

Second Edition

**Eastern
Economy
Edition**

Parallel Computers

Architecture and Programming

V. Rajaraman

C. Siva Ram Murthy



Second Edition

Eastern
Economy
Edition

Parallel Computers

Architecture and Programming

V. Rajaraman

C. Siva Ram Murthy



PARALLEL COMPUTERS
Architecture and Programming
SECOND EDITION

V. RAJARAMAN

Honorary Professor

Supercomputer Education and Research Centre
Indian Institute of Science Bangalore

C. SIVA RAM MURTHY

Richard Karp Institute Chair Professor

Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai

PHI Learning Private Limited

Delhi-110092

2016

PARALLEL COMPUTERS: Architecture and Programming, Second Edition

V. Rajaraman and C. Siva Ram Murthy

© 2016 by PHI Learning Private Limited, Delhi. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.

ISBN-978-81-203-5262-9

The export rights of this book are vested solely with the publisher.

Eleventh Printing (Second Edition) July, 2016

Published by Asoke K. Ghosh, PHI Learning Private Limited, Rimjhim House, 111, Patparganj Industrial Estate, Delhi-110092 and Printed by Mohan Makhijani at Rekha Printers Private Limited, New Delhi-110020.

To
the memory of my dear nephew Dr. M.R. Arun
— V. Rajaraman

To
the memory of my parents, C. Jagannadham and C. Subbalakshmi
— C. Siva Ram Murthy

Table of Contents

Preface

1. Introduction

1.1 WHY DO WE NEED HIGH SPEED COMPUTING?

1.1.1 Numerical Simulation

1.1.2 Visualization and Animation

1.1.3 Data Mining

1.2 HOW DO WE INCREASE THE SPEED OF COMPUTERS?

1.3 SOME INTERESTING FEATURES OF PARALLEL COMPUTERS

1.4 ORGANIZATION OF THE BOOK

EXERCISES

Bibliography

2. Solving Problems in Parallel

2.1 UTILIZING TEMPORAL PARALLELISM

2.2 UTILIZING DATA PARALLELISM

2.3 COMPARISON OF TEMPORAL AND DATA PARALLEL PROCESSING

2.4 DATA PARALLEL PROCESSING WITH SPECIALIZED PROCESSORS

2.5 INTER-TASK DEPENDENCY

2.6 CONCLUSIONS

EXERCISES

Bibliography

3. Instruction Level Parallel Processing

3.1 PIPELINING OF PROCESSING ELEMENTS

3.2 DELAYS IN PIPELINE EXECUTION

3.2.1 Delay Due to Resource Constraints

3.2.2 Delay Due to Data Dependency

3.2.3 Delay Due to Branch Instructions

3.2.4 Hardware Modification to Reduce Delay Due to Branches

3.2.5 Software Method to Reduce Delay Due to Branches

3.3 DIFFICULTIES IN PIPELINING

3.4 SUPERSCALAR PROCESSORS

3.5 VERY LONG INSTRUCTION WORD (VLIW) PROCESSOR

3.6 SOME COMMERCIAL PROCESSORS

3.6.1 ARM Cortex A9 Architecture

3.6.2 Intel Core i7 Processor

3.6.3 IA-64 Processor Architecture

3.7 MULTITHREADED PROCESSORS

3.7.1 Coarse Grained Multithreading

3.7.2 Fine Grained Multithreading

3.7.3 Simultaneous Multithreading

3.8 CONCLUSIONS

EXERCISES

BIBLIOGRAPHY

4. Structure of Parallel Computers

4.1 A GENERALIZED STRUCTURE OF A PARALLEL COMPUTER

4.2 CLASSIFICATION OF PARALLEL COMPUTERS

4.2.1 Flynn's Classification

4.2.2 Coupling Between Processing Elements

4.2.3 Classification Based on Mode of Accessing Memory

4.2.4 Classification Based on Grain Size

4.3 VECTOR COMPUTERS

4.4 A TYPICAL VECTOR SUPERCOMPUTER

4.5 ARRAY PROCESSORS

4.6 SYSTOLIC ARRAY PROCESSORS

4.7 SHARED MEMORY PARALLEL COMPUTERS

4.7.1 Synchronization of Processes in Shared Memory Computers

4.7.2 Shared Bus Architecture

4.7.3 Cache Coherence in Shared Bus Multiprocessor

4.7.4 MESI Cache Coherence Protocol

4.7.5 MOESI Protocol

4.7.6 Memory Consistency Models

4.7.7 Shared Memory Parallel Computer Using an Interconnection Network

4.8 INTERCONNECTION NETWORKS

4.8.1 Networks to Interconnect Processors to Memory or Computers to Computers

4.8.2 Direct Interconnection of Computers

4.8.3 Routing Techniques for Directly Connected Multicomputer Systems

4.9 DISTRIBUTED SHARED MEMORY PARALLEL COMPUTERS

4.9.1 Cache Coherence in DSM

4.10 MESSAGE PASSING PARALLEL COMPUTERS

4.11 Computer Cluster

4.11.1 Computer Cluster Using System Area Networks

4.11.2 Computer Cluster Applications

4.12 Warehouse Scale Computing

4.13 Summary and Recapitulation

EXERCISES

BIBLIOGRAPHY

5. Core Level Parallel Processing

5.1 Consequences of Moore's law and the advent of chip multiprocessors

5.2 A generalized structure of Chip Multiprocessors

5.3 MultiCore Processors or Chip MultiProcessors (CMPs)

5.3.1 Cache Coherence in Chip Multiprocessor

5.4 Some commercial CMPs

[5.4.1 ARM Cortex A9 Multicore Processor](#)
[5.4.2 Intel i7 Multicore Processor](#)
[5.5 Chip Multiprocessors using Interconnection Networks](#)
[5.5.1 Ring Interconnection of Processors](#)
[5.5.2 Ring Bus Connected Chip Multiprocessors](#)
[5.5.3 Intel Xeon Phi Coprocessor Architecture \[2012\]](#)
[5.5.4 Mesh Connected Many Core Processors](#)
[5.5.5 Intel Teraflop Chip \[Peh, Keckler and Vangal, 2009\]](#)
[5.6 General Purpose Graphics Processing Unit \(GPGPU\)](#)

EXERCISES

BIBLIOGRAPHY

[6. Grid and Cloud Computing](#)
[6.1 GRID COMPUTING](#)
[6.1.1 Enterprise Grid](#)
[6.2 Cloud computing](#)
[6.2.1 Virtualization](#)
[6.2.2 Cloud Types](#)
[6.2.3 Cloud Services](#)
[6.2.4 Advantages of Cloud Computing](#)
[6.2.5 Risks in Using Cloud Computing](#)
[6.2.6 What has Led to the Acceptance of Cloud Computing](#)
[6.2.7 Applications Appropriate for Cloud Computing](#)

6.3 CONCLUSIONS

EXERCISES

BIBLIOGRAPHY

[7. Parallel Algorithms](#)
[7.1 MODELS OF COMPUTATION](#)
[7.1.1 The Random Access Machine \(RAM\)](#)
[7.1.2 The Parallel Random Access Machine \(PRAM\)](#)
[7.1.3 Interconnection Networks](#)
[7.1.4 Combinational Circuits](#)
[7.2 ANALYSIS OF PARALLEL ALGORITHMS](#)
[7.2.1 Running Time](#)
[7.2.2 Number of Processors](#)
[7.2.3 Cost](#)
[7.3 PREFIX COMPUTATION](#)
[7.3.1 Prefix Computation on the PRAM](#)
[7.3.2 Prefix Computation on a Linked List](#)
[7.4 SORTING](#)
[7.4.1 Combinational Circuits for Sorting](#)
[7.4.2 Sorting on PRAM Models](#)
[7.4.3 Sorting on Interconnection Networks](#)
[7.5 SEARCHING](#)

7.5.1 Searching on PRAM Models

Analysis

7.5.2 Searching on Interconnection Networks

7.6 MATRIX OPERATIONS

7.6.1 Matrix Multiplication

7.6.2 Solving a System of Linear Equations

7.7 PRACTICAL MODELS OF PARALLEL COMPUTATION

7.7.1 Bulk Synchronous Parallel (BSP) Model

7.7.2 LogP Model

7.8 CONCLUSIONS

EXERCISES

BIBLIOGRAPHY

8. Parallel Programming

8.1 MESSAGE PASSING PROGRAMMING

8.2 MESSAGE PASSING PROGRAMMING WITH MPI

8.2.1 Message Passing Interface (MPI)

8.2.2 MPI Extensions

8.3 SHARED MEMORY PROGRAMMING

8.4 SHARED MEMORY PROGRAMMING WITH OpenMP

8.4.1 OpenMP

8.5 HETEROGENEOUS PROGRAMMING WITH CUDA AND OpenCL

8.5.1 CUDA (Compute Unified Device Architecture)

8.5.2 OpenCL (Open Computing Language)

8.6 PROGRAMMING IN BIG DATA ERA

8.6.1 MapReduce

8.6.2 Hadoop

8.7 CONCLUSIONS

EXERCISES

BIBLIOGRAPHY

9. Compiler Transformations for Parallel Computers

9.1 ISSUES IN COMPILER TRANSFORMATIONS

9.1.1 Correctness

9.1.2 Scope

9.2 TARGET ARCHITECTURES

9.2.1 Pipelines

9.2.2 Multiple Functional Units

9.2.3 Vector Architectures

9.2.4 Multiprocessor and Multicore Architectures

9.3 DEPENDENCE ANALYSIS

9.3.1 Types of Dependences

9.3.2 Representing Dependences

9.3.3 Loop Dependence Analysis

9.3.4 Subscript Analysis

[9.3.5 Dependence Equation](#)
[9.3.6 GCD Test](#)
[9.4 TRANSFORMATIONS](#)
[9.4.1 Data Flow Based Loop Transformations](#)
[9.4.2 Loop Reordering](#)
[9.4.3 Loop Restructuring](#)
[9.4.4 Loop Replacement Transformations](#)
[9.4.5 Memory Access Transformations](#)
[9.4.6 Partial Evaluation](#)
[9.4.7 Redundancy Elimination](#)
[9.4.8 Procedure Call Transformations](#)
[9.4.9 Data Layout Transformations](#)
[9.5 FINE-GRAINED PARALLELISM](#)
[9.5.1 Instruction Scheduling](#)
[9.5.2 Trace Scheduling](#)
[9.5.3 Software Pipelining](#)
[9.6 Transformation Framework](#)
[9.6.1 Elementary Transformations](#)
[9.6.2 Transformation Matrices](#)
[9.7 PARALLELIZING COMPILERS](#)
[9.8 CONCLUSIONS](#)
[EXERCISES](#)
[BIBLIOGRAPHY](#)
[10. Operating Systems for Parallel Computers](#)
[10.1 RESOURCE MANAGEMENT](#)
[10.1.1 Task Scheduling in Message Passing Parallel Computers](#)
[10.1.2 Dynamic Scheduling](#)
[10.1.3 Task Scheduling in Shared Memory Parallel Computers](#)
[10.1.4 Task Scheduling for Multicore Processor Systems](#)
[10.2 PROCESS MANAGEMENT](#)
[10.2.1 Threads](#)
[10.3 Process Synchronization](#)
[10.3.1 Transactional Memory](#)
[10.4 INTER-PROCESS COMMUNICATION](#)
[10.5 MEMORY MANAGEMENT](#)
[10.6 INPUT/OUTPUT \(DISK ARRAYS\)](#)
[10.6.1 Data Striping](#)
[10.6.2 Redundancy Mechanisms](#)
[10.6.3 RAID Organizations](#)
[10.7 CONCLUSIONS](#)
[EXERCISES](#)
[BIBLIOGRAPHY](#)
[11. Performance Evaluation of Parallel Computers](#)

11.1 BASICS OF PERFORMANCE EVALUATION

11.1.1 Performance Metrics

11.1.2 Performance Measures and Benchmarks

11.2 SOURCES OF PARALLEL OVERHEAD

11.2.1 Inter-processor Communication

11.2.2 Load Imbalance

11.2.3 Inter-task Synchronization

11.2.4 Extra Computation

11.2.5 Other Overheads

11.2.6 Parallel Balance Point

11.3 SPEEDUP PERFORMANCE LAWS

11.3.1 Amdahl's Law

11.3.2 Gustafson's Law

11.3.3 Sun and Ni's Law

11.4 SCALABILITY METRIC

11.4.1 Isoefficiency Function

11.5 PERFORMANCE ANALYSIS

11.6 CONCLUSIONS

EXERCISES

BIBLIOGRAPHY

Appendix

Index

Preface

There is a surge of interest today in parallel computing. A general consensus is emerging among professionals that the next generation of processors as well as computers will work in parallel. In fact, all new processors are multicore processors in which several processors are integrated in one chip. It is therefore essential for all students of computing to understand the architecture and programming of parallel computers. This book is an introduction to this subject and is intended for the final year undergraduate engineering students of Computer Science and Information Technology. It can also be used by students of MCA who have an elective subject in parallel computing. Working IT professionals will find this book very useful to update their knowledge about parallel computers and multicore processors.

Chapter 1 is introductory and explains the need for parallel computers. Chapter 2 discusses at length the idea of partitioning a job into many tasks which may be carried out in parallel by several processors. The concept of job partitioning, allocating and scheduling and their importance when attempting to solve problems in parallel is explained in this chapter. In Chapter 3 we deal with instruction level parallelism and how it is used to construct modern processors which constitute the heart of parallel computers as well as multicore processors. Starting with pipelined processors (which use temporal parallelism), we describe superscalar pipelined processors and multithreaded processors.

Chapter 4 introduces the architecture of parallel computers. We start with Flynn's classification of parallel computers. After a discussion of vector computers and array processors, we present in detail the various implementation procedures of MIMD architecture. We also deal with shared memory, CC-NUMA architectures, and the important problem of cache coherence. This is followed by a section on message passing computers and the design of Cluster of Workstations (COWs) and Warehouse Scale parallel computers used in Cloud Computing.

Chapter 5 is a new chapter in this book which describes the use of "Core level parallelism" in the architecture of current processors which incorporate several processors on one semiconductor chip. The chapter begins by describing the developments in both semiconductor technology and processor design which have inevitably led to multicore processors. The limitations of increasing clock speed, instruction level parallelism, and memory size are discussed. This is followed by the architecture of multicore processors designed by Intel, ARM, and AMD. The variety of multicore processors and their application areas are described. In this chapter we have also introduced the design of chips which use hundreds of processors.

Chapter 6 is also new. It describes Grid and Cloud Computing which will soon be used by most organizations for their routine computing tasks. The circumstances which have led to the emergence of these new computing environments, their strengths and weaknesses, and the major differences between grid computing and cloud computing are discussed.

Chapter 7 starts with a discussion of various theoretical models of parallel computers such as PRAM and combinational circuits, which aid in designing and analyzing parallel algorithms. This is followed by parallel algorithms for prefix computation, sorting, searching, and matrix operations. Complexity issues have been always kept in view while developing parallel algorithms. It also presents some practical models of parallel computation such as BSP, Multi-BSP, and LogP.

Chapter 8 is about programming parallel computers. It presents in detail the development of parallel programs for message passing parallel computers using MPI, shared memory parallel computers using OpenMP, and heterogeneous (CPU-GPU) systems using CUDA and OpenCL. This is followed by a simple and powerful MapReduce programming model that enables easy development of scalable parallel programs to process big data on large clusters of commodity machines.

In Chapter 9 we show the importance of compiler transformations to effectively use pipelined processors, vector processors, superscalar processors, multicore processors, and SIMD and MIMD computers. The important topic of dependence analysis is discussed at length. It ends with a discussion of parallelizing compilers.

Chapter 10 deals with the key issues in parallel operating systems—resource (processor) management, process/thread management, synchronization mechanisms including transactional memory, inter-process communication, memory management, and input/output with particular reference to RAID secondary storage system.

The last chapter is on performance evaluation of parallel computers. This chapter starts with a discussion of performance metrics. Various speedup performance laws, namely, Amdahl's law, Gustafson's law and Sun and Ni's law are explained. The chapter ends with a discussion of issues involved in developing tools for measuring the performance of parallel computers.

Designed as a textbook with a number of worked examples, and exercises at the end of each chapter; there are over 200 exercises in all. The book has been classroom tested at the Indian Institute of Science, Bangalore and the Indian Institute of Technology Madras, Chennai. The examples and exercises, together with the References at the end of each chapter, have been planned to enable students to have an extensive as well as an intensive study of parallel computing.

In writing this book, we gained a number of ideas from numerous published papers and books on this subject. We thank all those authors, too numerous to acknowledge individually. Many of our colleagues and students generously assisted us by reading drafts of the book and suggested improvements. Among them we thank Prof. S.K. Nandy and Dr. S. Balakrishnan of Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, Prof. Mainak Chaudhuri of IIT, Kanpur, and Arvind, Babu Shivnath, Bharat Chandra, Manikantan, Rajkarn Singh, Sudeeptha Mishra and Sumant Kowshik of Indian Institute of Technology Madras, Chennai. We thank Ms. T. Mallika of Indian Institute of Science, Bangalore, and Mr. S. Rajkumar, a former project staff member of Indian Institute of Technology Madras, Chennai, for word processing.

The first author thanks the Director, and the Chairman, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, for providing the facilities for writing this book. He also thanks his wife Dharma for proofreading the book and for her support which enabled him to write this book. The second author thanks the members of his family—wife Sharada, son Chandrasekhara Sastry and daughter Sarita—for their love and constant support of his professional endeavors.

We have taken reasonable care in eliminating any errors that might have crept into the book. We will be happy to receive comments and suggestions from readers at our respective email addresses: rajaram@serc.iisc.ernet.in, murthy@iitm.ac.in.

V. Rajaraman
C. Siva Ram Murthy

Introduction

Of late there has been a lot of interest generated all over the world on *parallel processors* and *parallel computers*. This is due to the fact that all current micro-processors are parallel processors. Each processor in a microprocessor chip is called a *core* and such a microprocessor is called a *multicore processor*. Multicore processors have an on-chip memory of a few megabytes (MB). Before trying to answer the question “What is a parallel computer?”, we will briefly review the structure of a single processor computer (Fig. 1.1). It consists of an *input unit* which accepts (or reads) the list of instructions to solve a problem (a *program*) and data relevant to that problem. It has a *memory* or *storage unit* in which the program, data and intermediate results are stored, a *processing element* which we will abbreviate as PE (also called a *Central Processing Unit* (CPU)) which interprets and executes instructions, and an *output unit* which displays or prints the results.

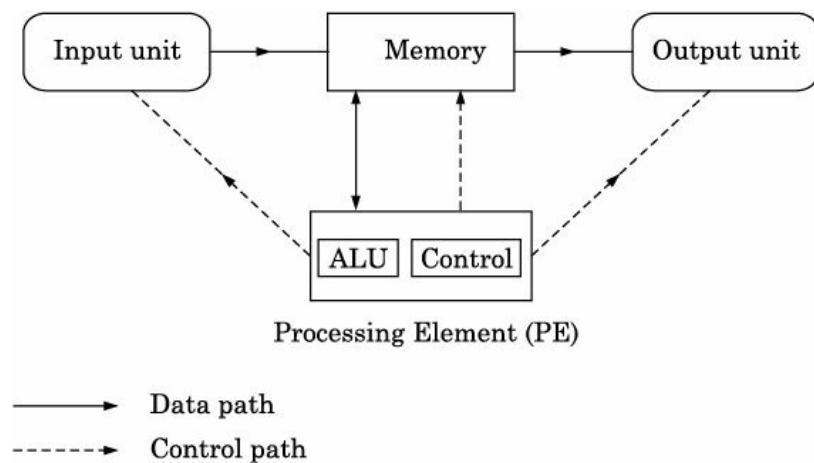


Figure 1.1 Von Neumann architecture computer.

This structure of a computer was proposed by John Von Neumann in the mid 1940s and is known as the *Von Neumann Architecture*. In this architecture, a program is first stored in the memory. The PE retrieves one instruction of this program at a time, interprets it and executes it. The operation of this computer is thus *sequential*. At a time, the PE can execute only one instruction. The speed of this sequential computer is thus limited by the speed at which a PE can retrieve instructions and data from the memory and the speed at which it can process the retrieved data. To increase the speed of processing of data one may increase the speed of the PE by increasing the clock speed. The clock speed increased from a few hundred kHz in the 1970s to 3 GHz in 2005. Processor designers found it difficult to increase the clock speed further as the chip was getting overheated. The number of transistors which could be integrated in a chip could, however, be doubled every two years. Thus, processor designers placed many processing “cores” inside the processor chip to increase its effective throughput. The processor retrieves a sequence of instructions from the main memory and stores them in an on-chip memory. The “cores” can then cooperate to execute these instructions in parallel.

Even though the speed of single processor computers is continuously increasing, problems which are required to be solved nowadays are becoming more complex as we will see in the next section. To further increase the processing speed, many such computers may be interconnected to work cooperatively to solve a problem. A computer which consists of a number

of inter-connected computers which cooperatively execute a single program to solve a problem is called a *parallel computer*. Rapid developments in electronics have led to the emergence of processors which can process over 5 billion instructions per second. Such processors cost only around \$100. It is thus possible to economically construct parallel computers which use around 4000 such multicore processors to carry out ten trillion (10^{13}) instructions per second assuming 50% efficiency.

The more difficult problem is to perceive parallelism in algorithms and develop a software environment which will enable application programs to utilize this potential parallel processing power.

1.1 WHY DO WE NEED HIGH SPEED COMPUTING?

There are many applications which can effectively use computing speeds in the trillion operations per second range. Some of these are:

- Numerical simulation to predict the behaviour of physical systems.
- High performance graphics—particularly visualization, and animation.
- Big data analytics for strategic decision making.
- Synthesis of molecules for designing medicines.

1.1.1 Numerical Simulation

Of late numerical simulation has emerged as an important method in scientific research and engineering design complementing theoretical analysis and experimental observations. Numerical simulation has many advantages. Some of these are:

1. Numerical modelling is versatile. A wide range of problems can be simulated on a computer.
2. It is possible to change many parameters and observe their effects when a system is modelled numerically. Experiments do not allow easy change of many parameters.
3. Numerical simulation is interactive. The results of simulation may be visualized graphically. This facilitates refinement of models. Such refinement provides a better understanding of physical problems which cannot be obtained from experiments.
4. Numerical simulation is cheaper than conducting experiments on physical systems or building prototypes of physical systems.

The role of experiments, theoretical models, and numerical simulation is shown in Fig. 1.2. A theoretically developed model is used to simulate the physical system. The results of simulation allow one to eliminate a number of unpromising designs and concentrate on those which exhibit good performance. These results are used to refine the model and carry out further numerical simulation. Once a good design on a realistic model is obtained, it is used to construct a prototype for experimentation. The results of experiments are used to refine the model, simulate it and further refine the system. This repetitive process is used until a satisfactory system emerges. The main point to note is that experiments on actual systems are not eliminated but the number of experiments is reduced considerably. This reduction leads to substantial cost saving. There are, of course, cases where actual experiments cannot be performed such as assessing damage to an aircraft when it crashes. In such a case simulation is the only feasible method.

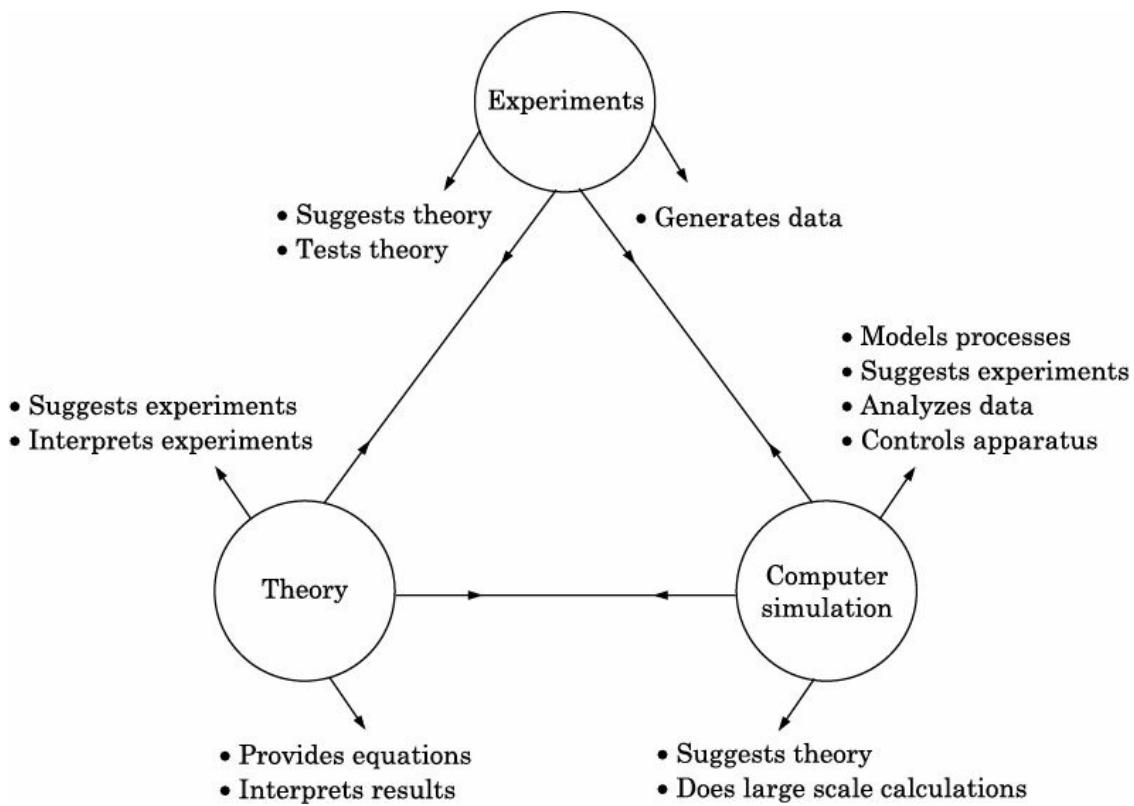


Figure 1.2 Interaction between theory, experiments and computer simulation.

With advances in science and engineering, the models used nowadays incorporate more details. This has increased the demand for computing and storage capacity. For example, to model global weather, we have to model the behaviour of the earth's atmosphere. The behaviour is modelled by partial differential equations in which the most important variables are the wind speed, air temperature, humidity and atmospheric pressure. The objective of numerical weather modelling is to predict the status of the atmosphere at a particular region at a specified future time based on the current and past observations of the values of atmospheric variables. This is done by solving the partial differential equations numerically in regions or grids specified by using lines parallel to the latitude and longitude and using a number of atmospheric layers. In one model (see Fig. 1.3), the regions are demarcated by using 180 latitudes and 360 longitudes (meridian circles) equally spaced around the globe. In the vertical direction 12 layers are used to describe the atmosphere. The partial differential equations are solved by discretizing them to difference equations which are in turn solved as a set of simultaneous algebraic equations. For each region one point is taken as representing the region and this is called a *grid point*. At each grid point in this problem, there are 5 variables (namely air velocity, temperature, pressure, humidity, and time) whose values are stored. The simultaneous algebraic equations are normally solved using an iterative method. In an iterative method several iterations (100 to 1000) are needed for each grid point before the results converge. The calculation of each trial value normally requires around 100 to 500 floating point arithmetic operations. Thus, the total number of floating point operations required for each simulation is approximately given by:

$$\begin{aligned}
 & \text{Number of floating point operations per simulation} \\
 &= \text{Number of grid points} \times \text{Number of values per grid point} \times \text{Number of trials} \times \text{Number of operations per trial}
 \end{aligned}$$

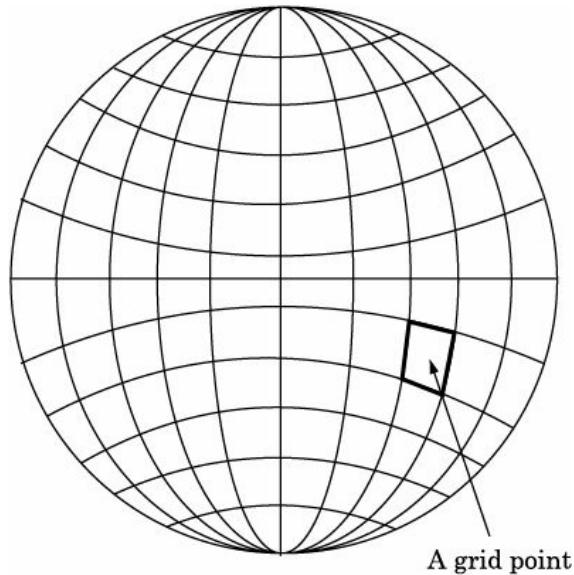


Figure 1.3 Grid for numerical weather model for the Earth.

In this example we have:

$$\text{Number of grid points} = 180 \times 360 \times 12 = 777600$$

$$\text{Number of values per grid point} = 5$$

$$\text{Number of trials} = 500$$

$$\text{Number of operations per trial} = 400$$

Thus, the total number of floating point operations required per simulation = $777600 \times 5 \times 500 \times 400 = 7.77600 \times 10^{11}$. If each floating point operation takes 100 ns the total time taken for one simulation = 7.8×10^4 s = 21.7 h. If we want to predict the weather at the intervals of 6 h there is no point in computing for 21.7 h for a prediction! If we want to simulate this problem, a floating point arithmetic operation on 64-bit operands should be complete within 10 ns. This time is too short for a computer which does not use any parallelism and we need a parallel computer to solve such a problem. In general the complexity of a problem of this type may be described by the formula:

$$\text{Problem complexity} = G \times V \times T \times A$$

where

G = Geometry of the grid system used

V = Variables per grid point

T = Number of steps per simulation for solving the problem

A = Number of floating point operations per step

For the weather modelling problem,

$$G = 777600, V = 5, T = 500 \text{ and } A = 400 \text{ giving problem complexity} = 7.8 \times 10^{11}.$$

There are many other problems whose complexity is of the order of 10^{12} to 10^{20} . For example, the complexity of numerical simulation of turbulent flows around aircraft wings and body is around 10^{15} . Some other areas where numerically intensive simulation is required are:

- Charting ocean currents
- Exploring greenhouse effect and ozone depletion
- Exploration geophysics, in particular, seismology

- Simulation of fusion and fission reactions to design hydrogen and atomic devices
- Designing complex civil and mechanical structures
- Design of drugs by simulating the effect of drugs at the molecular level
- Simulations in solid state and structural chemistry
- Simulation in condensed matter physics
- Analyzing data obtained from the large hadron collider experiment
- Protein folding
- Plate tectonics to forecast earthquakes

The range of applications is enormous and increasing all the time.

The use of computers in numerical simulation is one of the earliest applications of high performance computers. Of late two other problems have emerged whose complexity is in the range of 10^{15} to 10^{18} arithmetic operations. They are called *petascale* and *exascale* computing. We describe them as follows:

1.1.2 Visualization and Animation

In visualization and animation, the results of computation are to be realistically rendered on a high resolution terminal. In this case, the number of area elements where the picture is to be rendered is represented by G . The number of picture elements (called *pixels*) to be processed in each area element is represented by R and the time to process a pixel by T . The computation should be repeated at least 60 times a second for animation. Thus, GR pixels should be processed in 1/60 s. Thus, time to process a pixel = $1/(60 \times G \times R)$. Typically $G = 10^5$, $R = 10^7$. Thus, a pixel should be processed within 10^{-14} s. If N instructions are required to process a pixel then the computer should be able to carry out $N \times 10^{14}$ instructions per second. In general the computational complexity in this case is:

$$G \times R \times P \times N$$

where G represents the complexity of the geometry (i.e., number of area elements in the picture), R the number of pixels per area element, P the number of repetitions per second (for animation) and N the number of instructions needed to process a pixel. This problem has a complexity exceeding 10^{15} .

The third major application requiring intensive computation is data analytics or data mining which we describe next.

1.1.3 Data Mining

There are large databases of the order of peta bytes (10^{15} bytes) which are in the data archives of many organizations. Some experiments such as the Large Hadron Collider (LHC) generates peta and exa bytes of data which are to be analyzed. With the availability of high capacity disks and high speed computers, organizations have been trying to analyze data in the data archive to discover some patterns or rules. Consumer product manufacturers may be able to find seasonal trends in sales of some product or the effect of certain promotional advertisements on the sale of related products from archival data. In general, the idea is to hypothesize a rule relating data elements and test it by retrieving these data elements from the archive. The complexity of this processing may be expressed by the formula:

$$PC = S \times P \times N$$

where S is the size of the database, P the number of instructions to be executed to check a rule and N the number of rules to be checked. In practice the values of these quantities are:

$$S = 10^{15}$$

$$P = 100, N = 10$$

giving a value of PC (Problem Complexity) of 10^{18} . This problem can be solved effectively only if a computer with speeds of 10^{15} instructions per second is available.

These are just three examples of compute intensive problems. There are many others which are emerging such as realistic models of the economy, computer generated movies, and video database search, which require computers which can carry out tens of tera operations per second for their solution.

1.2 HOW DO WE INCREASE THE SPEED OF COMPUTERS?

There are two methods of increasing the speed of computers. One method is to build processing elements using faster semiconductor components and the other is to improve the architecture of computers using the increasing number of transistors available in processing chips. The rate of growth of speed using better device technology has been slow. For example, the basic clock of high performance processors in 1980 was 50 MHz and it reached 3 GHz by 2003. The clock speed could not be increased further without special cooling methods as the chips got overheated. However, the number of transistors which could be packed in a microprocessor continued to double every two years. In 1972, the number of transistors in a microprocessor chip was 4000 and it increased to more than a billion in 2014. The extra transistors available have been used in many ways. One method has put more than one arithmetic unit in a processing unit. Another has increased the size of on-chip memory. The latest is to place more than one processor in a microprocessor chip. The extra processing units are called *processing cores*. The cores share an on-chip memory and work in parallel to increase the speed of the processor. There are also processor architectures which have a network of “cores”, each “core” with its own memory that cooperate to execute a program. The number of cores has been increasing in step with the increase in the number of transistors in a chip. Thus, the number of cores is doubling almost every two years. Soon (2015) there will be 128 simple “cores” in a processor chip. The processor is one of the units of a computer. As we saw at the beginning of this chapter, a computer has other units, namely, memory, and I/O units. We can increase the speed of a computer by increasing the speed of its units and also by improving the architecture of the computer. For example, while the processor is computing, data which may be needed later could be fetched from the main memory and simultaneously an I/O operation can be initiated. Such an overlap of operations is achieved by using both software and hardware features.

Besides overlapping operations of various units of a computer, each processing unit in a chip may be designed to overlap operations of successive instructions. For example, an instruction can be broken up into five distinct tasks as shown in Fig. 1.4. Five successive instructions can be overlapped, each doing one of these tasks (in an assembly line model) using different parts of the CPU. The arithmetic unit itself may be designed to exploit parallelism inherent in the problem being solved. An arithmetic operation can be broken down into several tasks, for example, matching exponents, shifting mantissas and aligning them, adding them, and normalizing. The components of two arrays to be added can be streamed through the adder and the four tasks can be performed simultaneously on four different pairs of operands thereby quadrupling the speed of addition. This method is said to exploit *temporal parallelism* and will be explained in greater detail in the next chapter. Another method is to have four adders in the CPU and add four pairs of operands simultaneously. This type of parallelism is called *data parallelism*. Yet another method of increasing the speed of computation is to organize a set of computers to work simultaneously and cooperatively to carry out tasks in a program.

Instruction 1	Fetch Instruction	Decode Instruction	Execute	Memory Operation	Store Result	
Instruction 2	FI	DE	EX	MEM	SR	
Instruction 3	FI	DE	EX	MEM	SR	
Instruction 4	FI	DE	EX	MEM	SR	
Instruction 5	FI	DE	EX	MEM	SR	

FI: Fetch instruction, DE: Decode, EX: Execute, MEM: Memory operation, SR: Store result

Figure 1.4 The tasks performed by an instruction and overlap of successive instructions.

All the methods described above are called architectural methods which are the ones which have contributed to ten billions fold increase in the speed of computations in the last two decades. We summarize these methods in Table 1.1.

TABLE 1.1 Architectural Methods Used to Increase the Speed of Computers
<ul style="list-style-type: none"> * use parallelism in a single processor computer — Overlap execution of a number of instructions by pipelining, or by using multiple functional units, or multiple processor “cores”. — Overlap operation of different units of a computer. — Increase the speed of arithmetic logic unit by exploiting data and/or temporal parallelism. * use parallelism in the problem to solve it on a parallel computer. — Use number of interconnected computers to work cooperatively to solve the problem.

1.3 SOME INTERESTING FEATURES OF PARALLEL COMPUTERS

Even though higher speed obtainable with parallel computers is the main motivating force for building parallel computers, there are some other interesting features of parallel computers which are not obvious but nevertheless important. These are:

Better quality of solution. When arithmetic operations are distributed to many computers, each one does a smaller number of arithmetic operations. Thus, rounding errors are lower when parallel computers are used.

Better algorithms. The availability of many computers that can work simultaneously leads to different algorithms which are not relevant for purely sequential computers. It is possible to explore different facets of a solution simultaneously using several processors and these give better insight into solutions of several physical problems.

Better storage distribution. Certain types of parallel computing systems provide much larger storage which is distributed. Access to the storage is faster in each computer. This feature is of special interest in many applications such as information retrieval and computer aided design.

Greater reliability. In principle a parallel computer will work even if a processor fails. We can build a parallel computer's hardware and software for better fault tolerance.

1.4 ORGANIZATION OF THE BOOK

This book is organized as follows. In the next chapter we describe various methods of solving problems in parallel. In Chapter 3 we examine the architecture of processors and how instruction level parallelism is exploited in the design of modern microprocessors. We explain the structure of parallel computers and examine various methods of interconnecting processors and how they influence their cooperative functioning in Chapter 4. This is followed by a chapter titled Core Level Parallel Processing. With the improvement of semiconductor technology, it has now become feasible to integrate several billion transistors in an integrated circuit. Consequently a large number of processing elements, called “cores”, may be integrated in a chip to work cooperatively to solve problems. In this chapter we explain the organization of multicore processors on a chip including what are known as General Purpose Graphics Processing Units (GPGPU). Chapter 6 is on the emerging area of Grid and Cloud Computing. We explain how these computer environments in which computers spread all over the world are interconnected and cooperate to solve problems emerged and how they function. We also describe the similarities and differences between grid and cloud computing. The next part of the book concentrates on the programming aspects of parallel computers. Chapter 7 discusses parallel algorithms including prefix computation algorithms, sorting, searching, and matrix algorithms for parallel computers. These problems are natural candidates for use of parallel computers. Programming parallel computers is the topic of the next chapter. We discuss methods of programming different types of parallel machines. It is important to be able to port parallel programs across architectures. The solution to this problem has been elusive. In Chapter 8 four different explicit parallel programming models (a programming model is an abstraction of a computer system that a user sees and uses when developing programs) are described. These are: MPI for programming message passing parallel computers, OpenMP for programming shared memory parallel computers, CUDA and OpenCL for programming GPGPUs, and MapReduce programming for large scale data processing on clouds. Compilers for high level languages are corner stones of computing. It is necessary for compilers to take cognizance of the underlying parallel architecture. Thus in Chapter 9 the important topics of dependence analysis and compiler transformations for parallel computers are discussed. Operating Systems for parallel computers is the topic of Chapter 10. The last chapter is on the evaluation of the performance of parallel computers.

