedup fraction*, $S'(p)$, given by

$$S'(p) = \frac{f' t_p + (1-f')pt_p}{t_p} = p + (1-p)f' \tag{2.3}$$

The fraction, f', is the fraction of the computation on the multiprocessor that cannot be parallelized. This is different to f previously, which is the fraction of the computation on a single computer that cannot be parallelized. The conclusion drawn from Gustafson's law is that it should be possible to get high speedup if we scale up the problem size. For example, if f' is 5%, the scaled speedup computes to 19.05 with 20 processors, whereas with Amdahl's law with $f = 5\%$, the speedup computes to 10.26. Gustafson quotes results obtained in practice of very high speedup close to linear on a 1024-processor hypercube.

Others have explored speedup equations over the years, and it has reappeared with the introduction of multicore processors. Hill and Marty (2008) explored Amdahl's law with different architectural arrangements for multicore processors. Woo and Lee (2008) continued this work by considering Amdahl's law and the architectural arrangements in the light of energy efficiency, a key aspect of multicore processors. We will look at the architectural arrangements for multicore processors later.

2.3 Types of Parallel Computing Platforms

If we accept that it should be worthwhile to use a computer with multiple processors, the next question is how should such a system be constructed. Many parallel computing systems have been designed since the 1960s with various architectures. One thing they have in common is they are stored-program computers. Processors execute instructions from a memory and operate upon data. Flynn (1966) created a classification based upon the number of parallel instruction streams and number of data streams:

- Single instruction stream-single data stream (SISD) computer
- Multiple instruction stream-multiple data stream (MIMD) computer
- Single instruction stream-multiple data stream (SIMD) computer
- Multiple instruction stream-single data stream (MISD) computer

A sequential computer has a single instruction stream processing a single data stream (SISD). A general-purpose multiprocessor system comes under the category of a multiple instruction stream-multiple data stream (MIMD) computer. Each processor has its own instruction stream processing its own data stream. There are classes of problems that can be tackled with a more specialized multiprocessor structure able to perform the same operation on multiple data elements simultaneously. Problems include low-level image processing in which all the picture elements (pixels) need to be altered using the same calculation. Simulations of 2D and 3D structures can involve processing all the elements in the solution space using the same calculation. A single instruction stream-multiple data stream (SIMD) computer is designed specifically for executing such applications efficiently. Instructions are provided to perform a specified operation on an array of data elements simultaneously. The operation might be to

add a constant to each data element, or multiply data elements. In a SIMD computer, there are multiple processing elements but a single program. This type of design is very efficient for the class of problems it addresses, and some very large SIMD computers have been designed over the years, perhaps the first being the Illiac IV in 1972. The SIMD approach was adopted by supercomputer manufacturers, most notably by Cray computers. SIMD computers are sometimes referred to as vector computers as the SIMD instructions operate upon vectors. SIMD computers still need SISD instructions to be able to construct a program.

SIMD instructions can also be incorporated into regular processors for those times that appropriate problems are presented to it. For example, the Intel Pentium series, starting with the Pentium II in 1996, has SIMD instructions, called MMX (Multi-Media eXtension) instructions, for speeding up multimedia applications. This design used the existing floating-point registers to pack multiple data items (eight bytes, four 16-bit numbers, or two 32-bit numbers) that are then operated upon by the same operation. With the introduction of the Pentium III in 1999, Intel added further SIMD instructions, called SSE (Streaming SIMD extension), operating upon eight new 128-bit registers. Intel continued adding SIMD instructions. SSE2 was first introduced with the Pentium 4, and subsequently, SSE3, SSE4, and SSE5 appeared. In 2008, Intel announced AVX (Advanced Vector extensions) operating upon registers extended to 256 bits. Whereas, large SIMD computers vanished in the 1990s because they could not compete with general purpose multiprocessors (MIMD), SIMD instructions still continue for certain applications. The approach can also be found in graphics cards.