EXERCISES

- 1.1 The website <http://www.top500.org> lists the 500 fastest computers in the world. Find out the top 5 computers in this list. How many processors do each of them use and what type of processors do they use?
- 1.2 LINPACK benchmarks that specify the speed of parallel computers for solving 1000×1000 linear systems of equations can be found in the website <http://performance.netlib.org> and are updated by Jack Dongarra at the University of Tennessee, dongarra@cs.utk.edu. Look this up and compare peak speed of parallel computers listed with their speed.
- 1.3 SPEC marks that specify individual processor's performance are listed at the web site <http://www.specbench.org>. Compare the SPEC marks of individual processors of the top 5 fastest computers (distinct processors) which you found from the website mentioned in Exercise 1.1.
- 1.4 Estimate the problem complexity for simulating turbulent flows around the wings and body of a supersonic aircraft. Assume that the number of grid points are around 10^{11} .
- 1.5 What are the different methods of increasing the speed of computers? Plot the clock speed increase of Intel microprocessors between 1975 and 2014. Compare this with the number of transistors in Intel microprocessors between 1975 and 2014. From these observations, can you state your own conclusions?
- 1.6 List the advantages and disadvantages of using parallel computers.
- 1.7 How do parallel computers reduce rounding error in solving numeric intensive problems?
- 1.8 Are parallel computers more reliable than serial computers? If yes explain why.
- 1.9 Find out the parallel computers which have been made in India by CDAC, NAL, CRL, and BARC by searching the web. How many processors are used by them and what are the applications for which they are used?

BIBLIOGRAPHY

Barney, B., *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, USA, 2011.

(A short tutorial accessible from the web).

Computing Surveys published by the Association for Computing Machinery (ACM), USA is a rich source of survey articles about parallel computers.

Culler, D.E., Singh, J.P. and Gupta, A., *Parallel Computer Architecture and Programming*, Morgan Kauffman, San Francisco, USA, 1999.

(A book intended for postgraduate Computer Science students has a wealth of information).

DeCegama, A.L., *The Technology of Parallel Processing*, Vol. 1: Parallel Processing, Architecture and VLSI Hardware, Prentice-Hall Inc., Englewood Cliffs, NJ, USA, 1989.

(A good reference for parallel computer architectures).

Denning, P.J., “Parallel Computing and its Evolution”, *Communications of the ACM*, Vol. 29, No. 12, Dec. 1988, pp. 1363–1367.

Dubois, M., Annavaram, M., and Stenstrom, P., *Parallel Computer Organization and Design*, Cambridge University Press, UK, 2010. (A good textbook).

Gama, A., Gupta, A., Karpys, G., and Kumar, V., *An Introduction to Parallel Computing*, 2nd ed., Pearson, Delhi, 2004.

Hennesy, J.L., and Patterson, D.A., *Computer Architecture—A Quantitative Approach*, 5th ed., Morgan Kauffman-Elsevier, USA, 2012.

(A classic book which describes both latest parallel processors and parallel computers).

Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

(This is an 846 page book which gives a detailed description of not only parallel computers but also high performance computer architectures).

Keckler, S.W., Kundle, O. and Hofstir, H.P., (Eds.), *Multicore Processors and Systems*, Springer, USA, 2009.

(A book with several authors discussing recent developments in single chip multicore systems).

Lipovski, G.J. and Malak, M., *Parallel Computing: Theory and Practice*, Wiley, New York, USA, 1987.

(Contains descriptions of some commercial and experimental parallel computers).

Satyanarayanan, M., “Multiprocessing: An Annotated Bibliography,” *IEEE Computer*, Vol. 13, No. 5, May 1980, pp. 101–116.

Shen, J.P., and Lipasti, M.H., *Modern Processor Design*, Tata McGraw-Hill, Delhi, 2010. (Describes the design of superscalar processors).

The Magazines: Computer, Spectrum and Software published by the Institute of Electrical and Electronics Engineers (IEEE), USA, contain many useful articles on parallel computing.

Wilson, G.V., “The History of the Development of Parallel Computing”, 1993.
webdocs.cs.ualberta.ca/~paullu/c681/parallel.time.line.html.

Solving Problems in Parallel

In this chapter we will explain with examples how simple jobs can be solved in parallel in many different ways. The simple examples will illustrate many important points in perceiving parallelism, and in allocating tasks to processors for getting maximum efficiency in solving problems in parallel.

2.1 UTILIZING TEMPORAL PARALLELISM

Suppose 1000 candidates appear in an examination. Assume that there are answers to 4 questions in each answer book. If a teacher is to correct these answer books, the following instructions may be given to him:

Procedure 2.1 Instructions given to a teacher to correct an answer book

Step 1: Take an answer book from the pile of answer books.

Step 2: Correct the answer to Q_1 namely, A_1 .

Step 3: Repeat Step 2 for answers to Q_2, Q_3, Q_4 , namely, A_2, A_3, A_4 .

Step 4: Add marks given for each answer.

Step 5: Put answer book in a pile of corrected answer books.

Step 6: Repeat Steps 1 to 5 until no more answer books are left in the input.

A teacher correcting 1000 answer books using Procedure 2.1 is shown in Fig. 2.1. If a paper takes 20 minutes to correct, then 20,000 minutes will be taken to correct 1000 papers. If we want to speedup correction, we can do it in the following ways:

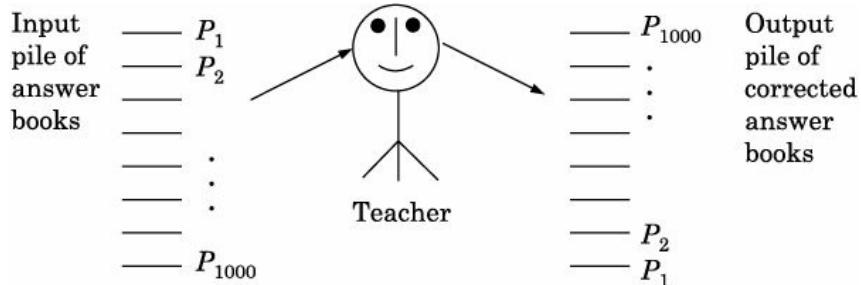


Figure 2.1 A single teacher correcting answer books.

Method 1: Temporal Parallelism

Ask four teachers to co-operatively correct each answer book. To do this the four teachers sit in one line. The first teacher corrects answer to Q_1 , namely, A_1 of the first paper and passes the paper to the second teacher who starts correcting A_2 . The first teacher immediately takes the second paper and corrects A_1 in it. The procedure is shown in Fig. 2.2.

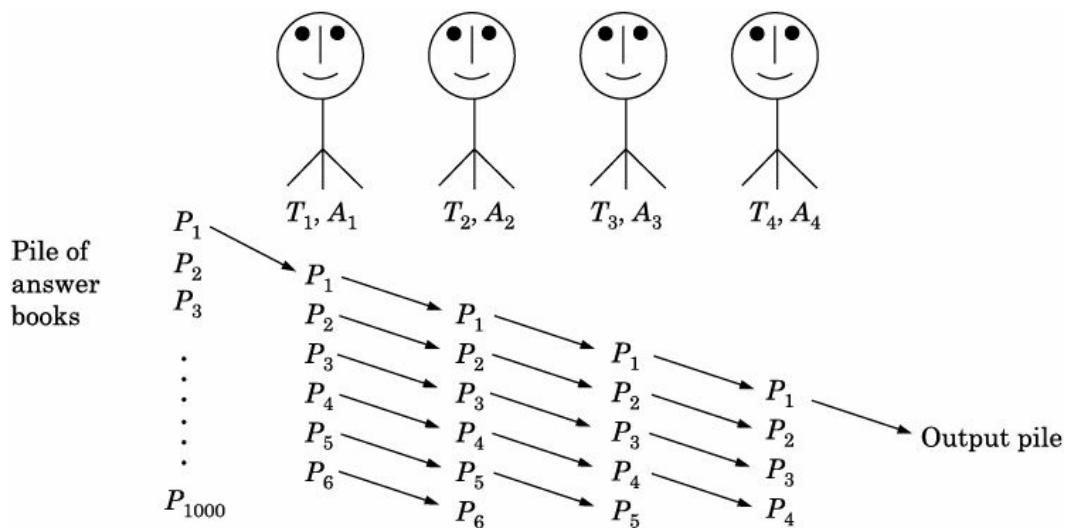


Figure 2.2 Four teachers working in a pipeline or assembly line.

It is seen from Fig. 2.2 that when the first three answer papers are being corrected some teachers are idle. However, from the fourth paper onwards all the teachers are busy correcting one answer book each; teacher 1 will be correcting A_1 of paper 4, teacher 2— A_2 of paper 3, teacher 3— A_3 of paper 2 and teacher 4— A_4 of paper 1. If the time taken to correct A_1 = time to correct A_2 = time to correct A_3 = time to correct A_4 = 5 minutes, then the first answer book will take 20 minutes to correct. After that, one corrected answer book will be ready every 5 minutes. The total time taken to correct 1000 papers will be $20 + (999 * 5) = 5015$ minutes. This time is about (1/4)th of the time taken by one teacher.

This method is a *parallel processing* method as 4 teachers work in parallel, that is, simultaneously to do the job in a shorter time. The type of parallel processing used in this method is called *temporal parallelism*. The term temporal means pertaining to time. As this method breaks up a job into a set of tasks to be executed overlapped in time, it is said to use temporal parallelism. It is also known as assembly line processing, pipeline processing, or vector processing. The terminology vector processing is appropriate in this case if we think of an answer book as a vector with 4 components; the components are the answers A_1 , A_2 , A_3 , and A_4 .

This method of parallel processing is appropriate if:

1. The jobs to be carried out are identical.
2. A job can be divided into many *independent* tasks (i.e., each task can be done independent of other tasks) and each can be performed by a different teacher.
3. The time taken for each task is same.
4. The time taken to send a job from one teacher to the next is negligible compared to the time needed to do a task.
5. The number of tasks is much smaller as compared to the total number of jobs to be done.

We can quantify the above points as worked out below:

Let the number of jobs = n

Let the time to do a job = p

Let each job be divisible into k tasks and let each task be done by a different teacher.

Let the time for doing each task = p/k .

Time to complete n jobs with no pipeline processing = np .

Time to complete n jobs with a pipeline organization of k teachers

$$= p + (n - 1) \frac{p}{k} = p \frac{(k + n - 1)}{k}$$

Speedup due to pipeline processing

$$= \frac{np}{p(k + n - 1)/k} = \frac{k}{1 + \frac{k - 1}{n}}$$

If $n \gg k$ then $(k - 1)/n \approx 0$ and the speedup is nearly equal to k , the number of teachers in the pipeline. Thus, the speedup is directly proportional to the number of teachers working in a pipeline mode provided the number of jobs is very large as compared to the number of tasks per job. The main problems encountered in implementing this method are:

1. Synchronization. Identical time should be taken for doing each task in the pipeline so that a job can flow smoothly in the pipeline without holdup.

2. Bubbles in pipeline. If some tasks are absent in a job “bubbles” form in the pipeline. For example, if there are some answer books with only 2 questions answered, two teachers will be forced to be idle during the time allocated to them to correct these answers.

3. Fault tolerance. The system does not tolerate faults. If one of the teachers takes a coffee break, the entire pipeline is upset.

4. Inter-task communication. The time to pass answer books between teachers in the pipeline should be much smaller as compared to the time taken to correct an answer by a teacher.

5. Scalability. The number of teachers working in the pipeline cannot be increased beyond a certain limit. The number of teachers would depend on how many independent questions a question paper has. Further, the time taken to correct the answer to each of the questions should be equal and the number of answer books n must be much larger than k , the number of teachers in the pipeline.

In spite of these problems, this method is a very effective technique of using parallelism as it is easy to perceive the possibility of using temporal parallelism in many jobs. It is very efficient as each stage in the pipeline can be optimized to do a specific task well. (We know that a teacher correcting the answer to one question in all answer books becomes an “expert” in correcting that answer soon and does it very quickly!). Pipelining is used extensively in processor design. It was the main technique used by vector supercomputers such as CRAY to attain their high speed.

2.2 UTILIZING DATA PARALLELISM

Method 2: Data Parallelism

In this method, we divide the answer books into four piles and give one pile to each teacher (see Fig. 2.3). Each teacher follows identical instructions given in Procedure 2.1 (see Section 2.1).

Assuming each teacher takes 20 minutes to correct an answer book, the time taken to correct all the 1000 answer books is 5000 minutes as each teacher corrects only 250 papers and all teachers work simultaneously. This type of parallelism is called *data parallelism* as the input data is divided into independent sets and processed simultaneously. We can quantify this method as shown below:

Let the number of jobs = n

Let the time to do a job = p

Let there be k teachers to do the job.

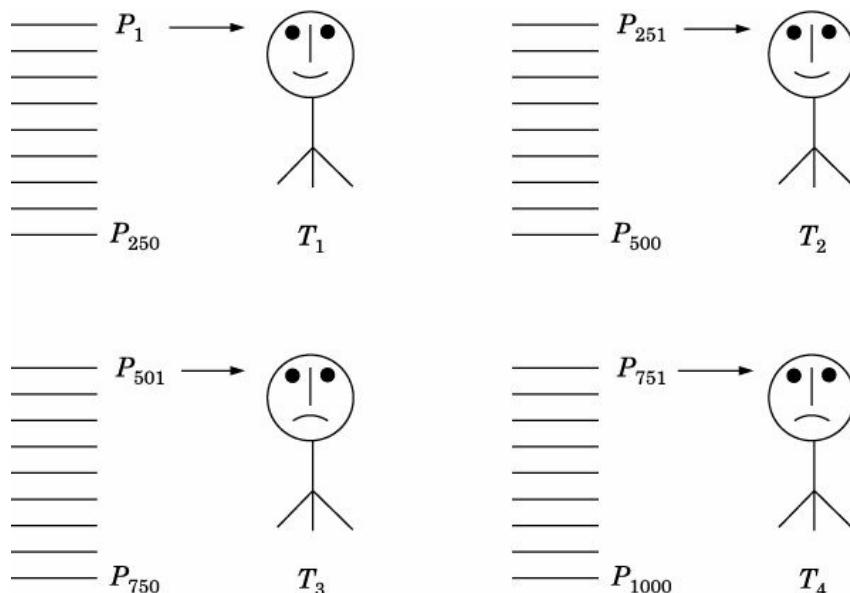


Figure 2.3 Four teachers working independently and simultaneously on 4 sets of answer books.

Let the time to distribute the jobs to k teachers be kq . Observe that this time is proportional to the number of teachers.

The time to complete n jobs by a single teacher = np

The time to complete n jobs by k teachers = $kq + np/k$

$$\frac{np}{kq + \frac{np}{k}} = \frac{knp}{k^2q + np} = \frac{k}{1 + (k^2q/np)}$$

Speedup due to parallel processing =

If $k^2q \ll np$ then the speedup is nearly equal to k , the number of teachers working independently. Observe that this will be true if the time to distribute the jobs is small.

Let us take a numerical example. Let $n = 1000$, $p = 20$ and $k = 4$. Let the time to create one subset of jobs $q = 1$. Then $kq = 4$.

$$\text{Speedup} = \frac{4}{1 + (16/20000)} = 3.997 \approx 4$$

If we define efficiency as the ratio of actual speedup to maximum possible speedup we get:

$$\text{Efficiency} = \frac{\text{Speedup}}{k} = \frac{3.997}{4} = 0.999$$

If $k = 100$ then

$$\text{Speedup} = \frac{100}{1 + (10000/20000)} = \frac{100}{1.5} = 66.7$$

$$\text{Efficiency} = \frac{\text{Speedup}}{k} = \frac{66.7}{100} = 0.667$$

Observe that the speedup is not directly proportional to the number of teachers as the time to distribute jobs to teachers (which is an unavoidable overhead) increases as the number of teachers is increased. The main advantages of this method are:

1. There is *no synchronization* required between teachers. Each teacher can correct papers independently at his own pace.
2. The problem of *bubbles* is *absent*. If a question is unanswered in a paper it only reduces the time to correct that paper.
3. This method is *more fault tolerant*. One of the teachers can take a coffee break without affecting the work of other teachers.
4. There is no communication required between teachers as each teacher works independently. Thus, there is no inter-task communication delay.

The main disadvantages of this method are:

1. The assignment of jobs to each teacher is pre-decided. This is called a *static assignment*. Thus, if a teacher is slow then the completion time of the total job will be slowed down. If another teacher gets many blank answer books he will complete his work early and will thereafter be idle. Thus, a static assignment of jobs is not efficient.
2. We must be able to divide the set of jobs into subsets of mutually independent jobs. Each subset should take the same time to complete.
3. Each teacher must be capable of correcting answers to all questions. This is to be contrasted with pipelining in which each teacher specialized in correcting the answer to only one question.
4. The time taken to divide a set of jobs into equal subsets of jobs should be small. Further, the number of subsets should be small as compared to the number of jobs.

However, there are many situations in practice (for example, distributing portions of an array to processors) where the task distribution time is negligible as only the starting and ending array indices are given to each processor. In such cases the efficiency will be almost 100%.

Method 3: Combined Temporal and Data Parallelism

We can combine Method 1 and Method 2 as shown in Fig. 2.4. Here two pipelines of teachers are formed and each pipeline is given half the total number of jobs. This is called *parallel pipeline processing*. This method almost halves the time taken by a single pipeline. If it takes 5 minutes to correct an answer book, the time taken by the two pipelines is $(20 + 499 \times 5) = 2515$ minutes.

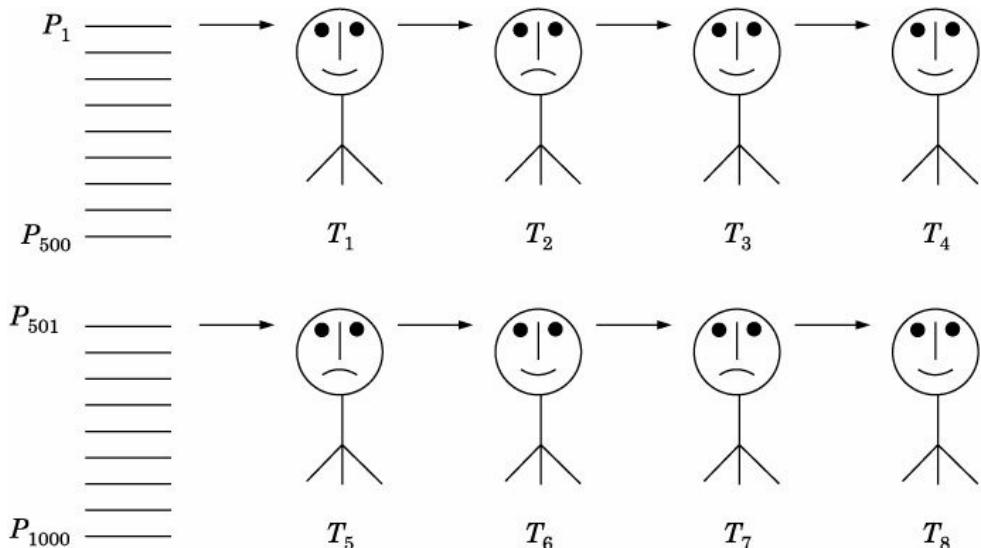


Figure 2.4 Eight teachers working in two pipelines to grade 2 sets of answer books.

Even though this method reduces the time to complete the set of jobs, it also has the disadvantages of both temporal parallelism and to some extent that of data parallelism. The method is effective only if the number of jobs given to *each* pipeline is much larger than the number of stages in the pipeline.

Multiple pipeline processing was used in supercomputers such as Cray and NEC-SX as this method is very efficient for numerical computing in which a number of long vectors and large matrices are used as data and could be processed simultaneously.

Method 4: Data Parallelism with Dynamic Assignment

This method is shown in Fig 2.5. Here a head examiner gives one answer paper to each teacher and keeps the rest with him. All teachers simultaneously correct the paper given to them. A teacher who completes correction goes to the head examiner for another paper which is given to him for correction. If a second teacher completes correction at the same time, then he queues up in front of the head examiner and waits for his turn to get an answer paper. The procedure is repeated till all the answer papers are corrected. The main advantage of this method is the balancing of the work assigned to each teacher dynamically as work progresses. A teacher who finishes his work quickly gets another paper immediately and he is not forced to be idle. Further, the time to correct a paper may widely vary without creating a bottleneck. The method is not affected by *bubbles*, namely, unanswered questions or blank answer papers. The overall time taken for paper correction will be minimized. The main disadvantages of this method are:

1. If many teachers complete correcting an answer paper simultaneously only one of them will get the next paper at once as the examiner can attend to only one teacher at a time. The other teachers have to wait in a queue to get their next answer paper.
2. The head examiner can become a bottleneck. If he takes a coffee break, all teachers will be idle! However, a teacher taking a coffee break does not cause breakdown of the system.
3. The head examiner himself is idle between handing out papers.
4. It is difficult to increase the number of teachers as it will increase the probability of many teachers completing their jobs simultaneously thereby leading to long queues of teachers waiting to get an answer paper.

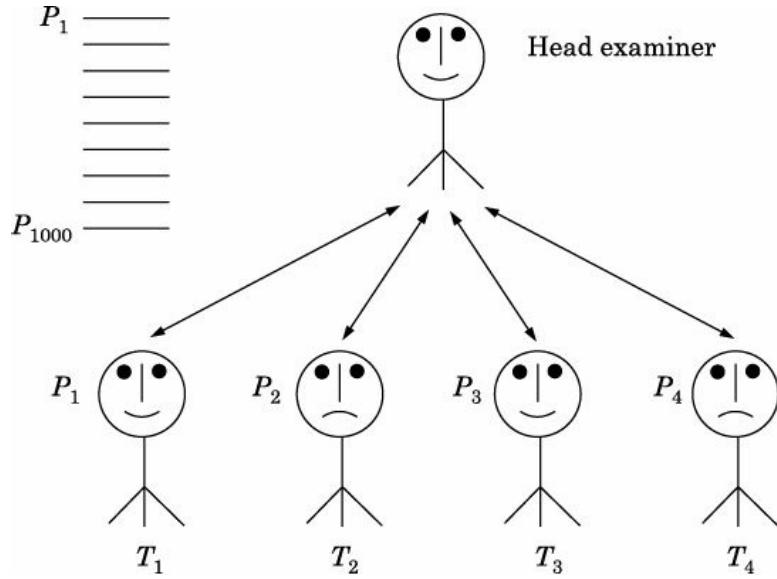


Figure 2.5 Scheduling of jobs by head examiner to a group of teachers.

If the speedup of a method is directly proportional to the number of processors, then the method is said to *scale well*. We saw that temporal parallel processing method scales well when the number of jobs is much larger than the number of tasks in a job.

We will now analyze the scalability of the dynamic assignment method:

Let the total number of papers = n .

If no parallel processing is employed then the time to correct n papers = np , where p is the average time to correct a paper.

Let k teachers be employed to work in parallel.

Let the time a teacher waits to get an answer book from the head examiner and to return it to the head examiner be q .

The time taken by each teacher to get a paper from the head examiner, grade, and return it = $(q + p)$.

Assuming that each teacher corrects (n/k) papers and all teachers work simultaneously the total time taken to correct all the papers by k teachers

$$= \frac{n(q + p)}{k}$$

Speedup due to parallel processing

$$\begin{aligned} &= np/[n(q + p)/k] \\ &= k/[1 + (q/p)] \end{aligned}$$

As long as $q \ll p$, the speedup approaches the ideal value. If this condition is not satisfied speedup is lower. For example, if the time to correct a paper = 10 minutes and the time to get and return a paper is 2.5 minutes then the speedup = 0.8 k an efficiency loss of 20%. Usually q is a function of k and goes up with k . The main reason for this is that a single head examiner has to cater too many teachers and he can cater to only one teacher at a time. In other words, there is contention for a common *shared resource* leading to queue formation and increase in service time. If $q = mk$ then

$$\begin{aligned} \text{Speedup} &= k/[1 + (mk/p)] \\ &= 1/[(1/k) + (m/p)] \end{aligned}$$

which is nearly equal to p/m , a constant, when k is large.

This illustrates the point that speedup will *saturate* in such a case and will not go up as more teachers are employed. In other words, the time to distribute papers goes on increasing as more teachers are employed and a teacher waits longer to get a paper than to correct it! The method is thus not scalable unless $(mk/p) \ll 1$.

Method 5: Data Parallelism with Quasi-dynamic Scheduling

Method 4 can be improved by giving each teacher unequal sets of answer papers to correct. For instance, teachers 1, 2, 3, 4 may be given respectively 7, 11, 13, 19 papers initially. When they complete their work they may be given further small bunches of papers. This will randomize the job completion time of each teacher and reduce the probability of queue formation in front of the head examiner.

The assignment of jobs to teachers in Method 3 is a *static schedule* as the assignment is done initially and not changed. In Method 4, one paper is handed out to a teacher as and when he completes correcting a paper. Thus, a teacher who corrects fast will get more papers to correct. This method of assigning jobs is called a *dynamic schedule*. This is a very good method provided the time taken to hand out each job is much smaller as compared to the time to actually do the job. Method 5 is called *quasi-dynamic schedule* which is in between purely static and purely dynamic schedules. The individual jobs in a purely dynamic schedule are *fine grained* in the sense that only one job (in our example, one paper) is assigned and the time taken to complete the job is small. In quasi-dynamic scheduling the jobs are *coarser grained* in the sense that a bunch of answer books are given to a teacher and the time taken to complete correction will be more than if one paper is to be corrected. Further, the completion time of coarse grained jobs will be somewhat spread out.

We summarise our observations as:

1. If the loads given to processors is balanced, i.e., almost equal, then a static assignment is very good.
2. One should resort to dynamic assignment of tasks to processors only if there is a wide variation in the completion time of tasks.
3. The task distribution overhead in quasi-dynamic assignment with coarse grained tasks is usually much smaller than in fine grained dynamic assignment and is thus a better method.

2.3 COMPARISON OF TEMPORAL AND DATA PARALLEL PROCESSING

We compare the two methods of parallel processing we discussed in the last two sections in Table 2.1.

TABLE 2.1 Comparison of Temporal and Data Parallel Processing	
<i>Temporal parallel processing (pipelining idea)</i>	<i>Data parallel processing</i>
1. Job is divided into a set of independent tasks and tasks are assigned for processing.	Full jobs are assigned for processing.
2. Tasks should take equal time. Pipeline stages should thus be synchronized.	Jobs may take different times. No need to synchronize beginning of jobs.
3. Bubbles in jobs lead to idling of processors.	Bubbles do not cause idling of processors.
4. Processors specialized to do specific tasks efficiently.	Processors should be general purpose and may not do all tasks efficiently.
5. Task assignment static.	Job assignment may be static, dynamic, or quasi-dynamic.
6. Not tolerant to processor faults.	Tolerates processor faults.
7. Efficient with fine grained tasks.	Efficient with coarse grained tasks and quasi-dynamic scheduling.
8. Scales well as long as number of data items to be processed is much larger than the number of processors in the pipeline and time taken to communicate task from one processor to the next is negligible.	Scales well as long as number of jobs is much greater than the number of processors and processing time is much higher than the time to distribute data to processors.

2.4 DATA PARALLEL PROCESSING WITH SPECIALIZED PROCESSORS

We saw that pipeline processing is effective if each answer takes the same time to correct and there are not many answer papers with unanswered questions. Data parallel processing is more tolerant but requires each teacher to be capable of correcting answers to all questions with equal ease. If we have a situation where we would like to constrain each teacher to correct only the answer to one question in each paper (may be to ensure uniformity in grading or because he is a specialist who can do that job very quickly) and the answer books are such that not all students answer all questions then a method using specialist data parallelism may be developed as explained below.

Method 6: Specialist Data Parallelism

In this method, we assume that there is a head examiner who assigns answer papers to teachers. The organization of this method is shown in Fig. 2.6. The procedure followed by the head examiner is given as Procedure 2.2. We assume that teacher 1 (T_1) grades A_1 , teacher 2 (T_2) grades A_2 and teacher i (T_i) the answer A_i to question Q_i .

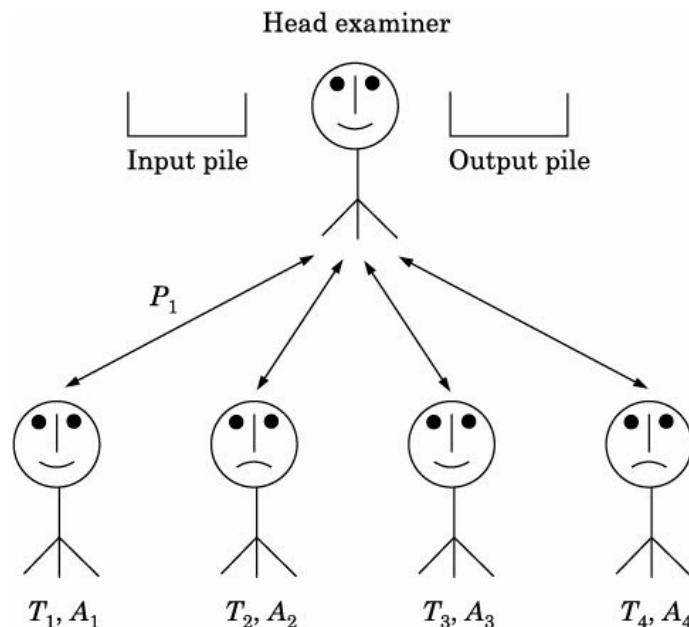


Figure 2.6 Head examiner assigning specific answers to each teacher to grade.

Procedure 2.2 Task assignment method followed by the head examiner

1. Give one answer paper to T_1, T_2, T_3, T_4 (Remark: Teacher T_i corrects only the answer to question Q_i).
2. When a corrected answer paper is returned check if all questions are graded. If yes add marks and put the paper in the output pile.
3. If no, check which questions are not graded.
4. For each i , if A_i is ungraded and teacher T_i is idle send it to teacher T_i or if any other teacher T_p is idle and an answer paper remains in input pile with A_p uncorrected send it to him.
5. Repeat Steps 2, 3 and 4 until no answer paper remains in the input pile and all

teachers are idle.

The main problem with this method is that the load is not balanced. If some answers take much longer time to grade than others then some of the teachers will be busy while others are idle. The same problem will occur if a particular question is not answered by many students. Further, the head examiner will waste a lot of time seeing which questions are unanswered and which teachers are idle before he is able to assign work to a teacher. In other words, the maximum possible speedup will not be attained.

Method 7: Coarse Grained Specialist Temporal Parallel Processing

The same problem can also be done without a head examiner. Here all teachers work independently and simultaneously at their pace. We will see, however, that each teacher will end up spending a lot of time inefficiently waiting for other teachers to complete their work. The model of processing is shown in Fig. 2.7. The answer papers are initially divided into 4 equal parts and one part is kept in an in-tray with each teacher. The teachers sit in a circle as shown in Fig. 2.7. Teacher T_1 takes a paper from his in-tray, grades answer A_1 to question Q_1 in it and places it in his out-tray. When no papers are in his in-tray, he checks if teacher T_2 's in-tray is empty. If yes he places his graded paper in T_2 's in-tray and waits for his in-tray to be filled. Teacher T_1 's in-tray will be filled by teacher T_4 . This is a circular pipeline arrangement with a coarse grain size. All papers would be graded when each teacher's out-tray is filled 4 times. The method is given more precisely as Procedure 2.3.

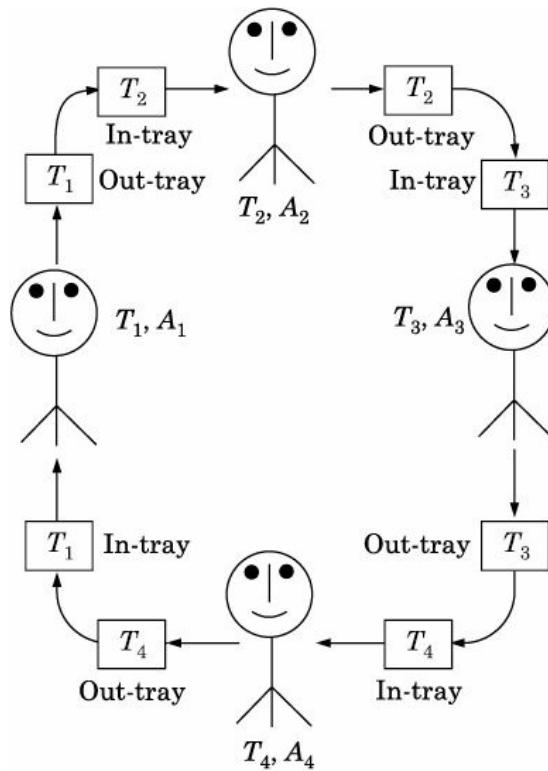


Figure 2.7 One teacher grades one answer in all papers—A circular pipeline method.

Procedure 2.3 Coarse grained specialist temporal processing

Answer papers are divided into 4 equal piles and put in the in-trays of each teacher. Each teacher repeats 4 times Steps 1 to 5. All teachers work simultaneously.

For teachers T_i ($i = 1$ to 4) do in parallel Steps 1 to 5.

Step 1: Take an answer paper from in-tray.

Step 2: Grade answer A_i to question Q_i and put it in out-tray.

Step 3: Repeat Steps 1 and 2 till no papers left in in-tray.

Step 4: Check if teacher $T_{(i+1)\bmod 4}$'s in-tray is empty.

Step 5: As soon as it is empty, empty own out-tray into the in-tray of that teacher.

Step 6: All answers will be graded (in other words the procedure will terminate) when each teacher's output tray is filled 4 times.

This section mainly illustrates the point that if processing of special tasks by special processors are to be done then balancing the load is essential and the promised speedup of parallel processing is not attainable unless this condition is fulfilled. It may also be observed that the method uses the concept of pipelined processing using a circular pipeline. Further, each stage in the pipeline has a chunk of work to do. This method does not require strict synchronization. It also tolerates bubbles due to unanswered questions in a paper or blank answer papers.

Method 8: Agenda Parallelism

In this method an answer book is thought of as an *agenda* of answers to be graded. All teachers are asked to work on the first item on the agenda, namely grade the answer to the first question in all papers. A head examiner gives one paper to each teacher and asks him to grade the answer A_1 to Q_1 . When a teacher finishes this he is given another paper in which he again grades A_1 . When A_1 of all papers are graded then A_2 is taken up by all teachers. This is repeated till all the answers in all papers are graded. This is data parallel method with dynamic schedule and fine grain tasks. This method for the problem being considered is not a good idea as the grain size is small and waiting time of teachers to receive a paper to grade may exceed the time to correct it! Method 4 is much better.

We consolidate and summarize the features of all the 8 methods of parallel processing in Fig. 2.8.

An interesting point brought out by this simple example is the rich opportunity which exists to exploit parallelism. Many jobs have inherent parallel features and if we think deeply many different methods for exploiting parallelism become evident. The important question is to decide which of the methods is good. The choice of a method depends on the nature of the problem, the type of input data, processors, interconnection between processors, etc. This will be discussed again in later chapters.

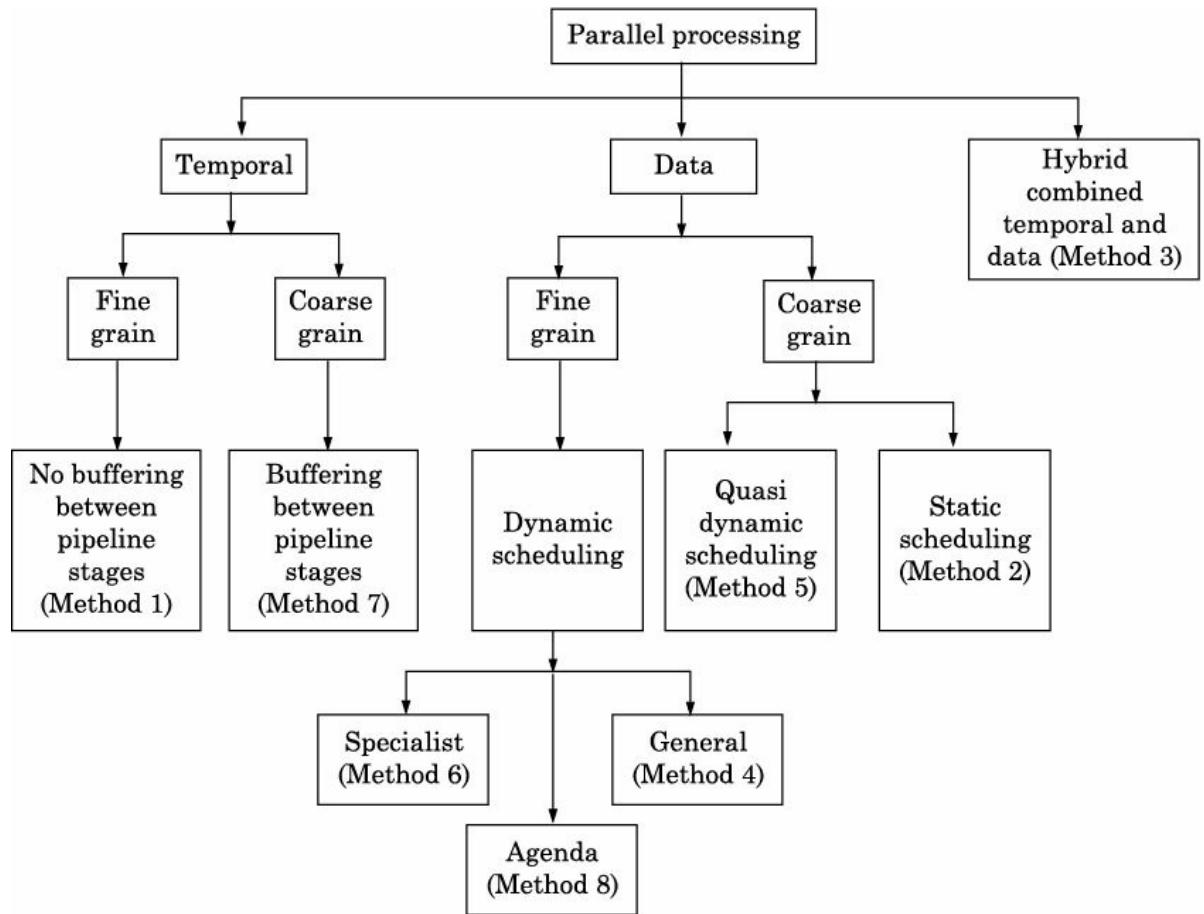


Figure 2.8 Chart showing various methods of parallel processing.

2.5 INTER-TASK DEPENDENCY

So far we have made the following assumptions in evolving methods of assigning tasks to teachers.

1. The answer to a question is independent of answers to other questions.
2. Teachers do not have to interact.
3. The same instructions are used to grade all answer books. In other words, the answer books are for the same subject.

In general a job consists of tasks which are inter-related. Some tasks can be done simultaneously and independently while others have to wait for the completion of previous tasks. For example, in the grading example we have discussed, the answer to a question may depend on the answers to previous questions. The inter-relations of various tasks of a job may be represented graphically as a *task graph*. In this graph circles represent tasks which in the example we have discussed are answers to be graded. A line with an arrow connecting two circles shows dependency of tasks. The direction of an arrow shows precedence. A task at the head of an arrow can be done after all tasks at their respective tails are done.

If the question paper is such that the answers to the 4 questions are independent of one another the answers to the four questions can be graded in any order. If grading the answer to Q_i is called T_i then the task graph for this case may be represented as shown in Fig. 2.9(a). If the question paper is such that the answer to question Q_{i+1} depends on the answer to Q_i ($i = 1, 2, 3, 4$), then Q_{i+1} cannot be graded (namely, task T_{i+1}) unless the answer to Q_i is graded (task T_i). The task graph for this case is shown in Fig. 2.9(b). If the question paper is such that T_2 is dependent on T_1 , and T_4 on T_1 and T_3 , then the task graph for this case may be drawn as shown in Fig. 2.9(c).

If the tasks are to be assigned to teachers for pipeline processing (Method 1) then if the tasks are independent (Fig. 2.9(a)) any task can be assigned to any teacher. If the task graph is Fig. 2.9(b) or (c), then T_1 should be assigned to the first teacher, T_2 to the second teacher who takes the paper from the first teacher and T_3 and T_4 assigned to teachers 3 and 4 respectively who sit next to teacher 2 in the assembly line (or pipeline).

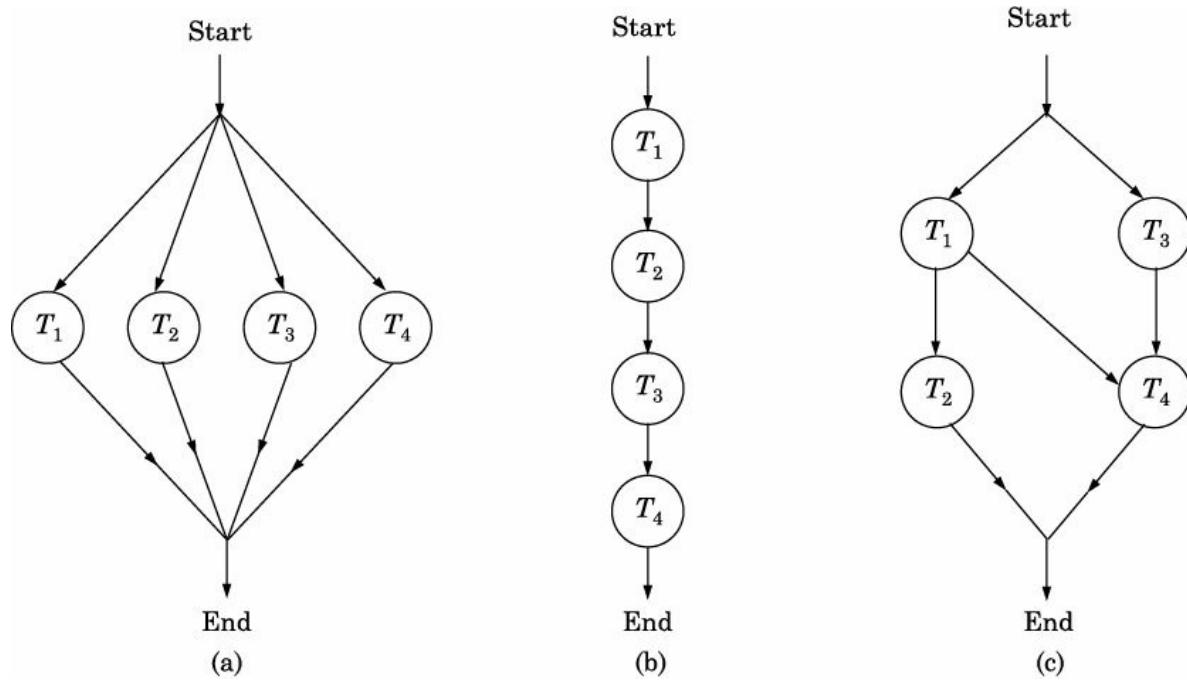


Figure 2.9 Task graphs for grading answer papers.

In the examples we have considered so far many identical jobs were processed in parallel. Another important case is where there is one complicated job which yields one end product. This job can be broken up into separate tasks which are cooperatively done faster by multiple processors. A simple everyday example of this type is cooking using a given recipe. We give as Procedure 2.4, the recipe for cooking Chinese vegetable fried rice.

Procedure 2.4 Recipe for Chinese vegetable fried rice

T₁: Clean and wash rice

T₂: Boil water in a vessel with 1 teaspoon salt

T₃: Put rice in boiling water with some oil and cook till soft (do not overcook)

T₄: Drain rice and cool

T₅: Wash and scrape carrots

T₆: Wash and string French beans

T₇: Boil water with 1/2 teaspoon salt in two vessels

T₈: Drop carrots and French beans separately in boiling water and keep for 1 minute

T₉: Drain and cool carrots and French beans

T₁₀: Dice carrots

T₁₁: Dice French beans

T₁₂: Peel onions and dice into small pieces.

T₁₃: Clean cauliflower. Cut into small pieces

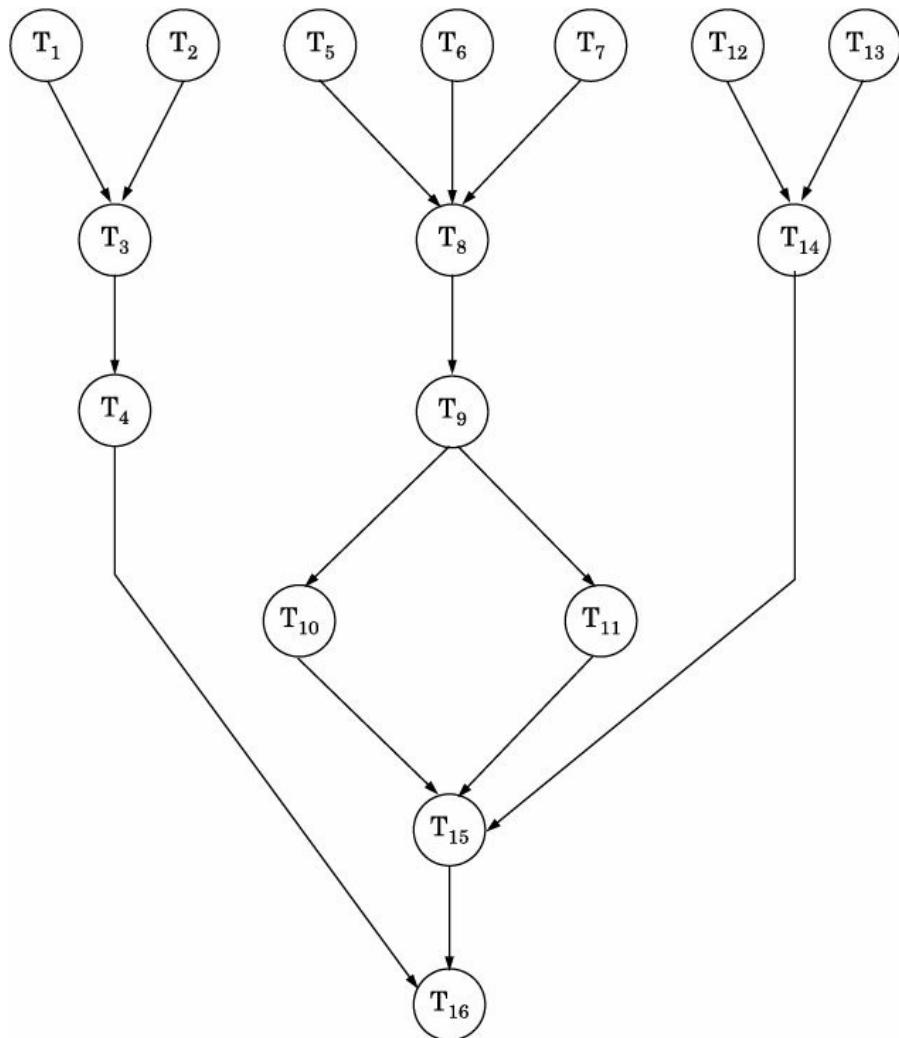
T₁₄: Heat oil in iron pan and fry diced onion and cauliflower for 1 minute in heated oil

T₁₅: Add diced carrots, French beans to above and fry for 2 minutes

T₁₆: Add cooled cooked rice, chopped spring onions, and soya sauce to the above and stir and fry for 5 minutes.

There are 16 tasks in cooking Chinese vegetable fried rice. Some of these tasks can be carried out simultaneously whereas others have to be done in sequence. For instance tasks

T_1 , T_2 , T_5 , T_6 , T_7 , T_{12} and T_{13} can all be done simultaneously, whereas T_8 cannot be done unless T_5 , T_6 , T_7 are done. A graph showing the relationship among tasks is given as in Fig. 2.10. The reader is urged to check the correctness of the graph.



T_1 : Clean and wash rice

T_2 : Boil water

T_3 : Cook rice

T_4 : Drain rice and cool

T_5 : Wash and scrape carrots

T_6 : Wash and string French beans

T_7 : Boil water

T_8 : Drop carrots and French beans in boiling water

T_9 : Drain and cool carrots and French beans

T_{10} : Dice carrots

T_{11} : Dice French beans

T_{12} : Peel onions and dice into small pieces

T_{13} : Clean cauliflower and cut

T_{14} : Heat oil and fry cauliflower and onions

T_{15} : Add diced carrots and French beans and fry

T_{16} : Add cooled cooked rice, onions, etc.

Figure 2.10 A task graph to cook Chinese vegetable fried rice.

Suppose this dish has to be made for 50 people and 4 cooks cooperate and cook. Tasks assigned to the cooks must be such that they work simultaneously and synchronize. The time taken for each task is given in Table 2.2.

With these timings an assignment of tasks for each cook is arrived by using Procedure 2.5 and is shown in Fig. 2.11. Observe that this schedule is obtained keeping the constraint of sequencing. This forces some cooks to be idle for sometime. In this schedule we have tried to minimize completion time of the job, but it is not the best schedule. We have also not attempted to equalize the load on cooks. The reader can try his ingenuity and obtain a better schedule. The minimum possible completion time requires T_2 , T_3 , T_4 and T_{16} to be done in sequence taking 40 units of time. The schedule of Fig. 2.11 takes 45 units. If no sequencing constraints were there and if all 4 cooks work simultaneously then the minimum time is:

$$(T_1 + T_2 + \dots + T_{16})/4 = 132/4 = 33 \text{ minutes}$$

Procedure 2.5 Assigning tasks in a task graph to cooks

Step 1: Find tasks which can be carried out in parallel in level 1. Sum their total time. In this case it is sum of times for tasks 1, 2, 5, 6, 7, 12, 13 = 65 minutes. As there are 4 cooks divide the work so that each cook has tasks assigned for around $65/4 = 16$ minutes.

TABLE 2.2 Time for Each Task in Procedure 2.4

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Time	5	10	15	10	5	10	8	1
Task	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}
Time	10	10	10	15	12	4	2	5

The Gantt chart illustrates the task assignments for four cooks over a timeline. The tasks are represented by horizontal bars, and the time axis is marked at 5-unit intervals. Cook 1 performs tasks T1 through T4 and T16. Cook 2 performs tasks T5 through T10 and T15. Cook 3 performs tasks T7 and T13, followed by T14. Cook 4 performs task T12, followed by an 11-unit idle period, then T11, followed by another idle period.

Figure 2.11 Assignment of tasks to cooks.

The assignment based on this logic is:

Cook 1 \rightarrow (T_1 , T_2) (15 minutes)

Cook 2 \rightarrow (T_5 , T_6) (15 minutes)

Cook 3 \rightarrow (T_7 , T_{13}) (20 minutes)

Cook 4 \rightarrow (T_{12}) (15 minutes)

Step 2: Find tasks which can be carried out in level 2. They are T_3 , T_8 , and T_{14} . There are 4 cooks. Further T_{14} cannot start unless T_{12} and T_{13} are completed. Thus, we allocate Cook 1 \rightarrow T_3 ; Cook 2 \rightarrow T_8 ; Cook 3 \rightarrow T_{14} ; Cook 4 \rightarrow No task. At the end of this step,

Cook 1 has worked for 30 minutes.

Step 3: The tasks which can be carried out next in parallel are T_4 and T_9 . T_4 has to follow T_3 and T_9 has to follow T_8 . We thus allocate Cook 1 $\rightarrow T_4$ and Cook 2 $\rightarrow T_9$.

Step 4: The tasks which can be allocated next are T_{10} and T_{11} each taking 10 minutes. They follow completion of T_9 . Assignments are Cook 2 $\rightarrow T_{10}$ and Cook 4 $\rightarrow T_{11}$. However, Cook 4 cannot start T_{11} till Cook 2 completes T_9 .

Step 5: In the next level only T_{15} can be allocated and as it has to follow completion of T_{10} and T_{11} . This can be allocated to Cook 2.

Step 6: Only T_{16} is now left. It cannot start till T_4 , T_{15} , and T_{14} are completed. From Fig. 2.11 we see that T_4 is last to finish. Thus, T_{16} is allocated to Cook 1.

By now you may be wondering what grading papers and Chinese cooking have to do with parallel computing. These two examples illustrate task graphs and bring out important types of problems encountered in programming computers in which parallelism can be exploited. If a large number of similar data records are to be processed using identical processing rules, the parallel processing methods given for grading answer books would be appropriate. Such problems are said to be *embarrassingly parallel*. On the other hand, the cooking problem has constraints which forces sequential processing of some tasks as they have to wait for other tasks to complete.

We will now consider a sequential procedure and examine how it may be parallelized. We assume that each student's record has seven fields, the roll number and marks obtained in 6 subjects. Procedure 2.6 specifies the processing to be carried out.

Procedure 2.6 Processing student grade

Initialise all variables

begin procedure

```
Task 1: {Read student record
          Accumulate count of students
          Add marks in each subject and find total marks and enter it in student
          record}
Task 2: Find class
          {From total marks find class and enter it in student record}
Task 3: Find statistics /* Will be used to find mean and standard deviation */
          {grand total:= grand total + total marks of each student
           sum square:= sum square + (total marks)2 of each student}
Task 4: Find histogram
          {Find sum of students with total marks in each of the following slots:
           0 to 10, 11 to 20, 21 to 30, 31 to 40 ..... 91 to 100}
Task 5: Write processed student record and histogram
end while /* All records read and processed */
Task 6: Find class average, standard deviation
          {class average = sum of marks/No. of students
           Standard deviation
           = Square root (sum square – class average2)/No. of students;
            Write class statistics}
```

end of procedure

If there are 3 processors available to do the processing how do we allocate tasks to processors?

Method 1: Pipeline Processing

Allocate Tasks 1 and 2 to Processor 1

Allocate Task 3 to Processor 2

Allocate Tasks 4 and 5 to Processor 3

Assuming this assignment takes equal time on each of the processors we can use pipeline processing as shown in Fig. 2.12. After the pipeline processing is over Task 6 may be assigned to Processor 1. Other processors will be idle during this time.

while there are student records in the input file *do*

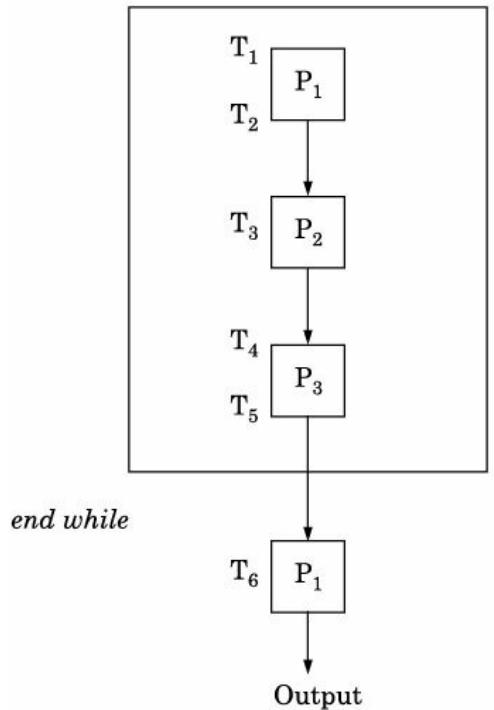


Figure 2.12 Processing student grades—pipeline processing.

Method 2: Data Parallel Processing—Version 1

Divide the input student record file into three almost equal parts and call them File 1, File 2 and File 3. Allocate tasks to processors as shown in Fig. 2.13. After processing is over, carry out Task 6 in Processor 1. Other processors will be idle.

If the number of records in a student file are not known or vary from case to case then the following method is appropriate:

Method 3: Data Parallel Processing—Version 2

while not end of student records in file *do*

 Task 1 in Processor 1

 Do in parallel

 Task 2, Task 3 in Processor 2

 Task 4, Task 5 in Processor 3

end while

 Do Task 6 in Processor 1

We depict this method in Fig. 2.14. The other methods described in Sections 2.1 to 2.4 may also be applied to this problem which we leave as an exercise to the reader.

If there is one complicated job which consists of many tasks which are dependent then the method given for Procedure 2.4 is applicable. The task is to compute:

$$a = \cos(\beta(x)) + \cos(g(y)) + e^{h(z)} \text{ and}$$

$$b = \sin(e^{h(z)}) + p(u) * e^{h(z)}$$

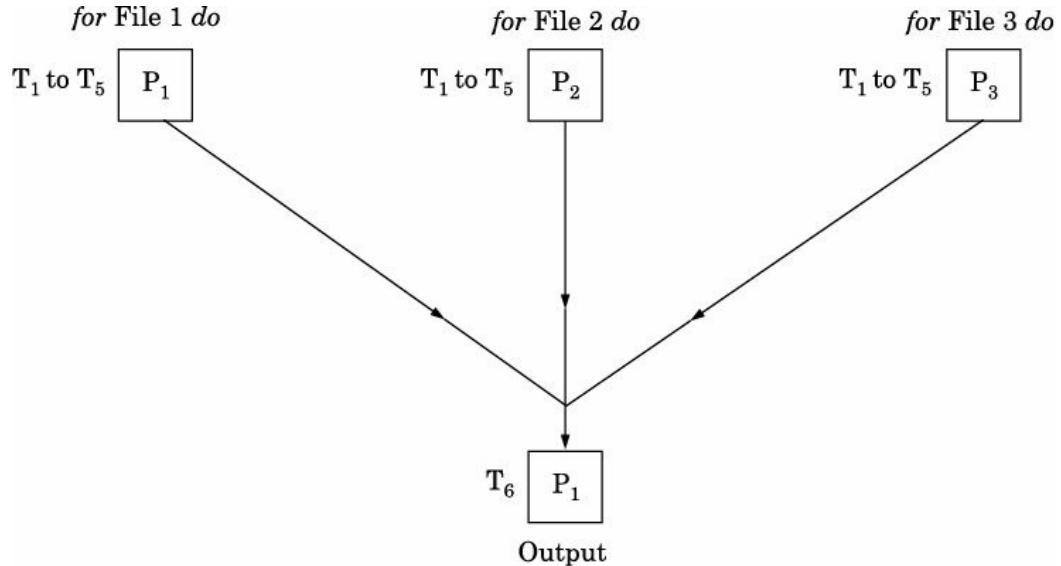


Figure 2.13 Processing student grades—data parallel processing—Version 1.

while there are student records in
the input file *do*

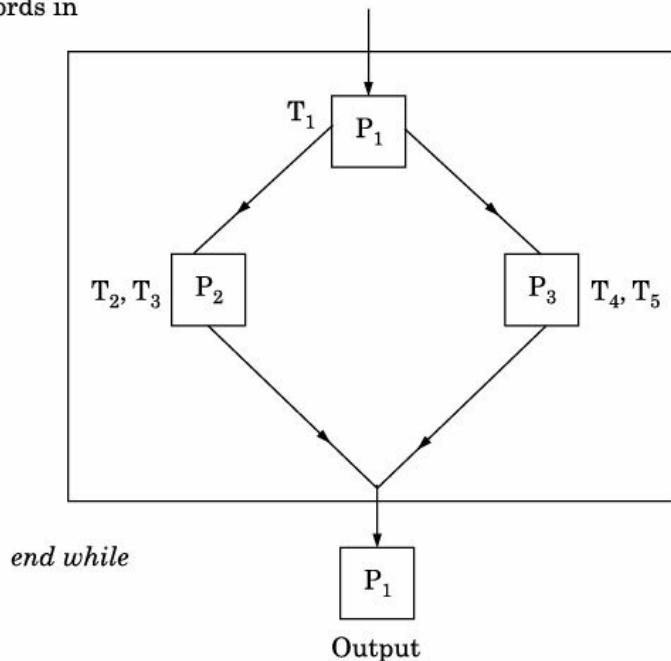


Figure 2.14 Processing student grades—data parallel processing—Version 2.

A task graph to compute this is shown in Fig. 2.15. The time taken by each task is given in the graph. This computation can be carried out by 4 processors as illustrated in Fig. 2.16(b). The time to compute each task is given in Fig. 2.16(a). The time to output *a* is 15 units and *b* is 20 units. The constraints on sequencing tasks as dictated by the task graph is used to assign tasks to processors shown in Fig. 2.16.

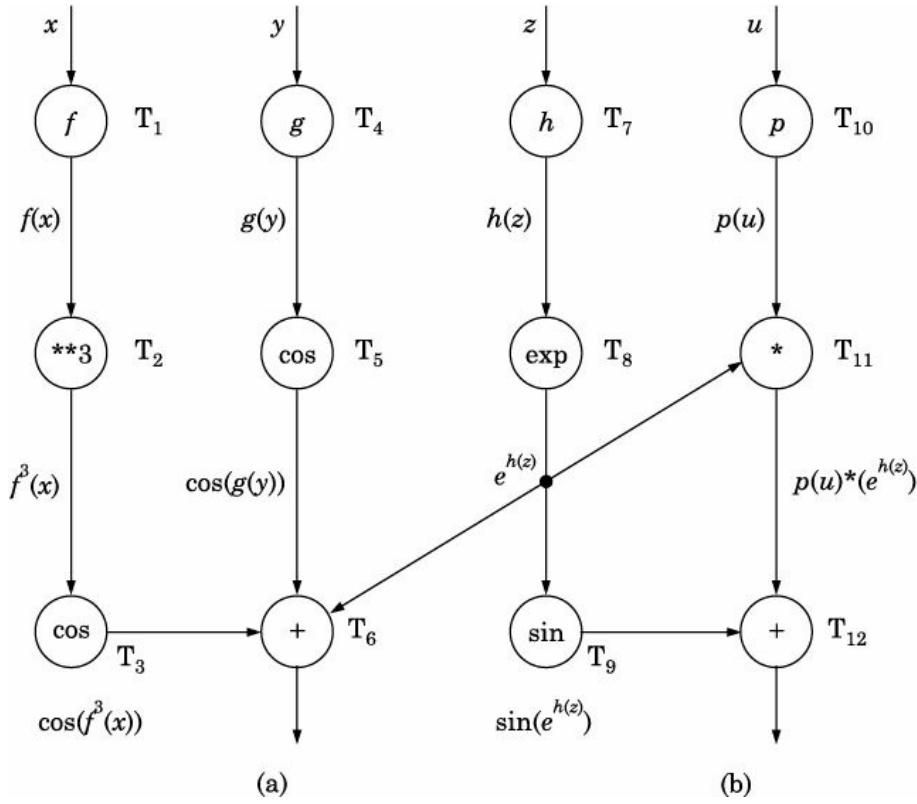


Figure 2.15 Task graph for a complex computation.

Embarrassingly Parallel Problems

There are some problems which are easy to parallelize. For example, consider the following program

```

for  $i := 1$  to 1000 do
     $a(i) := b(i) + c(i)$ 
end for

```

In this case we can break up the loop into 4 loops each loop index going from 1 to 250. Each of these loops may be assigned to a processor. All the processors can work independently and simultaneously to solve the problem in (1/4) time taken by a single processor. In fact many problems with array calculation in loops are normally easy to parallelize. This method of assignment of tasks is similar to Method 2 of Section 2.1.

There are of course other problems which are not easy to parallelize due to dependencies as we have already seen with some task graphs. A simple example is the program shown below:

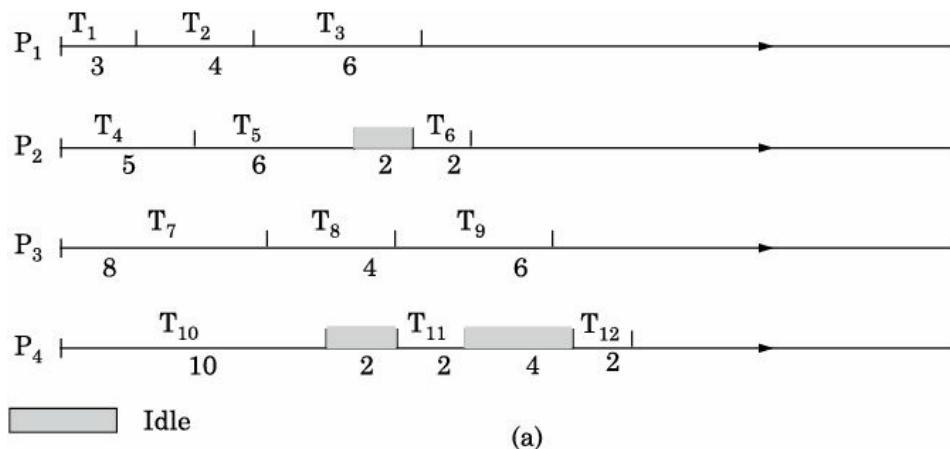
```

 $a(0) := 0$ 
 $a(1) := 1$ 
for  $i := 2$  to 1000 do
     $a(i) := a(i-1) + a(i-2)$ 
end for

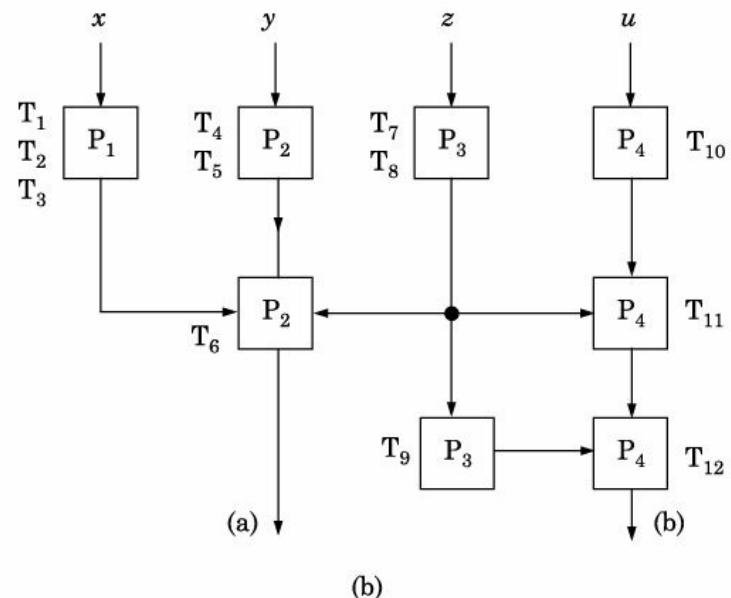
```

In this case $a(i-1)$ and $a(i-2)$ must be computed before $a(i)$. Therefore, there is a data dependency and parallelism is inhibited.

Even though many problems are potentially parallelizable very often some ingenuity is needed to obtain a suitable parallel algorithm as we will see in later chapters.



(a)



(b)

Figure 2.16 Task assignment of graph to 4 processors to work in parallel.

2.6 CONCLUSIONS

This chapter brought out many important points in parallel computing. We saw that there are many ways of solving a problem in parallel. The method to be picked depends on many factors. The issues pointed out in this chapter are:

- Is there a need for strict synchronization of tasks to be carried out in parallel in a method?
- How far is the method fault tolerant?
- Is balancing of load between processors feasible?
- Is there a need for a lot of data to be communicated between processors?
- Are processors forced to wait intermittently for data?
- Is there a common resource for which there is considerable demand in the proposed method? Does this lead to difficulty in scaling?
- What is the granularity of tasks? What is the ratio of computation time of a task and the time needed to communicate results between tasks?
- What is the ratio of the computation time of a task and the time the task has to wait after computation to access a common resource?
- How are tasks assigned to processors? Is a static schedule feasible or is a dynamic schedule necessary? Does a static schedule lead to minimum completion time of the job?
- Is the method scalable? In other words, does the method allow increase in the number of processors or computers working in parallel with a commensurate decrease in processing time?

Besides these issues, there are also constraints placed by the structure and interconnection of computers in a parallel computing system. Thus, picking a suitable method is also governed by the architecture of the parallel computer using which a problem is to be solved.

EXERCISES

- 2.1 An examination paper has 8 questions to be answered and there are 1000 answer books. Each answer takes 3 minutes to correct. If 4 teachers are employed to correct the papers in a pipeline mode, how much time will be taken to complete the job of correcting 1000 answer papers? What is the efficiency of processing? If 8 teachers are employed instead of 4, what is the efficiency? What is the job completion time in the second case? Repeat with 32 teachers and 4 pipelines.
- 2.2 An examination paper has 4 questions. The answers to these questions do not take equal time to correct. Answer to question 1 takes 2 minutes to correct, question 2 takes 3 minutes, question 3 takes 2.5 minutes and question 4 takes 4 minutes. Due to this speed mismatch, storage should be provided between teachers (e.g., a tray where a teacher finishing early can keep the paper for the slower teacher to take it). Answer the following questions assuming 1000 papers are to be corrected by 4 teachers.
- (i) What is the idle time of teachers?
 - (ii) What is the system efficiency?
 - (iii) How much tray space should be provided between teachers due to speed mismatch?
- 2.3 If Exercise 2.2 is to be solved using a data parallel processing method, what will be its efficiency? Compare with pipeline mode.
- 2.4 In an examination paper there are 5 questions and each will take on the average 10 minutes to correct. 2000 candidates write the examination. 5 teachers are employed to correct the papers using pipeline mode. Every question is not answered by all candidates. 10% of the candidates do not answer question 1, 15% question 2, 5% question 3, 5% question 4, and 25% question 5.
- (i) How much time is taken to complete grading?
 - (ii) What is the efficiency of pipeline processing?
 - (iii) Work out a general formula for calculating efficiency assuming n papers, k teachers and percent unanswered questions as p_1, p_2, \dots, p_k respectively.
 - (iv) What is the efficiency of the data parallel method?
 - (v) If data parallel method is used how much time will be taken to complete grading?
- 2.5 In the pipeline mode of processing we assumed that there is no communication delay between stages of the pipeline. If there is a delay of y between pipeline stages derive a speedup formula. What condition should y satisfy to ensure a speedup of at least $0.8 k$, where k is the number of stages in the pipeline?
- 2.6 Assume that there are k questions in a paper and k teachers grade them in parallel using specialist data parallelism (Method 6, Section 2.4). Assume the time taken to grade the k answers is $t_1, t_2, t_3, \dots, t_k$ respectively. Assume that there are n answer books and the time taken to despatch a paper to a teacher is kq .
- (i) Obtain a formula for the speedup obtainable from the method assuming that every student answers all questions.
 - (ii) If the percentage of students not answering questions 1, 2, 3, ..., k are respectively, $p_1, p_2, p_3, \dots, p_k$, modify the speedup formula.
- 2.7 Making the same assumptions as made in Exercise 2.6, obtain the speedup formula for Method 7 of the text (Procedure 2.3). Find speedup for both cases (i) and (ii).
- 2.8 Making the same assumptions as made in Exercise 2.6, obtain the speedup formula for

Method 8 (Agenda parallelism). Find speedup for both cases (i) and (ii).

- 2.9 In Method 4, we assumed that the head examiner gives one paper to each teacher to correct. Assume that instead of giving one paper each, x papers are given to each teacher.

(i) Derive the speedup formula for this case. State your assumptions.

(ii) If the percentage of unanswered questions are a_1, a_2, \dots, a_k respectively (for the k questions), obtain the speedup.

(iii) Compare (ii) with static assignment for the same case.

- 2.10 In Method 7, we assumed that each teacher has an in-tray and an out-tray. Instead if each teacher has only one in-tray and when he corrects a paper he puts it in the in-tray of his neighbour, modify the algorithm for correcting all papers. Compare this method with the method given in the text. Clearly specify the termination condition of the algorithm.

- 2.11 A recipe for making potato bondas is given below: Ingredients: Potatoes (100), onions (10), chillies (10), gram flour (1 kg), oil, salt.

Method:

Step 1: Boil potatoes in water till cooked.

Step 2: Peel and mash potatoes till soft.

Step 3: Peel onions and chop fine.

Step 4: Clean chillies and chop fine.

Step 5: Mix mashed potatoes, onion, green chillies, and salt to taste and make small balls.

Step 6: Mix gram flour with water and salt to taste till a smooth and creamy batter is obtained.

Step 7: Dip the potato balls in batter. Take out and deep fry in oil on low fire.

Step 8: Take out when the balls are fried to a golden brown colour.

Result: 300 bondas.

(i) Obtain a task graph for making bondas in parallel. Clearly specify the tasks in your task graph. Assign appropriate time to do each task.

(ii) If 3 cooks are employed to do the job and 3 stoves are available for frying, how will you assign tasks to cooks?

- 2.12 The following expressions are to be evaluated:

$$a = \sin(x^2y) + \cos(xy^2) + \exp(-xy^2)$$

$$b = g(p) + e^{-xyf(y)} + h(x^2) + f(y)g(p)$$

$$c = f(u^2) + \sin(g(p)) + \cos^2 h(y^2)$$

(i) Obtain a task graph for calculating a, b, c .

(ii) Assuming 4 processors are available, obtain a task assignment to processors assuming the following timings for various operations.

Squaring = 1, Multiplication = 1

sin = cos = exponentiation = 2

$g(x) = h(x) = f(x) = 3$

(iii) Compare the time obtained by your assignment with the ideal time needed to complete the job with 4 processors.

- 2.13 A program is to be written to find the frequency of occurrence of each of the vowels a, e, i, o, u (both lower case and upper case) in a text of 1,00,000 words. Obtain at least 3

different methods of doing this problem in parallel. Assume 4 processors are available.

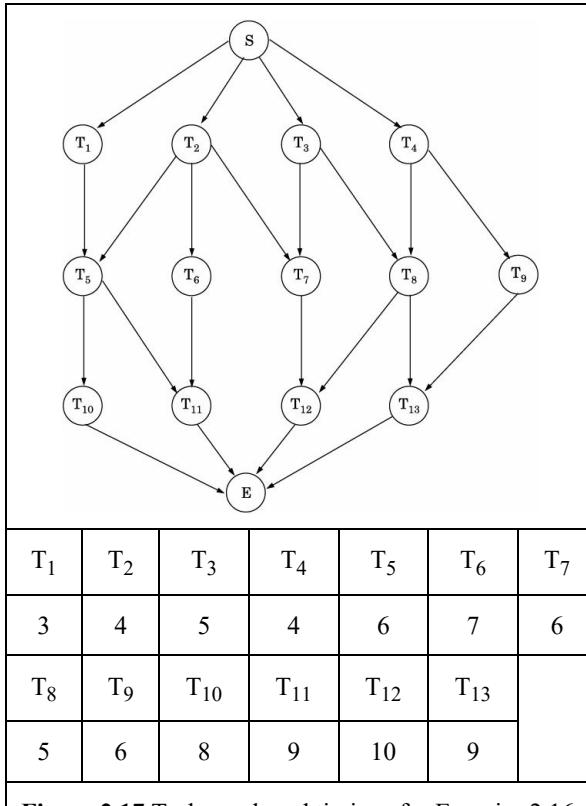
2.14 A company has 5000 employees and their weekly payroll is to be prepared. Each employee's record has employee number, hours worked, rate per hour, overtime worked (hours > 40 is considered overtime), overtime rate, percent to be deducted as tax and other deductions.

- (i) Write a sequential procedure to edit input for correctness, print the payroll, print sum of all the amounts paid, average payment per employee, a frequency table of payment made in 5 slabs and the employee number of the person(s) receiving maximum amount.
- (ii) Break up the sequential procedure into tasks and find which tasks can be carried out in parallel.
- (iii) If there are 3 processors available to carry out the job show how tasks can be assigned to processors.

2.15 For the task graph of Fig. 2.10 with the timings given in Table 2.2, assign tasks to three cooks.

- (i) Find the job completion time efficiency of your assignment.
- (ii) Repeat for 6 cooks. Compare with answers to (i) and comment.

2.16 A task graph with times of various tasks is given in Fig. 2.17. Assuming that 4 processors are available, assign tasks to processors.