General-purpose multiprocessor systems (MIMD computers) can be divided into two types:

1. Shared memory multiprocessor
2. Distributed memory multicomputer

Shared memory multiprocessor systems are a direct extension of single processor system. In a single processor system, the processor accesses a main memory for program instructions and data. In a shared memory multiprocessor system, multiple processors are arranged to have access to a single main memory. This is a very convenient configuration from a programming prospective as then data generated by one processor and stored in the main memory is immediately accessible by other processors. As in a single processor system, cache memory is present to reduce the need to access main memory continually, and commonly two or three levels of cache memory. However, it can be difficult to scale shared memory systems for a large number of processors because the connection to the common memory becomes a bottleneck. There are several possible programming models for a shared memory system. Mostly, they revolve around using threads, which are independent parallel code sequences within a process. We shall look at the thread programming model in more detail later. Multicore processors, at least with a small number of cores, usually employ shared memory configuration.

Distributed memory is an alternative to shared memory, especially for larger systems. In a distributed memory system, each processor has its own main memory and

the processor-memory pair operates as an individual computer. Then, the computers are interconnected. Distributed memory systems have spawned a large number of interconnection networks, especially in the 1970s and 1980s. Notable networks in that era include

- 2D and 3D Mesh networks with computing nodes connected to their nearest neighbor in each direction.
- Hypercube network—a 3D (binary) hypercube is a cube of eight nodes, one at each corner. Each node connects to one other node in each dimension. This construction can be extended to higher dimensions. In an n-dimensional binary hypercube, each node connects to n other nodes, one in each dimension.
- Crossbar switch network—nodes connect to all other nodes, each through a single switch (N^2 switches with N nodes, including switches to themselves).
- Multiple bus network—an extension of a bus architecture in which more than one bus is provided to connect components.
- Tree (switching) network—A network using a tree construction in which the vertices are switches and the nodes are at the leaves of the tree. A path is made through the tree to connect one node to another node.
- Multistage interconnection network—a series of levels of switches make a connection from one node to another node. There are many types of these networks characterized by how the switches are interconnected between levels. Originally, multistage interconnection networks were developed for telephone exchanges. They have since been used in very large computer systems to interconnect processors/computers.

Beginning in the late 1980s, it became feasible to use networked computers as a parallel computing platform. Some early projects used existing networked laboratory computers. In the 1990s, it became cost-effective to interconnect low-cost commodity computers (PCs) with commodity interconnects (Ethernet) to form a high-performance computing cluster, and this approach continues today. The programming model for such a distributed memory system is usually a message-passing model in which messages pass information between the computers. Generally, the programmer inserts message-passing routines in their code. The most widely used suite of message-passing libraries for clusters is MPI. With the advent of multicore computer systems, a cluster of multicore computers can form very high-performance computing platform. Now the programming model for such a cluster may be a hybrid model with threads on each multicore system and message passing between systems. For example, one can use both OpenMP for creating threads and MPI for message passing easily in the same C/C++ program.

Distributed memory computers can also extend to computers that are not physically close. Grid computing refers to a computing platform in which the computers are geographically distributed and interconnected (usually through the Internet) to form a

collaborative resource. Grid computing tends to focus on collaborative computing and resource sharing. For more information on Grid computing, see Wilkinson (2010).

2.4 Processor Design

From the early days of computing, there has been an obvious desire to create the fastest possible processor. Amdahl's law suggested that this would be a better approach than using multiple processors. A central architectural design approach widely adopted to achieve increased performance is by using pipelining. Pipelining involves dividing the processing of an instruction into a series of sequential steps and providing a separate unit within the processor for each step. Speedup comes about when multiple instructions are executed in series. Normally, each unit operates for the same time performing its actions for each instruction as they pass through and the pipeline operates in lock-step synchronous fashion. Suppose there are s pipeline units (stages) and n instructions to process in a series. It takes s steps to process the first instruction. The second instruction completes in the next step, the third in the next step and so on. Hence, n instructions are executed in $s+n-1$ steps, and the speedup compared to a non-pipeline processor is given by