BIBLIOGRAPHY

- Barney, B., *Introduction to Parallel Computing*, Lawrence Livermore Laboratories, USA, 2011. (Available on the Web).
- Carriero, N. and Gelernter, D., “How to Write Parallel Programs—A guide to the perplexed”, *Computing Surveys*, Vol. 21, No. 3, Sept. 1989, pp. 323–357.
- Frenkel, K.A. (Ed.), “Special Issue on Parallelism”, *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986.
- Gelernter, D. (Ed.), “Special Issue on Domesticating Parallelism”, *IEEE Computer*, Vol. 19, No. 8, Aug. 1986.
- Hillis, W.D. and Steele, G.L., “Data Parallel Algorithms”, *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1170–1183.
- Howe, C.D. and Moxon, B., “How to Program Parallel Processors”, *IEEE Spectrum*, Vol. 24, No. 9, Sept. 1987, pp. 36–41.
- Jamieson, L.H., Gannon, D.B. and Douglas, R.J. (Eds.), *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, 1987.
- Kuhn, R.H. and Padua, D.A., *Tutorial on Parallel Processing*, IEEE Computer Society Press, Los Angeles, 1981.
- Kung, H.T., “The Structure of Parallel Algorithms”, *Advances in Computers*, Vol. 19, M. Yovits (Ed.), Academic Press, New York, 1980, pp. 65–112.
- Lorin, H., *Parallelism in Hardware and Software: Real and Apparent Concurrency*, Prentice Hall, Englewood Cliffs, NJ, 1972.
- Rodrigue, G. (Ed.), *Parallel Computations*, Academic Press, New York, USA, 1982.
- West, S., “Beyond the One-track Mind”, *Science* 85, Nov. 1985, pp. 102–109.

Instruction Level Parallel Processing

In this chapter we will discuss how instruction level parallelism is employed in designing modern high performance microprocessors which are used as Processing Elements (PEs) of parallel computers. The earliest use of parallelism in designing PEs to enhance processing speed, was *pipelining*. Pipelining has been extensively used in Reduced Instruction Set Computers (RISC). RISC processors have been succeeded by superscalar processors that execute multiple instructions in one clock cycle. The idea in superscalar processor design is to use the parallelism available at the instruction level by increasing the number of arithmetic and functional units in a PE. This idea has been exploited further in the design of Very Long Instruction Word (VLIW) processors in which one instruction word encodes more than one operation. The idea of executing a number of instructions of a program in parallel by scheduling them suitably on a single processor has been the major driving force in the design of many recent processors. The number of transistors which can be integrated in a chip has been doubling every 24 months (Moore's law). Now a processor chip has over ten billion transistors. The major architectural question is how to effectively use this to enhance the speed of PEs. Currently most microprocessor chips use multiple processors called "cores" within a microprocessor chip. Clearly the trend is to design a high performance microprocessor as a parallel processor. In this chapter we will describe the use of pipelining in the design of PEs in some detail. We will also discuss superscalar and multithreaded processor architectures.

3.1 PIPELINING OF PROCESSING ELEMENTS

In the last chapter we discussed in detail the idea of pipelining. We saw that pipelining uses temporal parallelism to increase the speed of processing. One of the important methods of increasing the speed of PEs is pipelined execution of instructions.

Pipelining is an effective method of increasing the execution speed of processors provided the following “ideal” conditions are satisfied:

1. It is possible to break up an instruction into a number of independent parts, each part taking nearly equal time to execute.
2. There is so called locality in instruction execution. By this we mean that instructions are executed in sequence one after the other in the order in which they are written. If the instructions are not executed in sequence but “jump around” due to many branch instructions, then pipelining is not effective.
3. Successive instructions are such that the work done during the execution of an instruction can be effectively used by the next and successive instructions. Successive instructions are also independent of one another.
4. Sufficient resources are available in a processor so that if a resource is required by successive instructions in the pipeline it is readily available.

In actual practice these “ideal” conditions are not always satisfied. The non-ideal situation arises because of the following reasons:

1. It is not possible to break up an instruction execution into a number of parts each taking exactly the same time. For example, executing a floating point division will normally take much longer than say, decoding an instruction. It is, however, possible to introduce delays (if needed) so that different parts of an instruction take equal time.
2. All real programs have branch instructions which disturb the sequential execution of instructions. Fortunately statistical studies show that the probability of encountering a branch instruction is low; around 17%. Further it may be possible to predict when branches will be taken a little ahead of time and take some pre-emptive action.
3. Successive instructions are not always independent. The results produced by an instruction may be required by the next instruction and must be made available at the right time.
4. There are always resource constraints in a processor as the chip size is limited. It will not be cost effective to duplicate some resources indiscriminately. For example, it may not be useful to have more than two floating point arithmetic units.

The challenge is thus to make pipelining work under these non-ideal conditions.

In order to fix our ideas, it is necessary to take a concrete example. We will describe the architectural model of a small hypothetical computer. The computer is a *Reduced Instruction Set Computer* (RISC) which is designed to facilitate pipelined instruction execution. The computer is similar to SMAC2 (Small Computer 2) described by Rajaraman and Radhakrishnan [2008]. We will call the architecture we use SMAC2P (Small Computer 2 Parallel). It has the following units:

- A data cache (or memory) and an instruction cache (or memory). It is assumed that the instructions to be executed are stored in the instruction memory and the data to be read or written are stored in the data memory. These two memories have their own address and data registers. We call the memory address register of the instruction memory IMAR and that of data memory DMAR. The data register of the instruction memory is called IR and that of the data memory MDR.
- A Program Counter (PC) which contains the address of the next instruction to be executed. The machine is word addressable. Each word is 32 bits long.
- A register file with 32 registers. These are general purpose registers used to store operands and index values.
- The instructions are all of equal length and the relative positions of operands are fixed. There are 3 instruction types as shown in Fig. 3.1.
- The only instructions which access memory are load and store instructions. A load instruction reads a word from the data memory and stores it in a specified register in the register file and a store instruction stores the contents of a register in the register file in the data memory. This is called *load-store architecture*.
- An Arithmetic Logic Unit (ALU) which carries out one integer arithmetic or one logic operation in one clock cycle.

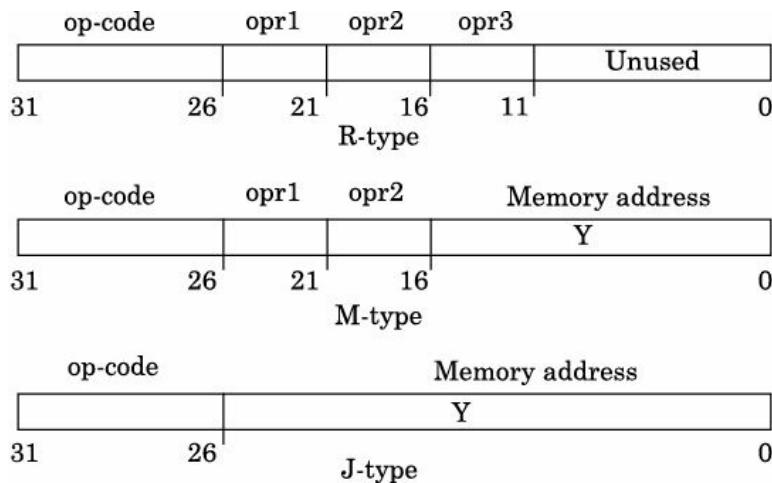


Figure 3.1 Instruction format of SMAC2P.

In the instruction formats shown in Figure 3.1, opr1, opr2, and opr3 are the addresses of registers in the register file. Y is the address of an operand stored in memory.

From the instruction formats of SMAC2P shown in Fig. 3.1, observe that with 6 bits op-code, 64 operations are possible. We will, however, consider a small set of instructions shown in Table 3.1.

TABLE 3.1 Instruction Set of SMAC2P

Operation mnemonic	Instruction		Semantics
	Type	Symbolic form	
ADD	R	ADD R ₁ , R ₂ , R ₃	C(R ₃) ← C(R ₁) + C(R ₂)
SUB	R	SUB R ₁ , R ₂ , R ₃	C(R ₃) ← C(R ₁) - C(R ₂)

MUL	R	MUL R ₁ , R ₂ , R ₃	C(R ₃) \leftarrow C(R ₁) * C(R ₂)
DIV	R	DIV R ₁ , R ₂ , R ₃	C(R ₃) \leftarrow C(R ₁) / C(R ₂)
LD	M	LD R ₂ , R ₃ , Y	C(R ₂) \leftarrow C(Y + C(R ₃)) (R ₃ has index value)
ST	M	ST R ₂ , R ₃ , Y	C(Y + C(R ₃)) \leftarrow C(R ₂)
JMP	J	JMP Y	PC \leftarrow Y
JMI	J	JMI Y	If negative bit in status register = 1 PC \leftarrow Y. Else do nothing
JEQ	M	JEQ R ₁ , R ₂ , Y	If (C(R ₁) = C(R ₂)) PC \leftarrow Y
LDI	M	LDI R ₂ , 0, Y	C(R ₂) \leftarrow Y (The 16 bits address treated as a 2's complement signed number)
INC	R	INC R ₆	C(R ₆) \leftarrow C(R ₆) + 1
DEC	R	DEC R ₆	C(R ₄) \leftarrow C(R ₆) - 1
BCT	M	BCT R ₄ , 0, Y	C(R ₄) \leftarrow C(R ₄) - 1 If (C(R ₄) = 0) then PC \leftarrow Y

This set has a reasonable variety of instructions to illustrate the basic ideas of pipeline processing. In Table 3.1 observe the load (LD) and store (ST) instructions. One of the registers is used as an index register to calculate the effective address. The other interesting instruction is load immediate (LDI) which uses the address part of the instruction as an operand. We will now examine the steps in the execution of the instructions of SMAC2P.

We will see that an instruction execution cycle can be broken up into 5 steps, each step taking one clock cycle. We will now explain each of these steps in detail.

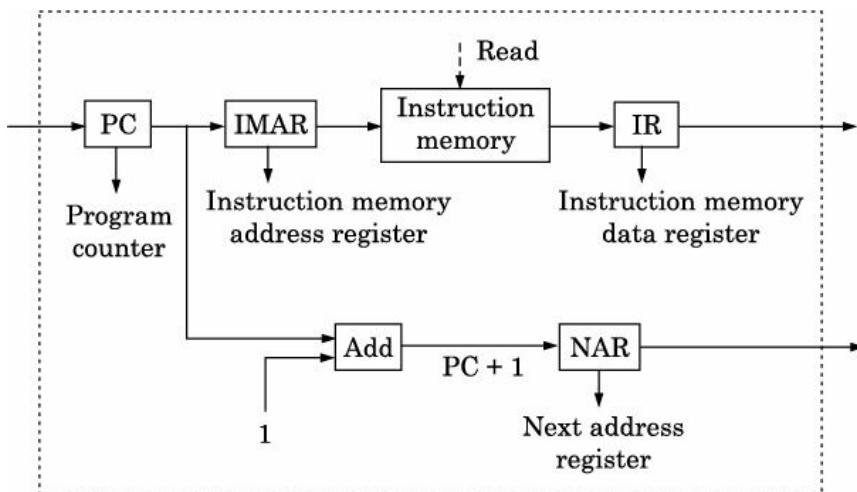


Figure 3.2 Block diagram for fetch instruction.

Step 1: Fetch an instruction from the instruction memory (FI)

The registers of relevance in this step are shown in Fig. 3.2. The data paths are shown as solid lines and a dotted line is used to show a control signal. The operations carried out during this step are:

```

IMAR ← PC
IR ← IMEM [IMAR] /* IMEM is instruction memory*/
NAR ← PC + 1 /* NAR is next address Register */

```

All these operations are carried out in one cycle. We have assumed a word addressable memory. $NAR \leftarrow PC + 4$ if the memory is byte addressable.

Step 2: Decode instruction and fetch register (DE)

In this step the instruction is decoded and the operand fields are used to read values of specified registers. As the structure of the instruction is simple with fixed fields, decoding and register fetch can be completed in one clock cycle.

The least significant 16 bits of M-type instructions (which is used as an immediate operand or jump address) are also retrieved and stored in a register named IMM. The operations carried out during this step are shown below:

```

B1 ← Reg [ IR21 ... 25 ]
B2 ← Reg [ IR16 ... 20 ]
IMM ← IR0 ... 15 (M-type instruction)

```

All these operations are carried out in one clock cycle. Observe that the outputs from the register file are stored in two buffer registers B1 and B2. The data flow for this step is shown in Fig. 3.3.

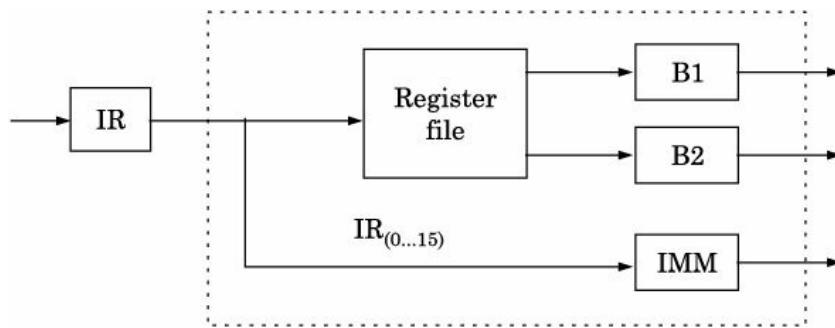


Figure 3.3 Data flow for decode and fetch register step.

Step 3: Execute instruction and calculate effective address (EX)

In this step the ALU operations are carried out. The instructions where ALU is used may be classified as shown below:

- Two registers as operands

$$B3 \leftarrow B1 <\text{operation}> B2$$

where $<\text{operation}>$ is ADD, SUB, MUL or DIV and
B3 is the register where the ALU output is stored

- Increment/Decrement (INC/DEC/BCT)

$$B3 \leftarrow B1 + 1 \text{ (Increment)}$$

$$B3 \leftarrow B1 - 1 \text{ (Decrement or BCT)}$$
- Branch on equal (JEQ)

$$B3 \leftarrow B1 - B2$$

If $B3 = 0$, set zero flag in status register = 1

- Effective address calculation (for LD/ST)

For load and store instructions, the effective address is stored in B3.

$$B3 \leftarrow B2 + IMM$$

The data flow for these operations is shown in Fig. 3.4. Observe that multiplexers (MUXes) have been used to select the appropriate alternate values to be input to ALU and output from ALU. Control signals (shown by dotted lines) are necessary for the MUXes.

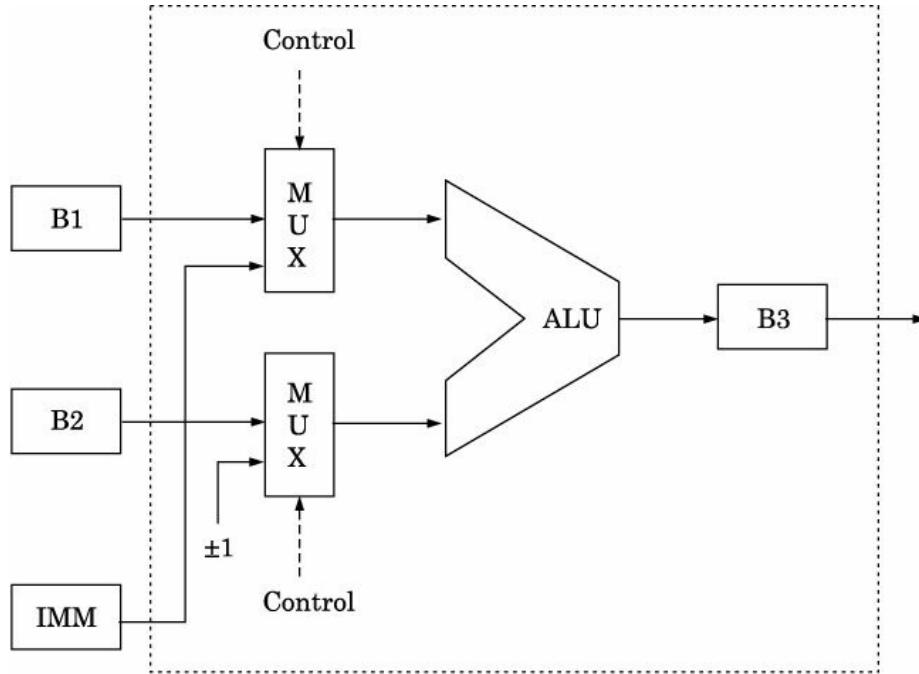


Figure 3.4 Data flow for ALU operation.

Step 4: Memory access (MEM)

In this step the load/store and address determination for branch instructions are carried out. The address for load or store operation was computed in the previous step and is in B3. The following operations are carried out for load/store.

$$DMAR \leftarrow B3$$

where DMAR is data memory address register

$$MDR \leftarrow DMEM [DMAR] \text{ load instruction (LD)}$$

$$DMEM [DMAR] \leftarrow MDR \leftarrow B1 \text{ store instruction (ST)}$$

For branch instructions we should find out the address of the next instruction to be executed in this step. The four branch instructions in our computer are JMP, JMI, JEQ and BCT (See Table 3.1).

The PC value in these four cases is as follows:

For JMP

$$PC \leftarrow IR_{0...25}$$

For JMI

$$\text{If (negative flag} = 1) PC \leftarrow IR_{0...25}$$

$$\text{else } PC \leftarrow NAR$$

For JEQ and BCT

$$\text{If (zero flag} = 1) PC \leftarrow IR_{0...15}$$

else $PC \leftarrow NAR$

For other instructions:

$PC \leftarrow NAR$

The data flow for this step is shown in Fig. 3.5.

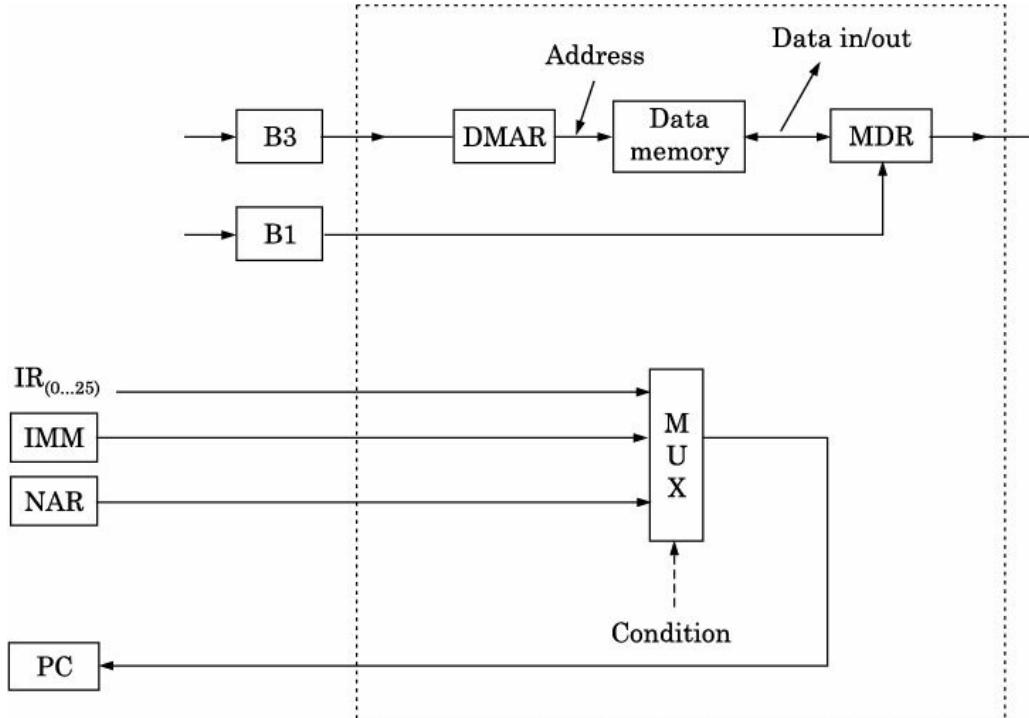


Figure 3.5 Data flow for load/store, next address step.

(Remember that $IMM = IR_{0...15}$ and JMP and JMI are J-type with $IR_{(0...25)}$)

Step 5: Store register (SR)

In this step the result of ALU operation or load immediate or load from memory instruction is stored back in the specified register in the register file. The three cases are:

Store ALU output in register file

$$Reg[IR_{11 \dots 15}] \leftarrow B3$$

Load immediate (LDI)

$$Reg [IR_{21 \dots 25}] \leftarrow IMM = IR_{0 \dots 15}$$

Load data retrieved from data memory in register file (LD)

$$Reg[IR_{21 \dots 25}] \leftarrow MDR$$

The data flow for this step is shown in Fig. 3.6.

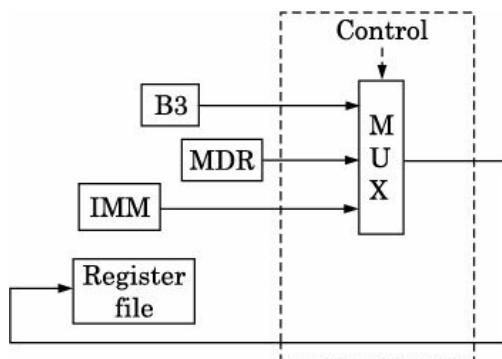


Figure 3.6 Data flow for the store register step.

These five steps were discussed individually. All these are combined to show the complete data flow during the instruction fetch and execute cycle in Fig. 3.7. In Table 3.2 we have consolidated the data flows, which occur during the 5 steps of instruction execution.

If we examine Fig. 3.7 carefully we find that unconditional branch (JMP) instruction need not take 5 clock cycles. It needs only 2 cycles because we may skip Steps 3, 4, 5 as soon as we decode the instruction and find it is an unconditional branch. However, for simplicity we have consolidated all jump instructions in Step 4.

Conditional branch instructions JEQ, JMI and BCT require 3 cycles. Instructions INC, DEC and LDI do not need a memory access cycle and can complete in 3 cycles.

The proportion of these instructions in actual programs has to be found statistically by executing a large number of programs and counting the number of such instructions. Such studies have been conducted on a large number of benchmark programs and the results have been reported in the literature. A ready reference is the book by Patterson and Hennessy [2012]. Data provided by such studies are often used while designing computers.

We will now calculate the average number of clock cycles needed to carry out an instruction in SMAC2P. If the percentage of unconditional branches is 5%, that of conditional branches 15% and immediate instructions 10%, then the average number of clock cycles per instruction is

$$\Sigma (\% \text{ instruction of type } p \times \text{clock cycles needed by instruction of type } p)/100.$$

In our example average clock cycles per instruction = $(5 \times 2 + 15 \times 3 + 10 \times 3 + 70 \times 5)/100 = 4.35$.

(This is for the non-pipelined case.)

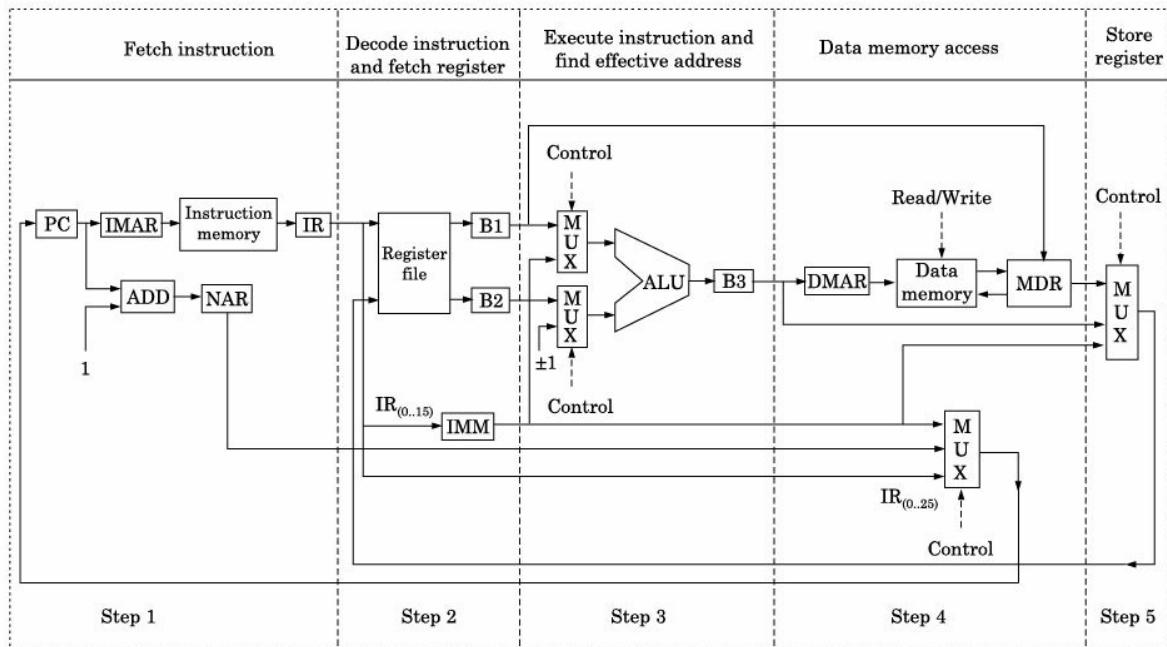


Figure 3.7 Consolidated data flow showing 5 steps in instruction execution.

TABLE 3.2 Consolidation of Data Flows of SMAC2P

<i>Step in execution</i>	<i>Data flow during step</i>	<i>Remarks</i>
Step 1 Fetch instruction (FI)	$IMAR \leftarrow PC$ $IR \leftarrow IMEM[IMAR]$ $NAR \leftarrow PC + 1$	Done for all instructions
Step 2 Decode instruction and fetch register (DE)	$B1 \leftarrow Reg[IR_{21..25}]$ $B2 \leftarrow Reg[IR_{16..20}]$ $IMM \leftarrow IR_{0..15}$	Done for all instructions
Step 3 Execute instruction or find effective address (EX)	$B3 \leftarrow B1 <op> B2$ or $B3 \leftarrow B1 \pm 1$ or $B3 \leftarrow B1 - B2$ If $B3 = 0$ set zero flag or $B3 \leftarrow B2 + IR_{0..15}$	ADD, SUB, MUL, DIV INC, DEC JEQ, BCT Effective address.
Step 4 Access memory Find address for branching (MEM)	$DMAR \leftarrow B3$ $MDR \leftarrow DMEM [DMAR]$ or $DMEM[DMAR] \leftarrow MDR$ $\leftarrow B1$ $PC \leftarrow IR_{0..25}$ $PC \leftarrow IR_{0..25}$ If negative flag is set or $PC \leftarrow IR_{0..15}$ If zero flag is set	Load memory address LD ST (store in memory) JMP JMI JEQ, BCT
Step 5 Store result in register (SR)	$Reg[IR_{11..15}] \leftarrow B3$ $Reg[IR_{21..25}] \leftarrow IMM$ $Reg[IR_{21..25}] \leftarrow MDR$	Result of ALU operation to register file LDI LD

If we want to execute the instructions of SMAC2P in pipelined mode, the five steps will be carried out in successive instructions as shown in Fig. 3.8. It is clear from the figure that the first instruction will be completed at the end of 5 clock cycles and if all conditions are “ideal” one instruction will be completed at the end of each cycle. Observe from Fig. 3.8 that from clock cycle 5 onwards, all 5 steps of instruction execution are to be carried out simultaneously. This implies that values in some of the registers have to be carried over from step to step. For instance, the value of IR read during Step 1 of instruction i should not be overwritten when instruction $(i + 1)$ is fetched. We will thus assume that between pipeline stages there are buffer registers to store all important data and data are transferred at each clock cycle from one stage to the next (see Fig. 3.9).

<i>Instructions</i>	1	2	3	4	5	6	7	8	9	10
i	FI	DE	EX	MEM	SR					
$i + 1$		FI	DE	EX	MEM	SR				
$i + 2$			FI	DE	EX	MEM	SR			
$i + 3$				FI	DE	EX	MEM	SR		

i + 4				FI	DE	EX	MEM	SR	
i + 5					FI	DE	EX	MEM	SR

FI: Fetch instruction
DE: Decode instruction
EX: ALU operation (Execute instruction and compute effective address)
MEM: Access data memory
SR: Store result in register

Figure 3.8 Pipelined execution of instructions of SMAC2P.

We will now find out the speedup due to pipelining in the ideal case.

Ideal case

Let the total number of instructions executed = m .

Let the number of clock cycles per instruction = n .

Time taken to execute m instructions with no pipelining = mn cycles

Time taken with pipelining (ideal) = $n + (m - 1)$ cycles.

$$\text{Speedup due to pipelining} = \frac{mn}{n + (m - 1)} = \frac{n}{\frac{n}{m} + \frac{(m - 1)}{m}}$$

If $m \gg n$ then $(n/m) \approx 0$ and $(m - 1)/m \approx 1$.

Thus speedup (ideal) = n .

The first non-ideal condition occurs due to the fact that extra buffer registers are introduced between pipeline stages as pointed out (see Fig. 3.9) and use of these registers lengthens each clock cycle. Thus, instead of 1 clock cycle per instruction it will take $(1 + e)$ clock cycles where e is a small fraction $\ll 1$.

Therefore, time taken with pipeline processing (non-ideal) = $n + (m - 1)(1 + e)$ cycles

$$\text{Speedup} = \frac{mn}{n + (m - 1)(1 + e)} = \frac{n}{(1 + e)}$$

If a non-pipelined computer takes 5 clock cycles per instruction and $e = 0.1$, the speedup is $5/1.1 = 4.5$.

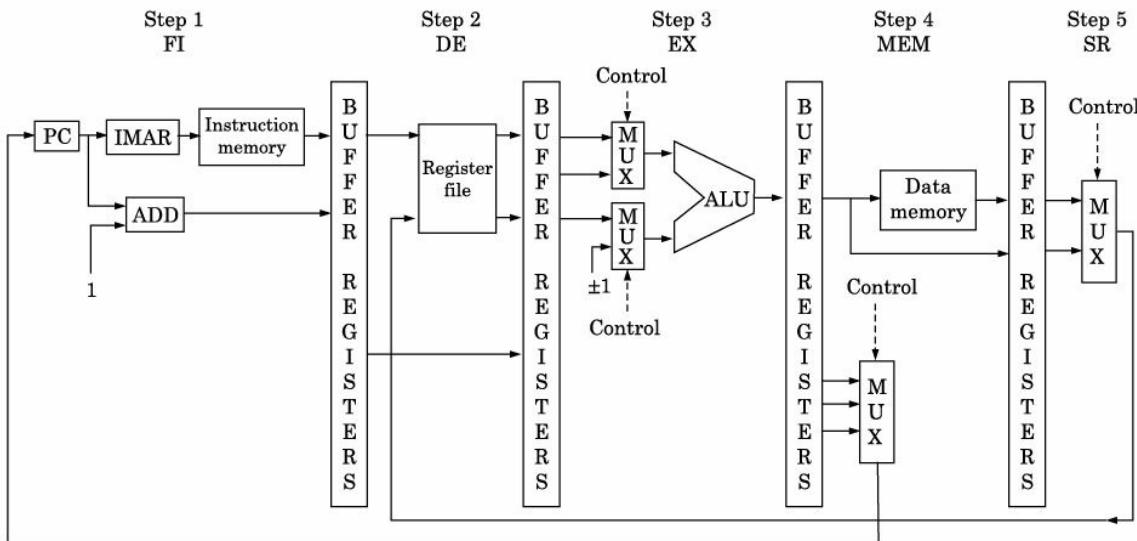


Figure 3.9 Pipelined instruction execution data flow with buffer registers.

In practice, the speedup is lower as the number of clock cycles needed per instruction in non-pipelined mode is smaller than n , the number of pipeline stages as we saw earlier in this section.

EXAMPLE 3.1

A non-pipelined computer uses a 10 ns clock. The average number of clock cycles per instruction required by this machine is 4.35. When the machine is pipelined it requires a 11 ns clock. We now find out the speedup due to pipelining.

Assume that the depth of the pipeline is n and the number of instructions executed is N . The speed-up due to pipelining is $(4.35 \times 10 \times N)/((n + N - 1) \times 11)$. As $(n + N - 1)/N$ is nearly 1, ($N \gg n$) the speedup is $(4.35 \times 10)/11 = 3.95$.

Besides the need to increase the period of each clock cycle, the ideal condition of being able to complete one instruction per clock cycle in pipeline execution is often not possible due to other non-ideal situations which arise in practice.

We will discuss the common causes for these non-ideal conditions and how to alleviate them in the next section.

3.2 DELAYS IN PIPELINE EXECUTION

Delays in pipeline execution of instructions due to non-ideal conditions are called *pipeline hazards*. As was pointed out earlier the non-ideal conditions are:

- Available resources in a processor are limited.
- Successive instructions are not independent of one another. The result generated by an instruction may be required by the next instruction.
- All programs have branches and loops. Execution of a program is thus not in a “straight line”. An ideal pipeline assumes a continuous flow of tasks.

Each of these non-ideal conditions causes delays in pipeline execution. We can classify them as:

- Delays due to resource constraints. This is known as *structural hazard*.
- Delays due to data dependency between instructions. This is known as *data hazard*.
- Delays due to branch instructions or control dependency in a program. This is known as *control hazard*.

We will discuss each of these next and examine how to reduce these delays.

3.2.1 Delay Due to Resource Constraints

Pipelined execution may be delayed due to the non-availability of resources when required during execution of an instruction. Referring to Fig. 3.8, during clock cycle 4, Step 4 of instruction i requires read/write in data memory and instruction $(i + 3)$ requires an instruction to be fetched from the instruction memory. If one common memory is used for both data and instructions (as is done in many computers) only one of these instructions can be carried out and the other has to wait. Forced waiting of an instruction in pipeline processing is called *pipeline stall*. In the design of our computer we used two different memories (with independent read/write ports) for instructions and data to avoid such a pipeline stall.

Pipeline execution may also be delayed if one of the steps in the execution of an instruction takes longer than one clock cycle. Normally a floating point division takes longer than, say, an integer addition. If we assume that a floating point operation takes 3 clock cycles to execute, the progress of pipeline execution is shown in Fig. 3.10. Instruction $(i + 1)$ in this example is a floating point operation which takes 3 clock cycles. Thus, the execution step of instruction $(i + 2)$ cannot start at clock cycle 5 as the execution unit is busy. It has to wait till clock cycle 7 when the execution unit becomes free to carry it out. This is a pipeline stall due to non-availability of required resource (structural hazard). How do we avoid this? One way is to speedup the floating point execution by using extra hardware so that it also takes one clock cycle. This will require more hardware resources. Before deciding whether it is worthwhile doing it, we compute the delay due to this resource constraint and if it is not too large we may decide to tolerate it. We have calculated the speedup due to pipelining in the ideal case as n , the number of stages in the pipeline. We will now calculate the speedup when there is a pipeline delay due to resource constraint.

Loss of Speedup Due to Resource Non-availability

Assume a fraction of f of total number of instructions are floating point instructions which

take $(n + k)$ clock cycles each. Assume m instructions are being executed.

Time to execute m instructions with no pipelining = $m(1 - f)n + mf(n + k)$ clock cycles.

Time taken with pipelining = $n + (m - 1)(1 + kf)$ clock cycles.

(Remember that the first instruction takes n cycles)

$$\begin{aligned}\text{Speedup} &= \frac{m(1 - f)n + mf(n + k)}{n + (m - 1)(1 + kf)} \\ &= \frac{n + kf}{\frac{n}{m} + \frac{m-1}{m}(1 + kf)} \approx \frac{n}{1 + kf}\end{aligned}$$

(Assuming $m \gg n$ and $n \gg kf$)

If 10% of the instructions are floating point instructions and floating point instructions take 2 extra clock cycles to execute, we have $k = 2$ and $f = 0.1$.

$$\therefore \text{Speedup} = n/(1 + 0.2) = 0.833n$$

As the loss of efficiency is less than 16.6% it may not be worthwhile expending considerable resources to speedup floating point execution in this example. In Fig. 3.10, which illustrates pipeline delay due to resource shortage, observe that in cycle 5 of instruction $i + 2$, the hardware detects that the execution unit is busy and thus suspends pipeline execution. In some pipeline designs whenever work cannot continue in a particular cycle, the pipeline is “locked”. All the pipeline stages except the one that is yet to finish (in this case the execution step of instruction $(i + 1)$) are stalled. In such a case the pipeline diagram will be the one as shown in Fig. 3.11. Observe that even though the decode step (DE) of instruction $(i + 3)$ can go on (as the decoding unit is free) because of the locking of the pipeline no work is done during this cycle. The main advantage of this method of locking the pipeline is ease of hardware implementation. Also by such locking we ensure that successive instructions will complete in the order in which they are issued. Many machines have used this technique. However, recent machines do not always lock a pipeline and let instructions continue if there are no resource constraints or other problems which we discuss next. In such a case completion of instruction may be “out of order”, that is, a later instruction may complete before an earlier instruction. If this is logically acceptable in a program, pipeline need not be locked.

<i>Clock cycle → Instructions</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	FI	DE	EX	MEM	SR											
i+1		FI	DE	EX	EX	EX	MEM	SR								
i+2			FI	DE	X	X	EX	MEM	SR							
i+3				FI	DE	X	X	EX	MEM	SR						
i+4					FI	DE	X	X	EX	EX	EX	MEM	SR			
i+5						FI	DE	X	X	X	X	EX	MEM	SR		
i+6							FI	DE	X	X	X	X	EX	MEM	SR	
i+7								FI	DE	X	X	X	X	EX	MEM	SR

X indicates idle period or stall of an instruction.

Figure 3.10 Delay in pipeline due to resource constraint.

<i>Clock cycle → Instructions</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	FI	DE	EX	MEM	SR											
i+1		FI	DE	EX	EX	EX	MEM	SR								
i+2			FI	DE	X	X	EX	MEM	SR							
i+3				FI	X	X	DE	EX	MEM	SR						
i+4					X	X	FI	DE	EX	EX	EX	MEM	SR			
i+5						X	X	X	FI	DE	X	X	EX	MEM	SR	
i+6									FI	X	X	DE	EX	MEM	SR	
i+7										X	X	FI	DE	EX	MEM	SR

Figure 3.11 Delay in pipeline with pipeline locking.

3.2.2 Delay Due to Data Dependency

Pipeline execution is delayed due to the fact that successive instructions are not always independent of one another. The result produced by an instruction may be needed by succeeding instructions and the results may not be ready when needed. Consider the following sequence of instructions:

ADD R1, R2, R3	$C(R3) \leftarrow C(R1) + C(R2)$
MUL R3, R4, R5	$C(R5) \leftarrow C(R3) * C(R4)$
SUB R7, R2, R6	$C(R6) \leftarrow C(R7) - C(R2)$
INC R3	$C(R3) \leftarrow C(R3) + 1$

Referring to Fig. 3.12, we see that the value of R3 (the result of the ADD operation) will not be available till clock cycle 6. Thus, the MUL operation should not be executed in cycle 4 as the value of R3 it needs will not be stored in the register file by the ADD operation and thus not available. Thus, the EX step of MUL instruction is delayed till cycle 6. This stall is due to *data dependency* and is called *data hazard* as the required data is not available when needed. The next instruction SUB R7, R2, R6 does not need R3. It also reads R2 whose value is not changed by MUL. Thus, it can proceed without waiting as shown in Fig. 3.12. The next instruction INC R3 can be executed in clock period 6 as R3 is already available in the register file at that time. However, as ALU is being used in clock cycle 6, it has to wait till cycle 7 to increment R3. The new value of R3 will be available only in cycle 9. This delay can be eliminated if the computer has a separate unit to execute INC operation.