$$s(p) = \frac{sn}{s+n-1} \tag{2.4}$$

assuming the non-pipelined processor must complete all s steps of one instruction before starting the next instruction and the steps take the same time. This is a very approximate comparison as the non-pipelined processor probably can be designed to complete the processing of one instruction in less time than s time steps of the pipelined approach. The speedup will tend to s for large n or tend to n for large s. This suggests that there should be a large number of uninterrupted sequential instructions or a long pipeline. Complex processors such as the Pentium IV can have long pipelines, perhaps up to 22 stages, but long pipelines also incur other problems that have to be addressed such as increased number of dependencies between instructions in the pipeline. Uninterrupted sequences of instructions will depend upon the program and is somewhat limited, but still pipelining is central for high-performance processor design. Pipelining is a very cost-effective solution compared to duplicating the whole processor.

The next development for increased single processor design is to make the processor capable of issuing multiple instructions for execution at the same time using multiple parallel execution units. The general term for such a design is a *superscalar processor*. It relies upon *instruction-level parallelism*, that is, being able to find multiple instructions in the instruction stream that can be executed simultaneously. As one can imagine, the processor design is highly complex, and the performance gains will depend upon how many instructions can actually be executed at the same time. There are other architectural improvements, including register renaming to provide dynamically allocated registers from a pool of registers.

Apart from designs that process a single instruction sequence, increased performance can be achieved by processing instructions from different program sequences switching from one sequence to another. Each sequence is a thread, and the technique is known as *multithreading*. The switching between threads might occur after each instruction (fine-grain multithreading) or when a thread is blocked (coarse-grain multithreading). Fine-grain multithreading suggests that each thread sequence will need its own register file. Interleaving instructions increases the distance of related instructions in pipelines and reduces the effects of instruction dependencies. With advent of multiple-issue processors that have multiple execution units, these execution units can be utilized more fully by processing multiple threads. Such multithreaded processor designs are called *simultaneous multithreading* (*SMT*) because the instructions of different threads are being executed simultaneously using the multiple execution units. Intel calls their version *hyper-threading* and introduced it in versions of the Pentium IV. Intel limited its simultaneous multithreading design to two threads. Performance gains from simultaneous multithreading are somewhat limited, depending upon the application and processor, and are perhaps in the region 10%–30%.

Up to the early 2000s, the approach taken by manufacturers such as Intel was to design a highly complex superscalar processor with techniques for simultaneous operation coupled with using a state-of-the-art fabrication technology to obtain the highest chip density and clock frequency. However, this approach was coming to an end. With clock frequencies reaching almost 4 GHz, technology was not going to provide a continual path upward because of the laws of physics and increasing power consumption that comes with increasing clock frequency and transistor count.

Power consumption of a chip has a static component (leakage currents) and a dynamic component due to switching. Dynamic power consumption is proportional to the clock frequency, the square of the voltage switched, and the capacitive load (Patterson and Hennessy 2009, p. 39). Therefore, each increase in clock frequency will directly increase the power consumption. Voltages have been reduced as a necessary part of decreased feature sizes of the fabrication technology, reducing the power consumption. As the feature size of the chip decreases, the static power becomes more significant and can be 40% of the total power (Asanovic et al. 2006). By the mid-2000s, it had become increasing difficult to limit the power consumption while improving clock frequencies and performance. Patterson calls this the *power wall*.

Wulf and McKee (1995) identified the *memory wall* as caused by the increasing difference between the processor speed and the memory access times. Semiconductor main memory has not kept up with the increasing speed of processors. Some of this can be alleviated by the use of caches and often nowadays multilevel caches, but still it poses a major obstacle. In addition, the *instruction-level parallelism wall* is caused by the increasing difficulty to exploit more parallelism within an instruction sequence. These walls lead to Patterson's "brick wall":

$$\text{Power wall} + \text{Memory wall} + \text{Instruction-Level wall} = \text{Brick wall}$$

for a sequential processor. Hence, enter the multicore approach for using the ever-increasing number of transistors on a chip. Moore's law originally predicted that the number of transistors on an integrated circuit chip would double approximately every

year, later predicting every two years, and sometimes quoted as doubling every 18 months. Now, a prediction is that the number of cores will double every two years or every fabrication technology.