<i>Clock cycle →</i>	1	2	3	4	5	6	7	8	9	10
ADD R1, R2, R3	FI	DE	EX	MEM	SR					
MUL R3, R4, R5		FI	DE	X	X	EX	MEM	SR		

SUB R7, R2, R6			FI	DE	EX	MEM	SR			
INC R3				FI	DE	X	X	EX	MEM	SR
Figure 3.12 Delay in pipeline due to data dependency.										

Observe that the third instruction SUB R7, R2, R6 completes execution before the previous instruction. This is called *out-of-order completion* and may be unacceptable in some situations. Thus, many machine designers lock the pipeline when it is stalled (delayed). In such a case the pipeline execution will be as shown in Fig. 3.13. Observe that this preserves the order of execution but the overall execution takes 1 more cycle as compared to the case where pipeline is not locked.

Clock cycle →	1	2	3	4	5	6	7	8	9	10
ADD R1, R2, R3	FI	DE	EX	MEM	SR					
MUL R3, R4, R5		FI	DE	X	X	EX	MEM	SR		
SUB R7, R2, R6			FI	X	X	DE	EX	MEM	SR	
INC R3				X	X	FI	DE	EX	MEM	SR

Figure 3.13 Locking pipeline due to data dependency.

A question which naturally arises is how we can avoid pipeline delay due to data dependency. There are two methods available to do this. One is a hardware technique and the other is a software technique. The hardware method is called *register forwarding*. Referring to Fig. 3.12, the result of ADD R1, R2, R3 will be in the buffer register B3. Instead of waiting till SR cycle to store it in the register file, one may provide a path from B3 to ALU input and bypass the MEM and SR cycles. This technique is called register forwarding. If register forwarding is done during ADD instruction, there will be no delay at all in the pipeline. Many pipelined processors have register forwarding as a standard feature. Hardware must of course be there to detect that the next instruction needs the output of the current instruction and should be fed back to ALU. In the following sequence of instructions not only MUL but also SUB needs the value of R3. Thus, the hardware should have a facility to forward R3 to SUB also.

ADD R1, R2, R3

MUL R3, R4, R5

SUB R3, R2, R6

Consider another sequence of instructions

LD R2, R3, Y
ADD R1, R2, R3
MUL R4, R3, R1
SUB R7, R8, R9

$C(R2) \leftarrow C(Y + C(R3))$
 $C(R3) \leftarrow C(R1) + C(R2)$
 $C(R1) \leftarrow C(R4) \times C(R3)$
 $C(R9) \leftarrow C(R7) - C(R8)$

Clock cycle →	1	2	3	4	5	6	7	8	9	10
LD R2, R3, Y	FI	DE	EX	MEM	SR					
ADD R1, R2, R3		FI	DE	X	X	EX	MEM	SR		

MUL R4, R3, R1			FI	DE	X	X	EX	MEM	SR	
SUB R7, R8, R9				FI	DE	X	X	EX	MEM	SR
(a)										
Clock cycle →	1	2	3	4	5	6	7	8	9	
LD R2, R3, Y	FI	DE	EX	MEM	SR					
SUB R7, R8, R9		FI	DE	EX	MEM	SR				
ADD R1, R2, R3			FI	DE	X	EX	MEM	SR		
MUL R4, R3, R1				FI	DE	X	EX	MEM	SR	
(b)										

Figure 3.14 Reducing pipeline stall by software scheduling.

The pipeline execution with register forwarding is shown in Fig. 3.14(a). Observe that the LD instruction stores the value of R2 only at the end of cycle 5. Thus, ADD instruction can perform addition only in cycle 6. MUL instruction even though it uses register forwarding is delayed first due to the non-availability of R3 as ADD instruction is not yet executed and in the next cycle as the execution unit is busy. The next instruction SUB, even though not dependent on any of the previous instructions is delayed due to the non-availability of the execution unit. Hardware register forwarding has not eliminated pipeline delay. There is, however, a software scheduling method which reduces delay and in some cases completely eliminates it. In this method, the sequence of instructions is reordered without changing the meaning of the program. In the example being considered SUB R7, R8, R9 is independent of the previous instructions. Thus, the program can be rewritten as:

```

LD      R2, R3, Y
SUB    R7, R8, R9
ADD    R1, R2, R3
MUL    R4, R3, R1

```

The pipeline execution of these instructions is shown in Fig. 3.14(b). Observe that SUB can be executed without any delay. However, ADD is delayed by a cycle as a value is stored in R2 by LD instruction only in cycle 5. This rescheduling has reduced delay by one clock cycle but has not eliminated it. If the program has two or more instructions independent of LD and ADD then they can be inserted between LD and ADD eliminating all delays.

3.2.3 Delay Due to Branch Instructions

The last and the most important non-ideal condition which affects the performance of pipeline processing is branches in a program. A branch disrupts the normal flow of control. If an instruction is a branch instruction (which is known only at the end of instruction decode step), the next instruction may be either the next sequential instruction (if the branch is not taken) or the one specified by the branch instruction (if the branch is taken). In SMAC2P the branch instructions are JMP (unconditional jump), JMI, JEQ and BCT(conditional jumps). From Fig. 3.7 we see that the jump address for JMP is known at decoding time of Step 2. For JMI also the jump address could be found during the decode step as the negative flag would have been set by the previous instruction. For JEQ and BCT instructions, however, only after

executing Step 3 one would know whether to jump or not. In our design we have shown the branch address assignment at the end of Step 4 uniformly for all branch instructions. Let us now examine the effect of this on pipeline execution using Fig. 3.15. In this figure instruction $(i + 1)$ is a branch instruction. The fact that it is a branch will be known to the hardware only at the end of decode (DE) step. Meanwhile, the next instruction would have been fetched. In our design whether the fetched instruction should be executed or not would be known only at the end of MEM cycle of the branch instruction (see Fig. 3.15). Thus, this instruction will be delayed by two cycles. If branch is not taken the fetched instruction could proceed to DE step. If the branch is taken then the instruction fetched is not used and the instruction from the branch address is fetched. Thus, there is 3 cycles delay in this case. Let us calculate the reduction of speedup due to branching.

Clock cycle instructions →	1	2	3	4	5	6	7	8	9	10
i	FI	DE	EX	MEM	SR					
$(i+1)$ branch		FI	DE	EX	MEM	SR				
$(i+2)$			FI	X	X	FI	DE	EX	MEM	SR
Figure 3.15 Delay in pipeline execution due to branch instruction.										

EXAMPLE 3.2

In our pipeline the maximum ideal speedup is 5. Let the percentage of unconditional branches in a set of typical programs be 5% and that of conditional branches be 15%. Assume that 80% of the conditional branches are taken in the programs.

$$\text{No. of cycles per instruction (ideal case)} = 1$$

$$\text{Average delay cycles due to unconditional branches} = 3 \times 0.05 = 0.15$$

$$\text{Average delay cycles due to conditional branches}$$

$$= \text{delay due to taken branches} + \text{delay due to non taken branches}$$

$$= 3 \times (0.15 \times 0.8) + 2 \times (0.15 \times 0.2)$$

$$= 0.36 + 0.06 = 0.42$$

$$\therefore \text{Speedup with branches} = \frac{5}{1 + 0.15 + 0.42} \approx 3.18$$

$$\% \text{ loss of speedup due to branches} = 36.4\%.$$

Thus, it is essential to reduce pipeline delay when branches occur.

There are two broad methods of reducing delay due to branch instructions. One of them is to tackle the problem at the hardware level and the other is to have software aids to reduce delay. We will first examine how hardware can be modified to reduce delay cycles.

3.2.4 Hardware Modification to Reduce Delay Due to Branches

The primary idea is to find out the address of the next instruction to be executed as early as possible. We saw in SMAC2P design the branch address of JMP and JMI can be found at the end of DE step. If we put a separate ALU in the decode stage of the pipeline to find (B1-B2) for JEQ and BCT instructions (see Table 3.2), we can make a decision to branch or not at the end of Step 2. This is shown in Table 3.3 by modifying Step 2 of Table 3.2.

TABLE 3.3 Modified Decode Step of SMAC2P to Reduce Stall During Branch

<i>Step in execution</i>	<i>Data flow during step</i>	<i>Remarks</i>
Decode instruction fetch register and find branch address	$B1 \leftarrow \text{Reg}[\text{IR}_{21..25}]$	
	$B2 \leftarrow \text{Reg}[\text{IR}_{16..20}]$	
	$B2 \leftarrow (B1 - B2)$	
	If zero flag $\text{PC} \leftarrow \text{IR}_{0..15}$	For BCT and JEQ
	$\text{PC} \leftarrow \text{IR}_{0..25}$	For JMP
	If negative flag $\text{PC} \leftarrow \text{IR}_{0..25}$	For JMI

It is clear from Table 3.3 that we have introduced an add/subtract unit in the decode stage to implement JEQ and BCT instructions in Step 2. This add/subtract unit will also be useful if an effective address is to be computed. By adding this extra circuitry we have reduced delay to 1 cycle if branch is taken and to zero if it is not taken. We will calculate the improvement in speedup with this hardware.

EXAMPLE 3.3

Assume again 5% unconditional jumps, 15% conditional jumps and 80% of conditional jumps as taken.

Average delay cycles with extra hardware

$$= 1 \times 0.05 + 1 \times (0.15 \times 0.8) = 0.05 + 0.12 = 0.17$$

\therefore Speedup with branches $= 5/1.17 \approx 4.27$

% loss of speedup $= 14.6\%$.

Thus, we get a gain of 22% in speedup due to the extra hardware which is well worth it in our example.

In SMAC2P there are only a small number of branch instructions and the extra hardware addition is simple. Commercial processors are more complex and have a variety of branch instructions. It may not be cost effective to add hardware of the type we have illustrated. Other technologies have been used which are somewhat more cost-effective in certain processors.

We will discuss two methods both of which depend on predicting the instruction which will be executed immediately after a branch instruction. The prediction is based on the execution time behaviour of a program. The first method we discuss is less expensive in the use of hardware and consequently less effective. It uses a small fast memory called a *branch prediction buffer* to assist the hardware in selecting the instruction to be executed immediately after a branch instruction. The second method which is more effective also uses a fast memory called a *branch target buffer*. This memory, however, has to be much larger and requires more control circuitry. Both ideas can, of course, be combined. We will first discuss the use of branch prediction buffer.

Branch Prediction Buffer

In this technique some of the lower order bits of the address of branch instructions in a program segment are used as addresses of the branch prediction buffer memory. This is shown in Fig. 3.16. The contents of each location of this buffer memory is the address of the next instruction to be executed if the branch is taken. In addition, two bits are used to predict whether a branch will be taken when a branch instruction is executed. If the prediction bits are 00 or 01, the prediction is that the branch will not be taken. If the prediction bits are 10 or

11 then the prediction is that the branch will be taken. While executing an instruction, at the DE step of the pipeline, we will know whether the instruction is a branch instruction or not. If it is a branch instruction, the low order bits of its address are used to look up the branch prediction buffer memory. Initially the prediction bits are 00. Table 3.4 is used to change the prediction bits in the branch prediction buffer memory.

TABLE 3.4 How Branch Prediction Bits are Changed								
Current prediction bits	00	00	01	01	10	10	11	11
Branch taken?	Y	N	Y	N	Y	N	Y	N
New prediction bits	01	00	10	00	11	01	11	10

The prediction bits are examined. If they are 10 or 11, control jumps to the branch address found in the branch prediction buffer. Otherwise the next sequential instruction is executed. Experimental results show that the prediction is correct 90% of the time. With 1000 entries in the branch prediction buffer, it is estimated that the probability of finding a branch instruction in the buffer is 95%. The probability of finding the branch address is at least $0.9 \times 0.95 = 0.85$.

There are two questions which must have occurred to the reader. The first is “How many clock cycles do we gain, if at all, with this scheme?” The second is “Why should there be 2 bits in the prediction field? Would it not be sufficient to have only one bit?” We will take up the first question. In SMAC2P with hardware enhancements there will not be any gain by using this scheme as the branch address will be found at the DE step of the pipeline. Without any hardware enhancements of SMAC2P two clock cycles will be saved when this buffer memory is used provided the branch prediction is correct. In many machines where address computation is slow this buffer will be very useful.

Address	Contents	Prediction bits
Low order bits of branch instruction address	Address where branch will jump	2 bits

Figure 3.16 The fields of a branch prediction buffer memory.

A single bit predictor incorrectly predicts branches more often, particularly in most loops, compared to a 2-bit predictor. Thus, it has been found more cost-effective to use a 2-bit predictor.

Branch Target Buffer

Unlike a branch prediction buffer, a branch target buffer is used at the instruction fetch step itself. The various fields of the Branch Target Buffer memory (BTB) are shown in Fig. 3.17.

Address	Contents	Prediction bits
Address of branch instruction	Address where branch will jump	1 or 2 bits (optional)

Figure 3.17 The fields of a branch target buffer memory.

Observe that the address field has the complete address of all branch instructions. The contents of BTB are created dynamically. When a program is executed whenever a branch statement is encountered, its address and branch target address are placed in BTB. Remember the fact that an instruction is a branch will be known only during the decode step. At the end of execution step, the target address of the branch will be known if branch is taken. At this

time the target address is entered in BTB and the prediction bits are set to 01. Once a BTB entry is made, it can be accessed at instruction fetching phase itself and target address found. Typically when a loop is executed for the first time, the branch instruction governing the loop would not be found in BTB. It will be entered in BTB when the loop is executed for the first time. When the loop is executed the second and subsequent times the branch target would be found at the instruction fetch phase itself thus saving 3 clock cycles delay. We explain how the BTB is used with a flow chart in Fig. 3.18. Observe that left portion of this flow chart describes how BTB is created and updated dynamically and the right portion describes how it is used. In this flow chart it is assumed that the actual branch target is found during DE step. This is possible in SMAC2P with added hardware. If no hardware is added then the fact that the predicted branch and actual branch are same will be known only at the execution step of the pipeline. If they do not match, the instruction fetched from the target has to be removed and the next instruction in sequence should be taken for execution.

Observe that we have to search BTB to find out whether the fetched instruction is in it. Thus, BTB cannot be very large. About 1000 entries are normally used in practice. We will now compute the reduction in speedup due to branches when BTB is employed using the same data as in Example 3.2.

EXAMPLE 3.4

Assume unconditional branches = 5%.

Conditional branches = 15%

Taken branches = 80% of conditional

For simplicity we will assume that branch instructions are found in BTB with probability 0.95. We will assume that in 90% cases, the branch prediction based on BTB is correct.

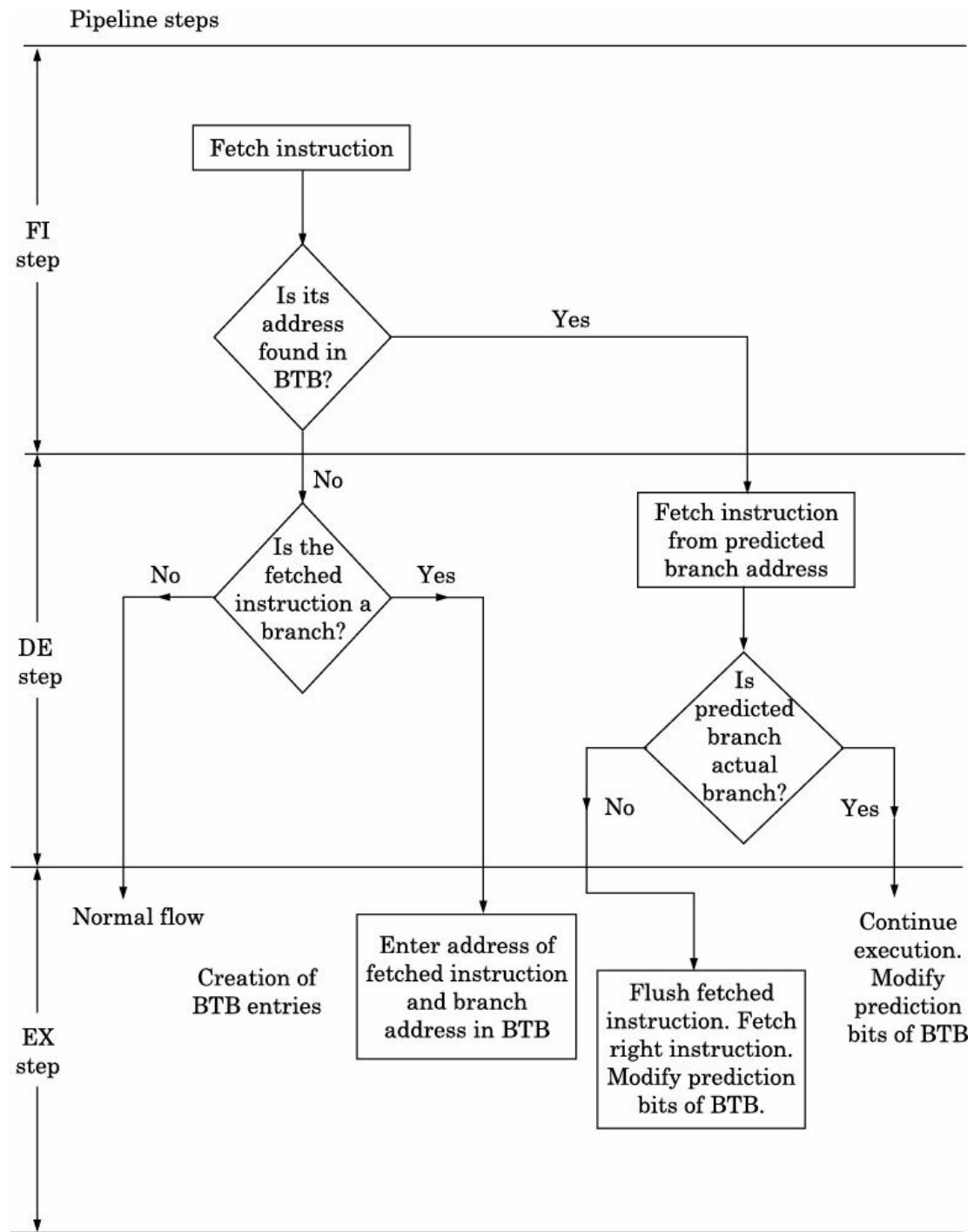


Figure 3.18 Branch Target Buffer (BTB) creation and use.

We will also assume that the branch address is put in PC only after MEM cycle. In other words, there is no special branch address calculation hardware.

By having a BTB the average delay cycles when unconditional branches are found in BTB = 0.

Average delay when unconditional branches are not found in BTB = 3.

Thus, average delay due to unconditional branches $2 \times 0.05 = 0.1$ (Probability of branch not found in BTB is 0.05).

(For unconditional branches there can be no misprediction.)

Average delay due to conditional branches when they are found in BTB = 0.

Average delay when conditional branches are not found in BTB = $(3 \times 0.8) + (2 \times 0.2) =$

2.8.

As probability of not being in BTB = 0.05, the average delay due to conditional branches = $0.05 \times 2.8 = 0.14$.

Average delay due to misprediction of conditional branches when found in BTB = $0.1 \times 3 \times 0.95 = 0.285$ (As probability of conditional branch being in BTB is 0.95). As 5% of instructions are unconditional branches and 15% are conditional branches in a program, the average delay due to branches = $0.05 \times 0.1 + 0.15 \times (0.14 + 0.285) = 0.005 + 0.064 = 0.069$.

Therefore, speedup with branches when BTB is used = $5/(1 + 0.069) = 4.677$.

% loss of speedup due to branches = 6%.

Compare with the loss of speedup of 36.4% found in Example 3.2 with no BTB. Thus, use of BTB is extremely useful.

3.2.5 Software Method to Reduce Delay Due to Branches

This method assumes that there is no hardware feature to reduce delay due to branches. The primary idea is for the compiler to rearrange the statements of the assembly language program in such a way that the statement following the branch statement (called a *delay slot*) is always executed once it is fetched without affecting the correctness of the program. This may not always be possible but analysis of many programs shows that this technique succeeds quite often. The technique is explained with some examples.

EXAMPLE 3.5

Observe that in the rearranged program the branch statement JMP has been placed before ADD R1, R2, R3 (Fig. 3.19). While the jump statement is being decoded, the ADD statement would have been fetched in the rearranged code and can be allowed to complete without changing the meaning of the program. If no such statement is available then a No Operation (NOP) statement is used as a filler after the branch so that when it is fetched it does not affect the meaning of the program. Another technique is also used by the compiler to place in the delay slot the target instruction of the branch. This is illustrated in Example 3.6.

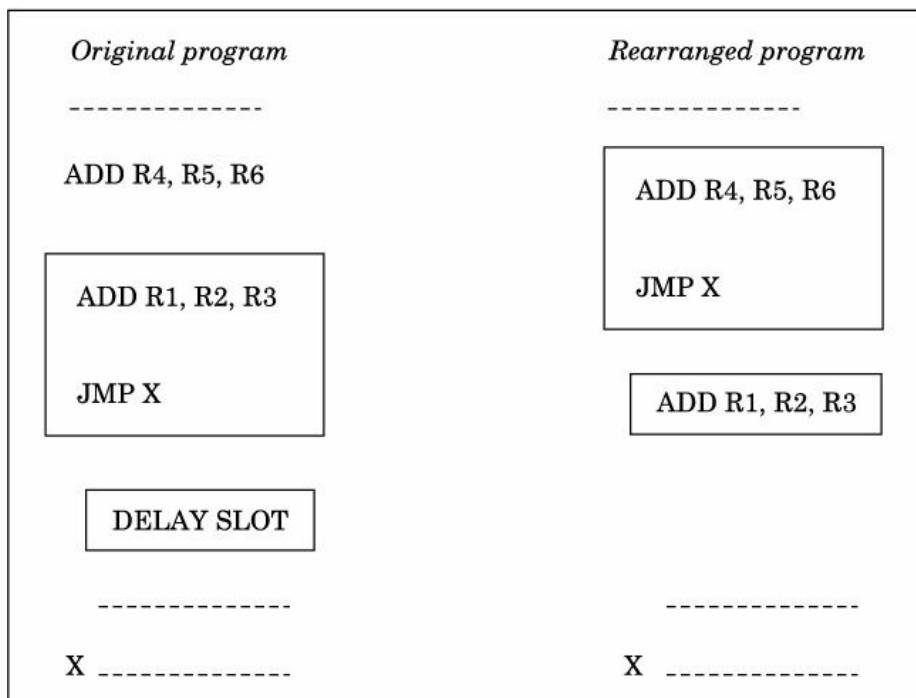


Figure 3.19 Rearranging compiled code to reduce stalls.

EXAMPLE 3.6

Observe that the delay slot is filled by the target instruction of the branch (Fig. 3.20). If the probability of the branch being taken is high then this procedure is very effective. When the branch is not taken control will fall through and the compiler should “undo” the statement executed in the delay slot if it changes the semantics of the program. Some pipelined computers provide an instruction “No Operation if branch is unsuccessful” so that the delay slot automatically becomes No Operation and the program’s semantics does not change.

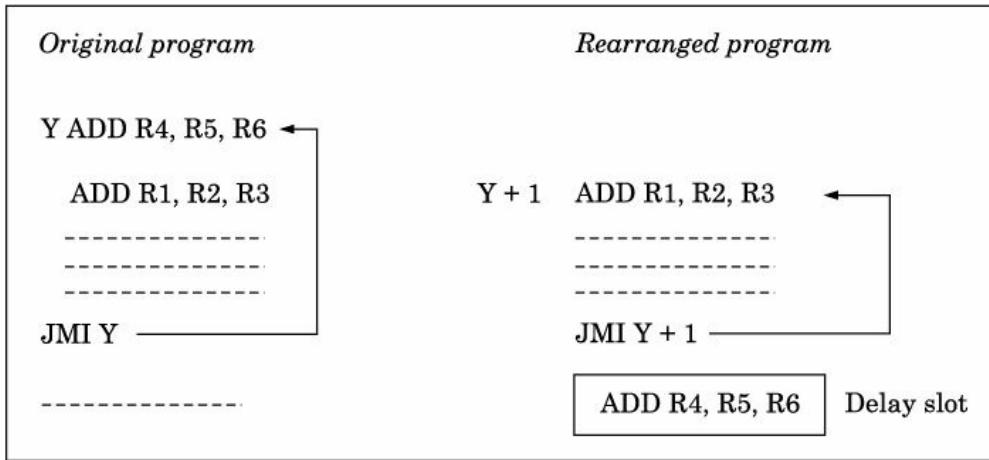


Figure 3.20 Rearranging code in a loop to reduce stalls.

Forward branching can also be handled by placing the target instruction in the delay slot as shown in Fig. 3.21.

There are other software techniques such as unrolling a loop to reduce branch instructions which we do not discuss here. They are discussed in detail in Chapter 9 on Compiler Transformations.

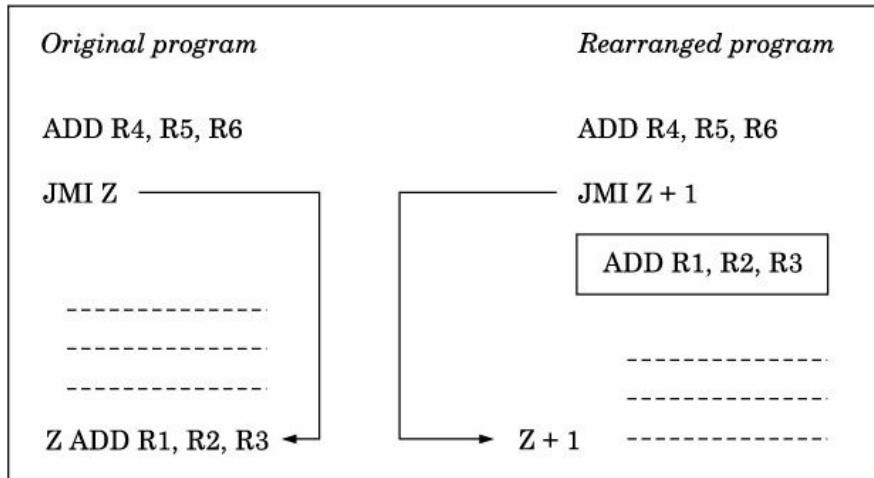


Figure 3.21 Hoisting code from target to delay slot to reduce stall.

3.3 DIFFICULTIES IN PIPELINING

We have discussed in the last section problems which arise in pipeline execution due to various non-ideal conditions in programs. We will examine another difficulty which makes the design of pipeline processors challenging. This difficulty is due to interruption of normal flow of a program due to events such as illegal instruction codes, page faults, and I/O calls. We call all these *exception conditions*. In Table 3.5 we list a number of exception conditions. For each exception condition we have indicated whether it can occur in the middle of the execution of an instruction and if yes during which step of the pipeline. Referring to Table 3.5 we see, for example, that memory protection violation can occur either during the instruction fetch (FI) step or load/store memory (MEM) step of the pipeline. In Table 3.5 we have also indicated whether we could resume the program after the exception condition or not. In the case of undefined instruction, hardware malfunction or power failure we have no option but to terminate the program. If the Operating System (OS) has a checkpointing feature, one can restart a half finished program from the last checkpoint.

An important problem faced by an architect is to be able to attend to the exception conditions and resume computation in an orderly fashion whenever it is feasible. The problem of restarting computation is complicated by the fact that several instructions will be in various stages of completion in the pipeline. If the pipeline processing can be stopped when an exception condition is detected in such a way that all instructions which occur before the one causing the exception are completed and all instructions which were in progress at the instant exception occurred can be restarted (after attending to the exception) from the beginning, the pipeline is said to have *precise exceptions*. Referring to Fig. 3.22, assume an exception occurred during the EX step of instruction $(i + 2)$. If the system supports precise exceptions then instructions i and $(i + 1)$ must be completed and instructions $(i + 2)$, $(i + 3)$ and $(i + 4)$ should be stopped and resumed from scratch after attending to the exception. In other words whatever actions were carried out by $(i + 2)$, $(i + 3)$ and $(i + 4)$ before the occurrence of the exception should be cancelled. When an exception is detected, the following actions are carried out:

TABLE 3.5 Exception Types in a Computer

Exception type	Occurs during pipeline stage?		Resume or terminate
	Yes/No	Which stage?	
I/O request	No		Resume
OS request by user program	No		Resume
User initiates break point during execution	No		Resume
User tracing program	No		Resume
Arithmetic overflow or underflow	Yes	EX	Resume
Page fault	Yes	FI, MEM	Resume
Misaligned memory access	Yes	FI, MEM	Resume
Memory protection violation	Yes	FI, MEM	Resume
Undefined instruction	Yes	DE	Terminate

Hardware failure	Yes	Any	Terminate
Power failure	Yes	Any	Terminate

1. As soon as the exception is detected turn off write operations for the current and all subsequent instructions in the pipeline [Instructions $(i + 2)$, $(i + 3)$ and $(i + 4)$ in Fig. 3.22].
2. A trap instruction is fetched as the next instruction in pipeline (Instruction $i + 5$ in Fig. 3.22).
3. This instruction invokes OS which saves address (or PC of the program) of faulting instruction to enable resumption of program later after attending to the exception.

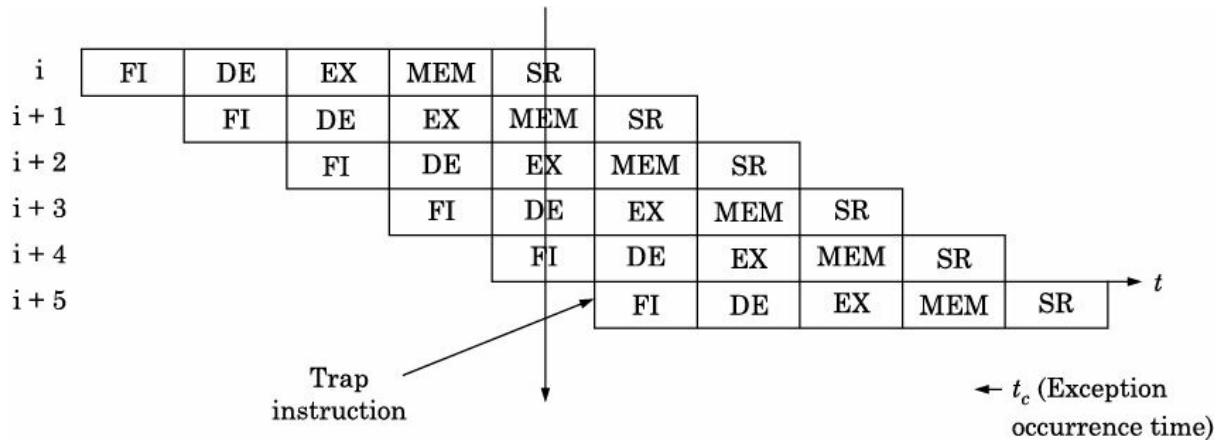


Figure 3.22 Occurrence of an exception during pipeline execution.

We have briefly discussed the procedure to be followed when exceptions occur. There are many other issues which complicate the problem, particularly when the instruction set is complex and multi-cycle floating point operations are to be performed. For a detailed discussion the reader should refer to the book by Patterson and Hennessy [2012].

3.4 SUPERSCALAR PROCESSORS

In a pipelined processor we assume that each stage of pipeline takes the same time, namely, one clock interval. In practice some pipeline stages require less than one clock interval. For example, DE and SR stages normally may require less time. Thus, we may divide each clock cycle into two phases and allocate intervals appropriate for each step in the instruction cycle. Referring to Fig. 3.23, we have allocated a full clock cycle to instruction fetch (FI), execution (EX) and Data Memory load/store (MEM) steps while half a cycle has been allocated to the decode (DE) and store register (SR) steps. Further if the steps are subdivided into two phases such that each phase needs different resources thereby avoiding resource conflicts, pipeline execution can be made faster as shown in Fig. 3.23. This method of pipeline execution is known as *superpipelining*. Observe that the normal pipeline processing takes 7 clock cycles to carry out 3 instructions whereas in superpipelined mode we need only 5 clock cycles for the same three instructions. In the steady state one instruction will take half a clock cycle under ideal conditions. All the difficulties associated with pipeline processing (namely various dependencies) will also be there in superpipelined processing. As superpipelined processing speeds up execution of programs, it has been adopted in many commercial high performance processors such as MIPS R4000.

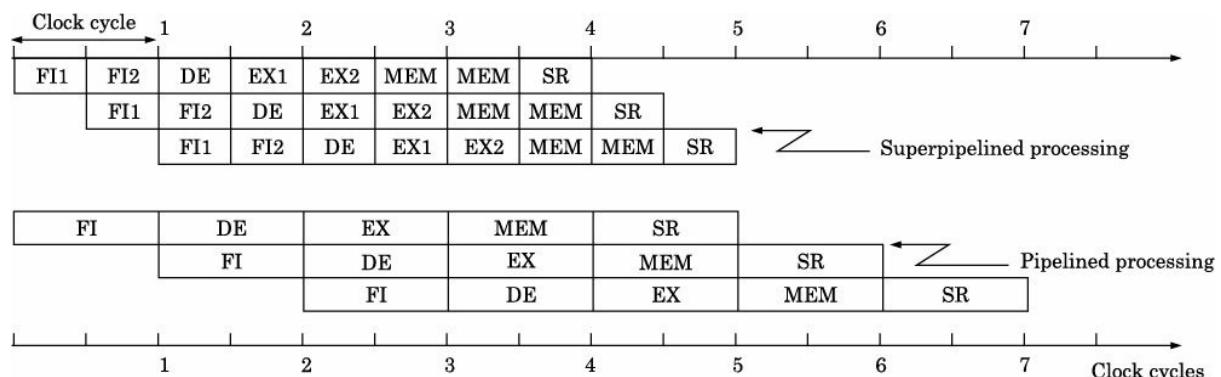


Figure 3.23 Superpipelined and pipelined processing.

Another approach to improve the speed of a processor is to combine temporal parallelism used in pipeline processing and data parallelism by issuing several instructions from an instruction thread simultaneously in each cycle. The instructions issued may even be not in the order found in the thread but in an order which maximizes the use of hardware resources. This is called *superscalar* processing. Pipeline execution with 2 instructions issued simultaneously is shown in Fig. 3.24. Observe that 6 instructions are completed in 7 clock cycles. In the steady state two instructions will be completed every clock cycle under ideal conditions. For successful superscalar processing, the hardware should permit fetching several instructions simultaneously from the instruction memory. The data cache must also have several independent ports for read/write which can be used simultaneously. If the instruction is a 32-bit instruction and we fetch 2 instructions, 64-bit data path from instruction memory is required and we need 2 instruction registers. Executing multiple instructions simultaneously would require multiple execution units to avoid resource conflicts. The minimum extra execution units required will be a floating point arithmetic unit in addition to an integer arithmetic unit and a separate address calculation arithmetic unit. If many instructions are issued simultaneously, we may need several floating point and integer execution units.

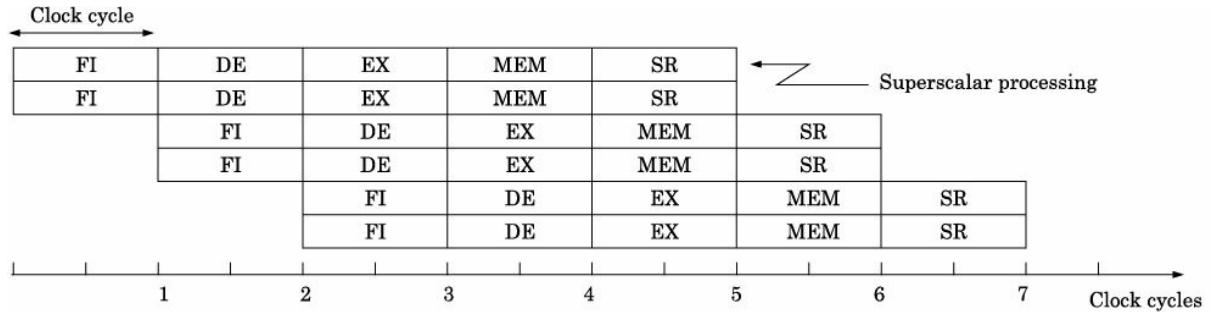


Figure 3.24 Superscalar processing.

As programs are generally written without any concern for parallel execution, the delay due to data dependency of successive instructions during pipeline execution can become quite high in superscalar processors unless there are features available in hardware to allow execution of instructions out of order while ensuring the correctness of the program. Another approach is to transform the source code by a compiler in such a way that the resulting code has reduced dependency and resource conflicts so that parallelism available in hardware is fully utilized. The software techniques will be discussed in Chapter 9. We will discuss some of the hardware techniques in this section.

Consider the sequence of instructions given in Table 3.6. We assume that the processor has two integer execution units and one floating point execution unit which can all work in parallel. We develop the superscalar execution sequence with 2 instructions fetched per cycle in Fig. 3.25. Observe that execution of I2 is delayed by 2 cycles as value of R1 will be available only in cycle 5. (We have assumed that integer division requires 2 clock cycles).

TABLE 3.6 A Sequence of Instructions for Superscalar Execution

<i>Instruction identity</i>	<i>Instruction</i>	<i>Number of cycles for execution</i>	<i>Arithmetic unit needed</i>
I1	$R1 \leftarrow R1/R5$	2	Integer
I2	$R3 \leftarrow R1 + R2$	1	Integer
I3	$R2 \leftarrow R5 + 3$	1	Integer
I4	$R7 \leftarrow R1 - R11$	1	Integer
I5	$R6 \leftarrow R4 \times R8$	2	Floating point
I6	$R5 \leftarrow R1 + 6$	1	Integer
I7	$R1 \leftarrow R2 + 1$	1	Integer
I8	$R10 \leftarrow R9 \times R8$	2	Floating point

Cycles → 1 2 3 4 5 6 7

I1	FI	DE	EX1	EX2	MEM	SR	
I2	FI	DE	X	X	EXI	MEM	SR
I3		FI	DE	EXI	MEM	SR	
I4		FI	DE	X	EXI	MEM	SR
⋮							

Figure 3.25 Developing superscalar execution sequence for instructions in Table 3.6.

Assuming hardware register forwarding I2 can execute in cycle 5. I3 does not apparently depend on previous instructions and can complete in cycle 6 if allowed to proceed. This would however, lead to an incorrect R3 when the preceding instruction I2 completes as I2 will take the value of R2 computed by I3 instead of the earlier value of R2 (Remember that we have assumed internal register forwarding). This mistake is due to what is known as *anti-dependency* between instructions I2 and I3. In other words a value computed by a later instruction is used by a previous instruction. This is also called *Write After Read* (WAR) hazard and is not allowed. Such problems will arise when we allow out-of-order completion of instructions and one has to watch out for these problems in such a situation. In this example we have delayed execution of I3 till I2 completes as shown in Fig. 3.26. A question which may arise is; “Can the execution steps of I2 and I3 be done simultaneously in cycle 5?” If I2 reads R2 before I3 updates it, there will be no problem. We, however, take a conservative approach and wait for the completion of the execution step of I2 before commencing the execution of I3. Continuing with this execution sequence being developed in Fig. 3.26, we see that instruction I4 has to wait for I1 to complete as it requires the value of R1. I5 has no dependency and can complete without delay. Observe that in cycle 5 both instructions I2 and I4 need integer arithmetic units which we assume is available. I5 needs a floating point unit which is available. I6 needs the value of R1 which is available in cycle 5. It, however, cannot complete before I3 which uses R5. (I3 and I6 have anti-dependency). Thus I6 starts after I3 completes execution. The next instruction I7, $R1 \leftarrow R2 + 1$ cannot start till I6 completes as it is anti-dependent on I6. Observe also that the destination register of I7 is the same as that of I1. Thus, I1 must also complete before I7 begins. This dependency when the destination registers of two instructions are the same is called *output dependency*. It is also known as *Write After Write* (WAW) hazard. In this example I7 can start execution in cycle 8. The last instruction in our example I8 has no dependency with any of the previous instructions. It can thus proceed without delay. However, it gets delayed because of resource constraints as it needs the floating point execution unit in cycle 6 but it is being used by instruction I5. When the execution of I5 completes in cycle 6, I8 can use it in cycle 7. Thus, I8 is delayed by one cycle and completes in cycle 10. If we have 2 floating point units then I8 can complete in cycle 9. However, I7 completes only in cycle 10 (see Fig. 3.26).

	Cycles →	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR					
I2	FI	DE	X	X	EXI	MEM	SR				
I3	FI	DE	X	X	X	EXI	MEM	SR			
I4	FI	DE	X	X	EXI	MEM	SR				
I5		FI	DE	EXF	EXF	MEM	SR				
I6		FI	DE	X	X	EXI	MEM	SR			
I7			FI	DE	X	X	EXI	MEM	SR		
I8			FI	DE	X	EXF	EXF	MEM	SR		


Idle cycle

Figure 3.26 Superscalar pipeline execution of instructions given in Table 3.6.

We will now explore whether any speedup is possible using extra resources. One method which has been adopted is to use more registers. We must first detect which registers are used

by prior instructions and whether they are source or destination registers. If the destination register of an instruction I is used as the source of the next or succeeding instruction K then K is *flow dependent* on I. It is known as *Read After Write* (RAW) hazard. This is also known as *true dependency* as K cannot proceed unless I is completed. In our example I2, I4 and I6 are flow dependent on I1. If the destination register of a *later* instruction, say K, is used as a source register of an *earlier* instruction, say I, then K and I are *anti-dependent* (WAR hazard). In our example I3 and I2 are anti-dependent. I3 and I6 are also anti-dependent. Finally if the same register is used as a destination register in two different instructions, these two instructions are said to be *output dependent*. In our example I1 and I7 are output dependent. If two instructions are output dependent then the instruction appearing earlier must complete before the later instruction is executed. Output dependency, as was pointed out earlier, is also known as *Write After Write* (WAW) hazard.

Register Score Boarding and Renaming

One method which is used by hardware to detect dependencies is to have a separate register called a *score board* which has one bit to identify each register of the register file. This bit is set if the corresponding register is in use. A computer with 32 registers will have a 32-bit score board to indicate the busy/free status of registers. When an instruction uses a set of registers, the bits corresponding to those registers are set to 1 in the score board. They are reset to 0 as soon as the instruction using that register completes execution. Thus, the hardware will examine the score board to see if the registers specified in the instruction are free before executing it.

In order to speedup execution we must reduce delays due to anti-dependency and output dependency as it is not possible to reduce delays due to flow dependency. One of the methods to reduce delays is to use more resources judiciously. In this case the delay is due to the same registers being used in multiple instructions. We thus allocate extra registers to eliminate anti dependency and output dependency. For the example being considered we assign new registers (called *register renaming*) as shown in Table 3.7. With this renaming (I2, I3), (I3, I6) and (I6, I7) are not anti-dependent. The instructions (I1, I7) also cease to be output dependent.

TABLE 3.7 Register Renaming		
	<i>Old instruction</i>	<i>Instruction with register renamed</i>
I1	$R1 \leftarrow R1/R5$	$R1 \leftarrow R1/R5$
I2	$R3 \leftarrow R1 + R2$	$R3 \leftarrow R1 + R2$
I3	$R2 \leftarrow R5 + 3$	$R2N \leftarrow R5 + 3$
I4	$R7 \leftarrow R1 - R11$	$R7 \leftarrow R1 - R11$
I5	$R6 \leftarrow R4 \times R8$	$R6 \leftarrow R4 \times R8$
I6	$R5 \leftarrow R1 + 6$	$R5N \leftarrow R1 + 6$
I7	$R1 \leftarrow R2 + 1$	$R1N \leftarrow R2N + 1$
I8	$R10 \leftarrow R9 \times R8$	$R10 \leftarrow R9 \times R8$

With this modified instructions, the execution sequence is shown in Fig. 3.27. We see that

the delay in the pipeline is reduced and the execution completes in 9 cycles, a saving of one cycle as compared to the schedule of Fig. 3.26. In developing the execution sequence we have assumed that 2 floating point units and 2 integer units are available. After completing the sequence of instructions, the renamed registers overwrite their parents (R2 is R2N's parent).

Cycles →	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR				
I2	FI	DE	X	X	EXI	MEM	SR			
I3		FI	DE	EXI	MEM	SR				
I4		FI	DE	X	EXI	MEM	SR			
I5			FI	DE	EXF	EXF	MEM	SR		
I6			FI	DE	X	EXI	MEM	SR		
I7				FI	DE	EXI	MEM	SR		
I8				FI	DE	EXF	EXF	MEM	SR	

Figure 3.27 Superscalar pipeline execution with register renaming.

The next question we ask is, “Is it possible to reschedule issue of instructions to reduce the time needed to carry out the instructions while maintaining the correctness of the results?” It is indeed possible as we illustrate now. The hardware of superscalar processors provides an instruction buffer with a capacity of several instructions. From this buffer called an *instruction window*, the system picks up 2 instructions (in our example) to schedule in such a way that the completion time is minimized. The larger the size of the window, the greater is the opportunity to optimize. We will assume a small window of only 4 instructions. The opportunity for optimization is better with register renaming. We will thus use the instruction sequence of Table 3.7 with renamed registers. The instructions in the window at start are I1, I2, I3, I4. I2 cannot be scheduled before I1 as the result of executing I1 is used by I2. I3 can however be interchanged with I2. Thus, I1 and I3 are scheduled first. In the next cycle I2, I4, I5 and I6 are taken to the window for scheduling. As I4 is dependent on I1 and will be delayed, we schedule I5 along with I2 as shown in Fig. 3.28. In the next cycle we retrieve I4, I6, I7 and I8. I4 can be scheduled without delay and we do it. Neither I6 nor I7 can be scheduled without delay as both the integer units are busy. We thus take I8 which can be scheduled as it has no dependency and needs a floating point unit which is available. Thus, we schedule I4 and I8. Now, only I6 and I7 need to be executed and both can be scheduled as they have no resource conflicts or dependencies. The final schedule is shown in Fig. 3.28. Observe that we have saved one cycle and completed the 8 instructions in 8 cycles. This is in fact the best we can do in this case.

Two questions arise at this point. One is, “What happens if there are branches?” No new method is required. We can use the same procedure of setting branch prediction bits and employ branch target buffer (Fig. 3.18) as we did for normal pipeline execution. The second question is regarding completion of execution of instruction in an order which is not the same as shown for the static program. For example in Fig. 3.28, I3 completes before I1 and I2. Further the use of branch prediction and speculative execution may lead to the completion of some instructions which may have to be abandoned as the expected branch is not taken. It is thus incorrect to update memory and program-visible registers as soon as an instruction is executed. The results should be stored in a buffer temporarily and committed permanently to the appropriate storage to ensure that the sequential model of execution of the program is not

violated. This is called *committing* or *retiring* an instruction.

Cycles →	1	2	3	4	5	6	7	8	9	10
I1	FI	DE	EX1	EX2	MEM	SR				
I3	FI	DE	EXI	MEM	SR					
I2	FI	DE		X	EXI	MEM	SR			
I5	FI	DE	EXF	EXF	MEM	SR				
I4	FI	DE	EXI	MEM	SR					
I8	FI	DE	EXF	EXF	MEM	SR				
I6		FI	DE	EXI	MEM	SR				
I7		FI	DE	EXI	MEM	SR				

Figure 3.28 Rescheduled instructions in a superscalar processor.

A simplified block diagram of the pipeline stages of a superscalar processor architecture is shown in Fig. 3.29. This figure consolidates the various points we have discussed so far in this section. The fetch unit fetches several instructions in each clock cycle from the instruction cache. The decode unit decodes the instructions and renames registers so that false data dependencies are eliminated and only true dependencies remain. If the instruction is a branch instruction, branch prediction algorithm is used to identify the next instruction to be executed. The instructions are sent to an instruction buffer where they stay temporarily till the source operands are identified and become available. Ready instructions are despatched based on the availability of functional units. The operands are fetched either from the register file or from the bypass path from earlier instructions which produced these operands. The results are reordered in the correct sequential order and committed, that is, either stored in the data cache or in the register file.

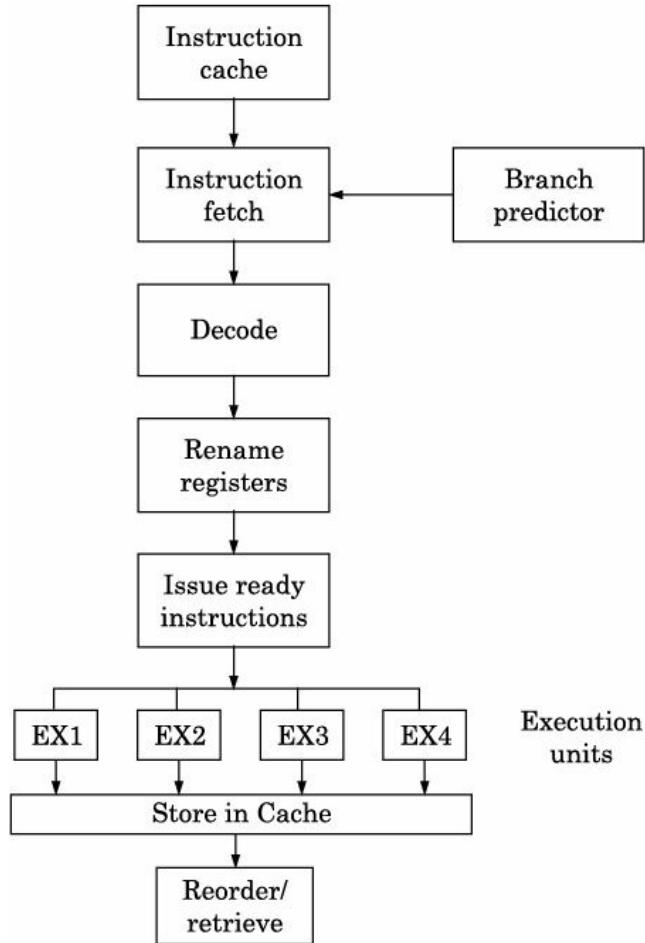


Figure 3.29 Superscalar processor pipeline stages.

Superscalar processing depends on available parallelism in groups of instructions of programs. If the available parallelism is more, then one may issue several instructions in the same cycle. Machines have been built which issue 4 instructions in each cycle and execute them in pipeline mode. In order to profitably use this available parallelism in programs, the processor must also have extra functional units. *Hardware or machine parallelism* indicates the resource availability in hardware and *instruction level parallelism*, the parallelism inherent in programs. Many studies have shown that 4 or 5 instructions can be potentially executed in parallel in common application programs. Thus in theory processors can issue 4 or 5 instructions in each cycle. As we have observed the most serious impediments in pipelined processing are data dependencies and branching. One of the most important and useful methods to reduce branching while exposing more instruction parallelism is *loop unrolling*. In this the instructions in the loop are replicated a number of times equal to the number of iterations of the loop. This unrolling is done by the compiler which often unrolls up to 4 iterations. Details of this method are discussed in Chapter 9.

Many processors, both RISC and CISC, are superscalar. MIPS 10000 and Power PC are examples of superscalar RISCs. Pentium 4 is a superscalar CISC.

3.5 VERY LONG INSTRUCTION WORD (VLIW) PROCESSOR

The major problem in designing superscalar processors is, besides the need to duplicate instruction register, decoder and arithmetic unit, it is difficult to schedule instructions dynamically to reduce pipeline delays. Hardware looks only at a small window of instructions. Scheduling them to use all available processing units, taking into account dependencies, is sub-optimal. Compilers can take a more global view of the program and rearrange code to better utilize the resources and reduce pipeline delays. An alternative to superscalar architecture has thus been proposed which uses sophisticated compilers to expose a sequence of instructions which have no dependency and require different resources of the processor. In this architecture a single word incorporates many operations which are independent and can thus be pipelined without conflicts. If such operations cannot be found by the compiler to fill a word then the slot is filled with a no-op. Typically two integer operations, two floating point operations, two load/store operations and a branch may all be packed into one long instruction word which may be anywhere between 128 and 256 bits long. The processor must have enough resources to execute all operations specified in an instruction word simultaneously. In other words, it must have 2 integer units, 2 floating point units, two data memories and a branch arithmetic unit in this example. The challenging task is to have enough parallelism in a sequence of instructions to keep all these units busy. Parallelism in programs is exposed by unrolling loops, scheduling instructions across branches by examining the program globally and by what is known as *trace scheduling*. Trace scheduling is based on predicting the path taken by branch operations at compile time using some heuristics or by hints given by the programmer. If the branches follow the predicted path, the code becomes a “straight line code” which facilitates packing sequences of instructions into long instruction words and storing them in the instruction memory of VLIW [Fisher, 1983] processor and pipelining the instructions without stalls. (Trace scheduling is discussed in Chapter 9). The main challenges in designing VLIW processors are:

- Lack of sufficient instruction level parallelism in programs.
- Difficulties in building hardware.
- Inefficient use of bits in a very long instruction word.

A major problem is the lack of sufficient parallelism. Assume that the two Floating Point Units (FPUs) are pipelined to speedup floating point operations. If the number of stages in the pipeline of FPU is 4, we need at least 10 floating point operations to effectively use such a pipelined FPU. With two FPUs we need at least 20 independent floating point operations to be scheduled to get the ideal speedup. We also need besides this, 2 integer operations. The overall instruction level parallelism may not be this high.

The VLIW hardware also needs much higher memory and register file bandwidth to support wide words and multiple read/write to register files. For instance, to sustain two floating point operations per cycle, we need two read ports and two write ports to register file to retrieve operands and store results. This requires a large silicon area on the processor chip.

Lastly it is not always possible to find independent instructions to pack all 7 operations in each word. On the average about half of each word is filled with no-ops. This increases the required memory capacity.

Binary code compatibility between two generations of VLIWs is also very difficult to maintain as the structure of the instruction will invariably change.

Overall VLIW, though an interesting idea, has not become very popular in commercial high performance processors.

3.6 SOME COMMERCIAL PROCESSORS

In this section we describe three processors which are commercially available. The first example is a RISC superscalar processor, the second is one of the cores of a CISC high performance processor, and the third a long instruction word processor called an EPIC processor.

3.6.1 ARM Cortex A9 Architecture

ARM Cortex A9 is a micro architecture which implements version 7 of the ARM instruction set [Tannenbaum and Austin, 2013]. It is a RISC architecture designed by ARM Ltd., and sold to chip manufacturers such as Texas Instruments who implement it with slight modification which is appropriate for their applications. It is widely used in embedded systems which require low power. One such device is OMAP4430 of Texas Instruments. ARM Cortex A9 is also used as one of the cores of a multicore chip which we will describe in Chapter 5.

We give the block diagram of Cortex A9 in Fig. 3.30. The instruction cache of Cortex A9 is 32KB and uses 32 byte cache lines. As ARM is a RISC processor with 32-bit instruction length, the instruction cache can accommodate 8 K instructions. A special feature of the Cortex A9 architecture is a fast *loop look-aside cache*. If a program is executing a non-nested small loop, all the instructions of the loop are stored in this cache and fetched. This eliminates the need to look up the branch prediction unit which can be put in low power mode. The instruction fetch unit assembles 4 instructions which are ready for execution in each clock cycle. When a decoded instruction is a conditional branch, a branch predictor cache with 4 K addresses is looked up to predict whether the branch will be taken. If it predicts that the branch will be taken, the branch address is read from a branch target address cache which has 1 K entries. The output of the instruction fetch unit is decoded to find which execution units and inputs are required to carry out the instructions. Registers are renamed after decoding to eliminate WAR hazards. The results are placed in an instruction queue from which they are issued to the execution unit when the necessary inputs are available. The instructions may be issued out of order. In Fig. 3.30 we have shown the available functional units. After execution of the instructions the results may be out of order. The retirement unit outputs the results in the correct sequential order.

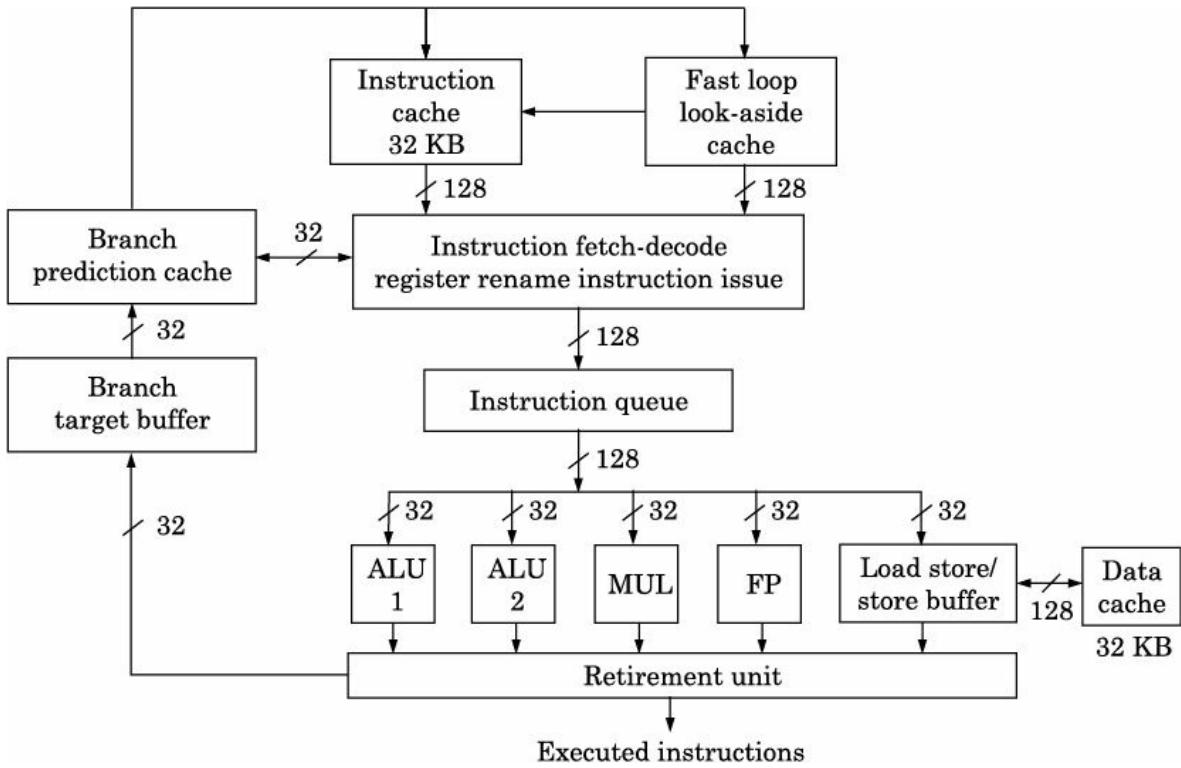


Figure 3.30 Block diagram of ARM cortex A9 microarchitecture.

The execution pipeline of Cortex A9 has 11 stages. In Table 3.8 we show what happens during each pipeline stage. We have given in this subsection the essential features of a RISC design (namely ARM Cortex A9) leaving out many details. This example is intended to illustrate how the theory we discussed in the previous sections is used in practice. For greater details on ARM Cortex A9 a reader must look up the appropriate ARM manual available on the web.

TABLE 3.8 Operations Performed During Eleven Pipeline Stages of ARM Cortex A9		
No.	Pipeline stage	Operations performed during the stage
1.	FI 1	Fetch address of next instruction.
2.	FI 2	Branch prediction/service trap or interrupt.
3.	FI 3	Fetch up to 4 instructions and put in buffer.
4.	DE 1	Decode instruction. Determine execution units and inputs needed.
5.	DE 2	
6.	RE	Rename registers to eliminate WAR and WAW hazards during out-of-order execution. Assign registers from a pool of unused physical registers.
7.	I 1	Up to 4 instructions issued based on availability of inputs and execution units. They may be out of order.
8.	EX 1	Instructions executed. Conditional branch execution done in EX1 and branch/no branch determined. If mispredicted, a signal is sent to FI 1 to void pipeline. The execution units available are shown in Fig. 3.30.
9.	EX 2	
10.	EX 3	

3.6.2 Intel Core i7 Processor

We will discuss in this subsection the micro architecture of a popular Intel processor core i7 [Tannenbaum and Austin, 2013]. It is a 64-bit processor and has the same instruction set architecture as that of the earlier Intel processors, namely, 80486, and Pentium series. It has the same registers, identical instruction set as earlier x86 series processors allowing old processor code to be run without alteration on this processor. In addition to the old instruction set i7 has some new instructions to support cryptography.

The i7 is a multicore CPU. In other words, in one chip several identical processors are embedded which share a memory. (We will describe multicore processors in detail in Chapter 5). The processor is also multithreaded. In other words, multiple short sequences of instructions may be kept ready and may be scheduled as a unit. Core i7 is a superscalar processor which can carry out up to 4 instructions per cycle. In the rest of this section we will describe the micro architecture of i7 to illustrate how a CISC processor can use the ideas which we described in the last section for a RISC superscalar processor.

As we pointed out earlier, the instruction set of i7 is the same as the earlier Pentium processors of Intel so that old programs may be run without any changes on this new generation processor. The instruction set of i7 is very complex. Each instruction can be between 1 and 17 bytes long. A huge number of instructions are supported and there is no uniformity in the structure of instructions unlike RISC. Thus, it is extremely difficult to pipeline execution of original instructions. The problem was solved by Intel architects by translating each i7 instruction (called macro-operation) into a set of micro-operations of uniform length. This is done by hardware known as micro-program control unit. Thus, one i7 machine language instruction gets converted into a sequence of micro-operations. All micro-operations have the same length and they take the same amount of time to execute if there are no hazards during execution. The branch statements are also micro-instructions and are easier to predict. The micro-instructions can thus be scheduled on a superscalar RISC-like pipeline. The pipeline used by i7 has 14 stages and 4 instructions can be issued at a time. The exact functions carried out in each pipeline stage are not clearly specified. i7 uses an out-of-order speculative execution pipeline.

In Fig. 3.31 we give a block diagram which provides a simplified overview of the micro-architecture of i7. We will expand and explain the function of each of the blocks now.

At the top of Fig. 3.31 is a 32 KB instruction cache which stores a sequence of i7 instructions as obtained from the compiled code. The instruction fetch unit (block numbered 2 in Fig. 3.31) determines the address from where an instruction is to be fetched by consulting a branch prediction unit. The predictor uses a two-level branch target buffer organization. The first level stores the history of a small number of branches and is fast. The second level stores information about a very large number of branches but is slower. This organization improves the prediction accuracy of large programs. A misprediction penalty is about 15 cycles. Using the predicted address the instruction fetch unit fetches 16 bytes from the instruction cache. The fetched bytes would normally belong to several i7 instructions called *macro-operations*. A procedure called *macro-op fusion* is performed in this unit. A sequence of two instructions for example, a compare followed by a branch are fused into one operation. The pre-decode unit disassembles the 16 bytes which were retrieved into individual i7 instructions. This operation is not easy as the length of an i7 instruction can be anywhere between 1 and 17 bytes. The pre-decoder must thus examine a number of bytes in order to detect instruction boundaries. The individual i7 instructions along with the fused

instructions (if any) are put in an instruction queue which has 18 entries (Block 3 Fig. 3.31). The instructions from this queue are taken and translated into RISC-like micro-operations. Four micro-operations can be produced in one cycle. The instructions of i7 with complex semantics are sent to the complex decoder and the others to the simple decoders (Block 4 of Fig. 3.31). The resulting micro-operations are stored in the same order as the original i7 instructions in a micro-operation buffer which can store 28 micro-operations. A loop detection unit finds out if any sequence of micro-operations (less than 28) forms a loop. Loops can be scheduled as a unit and the instructions of the loop need not be fetched/decoded thereby saving time. Similar to macro-op fusion, this unit also performs micro-op fusion, combining two micro-operations which always occur together, for example, load/ALU operation. This conserves space in the buffer.

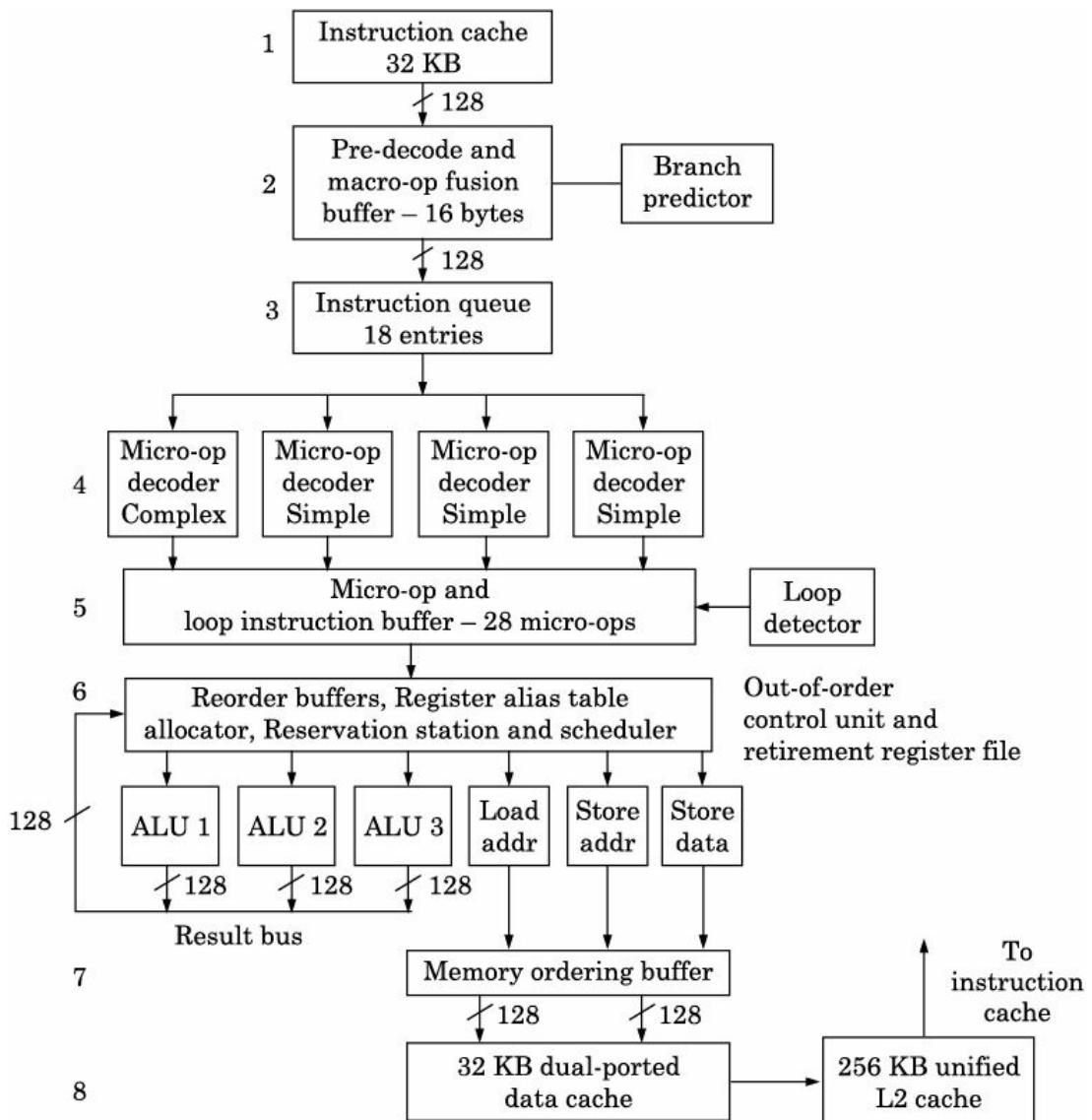


Figure 3.31 Block diagram of i7 micro architecture.

From the micro-op buffer instructions are sent to a unit (Block 6) which performs many of the operations required by a superscalar RISC processor. This is the out-of-order control logic. Up to 4 micro-operations are issued each cycle from the micro-op buffer. A re-order buffer has 168 entries. This unit keeps track of micro-operations till they are retired. The allocation/rename unit checks if all the resources needed to execute a micro-operation are available. A micro-operation may stall because it has to write to a register that is being used

by an earlier micro-operation to read or write. As we have seen earlier these are WAR and WAW dependencies. There are 160 scratch registers available which can be used to rename the target registers enabling the micro-operations to execute without stall. If no scratch register is available or the micro-operation has a RAW dependency, the allocator flags this in the reorder buffer entry. The micro-operations which are ready for execution (which need not be in-order) are sent to a 36 entry reservation station to be scheduled for execution in the execution units.

There are six execution units; 3 ALUs, 1 load unit and 2 store units. Thus, six micro-operations can be issued in each clock cycle. Individual execution units execute the appropriate micro-operation and the results returned to register retirement unit. The retirement unit updates the register state and ensures that the retirement of instructions is in-order. When an instruction is retired the entry corresponding to this instruction in the re-order buffer is marked complete, any pending writes to the retirement unit are executed and the instruction deleted from the reorder buffer. Finally the results (in-order) are stored in the 32 KB data cache which is periodically flushed to L2 cache.

If during execution of a program there is an interrupt, the instructions which have not yet been retired by the retirement unit are aborted. Thus, i7 has a precise interrupt.

One of the major problems with i7 class of processors is the high power requirements. The i7 processor consumes 130 W, one cannot touch the chip as it becomes very hot. If the clock frequency is increased beyond 3.3 GHz, the power consumption and consequently the heat to be dissipated will be so high that air cooling will not be sufficient. Thus, one of the attempts of architects is to reduce heat dissipation by placing some of the parts of the system in sleep mode. For example, when a loop is executed after it is detected in the loop instruction buffer (Block 5 of Fig. 3.31), Blocks 1,2,3 and 4 are placed in sleep mode to reduce power dissipation. Thus, architects of all high performance processors are currently more concerned with reducing power dissipation whenever an opportunity arises.

3.6.3 IA-64 Processor Architecture

IA-64 is an architecture designed by Intel in collaboration with Hewlett-Packard. It is a full fledged 64-bit computer with 64-bit registers and 64-bit address space. It is called Explicitly Parallel Instruction Computer (EPIC) by Intel. An important idea in the design is to use an intelligent compiler which exposes the parallelism available in the program and reduces the work to be done at run time by the hardware. All the parallel hardware features such as the number of floating point and integer units is known to the compiler to enable it to schedule instructions based on the availability of resources. One of the bottlenecks we have noted in all architectures is the speed mismatch between the main memory and the processor. This mismatch is alleviated using caches. In IA-64, the architects have provided a large number of registers to reduce memory references. As the compiler knows these registers and their intended use the machine code is optimized. IA-64 has 128, 64-bit general purpose registers of which 32 are static and the rest may be used as a stack and assigned dynamically for various tasks. Besides this it has 128 floating point registers which are 82 bits long, 128, 64-bit registers which can be used by the compiler for various applications. There are 8, 64-bit branch registers and 64, 1-bit registers called predicate registers whose purpose will be evident soon.

IA-64 [www.intel.com] instruction word is 128-bits long and its format is given in Fig. 3.32. Observe that each instruction word has a bundle of three instructions, each of 41-bits, packed in it. The intelligent compiler explicitly detects which instructions can be carried out in parallel. This information is put in a 5-bit template field which identifies the instructions in

the current bundle of three instructions (in the instruction word) and in the following bundle that can be carried out in parallel.

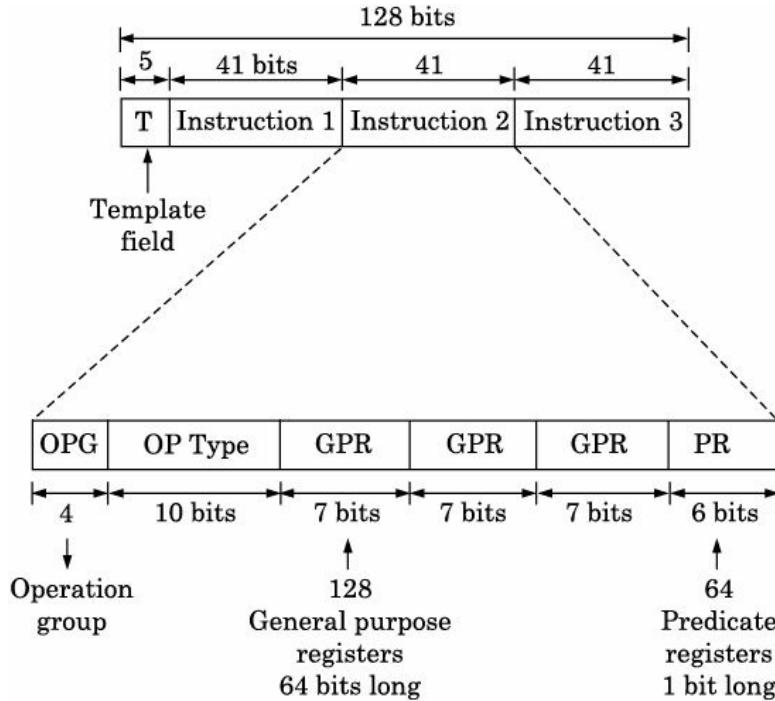


Figure 3.32 IA-64 instruction format.

An instruction has 3 GPR fields of 7-bits, each of which specify 128 integer or 128 floating point registers. A 6-bit field defines 64 predicate registers. This is an innovation in the IA-64 architecture. These registers are used to speculatively execute instructions across branches. We explain the working of this idea, called *predication* by Intel, with the program segment shown in Fig. 3.33. The objective of the compiler of IA-64 is to get maximum possible parallelism and execute as much straight line code as possible in many pipelines working in parallel. We have seen that branches delay pipelines and we used branch prediction techniques. Instead of predicting which branch will be taken and carrying out instructions along that branch (and later on finding that the prediction was wrong) IA-64 schedules instructions in *both* the branches till such a point that results are not committed to memory. However, only one of the branches would be valid depending on the result of the test of the predicate. Thus when a predicate instruction (13 in this example) is encountered, it attaches a predicate register say P1 to that instruction and to each of the instructions along the two branches. The P1 field is set to 1 along the true branch and 0 along false branch (see Fig. 3.34). When the predicate is evaluated P1 register gets a value of 1 or 0 depending on whether x is true or false. If x is true, all instructions with $P1 = 0$ are discarded and those with $P1 = 1$ are committed. The evaluation along one of the branches is wasted. The compiler has to decide whether cycles we lose due to incorrect prediction wastes more cycles as compared to using predicate bit. Thus when the compiler has a global look at the program, it will assign only selected branches for speculative evaluation. Remember that there are only 64 prediction registers and every sixth instruction in a program is normally a branch.

I1
I2
I3: If x then A else B (Assign predicate register P1 to this branch instruction)
A: I41 (P1 = 1)
I51 (P1 = 1)

I61 (P1 = 1)
.
.
.
B: I40 (P1 = 0)
I50 (P1 = 0)
I60 (P1 = 0)
.
.
.

Figure 3.33 Program to illustrate use of predicate register.

I1	I2	I3	P1
P1	P1	P1	
I41	1	I40	0
I50	0		
I51	1	I60	0
I61	1		

Figure 3.34 Predicate register bits for speculative evaluation.

Another innovation attempted in IA-64 is *speculative loading* of data from memory to registers. This is to alleviate memory-CPU speed mismatch. The idea is to re-order instructions and issue a speculative load ahead of a branch instruction. A speculative check instruction is inserted where the original load instruction was present. If branch is taken as speculated, the speculative check commits the data. Otherwise it discards the data.

Overall the primary aim in IA-64 architecture is to do an aggressive compiler transformation, detect parallelism at compile time, carry out parallel instructions speculatively and use resources in the processor with greatest possible efficiency.

3.7 MULTITHREADED PROCESSORS

Superscalar and VLIW processors which we discussed in the last section use Instruction Level Parallelism (ILP) to speed up computation. It is found in practice that there may not be enough ILP to effectively use the parallelism in programs. Thus, another method has been explored to use what is known as *thread level parallelism* which we will describe in this section.

We define a *thread* as a short sequence of instructions schedulable as a unit by a processor. A *process* in contrast normally consists of a long sequence of instructions which is scheduled by the operating system to be executed on a processor. Consider the following loop:

```
for i := 1 to 10 do
    a(i) := b(i) + c(i)
    x(i) := y(i) * z(i)
end for
```

This loop can be “unrolled” and a thread created for each value of i . We will then have 10 threads which can potentially be executed in parallel provided sufficient resources are available. A multithreaded processor [Culler, Singh, and Gupta, 1999] is designed to execute a thread and switch to another thread or issue several threads simultaneously as decided by the designer. We will describe in this section three types of multithreaded processors. They are:

- Coarse grained multithreaded processors
- Fine grained multithreaded processors and
- Simultaneous multithreaded processors

All these processors have a basic pipelined execution unit and in effect try to make best use of pipeline processing by using thread parallelism.