2.5 Multicore Processor Architectures

2.5.1 General

The term multicore processor describes a processor architecture that has multiple independent execution units (cores). (The term *many-core* indicates a large number of cores are present, but we shall simply use the term multicore.) How does a multicore approach get around Patterson's brick wall? With instruction-level parallelism at its limit, we turn to multiple processors to process separate instruction sequences. If we simply duplicated the processors on a chip and all processors operated together, the power would simply increase proportionally and beyond the limits of the chip. Therefore, power consumption must be addressed. The processor cores have to be made more power efficient. One approach used is to reduce the clock frequency, which can result in more than proportional reduction on power consumption. Although reducing the clock frequency will reduce the computational speed, the inclusion of multiple cores provides the potential for increased combined performance if the cores can be used effectively (a big "if"). All multicore designs use this approach. The complexity of each core can be reduced, that is, not use a processor of an extremely aggressive superscalar design. Power can be conserved by switching off parts of the core that are not being used. Temperature sensors can be used to reduce the clock frequency and cut off circuits if the power exceeds limits.

2.5.2 Symmetric Multicore Designs

The most obvious way to design a multicore processor is to replicate identical processor designs on the die (integrated circuit chip) as many times as possible. This is known as a *symmetric multicore* design. Existing processor designs might be used for each core or designs that are based upon existing designs. As we described, the tendency in processor design has been to make processors complex and superscalar for the greatest performance. One could replicate complex superscalar processors on the chip, and companies such as Intel and AMD have followed this approach as their first entry into multicore products, beginning with dual core.

The processor is of course only one part of the overall system design. Memory is required. Main semiconductor memory operates much slower than a processor (the memory wall), partly because of its size and partly because of the dynamic memory design used for high capacity and lower costs. Hence, high-speed cache memory is added near the processor to hold recently used information that can be accessed much more quickly than from the main memory. Cache memory has to operate much faster than the main memory and uses a different circuit design. Speed, capacity, and cost in

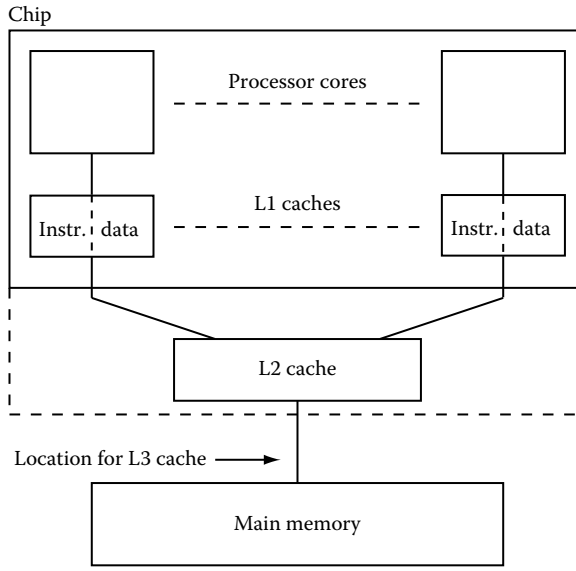


FIGURE 2.2: Symmetric multicore design.

semiconductor are related. As memory capacity is increased on a memory chip, the tendency is for the device to operate slower for given technology in addition to its cost increasing. This leads to a memory hierarchy with multiple levels of cache memory, that is, an L1 cache, an L2 cache, and possibly an L3 cache, between the processor and main memory. Each level of caches is slower and larger than the previous level and usually includes the information held in the previous level (although not necessarily depending upon the design).