3.7.1 Coarse Grained Multithreading

In a coarse grained multithreaded processor, a program is broken into many threads which are independent of one another. Each thread has an independent stack space but shares a common global address space. To explain the way the processor works assume that there are 4 threads A, B, C and D which are ready and can be scheduled for execution. Let A be scheduled first. Let instruction I₁ of thread A be issued first followed by instructions I₂, I₃, I₄, etc., in successive cycles. The progress of I₈, for example, in the pipeline (see Fig. 3.35) may be delayed due to data dependency, control dependency, resource non-availability or long delay due to cache miss (i.e., required data is not in cache). If the delay is one or two clock cycles, one may tolerate it. However, if the delay is several cycles, (for example, if the required data is not in the cache, it may delay progress of I₈ by 10 cycles or if I₇ is performing a floating point divide I₈ may be delayed by several cycles if it also wants to do a floating point operation and no additional FPU is available), then the processor will suspend thread A and switch to another thread, say B. Before switching to thread B, the “status” of A should be saved so that it can be resumed when the required data or resource is available. The status of A consists of the value of PC, program status word, instruction register, and processor registers. It is, of course, not appropriate to store the status in the cache because it will take several cycles to store and retrieve them later. The best solution is to have a set of status

registers reserved for each thread and simply switch from one set to another. This is feasible if the number of threads is not too many (say less than 10). Thread B can now start issuing instructions. If it gets held up due to some reason then the processor switches to thread C. In the meanwhile if thread A is ready again to resume execution, the processor will switch back to A. The general idea is shown in Fig. 3.36. This is somewhat like multiprogramming but unlike multiprogramming where we have several long running processes controlled by the operating system, we have much smaller time scale, measured in number of processor clock cycles.

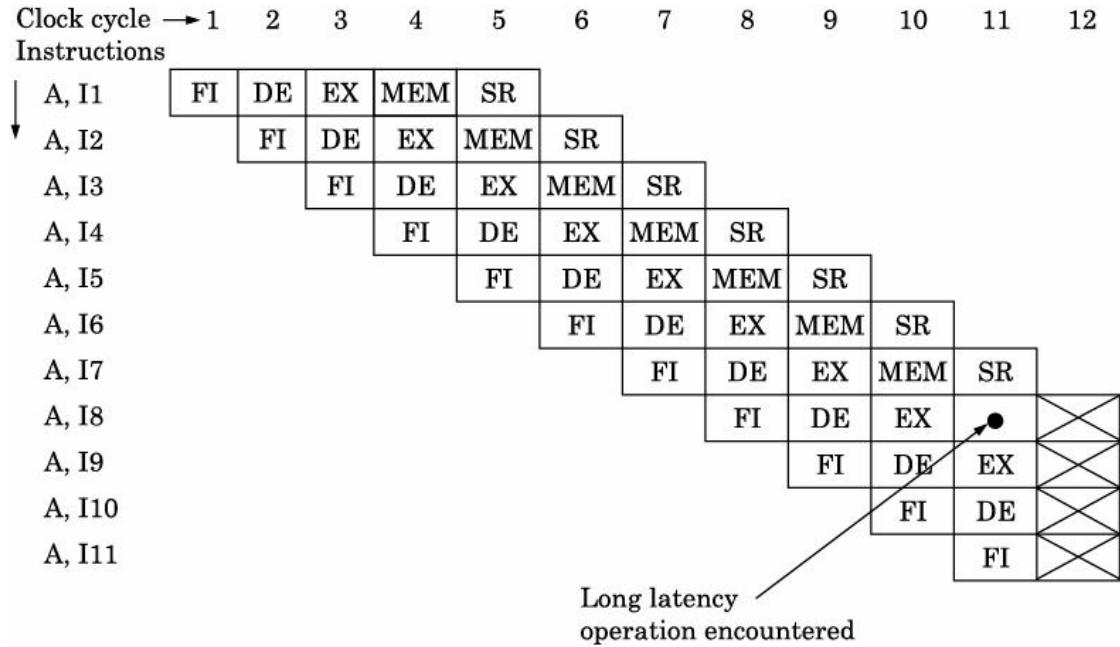


Figure 3.35 Pipeline interruption due to long latency operation.

The major design questions which arise are:

1. How many threads should be supported?
2. What is the processor efficiency?

In order to answer these questions we must identify the parameters which affect the performance of the processor. These are:

1. The average number of instructions which a thread executes before it suspends (let us say it is p).
2. The delay when a processor suspends a thread and switches to another one (let it be q cycles).
3. The average waiting time of a thread before it gets the resource it needs to resume execution called latency (let it be w cycles).

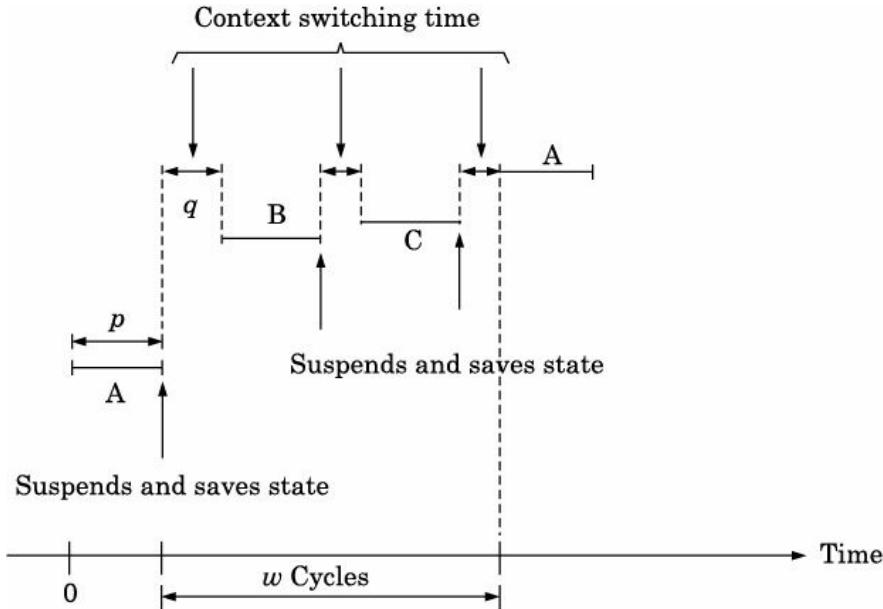


Figure 3.36 Processing of multiple threads in a processor.

We will count all these in number of processor cycles. To find out the number of threads n to reduce the effects of latency (referring to Fig. 3.36), we use the simple formula:

$$nq + (n - 1)p \leq w$$

$$n \leq \frac{p + w}{p + q}$$

(Remember that p instructions take p cycles ideally in a pipeline machine). Let us substitute numbers to get an idea of the order of magnitude. Let $p = 7$, $w = 15$. If the number of pipeline stages is five and cache miss occurs in the MEM cycle, three more instructions would have been issued and the previous instruction would not have been completed. The best policy is to let the previous instruction complete its work and abandon the next three instructions. Referring to Fig. 3.35 if a delay occurs when I8 is being executed in its MEM cycle, we should allow I7 to complete and abandon I9, I10, I11. The reason we abandon the succeeding instructions is because they may depend on the suspended instruction. Thus, we should wait for at least 1 cycle in this example to complete I7 before switching to another thread. Therefore, $q = 1$ in this case. In longer pipelines q will be larger. In the worst case if instruction fetching is delayed, the delay can be equal to the depth of the pipeline.

Thus,

$$n = \frac{7 + 15}{7 + 1} = \frac{22}{8} \approx 3$$

Observe that larger p and smaller w reduces the number of threads to be supported.

A rough calculation of efficiency of processor is:

$$\frac{p}{p + q}$$

In the worst case with $q = 5$

$$\text{Efficiency} = \frac{7}{7 + 5} \approx 58\%$$

This poor utilization is due to the fact that the average non-blocked length of a thread, in this example, is small relative to context switching time. Coarse grained multithreading has

poor performance because the instructions in the pipeline when the thread is suspended are abandoned and this equals the length of the pipeline (in the worst case). It is too expensive in hardware to let them continue while another thread is invoked as we have to provide independent resources.

The question thus naturally arises whether there are better schemes. One such attempt is fine-grained multithreading.

3.7.2 Fine Grained Multithreading

In fine grained multithreading, every instruction is issued from a different thread. Referring to Fig. 3.37 observe that instruction I1 of thread A is issued, followed by instruction I1 of thread B and instruction I1 of thread C and so on. In this example we have chosen five threads equal to the depth of the pipeline and one instruction from each thread is issued. After one cycle we get back to thread A from which instruction I2 is issued, the first instruction I1 of A would have been completed when I2 from thread A is issued. Thus, I2 of A will not be delayed due to data or control dependency. However, if I1 of A is delayed due to data not being in the cache then it is better to delay A and take an instruction from another “ready” thread. In this processor architecture each thread has its own “private” set of registers including status registers. When a thread is suspended due to a long latency operation, all its registers (including general purpose registers in the register file) are “frozen”. When the next thread is invoked, the processor switches context to this thread’s registers. In other words, the processor must have a set of registers private to each thread so that switching from one thread to another can be done in one cycle. If we have to save the registers in a cache, it will take too long. In this architecture, a set of around 10 to 20 threads which can be executed in parallel are kept in individual buffers in a fast memory. Instructions are issued from successive buffers. As soon as an instruction of a thread is delayed because of a cache miss (say) or non-availability of a resource or a long latency operation such as a floating point division, no further instructions are issued from that thread. It is suspended and the thread’s buffer is flagged as blocked. The flag is reset when the situation causing the block is cleared. If say, instruction I1 of thread E is blocked and it takes 12 cycles to clear it, issue of further instructions from E is suspended for 12 more cycles. When the block is cleared, the flag of thread E’s buffer is reset and next instruction can be issued from E. When thread E is blocked, instructions can be taken from a sixth thread F so that at any given time five threads are active equal to the depth of the pipeline (see Fig. 3.38). We will now do a rough calculation of the number of threads needed to keep the pipeline busy.

	Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
Instructions													
A1 A, I1		FI	DE	EX	MEM	SR							
B1 B, I1		FI	DE	EX	MEM	SR							
C1 C, I1		FI	DE	EX	MEM	SR							
D1 D, I1		FI	DE	EX	MEM	SR							
E1 E, I1		FI	DE	EX	MEM	SR							
A2 A, I2		FI	DE	EX	MEM	SR							
B2 B, I2		FI	DE	EX	MEM	SR							
C2 C, I2		FI	DE	EX	MEM	SR							

Figure 3.37 Fine grained multithreading.

Thread A	Thread B	Thread C	Thread D	Thread E	Thread F	Thread G
A_n • • • A1	B_n • • • B1	C_n • • • C1	D_n • • • D1	E_n • • • E1	F_n • • • F1	G_n • • • G1
A1	B1	C1	D1	(X) E1	-	-
A2	B2	C2	(X) D2	-	F1	-
A3	B3	(X) C3	-	-	F2	G1
A4	(X) B4	-	-	E2	F3	G2
(X) A5	-	-	D3	E3	F4	G3
-	-	C4	D4	E4	(X) F5	G4
-	B5	C5	D5	E5	-	(X) G5
A6	B6	C6	D6	E6	-	-

Circled entries are suspended due to long latency operation. A1 is instruction I1 of thread A. The pipeline depth is 5.

Figure 3.38 Issue of instructions from multiple thread buffers.

Let w be the number of cycles needed by a blocked thread to resume and let d be the depth of the pipeline. Then the number of threads n must be greater than the depth of the pipeline. The latency must be covered by extra threads. If the probability of a long latency operation is q then the number of threads needed can be roughly calculated as:

$$n > d + w \times q$$

If $d = 5$, $w = 10$ and $q = 0.2$ then $n = 7$.

Observe that longer the pipeline depth larger is the number of threads required. Longer delay in clearing the hold up of an instruction also requires more threads to be available.

Different designers have taken different views on designing multithreaded processors. HEP (Heterogeneous Element Processor) [Smith, 1995] was the first commercially designed multithreaded processor. In this processor no cache was used for storing data with the result that when a thread needed data from memory, it was held up for about 40 CPU cycles (as main memory speeds are much lower than processor speeds). This machine thus had a provision for 128 threads. Each thread had its own register file and processor registers so that context switching from thread to thread could be done every cycle. If a machine has 32 registers and 8 status registers each of 4 bytes we require a fast store of 20 KB in the processor. The philosophy of design in this case was tolerating latency by multithreading. The same philosophy is followed by a multithreaded processor called *Tera*. This processor had provision for 128 active threads, pipeline depth of 16 and had 32 KB on-processor memory for registers (32 per thread) and 1K branch target registers. It had a long instruction word with three instructions per instruction word. Thus, each time an instruction word is issued three operations can be performed (unlike the simple example we have taken). This provision complicates the design considerably. *Tera* was specifically built as a processor to be used in parallel computers. As memory and message delays are considerable in parallel computers, the claim of the designers of *Tera* was that a latency tolerant processor is much more appropriate for use in these computers. A hybrid scheme which combines ideas from coarse-grained multithreading and fine-grained multithreading has also been proposed [Laudon, Gupta and Horowitz, 1994]. In this scheme a cache is a part of the processor and

thus memory latency is reduced. The number of threads is also reduced so that total resources needed to save the context of threads is reduced. The design allows more than one instruction from a thread in the pipeline (as in coarse grain scheme). Simulation studies have shown that this hybrid architecture has better performance on a set of benchmarks [Laudon, Gupta and Horowitz, 1994].

3.7.3 Simultaneous Multithreading

Simultaneous multithreading (SMT) attempts to use both instruction level parallelism employed by superscalar processors and thread level parallelism which is available when many independent threads are processed by a processor. Threads which can be potentially carried out in parallel are kept ready. Two or more instructions are issued simultaneously from one thread (may be out of order) in each cycle just as in superscalar architecture. In the succeeding cycles instructions from other threads are scheduled (see Fig. 3.39). Instructions I1 and I2 are issued from thread A in the first cycle followed by instructions I1 and I2 of thread B in the second cycle and I3, I4 from thread A in the third cycle and so on as shown in the figure. The instructions issued from each thread may not necessarily be in the program order. SMT architecture assumes enough cache in the chip for storing data and instructions besides storing multiple set of registers for storing the contexts of each of the threads. It is also assumed that many functional units are available. As we have seen the main aim of a coarse grained or fine grained multithreaded design is to reduce the delays in the pipeline when there is a data or control dependency or long delay operation. It does not take advantage of instruction level parallelism available in the thread. The main aim of superscalar design is to use the instruction level parallelism available in a thread. It can, however, not reduce delay in execution of individual threads due to dependencies. Simultaneous multithreading attempts to use the good features of both ideas. It should, however, have considerable resources; extra instruction fetch units, register sets with high bandwidth (needed by multithreaded processor) to save contexts, extra reorder buffers in retirement units, and many functional units (needed by superscalar processor). It also increases the complexity of instruction scheduling relative to superscalar processors and increases the contention for shared resources, particularly the memory subsystem.

	Clock cycle → 1 2 3 4 5 6 7				
	Instructions				
A, I1	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR
A, I2	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR
B, I1	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR
B, I2	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR
A, I3	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR
A, I4	FI	DE	MEM	EX	SR
	FI	DE	MEM	EX	SR

Figure 3.39 Pipeline execution in simultaneous multithreading.

Referring to Fig. 3.24 which gives the pipeline stages of a superscalar processor, a question arises on how this will be modified for a SMT processor. The simplest method would be to have independent instruction fetch unit, branch predictor and retirement unit for each thread and share the rest of the hardware. It is expensive to dual port the instruction cache to fetch instructions of two independent threads but it simplifies the design. It is much neater to have an independent branch prediction unit for each thread as otherwise tagging each thread and modifying the predictor complicates design. Lastly handling precise

interrupts require that each thread be retired in order. This is done neatly if there is one retirement unit for each thread.

In Table 3.9 we compare coarse-grained, fine-grained, and simultaneous multi-threaded processor.

TABLE 3.9 Comparison of Multithreaded Processors		
<i>Type of multithreading</i>	<i>What are extra resources required?</i>	<i>When is context switched?</i>
Coarse grained	Instruction fetch buffers, register files for threads, control logic/state	On pipeline stall
Fine grained	Register files, control logic/state	Every cycle
Simultaneous multithreading	Instruction fetch buffers, register files, control logic/state, return address stack, reorder buffer/retirement unit	No context switching

3.8 CONCLUSIONS

The main computing engine of a parallel computer is a microprocessor used as the PE. It is thus important to design high speed PEs in order to build high performance parallel computers. The main method used to build high speed PEs is to use parallelism at the instruction execution level. This is known as *instruction level parallelism*. The earliest use of parallelism in a PE was using temporal parallelism by pipelining instruction execution. RISC processors were able to execute one instruction every clock cycle by using pipelined execution. Pipelined execution of instructions is impeded by dependencies between instructions in a program and resource constraints. The major dependencies are data dependency and control dependency. Loss of speedup due to these dependencies are alleviated by both hardware and software methods. The major method used to alleviate data dependency using hardware is by register forwarding, i.e., forwarding values in registers to ALU of PE internally instead of storing it in memory. Source instructions may be rearranged by a compiler to reduce data dependency during execution. To alleviate loss of speedup due to control dependency hardware is enhanced by providing two fast memories called *branch prediction buffer* and *branch target buffer*. They primarily work by predicting which path a program will take when it encounters a branch and fetching the instruction from this predicted branch. It has been found that this method rearranges the instructions in the program in such a way that an instruction fetched need not be abandoned due to a branch instruction. Delay in pipelined execution of programs due to resource constraints is normally solved by providing resources such as an additional floating point arithmetic unit or speeding up the unit which causes delays. Adding more resources is done judiciously by executing a number of benchmark problems and assessing cost-benefit.

Besides using temporal parallelism to enhance the speed of PEs, data parallelism is used in what are known as superscalar RISC processors. Superscalar processing depends on available parallelism in groups of instructions of programs. Machines have been built which issue 4 instructions in each cycle and execute them in parallel in pipeline mode. These machines require additional functional units to simultaneously execute several instructions.

In superscalar RISC processors, the hardware examines only a small window of instructions and schedules them to use all available functional units, taking into account dependencies. This is sub-optimal. Compilers can take a more global view of a program and rearrange instructions in it to utilize PE's resources better and reduce pipeline delays. VLIW processors use sophisticated compilers to expose a sequence of instructions which have no dependency and require diverse resources available in the processor. In VLIW architecture, a single word incorporates many operations. Typically two integer operations, two floating point operations, two load/store operations and a branch may be all packed into one long instruction word which may be anywhere between 128 to 256 bits long. Trace scheduling is used to optimally use the resources available in VLIW processors. Early VLIW processors were primarily proposed by University researchers. These ideas were used by commercial PEs, notably the IA-64 processor of Intel.

A program may be decomposed into small sequences of instructions, called threads, which are schedulable as a unit by a processor. Threads which are not dependent on one another can be executed in parallel. It is found that many programs can be decomposed into independent threads. This observation has led to the design of multithreaded processors which execute threads in parallel. There are three types of multithreaded processors; coarsegrained, finegrained and simultaneous. All multithreaded processors use pipelined execution of

instructions. In coarsegrained multithreaded processors, a thread executes till it is stalled due to a long latency operation at which time another thread is scheduled for execution. In finegrained multithreading, one instruction is fetched from each thread in each clock cycle. This eliminates data and control hazards in pipelined execution and tolerates long latency if there is a cache miss. This PE was designed specifically for use in parallel computers and was used in a commercial parallel computer (TeraMTA-32). In simultaneous multithreaded processors several threads are scheduled to execute in each clock cycle. There is considerable current research being conducted on multithreading as it promises latency tolerant PE architecture.

Levels of integration have reached a stage where five billion transistors are integrated in an integrated circuit chip in 2015. This has allowed a number of processing elements called “cores” to be built on a chip. We will discuss the architecture of multi-core processors in Chapter 5. It is evident that all future PEs will be built to exploit as much parallelism as possible at instruction and thread level leading to very high performance. These PEs will be the computing engines of future parallel computers thereby enhancing the speed of parallel computers considerably in the coming years.

EXERCISES

- 3.1 We assume a five stage pipeline processing for the hypothetical computer SMAC2P. Assume that the processor uses a 4 stage pipeline. What would you suggest as the 4 pipeline stages? Justify your answer.
- 3.2 Develop data flows of SMAC2P with a 4 stage pipeline (a table similar to Table 3.2 is to be developed by you).
- 3.3 If the percentage of unconditional branches is 10%, conditional branches 18%, and immediate instructions 8% in programs executed in SMAC2P, compute the average clock cycles required per instruction.
- 3.4 Assume 5 stage pipelining of SMAC2P with percentage mix of instructions given in Exercise 3.3. Assuming 8 ns clock, and 80% of the branches are taken find out the speedup due to pipelining.
- 3.5 Repeat Exercise 3.4 assuming a 4 stage pipeline.
- 3.6 In our model of SMAC2P we assumed availability of separate instruction and data memories. If there is a single combined data and instruction cache, discuss how it will affect pipeline execution. Compute loss of speedup due to this if a fraction p of total number of instructions require reference to data stored in cache.
- 3.7 Describe how a floating point add/subtract instruction can be pipelined with a 4 stage pipeline. Assume normalized floating point representation of numbers.
- 3.8 If a pipelined floating point unit is available in SMAC2P and if 25% of arithmetic operations are floating point instructions, calculate the speedup due to pipeline execution with and without pipelined floating point unit.
- 3.9 Draw pipeline execution diagram during the execution of the following instructions of SMAC2P.

MUL R1, R2, R3
ADD R2, R3, R4
INC R4
SUB R6, R3, R7

Find out the delay in pipeline execution due to data dependency of the above instructions.

- 3.10 Describe how the delay can be reduced in execution of the instructions of Exercise 3.9 by (i) Hardware assistance, and (ii) Software assistance.
- 3.11 Repeat Exercise 3.9, if MUL instruction takes 3 clock cycles whereas other instructions take one cycle each.
- 3.12 Given the following SMAC2P program

LOOP	LD	R3, 0, Temp 1	$R3 \leftarrow \text{Temp 1}$
	SUB	R3, R4, R5	$R5 \leftarrow R3 - R4$
	JMI	XX	
	ST	R3, 0, Temp	$\text{Temp} \leftarrow R3$
	LD	R4, 0, Temp	$R4 \leftarrow \text{Temp}$

XX	DEC R1	R1 \leftarrow R1 - 1
	JEQ R1, R0, NEXT	If R1 = R0 go to NEXT
	JMP LOOP	
NEXT	ST R4, 0, Large	Large \leftarrow R4

- (i) With R1 = 2, R4 = 6, Temp 1 = 4, draw the pipeline execution diagram.
- (ii) With R0 = 0, R3 = 3, R4 = 4, Temp 1 = 6, draw the pipeline execution diagram.

3.13 Explain how branch instructions delay pipeline execution. If a program has 18% conditional branch instructions and 4% unconditional branch instructions and if 7% of conditional branches are taken branches, calculate the loss in speedup of a processor with 4 pipeline stages.

3.14 What modifications would you suggest in SMAC2P hardware to reduce delay due to branches? Using the data of Exercise 3.13, calculate the improvement in speedup.

3.15 What is the difference between branch prediction buffer and branch target buffer used to reduce delay due to control dependency?

3.16 Using the data of Exercise 3.13, compute the reduction in pipeline delay when branch prediction buffer is used.

3.17 Using the data of Exercise 3.13, compute the reduction in pipeline delay when branch target buffer is used.

3.18 Estimate the size of branch target buffer (in number of bytes) for SMAC2P using the data of Exercise 3.13.

3.19 How can software methods be used to reduce delay due to branches? What conditions should be satisfied for software method to succeed?

3.20 What conditions must be satisfied by the statement appearing before a branch instruction so that it can be used in the delay slot?

3.21 For the program of Exercise 3.12, is it possible to reduce pipelining delay due to branch instructions by rearranging the code? If yes show how it is done.

3.22 What do you understand by the term precise exception? What is its significance?

3.23 When an interrupt occurs during the execution stage of an instruction, explain how the system should handle it.

3.24 What is the difference between superscalar processing and superpipelining? Can one combine the two? If yes, explain how.

3.25 What extra resources are needed to support superscalar processing?

- (i) For the following sequence of instructions develop superscalar pipeline execution diagrams similar to that given in Fig. 3.26. Assume there is 1 one floating point and 2 integer execution units.

<i>Instruction</i>	<i>Number of cycles needed</i>	<i>Arithmetic unit needed</i>
R2 \leftarrow R2 \times R6	2	Floating point
R3 \leftarrow R2 + R1	1	Integer
R1 \leftarrow R6 + 8	1	Integer
R8 \leftarrow R2 - R9	1	Integer
R5 \leftarrow R4/R8	2	Floating point
R6 \leftarrow R2 + 4	1	Integer

$R2 \leftarrow R1 + 2$	1	Integer
$R10 \leftarrow R9 \times R8$	2	Floating point

- (ii) If there are 2 integer and 2 floating point execution units, repeat (i).
- (iii) Is it possible to rename registers to reduce the number of execution cycles?
- (iv) Reschedule instructions (if possible) to reduce the number of cycles needed to execute this set of instructions.

3.26 Distinguish between flow dependency, anti-dependency and output dependency of instructions. Give one example of each of these dependencies.

- (i) Why should one be concerned about these dependencies in pipelined execution of programs?
- (ii) If in pipelined execution of programs out of order completion of instructions is avoided, will these dependencies matter?

3.27 What is score boarding? For the sequence of instructions given in Exercise 3.25, develop a score board. Illustrate how it is used in register renaming.

3.28 (i) Define a trace.

- (ii) How is it used in a VLIW processor?
- (iii) List the advantages and disadvantages of VLIW processor.

3.29 (i) Is ARM Cortex A9 a RISC or a CISC processor?

- (ii) Is it superscalar or superpipelined?
- (iii) Is it a VLIW processor?
- (iv) How many integer and floating point execution units does it have?
- (v) Show pipeline execution sequence of instructions of Exercise 3.25 on ARM Cortex A9.

3.30 (i) Is Intel Core i7 processor a RISC or CISC processor?

- (ii) Is it superscalar or superpipelined?
- (iii) What is the degree of superscalar processing in i7?
- (iv) Show pipelined execution sequence of instructions of Table 3.6 on a i7 processor.

3.31 (i) Is IA-64 a VLIW processor?

- (ii) Is it a superscalar processor?
- (iii) How many instructions can it carry out in parallel?
- (iv) Explain how it uses predicate registers in processing.

3.32 (i) Define the term thread.

- (ii) What is the difference between a thread, a trace and a process?
- (iii) What is multithreading?
- (iv) List the similarities and differences between a coarse grained multi-threaded processor, fine grained multithreaded processor and a simultaneous multithreaded processor.

3.33 What are the similarities and differences between multithreading and multiprogramming?

3.34 If a computer does not have an on-processor cache which type of multi-threaded processor would you use and why?

3.35 What type of multithreading is used by:

- (i) HEP processor?
- (ii) Tera processor?

3.36 Explain with a pipeline diagram how the instructions in Exercise 3.12 will be carried

out by a fine grained multithreaded processor if $R_1 = 10$.

- 3.37 Assume that the average number of instructions a thread executes before it suspends is 15, the delay when a thread suspends and switches to another one is 3 cycles and the average number of cycles it waits before it gets the resource it needs is 30. What is the number of threads the processor should support to hide the latency? What is the processor efficiency?
- 3.38 Explain how simultaneous multithreading is superior to multithreading. What extra processor resources are required to support simultaneous multithreading?

BIBLIOGRAPHY

- Culler, D.E., Singh, J.P. and Gupta, A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, USA, 1999.
- Fisher, J.A., “Very Long Instruction Word Architectures and ELI-512”, *Proceedings of ACM Symposium on Computer Architecture*, Sweden, 1983.
- IA-64 (Website: www.intel.com)
- Laudon, J., Gupta, A. and Horowitz, A., *Multithreaded Computer Architecture*, [Iannucci, R.A. (Ed.)], Kluwer Academic, USA, 1994.
- Patterson, D.A. and Hennessy, J.L., *Computer Architecture—A Quantitative Approach*, 5th ed., Reed Elsevier India, New Delhi, 2012.
- Rajaraman, V. and Radhakrishnan, T., *An Introduction to Digital Computer Design*, 5th ed., PHI Learning, New Delhi, 2008.
- Shen, J.P., and Lipasti, M.H., *Modern Processor Design*, Tata McGraw-Hill, New Delhi, 2010.
- Smith, B.J., *The Architecture of HEP in Parallel MIMD Computation*, [Kowalik, J.S. (Ed.)], MIT Press, USA, 1985.
- Stallings, W., *Computer Organization and Architecture*, 4th ed., Prentice-Hall of India, New Delhi, 1996.
- Tannenbaum, A.S. and Austin, T., *Structured Computer Organization*, 6th ed., Pearson, New Delhi, 2013.
- Theme Papers on One Billion Transistors on a Chip, *IEEE Computer*, Vol. 30, No. 9, Sept. 1997.
- Tullsen, D.M., Eggers, S.J. and Levy, H.M., “Simultaneous Multithreading: Maximizing On-chip Parallelism”, *Proceedings of ACM Symposium on Computer Architecture*, 1995.

Structure of Parallel Computers

In the last chapter we saw how parallelism available at the instruction level can be used extensively in designing processors. This type of parallelism is known in the literature as fine grain parallelism as the smallest grain is an instruction in a program. In this chapter we will examine how processors may be interconnected to build parallel computers which use a coarser grain, for example, threads or processes executing in parallel.

4.1 A GENERALIZED STRUCTURE OF A PARALLEL COMPUTER

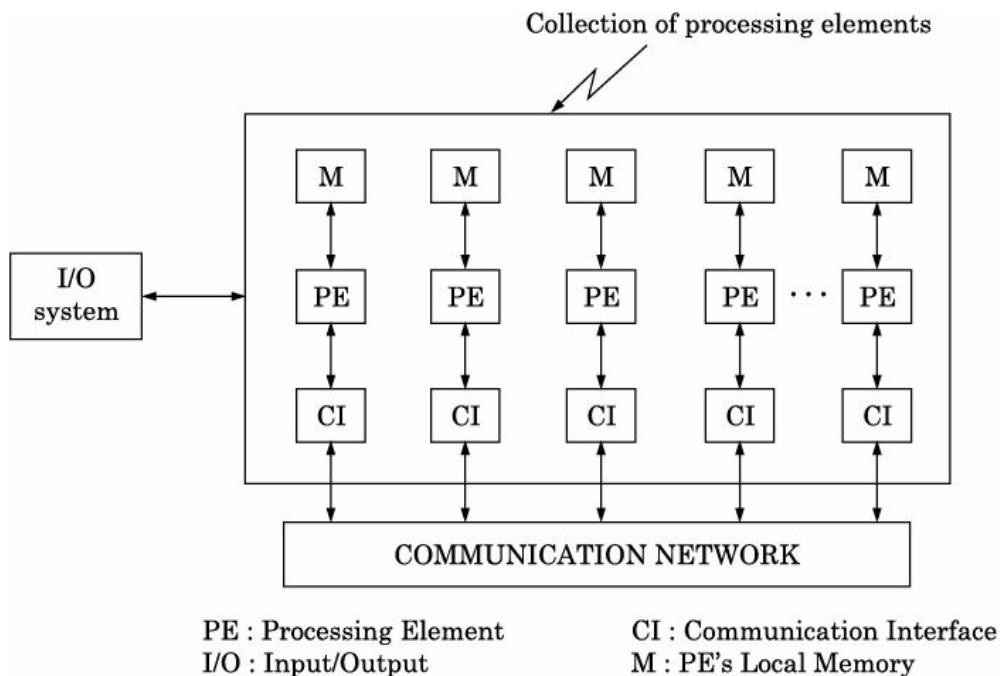
A parallel computer is defined as:

an interconnected set of Processing Elements (PEs) which cooperate by communicating with one another to solve large problems fast.

We see from this definition that the keywords which define the structure of a parallel computer are PEs, communication and cooperation. Figure 4.1 shows a generalized structure of a parallel computer. The heart of the parallel computer is a set of PEs interconnected by a communication network. This general structure can have many variations based on the type of PEs, the memory available to PEs to store programs and data, and how memories are connected to the PEs, the type of communication network used and the technique of allocating tasks to PEs and how they communicate and cooperate. The variations in each of these lead to a rich variety of parallel computers. The possibilities for each of these blocks are:

Type of Processing Elements

1. A PE may be only an Arithmetic Logic Unit (ALU) or a tiny PE. The ALUs may use 64-bit operands or may be quite tiny using 4-bit operands.
2. A PE may also be a microprocessor with only a private cache memory or a full fledged microprocessor with its own cache and main memory. We will call a PE with its own private memory a Computing Element (CE). PEs themselves are becoming quite powerful and may themselves be parallel computers as we will see the next chapter.
3. A PE may be server or a powerful large computer such as a mainframe or a Vector Processor.



Remarks: PE may have a private memory in which case it is called a Computing Element (CE).

Figure 4.1 A generalized structure of a parallel computer.

Number of Processing Elements

1. A small number (10 to 100) of powerful PEs.
2. A medium number of microprocessors (100 to 10000).
3. A large number of servers (>100,000) or tiny PEs (e.g., ALUs) (> 100,000).

Communication Network

1. A fixed interconnection network. A large variety of interconnection networks have been proposed and built.
2. A programmable interconnection network. In this case switches are used for interconnection which can be programmed using bit patterns to change the connections.
3. A single bus (i.e., a parallel set of wires) or a set of buses as the communication network.
4. A memory shared by all the PEs which is used to communicate among the PEs.
5. A combination of two or more of the above.

Memory System

1. Each PE has its own private caches (multi-level) and main memory.
2. Each PE has its own private caches (multi-level) but all PEs share one main memory which is uniformly addressed and accessed by all PEs.
3. Each PE has its own private caches (multi-level) and main memory and all of them also share one large memory.

Mode of Cooperation

1. Each CE (a PE with its own private memory) has a set of processes assigned to it. Each CE works independently and CEs cooperate by exchanging intermediate results.
2. All processes and data to be processed are stored in the memory shared by all PEs. A free PE selects a process to execute and deposits the results in the shared memory for use by other PEs.
3. A host CE stores a pool of tasks to be executed and schedules tasks to free CEs dynamically.

From the above description we see that a rich variety of parallel computer structures can be built by permutation and combination of these different possibilities.

4.2 CLASSIFICATION OF PARALLEL COMPUTERS

The vast variety of parallel computer architecture may be classified based on the following criteria:

1. How do instructions and data flow in the system? This idea for classification was proposed by Flynn [1972] and is known as Flynn's classification. It is considered important as it is one of the earliest attempts at classification and has been widely used in the literature to describe various parallel computer architectures.
2. What is the coupling between PEs? Coupling refers to the way in which PEs cooperate with one another.
3. How do PEs access memory? Accessing relates to whether data and instructions are accessed from a PE's own private memory or from a memory shared by all PEs or a part from one's own memory and another part from the memory belonging to another PE.
4. What is the quantum of work done by a PE before it communicates with another PE? This is commonly known as the *grain size* of computation.

In this section we will examine each of these classifications. This will allow us to describe a given parallel computer using adjectives based on the classification.

4.2.1 Flynn's Classification

Flynn classified parallel computers into four categories based on how instructions process data. A computer with a single processor is called a *Single Instruction stream Single Data stream (SISD) Computer*. In such a computer, a single stream of instructions and a single stream of data are accessed by the PE from the main memory, processed and the results stored back in the main memory.

The next class of computers which have multiple processors is known as a *Single Instruction stream Multiple Data stream (SIMD) Computer*. A block diagram of such a computer is shown in Fig. 4.2. Observe that in this structure there is no explicit communication among processors. However, data paths between nearest neighbours are used in some structures. SIMD computers have also been built as a grid with communication between nearest neighbours (Fig. 4.3). All processors in this structure are given identical instructions to execute and they execute them in a lock-step-fashion (simultaneously) using data in their memory.

The unique characteristic of SIMD computers is that all PEs work synchronously controlled by a single stream of instructions. An instruction may be broadcast to all PEs and they can process data items fed to them using this instruction. If instead of a single instruction, the PEs use identical programs to process different data streams, such a parallel computer is called a *Single Program Multiple Data (SPMD) Computer*. This model is analogous to the data parallel processing model we described in Chapter 2 (Section 2.2).

SIMD computers are used to solve many problems in science which require identical operations to be applied to different data sets synchronously. An example is adding a set of

arrays simultaneously (e.g.), $\sum_{i=1}^n \sum_{k=1}^m (a_{ik} + b_{ik})$. Such computers are known as *array processors* which we will discuss later in this chapter.

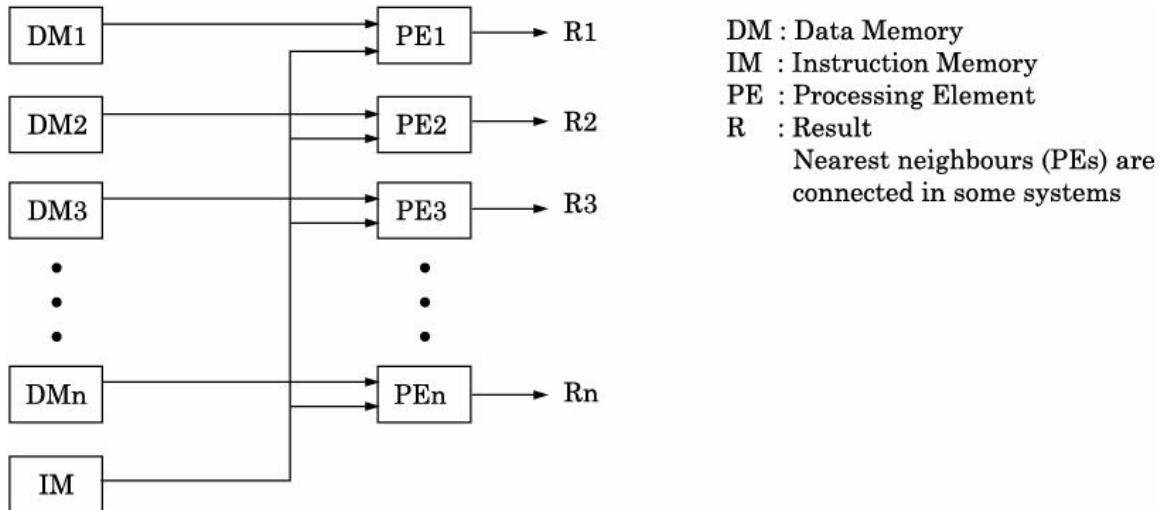
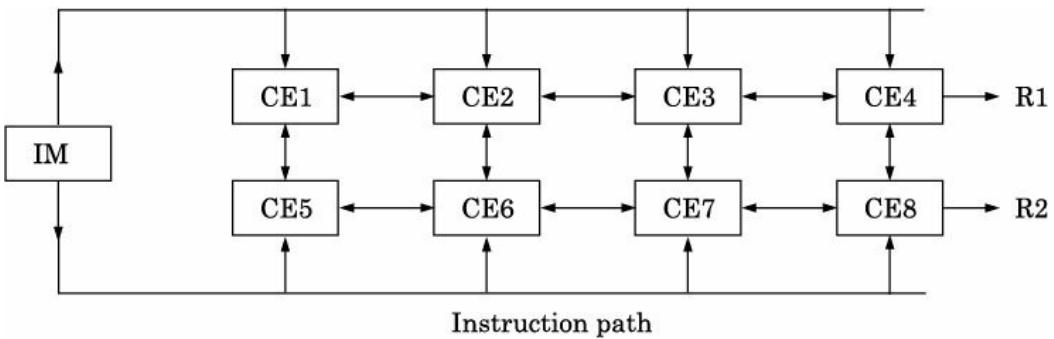


Figure 4.2 Structure for a single instruction multiple data computer.