Figure 2.2 shows one possible symmetric multicore design in which each processor core has its own L1 cache fabricated on-chip and close to the core and a single shared external L2 between the multicore chip and the main memory. This configuration would be a simple extension of single processor having an on-chip L1 cache. The L2 cache could also be fabricated on the chip given sufficient chip real estate and an example of such a design is the Intel Core Duo, with two cores on one chip (die) and a shared L2 cache. An L3 cache can be placed between the L2 cache and the main memory as indicated in Figure 2.2 There are several possible variations, including having each core have its own L2 cache and groups of cores sharing an L2 cache on-chip. The Intel Core i7, first released in November 2008, is designed for four, six, or eight cores on the same die. Each core has its own data and instruction L1 caches, its own L2 cache and a shared L3 cache, all on the same die. The cores in the Intel Core i7 also use simultaneously multithreading (two threads per core).

The design shown in Figure 2.2 will not currently scale to a very large number of processors if complex processors are used. However, in a symmetric multicore design, each core need not be a high-performance complex superscalar core. An alternative is to use less complex lower-performance lower-power cores but more of them.

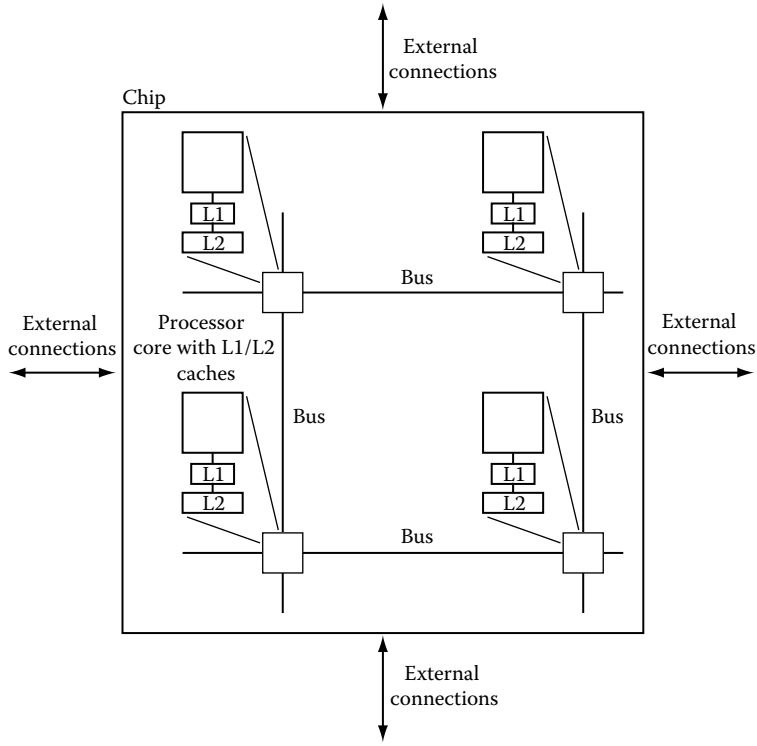


FIGURE 2.3: Symmetric multicore design using a mesh interconnect.

This approach offers the possibility of fabricating more cores onto the chip, although each core might not be as powerful as a high-performance complex superscalar core. Figure 2.3 shows an arrangement using a 2D bus structure to interconnect the cores. Using a large number of less complex lower-performance lower-power cores is often targeted toward a particular market. An example is the picaChip designed for wireless infrastructure and having 250–300 DSP cores. Another example is TILE64 with 64 cores arranged in a 2D array for networking and digital video processing.

2.5.3 Asymmetric Multicore Designs

In an *asymmetric multicore design*, different cores with different functionality are placed on the chip rather than have one uniform core design. Usually, the configuration is to have one fully functional high-performance superscalar core and large number of smaller less powerful but more power-efficient cores, as illustrated in Figure 2.4. These designs are often targeted toward specific applications. An example is the Sony/Toshiba/IBM Cell Broadband Engine Architecture (cell) used in the PlayStation 3 game console. (Asymmetric design is used in Microsoft's Xbox 360 video game console.) Cell processors are combined with dual-core Opteron