CE : Processing Elements with private data memory
IM : Instruction Memory

Figure 4.3 Regular structure of SIMD computer with data flowing from neighbours.

The third class of computers according to Flynn's classification is known as *Multiple Instructions stream Single Data stream (MISD) Computers*. This structure is shown in Fig. 4.4. Observe that in this structure different PEs run different programs on the same data. In fact pipeline processing of data explained in Section 2.1 is a special case of this mode of computing.

In the example we took in Section 2.1, the answer books passed from one teacher to the next corresponds to the data stored in DM and instructions to grade different questions given to the set of teachers is analogous to the contents of IM₁ to IM_n in the structure of Fig. 4.4. In pipeline processing the data processed by PE₁, namely, R₁ is fed to PE₂, R₂ to PE₃ etc., and DM contents will be input to only PE₁. This type of processor may be generalized using a 2-dimensional arrangement of PEs. Such a structure is known as a *systolic processor*. Use of MISD model in systolic processing will be discussed later in this chapter.

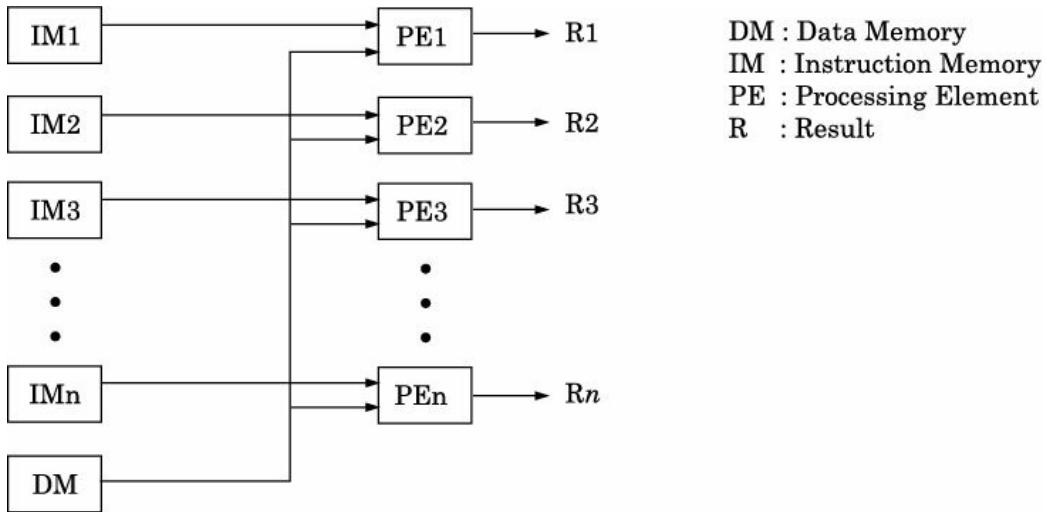


Figure 4.4 Structure of a Multiple Instruction Single Data (MISD) Computer.

The last and the most general model according to Flynn's classification is *Multiple Instructions stream Multiple Data stream (MIMD) Computer*. The structure of such a computer is shown in Fig. 4.5.

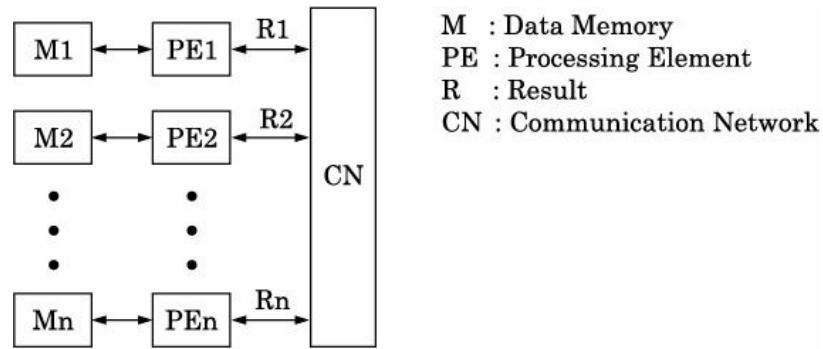


Figure 4.5 Structure of a Multiple Instruction Multiple Data (MIMD) Computer.

Observe that this is similar to the model of parallel computer we gave in Fig. 4.1. We will devote most of our discussions in this chapter to MIMD computers as these are general purpose parallel computers.

4.2.2 Coupling Between Processing Elements

The autonomy enjoyed by the PEs while cooperating with one another during problem solving determines the degree of coupling between them. For instance a parallel computer consisting of servers (usually a single board computer) connected together by a local area network such as an Ethernet is *loosely coupled*. In this case each server works independently. If they want to cooperate, they will exchange messages. Thus, logically they are autonomous and physically they do not share any memory and communication is via I/O channels. A *tightly coupled* parallel computer, on the other hand, shares a common main memory. Thus, communication among PEs is very fast and cooperation may be even at the level of instructions carried out by each PE as they share a common memory. In Fig. 4.6, we summarize this discussion.

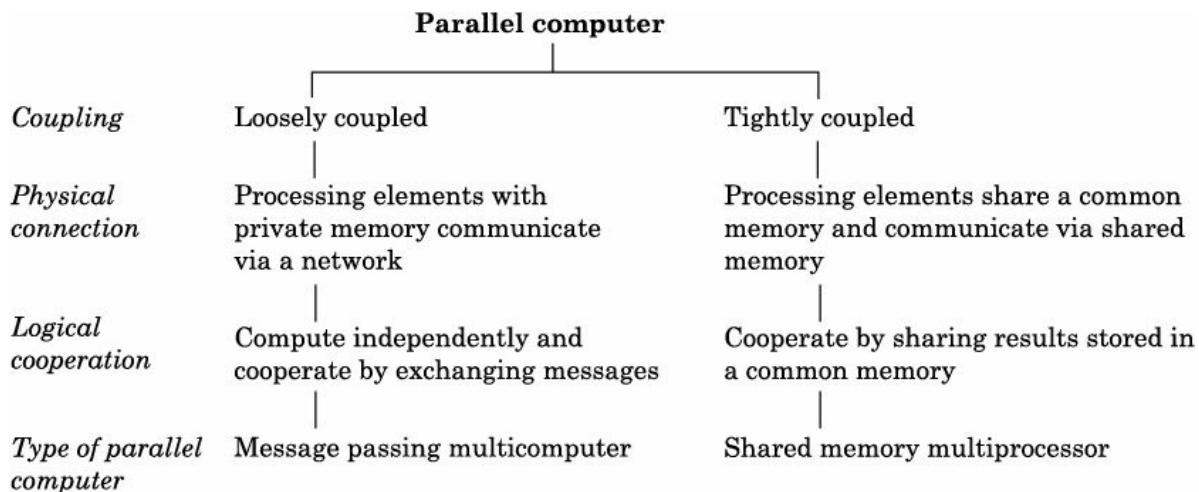
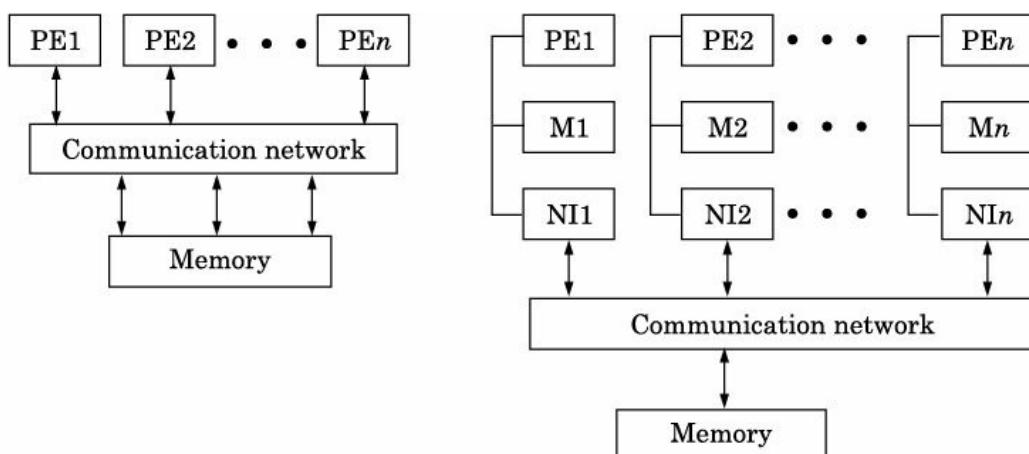


Figure 4.6 Classification as loosely or tightly coupled system.

4.2.3 Classification Based on Mode of Accessing Memory

In a shared memory computer all the processors *share a common global address space*. In other words, programs are written for this parallel computer assuming that all processors address a common shared memory. Physically the main memory may be a single memory bank shared by all processors or each processor may have its own local memory and may or may not share a common memory. These two models may be called Shared Memory (SM) computer and Distributed Shared Memory (DSM) computer respectively. The time to access a word in memory is constant for all processors in the first case. Such a parallel computer is said to have a *Uniform Memory Access* (UMA). In a distributed shared memory computer the time taken to access a word in its local memory is smaller than the time taken to access a word stored in the memory of another computer or a common shared memory. Thus, this system (DSM system) is said to have a *Non Uniform Memory Access* (NUMA). If a remote memory is accessed by a PE using a communication network, it may be 10 to 1000 times slower than accessing its own local memory. In Fig. 4.7, we can see the physical organization of these two types of parallel computers.



PEs have no local memory. All PEs share a common shared memory in UMA systems

(a) UMA parallel computer

PE : Processing Element

M : Memory

NI : Network Interface

There may or may not be a shared physical memory

(b) NUMA parallel computer

Figure 4.7 UMA and NUMA Parallel Computer Structures.

4.2.4 Classification Based on Grain Size

We saw that the quantum of work done by a PE before it communicates with another processor is called the *grain size*. The grain sizes are classified as very fine grain, fine grain, medium grain, and coarse grain. The grain size determines the frequency of communication between PEs during the solution of a problem. This will correspond to the number of instructions carried out by a PE before it sends the result to a cooperating PE. The smallest grain size is a single instruction. We saw in the last chapter how instruction level parallelism is used to make single processors work fast. Instruction level parallelism is exploited by having multiple functional units and overlapping steps during instruction execution. We will call it very fine grain parallelism. Compilers are very useful in exposing very fine grain parallelism and processors, such as VLIW use it effectively. In this chapter we are concerned with fine, medium and coarse grain processing as applied to parallel computers which use microprocessors as PEs. In this case fine grain parallelism will correspond to a thread. A thread typically may be the instructions belonging to one iteration of a loop, typically 100 machine instructions. Medium grain parallelism, on the other hand, will correspond to a procedure (or subroutine). It will typically be 1000 machine instructions. Coarse grain will correspond to a complete program. Tightly coupled parallel computers can exploit fine grain parallelism effectively as the processors cooperate via a shared memory and time to access a shared memory is small (a few machine cycles). The main point is that the compute time must be much larger than the communicating time. Loosely coupled parallel computers, on the other hand, can exploit only medium or coarse grain parallelism as the time to communicate results using a communication network will be a few 100 clock cycles and the quantum of work done by a processor must be much larger than this.

To explain the points made above we will consider an n -processor machine solving a problem.

Assume it takes k cycles to solve a problem on a single processor. If n processors are used, ideally each processor (if it computes in a fully overlapped mode) should take k/n cycles to solve the problem.

Let $k/n = p$ cycles

Let each processor compute continuously for p_i cycles and spend q_i cycles communicating with other processors for cooperatively solving the problem. During these q_i cycles, the processor is idle. Thus, time taken to compute and communicate once = $p_i + q_i$. The number of communication events will be inversely proportional to the grain size. If there are m communication events, the time taken by each processor to compute

$$\begin{aligned} &= \sum_{i=1}^m (p_i + q_i) = \sum_{i=1}^m p_i + \sum_{i=1}^m q_i \\ &= p + \sum_{i=1}^m q_i \end{aligned}$$

Total time to compute is increased by $\sum_{i=1}^m q_i$ cycles.

Each communication event in a loosely connected system has an overhead having a fixed part (due to the need to take the assistance of the operating system to communicate) and a variable part which depends on the speed of the communication channel.

Thus, $q_i = T + s_i$

where T is the fixed overhead.

If there are m communication events then the total communication time

$$= \sum_{i=1}^m q_i = mT + \sum_{i=1}^m s_i$$

$$\therefore \text{Total time taken to compute in parallel} = p + mT + \sum_{i=1}^m s_i$$

Thus, the total time taken to compute in parallel = $p + mT + \sum_{i=1}^m s_i$

$$\text{Speedup} = k / \left(p + mT + \sum_{i=1}^m s_i \right)$$

$$= \frac{k/p}{1 + \left\{ \left(mT + \sum_{i=1}^m s_i \right) / p \right\}} = \frac{n}{1 + \frac{m}{p} (T + s)}$$

where we have assumed for simplicity that $s_i = s$. To fix our ideas we will substitute some values in the above equation.

Let the overhead T for each communication event be 100 cycles and the communication times be 20 cycles. Let $n = 100$ (number of processors) and the total compute time p of each processor = 50000 cycles and m the number of communication events be 100.

$$\text{Speedup} = \frac{100}{1 + \frac{100(100 + 20)}{50000}} = \frac{100}{1 + \frac{12000}{50000}} = \frac{100}{1 + 0.24} \approx 80$$

Thus, there is 20% loss in speedup when the grain size is 500 cycles. If the processors are pipelined, one instruction is carried out in each cycle. Thus, if the parallel computer uses such processors and if each processor communicates once every 500 instructions, the overall efficiency loss is 20%.

In general if the loss in efficiency is to be less than 10% then $m/p(T + s) < 0.1$. This implies that the gain size $p/m > 10(T + s)$.

If $T = 100$ and $s = 20$ then grain size > 1200 cycles. If one instruction is carried out every cycle, the grain size is 1200 instructions. Thus, each processor of a loosely coupled parallel computer may communicate only once every 1200 instructions if loss of efficiency is to be kept low.

On the other hand in a tightly coupled system T is 0 as the memory is shared and no operating system call is necessary to write in the memory. Assuming s of the order of 2 cycles the grain size for 10% loss of speedup is 20 cycles. In other words, after every 20 instructions a communication event can take place without excessively degrading performance.

The above calculations are very rough and intended just to illustrate the idea of grain size. In fact we have been very optimistic in the above calculations and made many simplifying assumptions. These points will be reiterated later with a more realistic model.

4.3 VECTOR COMPUTERS

One of the earliest ideas used to design high performance computers was the use of temporal parallelism (i.e., pipeline processing). Vector computers use temporal processing extensively. The most important unit of a vector computer is the pipelined arithmetic unit. Consider addition of two floating point numbers x and y . A floating point number consists of two parts, a mantissa and an exponent. Thus, x and y may be represented by the tuple $(\text{mant } x, \text{exp } x)$ and $(\text{mant } y, \text{exp } y)$ respectively. Let $z = x + y$. The sum z may be represented by $(\text{mant } z, \text{exp } z)$. The task of adding x and y can be broken up into the following four steps:

Step 1: Compute $(\text{exp } x - \text{exp } y) = m$

Step 2: If $m > 0$ shift mant y , m positions right and fill its leading bit positions with zeros. Set $\text{exp } z = \text{exp } x$. If $m < 0$ shift mant x , m positions right and fill the leading bits of mant x with zeros. Set $\text{exp } z = \text{exp } y$. If $m = 0$ do nothing. Set $\text{exp } z = \text{exp } x$.

Step 3: Add mant x and mant y . Let $\text{mant } z = \text{mant } x + \text{mant } y$.

Step 4: If $\text{mant } z > 1$ then shift mant z right by 1 bit and add 1 to exponent. If one or more significant bits of $\text{mant } z = 0$ shift mant z left until leading bit of mant z is not zero. Let the number of shifts be p . Subtract p from $\text{exp } z$.

A block diagram of a pipelined floating point adder is shown in Fig. 4.8. Such a pipelined adder can be used very effectively to add two *vectors*. A vector may be defined as an ordered sequence of numbers. For example $x = (x_1, x_2, \dots, x_n)$ is a vector of length n . To add vectors x and y , each of length n , we feed them to a pipelined arithmetic unit as shown in Fig. 4.9. One pair of operands is shifted into the pipeline from the input registers every clock period, let us say T .

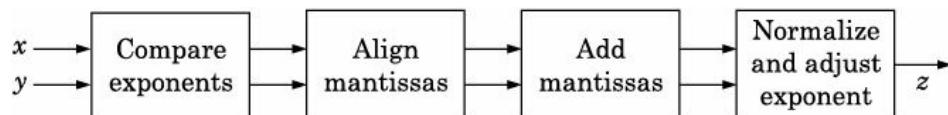


Figure 4.8 A four stage pipelined floating point adder.



Figure 4.9 A four stage pipelined adder.

The first component z_i will take time $4T$ but subsequent components will come out every T seconds. If two vectors each 64 components long are fed to the pipelined adder, the time taken to add them will be $67T$. Thus, clocks per sum is $(67/64) = 1.05$ which is almost one sum every clock period. The efficiency of pipeline addition increases when the vectors become longer. The pipeline stages may also be increased which will reduce T , the period required for each stage.

We discussed vector add. Vector subtract is not different (it is the complement of addition). Pipelined multiply and divide units are more complicated to build but use the same general principle. Observe that pipelined arithmetic operations do not have problems of data dependency as each component of a vector is independent of the other. There is no control dependency as the entire vector is considered as a unit and operations on vectors are implemented without any loop instruction.

A block diagram of a vector arithmetic unit of a typical vector computer is shown in Fig. 4.10 [Rajaraman, 1999]. Observe that an important part is a set of vector registers which feed the vector pipelined adders and multipliers. Typically vector registers used to store 64 floating point numbers, which are each 64 bits long. This was the vector register size used by Cray computers which were the most commonly used supercomputers of the 80s. Recent vector machines built by NEC of Japan [Nakazato, et al., 2008] use vector registers which store 256 double precision (64 bits) numbers. Observe in Fig. 4.10 that the output of the pipeline adder unit may be fed to another pipelined multiply unit. As the components of $(A + B)$, namely, $(a_1 + b_1), (a_2 + b_2) \dots (a_n + b_n)$ are generated, they are multiplied by the components of C, namely, $c_1 c_2 \dots c_n$ to produce $(a_1 + b_1) \times c_1, (a_2 + b_2) \times c_2, (a_3 + b_3) \times c_3, \dots (a_n + b_n) \times c_n$ respectively and stored in the vector register D. This method of feeding the output of one pipelined arithmetic unit to another is called *vector chaining*. With chaining the speed of computation is doubled as add and multiply are performed in one clock cycle.

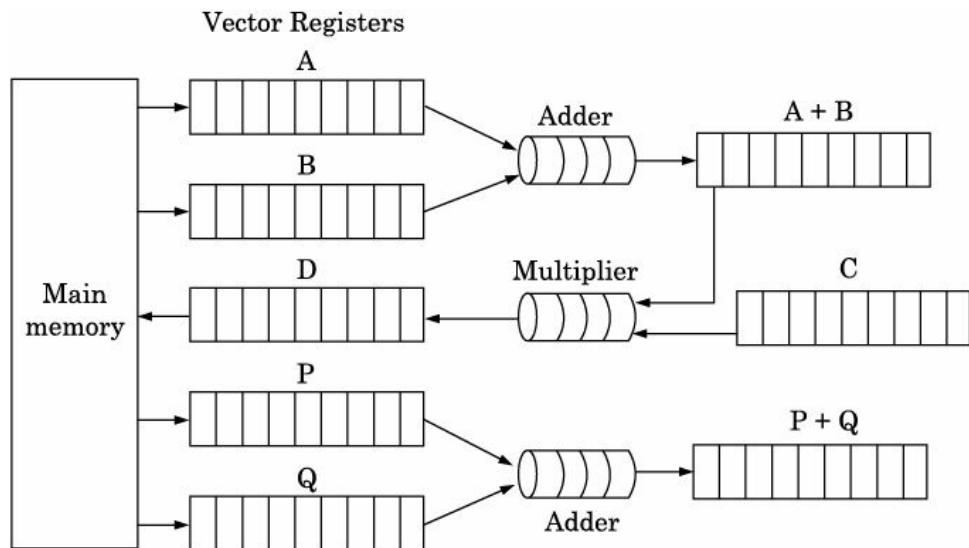


Figure 4.10 Use of pipelined arithmetic in a vector computer.

Besides vector chaining, many independent pipelined arithmetic units are provided in a vector computer. In Fig. 4.10, for example, there is a second pipelined adder unit using vector registers P and Q. These registers and an associated arithmetic unit may be used independently and simultaneously with other pipelined units provided the program being executed can use this facility.

One of the most important parts of a vector computer is a fast main memory from where data is fed to vector registers. Typically the speed of retrieval from the main memory is around 10 times slower than the speed of processing. To alleviate this speed mismatch, the main memory of vector computers is organized as a number of “banks”. Successive components of a vector are stored in successive memory banks and command for retrieval is issued to the banks successively and the outputs are stored in a buffer from where they are shifted into the vector register. If suppose the access time to a memory bank is 12 clock cycles and if there are 16 banks, it will take 12 cycles to get the first value. Soon after the first word is retrieved from a bank, command to retrieve the next value stored in it can be issued. This is overlapped with retrieval and shifting of values into the vector register so that effectively one word per clock is shifted into the vector register. This is illustrated in Fig. 4.11. Observe that the number of banks should be larger than or equal to the clock cycles needed to read a word from memory.

Organization of data in memory

Memory banks

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)	A(11)	A(12)	A(13)	A(14)	A(15)	A(16)
A(17)	A(18)	A(19)													A(32)
A(33)	A(34)														A(48)
A(49)	A(50)	A(51)													A(63) A(64)

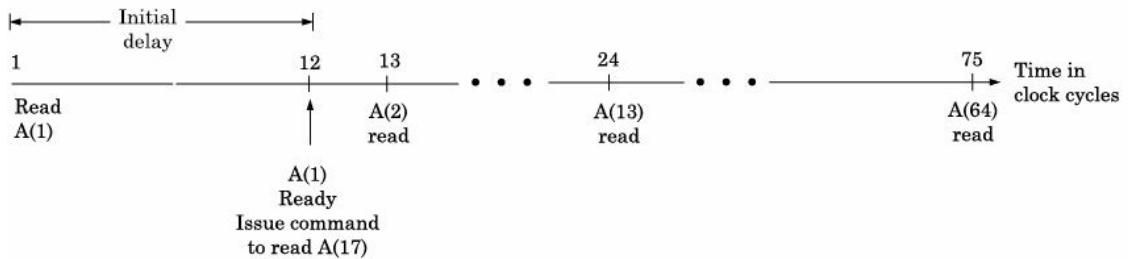


Figure 4.11 Retrieval of vector components from memory banks.

In some vector supercomputers such as Cray YMP two vectors may be read simultaneously from the memory and added. This implies that we must have multiple memory banks commensurate with the clocks needed to read a word. For example if memory access is 6 clock cycles then 16 banks would suffice if we store vectors A and B appropriately in the banks.

Another important problem in designing vector computing units is handling of *vector stride*. The stride is the subscript difference between successive elements of a vector. For example, in the loop

```

for i := 1 to 100 with step size of 5 do
    a(i) = b(i) + c(i)
end for

```

the stride is 5 as successive elements of the vectors *b* and *c* which are added are: *b*(1), *b*(6), *b*(11), ..., *c*(1), *c*(6), *c*(11) In this case the vector unit must retrieve *b*(1), *b*(5), *b*(11), ..., etc., and store them in successive slots of a vector register. If the stride equals the number of banks in the main memory retrieval of vector elements will be slowed down as successive requests for retrieval will go to the same memory bank.

Vector instructions in vector computers have features to specify a stride value in addition to a starting address. This facilitates loading into vector registers vector components separated by the value of stride.

4.4 A TYPICAL VECTOR SUPERCOMPUTER

A block diagram of the various units of a vector supercomputer such as the series of machines designed by Cray is shown in Fig. 4.12. The main logical units are:

- Input/Output processors
- Main memory
- Instruction registers, scalar registers, and vector registers
- Pipelined processors
- Secondary storage system
- Frontend computer system

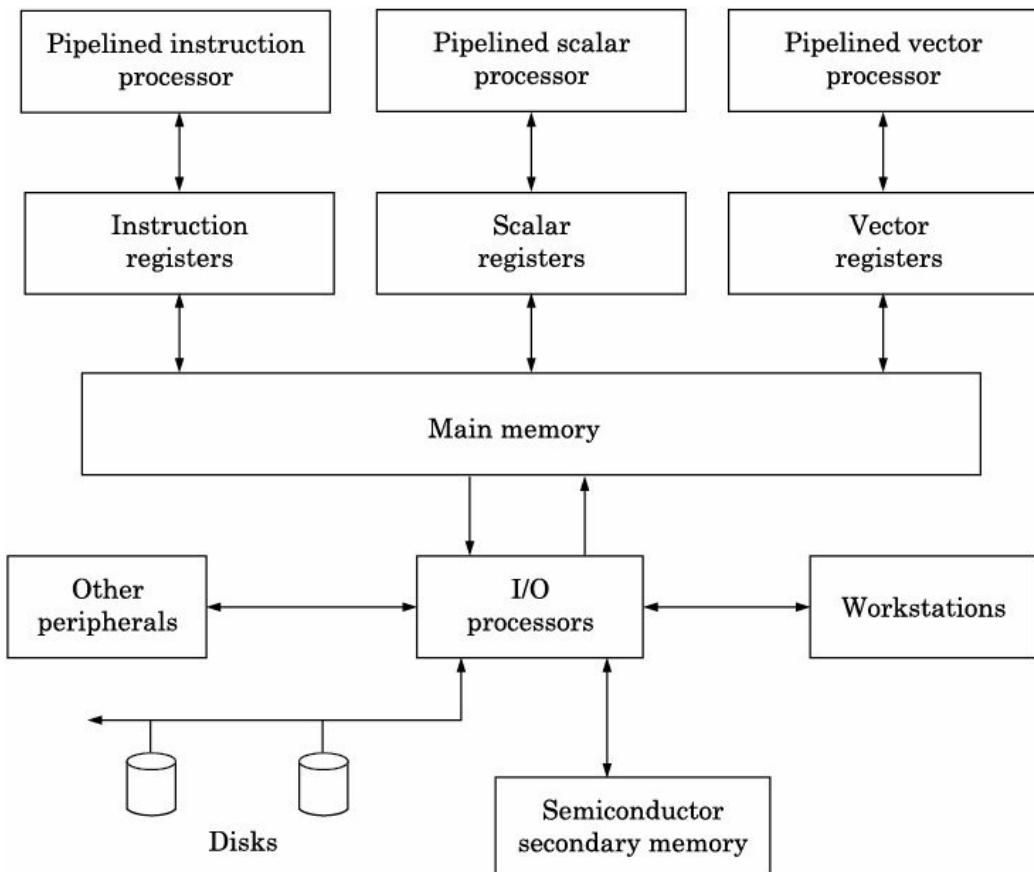


Figure 4.12 Block diagram of a typical vector supercomputer.

The most important feature of vector supercomputers is extensive use of parallelism achieved by overlapping the operations of I/O processor, instructions processor, scalar processor and the vector processor. Each of the processors, in turn, use whatever parallelism that can be exploited.

We have already seen that the main memory is organized as a set of banks from which retrieval can be overlapped to alleviate the speed mismatch between vector arithmetic unit and the memory. All vector machines also have a fast scalar unit. Scalar unit is like an ordinary ALU performing operations on individual operands. Scalar unit is also pipelined and usually has a large set of registers, typically around 128. We describe a modern vector supercomputer in the following paragraph.

NEC SX-9 [Nakazato et al., 2008] and its successor SX ACE having a similar architecture

are examples of typical modern vector computers manufactured by NEC of Japan. SX-9 was released in 2008 and SX ACE in 2013. The CPU configuration of this computer is shown in Fig. 4.13. This computer consists of a scalar unit and vector units. The scalar unit is a 64-bit RISC architecture. It has 128, 64-bit general purpose registers and a data cache and an instruction cache each storing 32 KB. The scalar unit has 6 execution pipelines; two for floating point arithmetic, two for integer arithmetic and two for memory operations. The processor is a typical superscalar out of order execution unit with speculative instruction execution. It has a powerful branch prediction system. It has a reorder buffer with 64 entries.

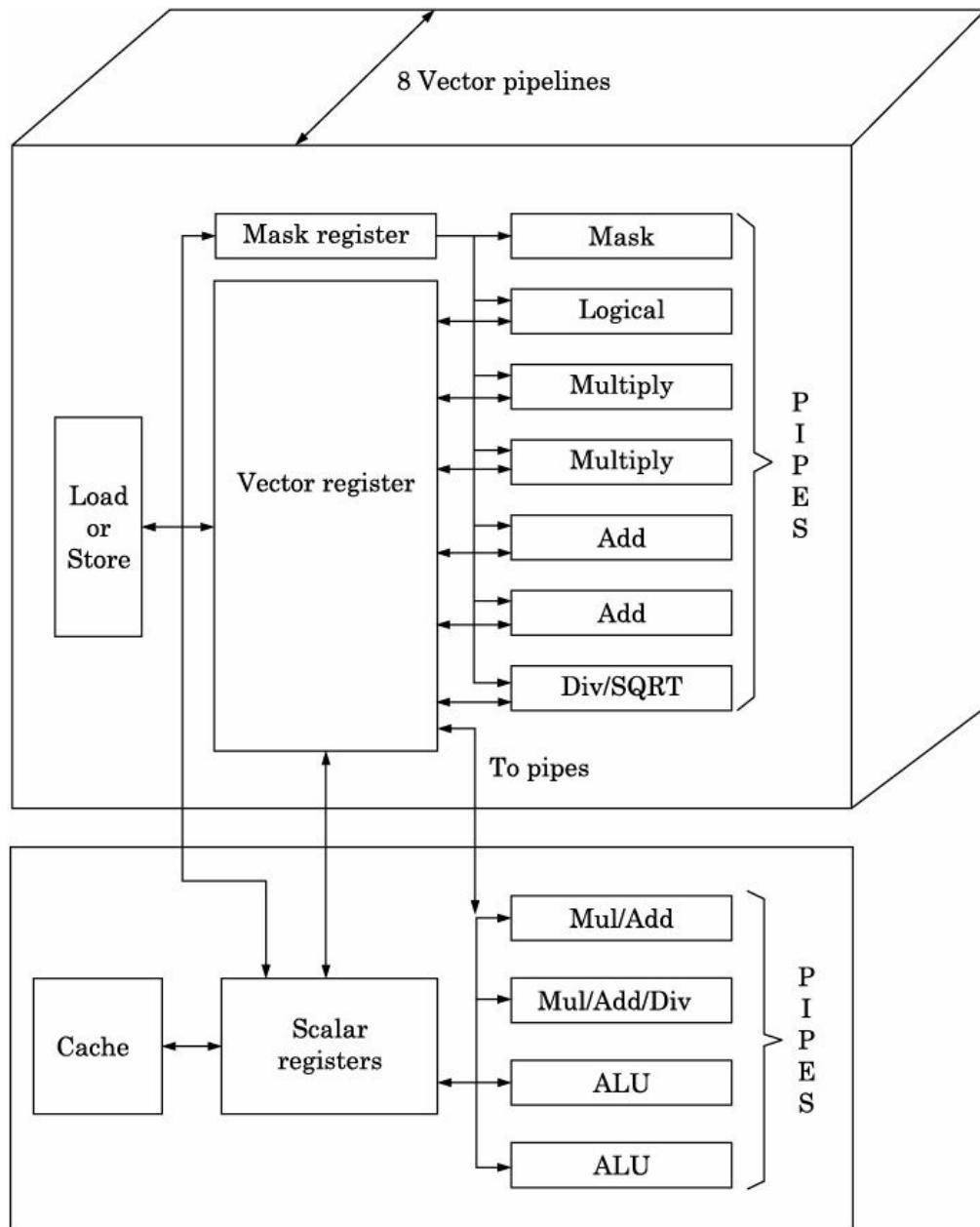


Figure 4.13 CPU configuration of NEC SX-9.

The vector unit has 72 vector registers each of which can store 256 component vectors each 64 bits long. Pipelines can operate concurrently. The pipelines are capable of performing add, multiply/divide, shift, load/store and mask. It has 16 mask registers each of which stores 64 bits. Mask registers are used to select specified components of a vector and are useful in many applications particularly with sparse matrices. There are 8 identical vector

units which can operate simultaneously. The processor has what is called Assignable Data Buffer (ADB) by NEC. ADB is used primarily to selectively buffer vector data. Data can be transferred from ADB to vector unit faster. Buffering of data reduces CPU to memory traffic. Both scalar and vector data can be stored in ADB as required by an application. The Main Memory Unit (MMU) consists of 512 memory cards each storing 2 GB for a total capacity of 1 TB. The main memory unit has 32768 ways interleaved to increase the speed of access of data by the vector processor. The memory to CPU speed is 8 GB/s per MMU card. Thus with 512 MMU cards, the maximum memory to CPU speed is 4 TB/s. Special operations known as *gather* and *scatter* on vector operands is also provided in hardware. Given a vector $a(i)$, $i = 1, 256$ and a stride of 4, a gather operation will create a vector $a(1), a(5), a(9), \dots, a(13) \dots a(253)$ and store it in a vector register. A *scatter* operation is the opposite of gather. Given a vector $b(1), b(8), b(15), \dots, b(256)$, a scatter operation will place these elements in their right slots in the vector as shown in Fig. 4.14. We will see later in Chapter 8, the use of gather and scatter operations in programming.

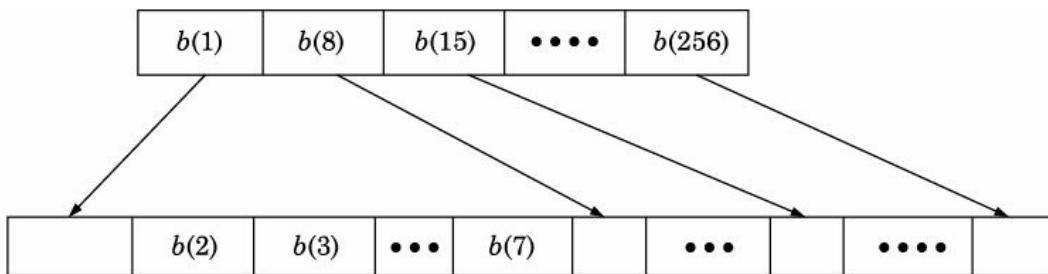


Figure 4.14 Illustrating scatter operation.

The NEC series of supercomputers, as we have described, use a large number of vector registers into which operands from main memory are read and processed. The results after processing are stored back in vector registers. This is known as *load-store* architecture. As long as the length of vectors to be processed is smaller than the length of vector registers, this works very well and speed increases linearly with vector length. When the vector length is larger than the vector register length (> 256 in NEC SX-9), it is necessary to execute one more vector instruction to get the extra elements and the speed drops by a small amount. Normally it has been found that most applications have vector lengths less than 256.

We will now compute the speed of vector computers (using a very simple model) and get an idea of optimal length of vector registers needed. Assuming a k stage pipelined arithmetic unit and a clock period of τ , the time to add two vectors of length n would be:

$$T_n = s\tau + k\tau + (n - 1)\tau = (s + k + n - 1)\tau$$

where $s\tau$ is a fixed startup time to interpret and start vector instruction. When $n \rightarrow \infty$, we get

$$T_{\text{ideal}} = n\tau$$

which is the “ideal” time for long vectors.

$$\frac{T_n}{T_{\text{ideal}}} = \frac{(s + k + n - 1)}{n}$$

If $s = 8$ and $k = 4$, we obtain

$$\frac{T_n}{T_{\text{ideal}}} = \frac{11 + n}{n}$$

If we want T_n to be 10% higher than the ideal time, we obtain

$$\frac{T_n}{T_{\text{ideal}}} = 1.1 = \frac{11 + n}{n}$$

which gives $n = 110$.

If the vector length n is increased to 256, the vector unit will take only 1% more time than the ideal time.

In this section we looked at the architecture of vector computers which were the fastest parallel computers in the 70s up to late 80s. The main advantage of vector computers is that a single vector instruction does a great deal of work. For example a single vector add multiply instruction on 256 components vector needs only one instruction whereas in a scalar machine we need multiple loop instructions with a large number of branches. Number of accesses to memory is also reduced. Thus, instruction caches are not required in vector machines. The main disadvantage of vector computers is their specialized nature as vectorizable problems are needed to use them effectively. They are considered niche products for high speed scientific computing. This is, however, changing as many new applications of vector processors are being discovered particularly in image and video processing. Besides this vector processing can also be used in cryptography, sound processing, and data mining. Thus special purpose integrated circuit chips are being designed for these applications. NEC is one of the few companies which persisted with vector architecture for general purpose computing and designed a series of machines called SX series. NEC changed the technology used with the improvements in semiconductor technology and reduced the power consumption and cooling requirements. Currently their model SX-ACE incorporates four vector processors in one chip. Multiple chips are interconnected to architect a parallel vector supercomputer.

4.5 ARRAY PROCESSORS

A vector computer adds two vectors $(a_1, a_2, a_3, \dots, a_n)$ and $(b_1, b_2, b_3, \dots, b_n)$ by streaming them through a pipelined arithmetic unit. Another method of adding vectors would be to have an array of n PEs, where each PE stores a pair of operands (see Fig. 4.15). An add instruction is broadcast to all PEs and they add the operands stored in them simultaneously. If each PE takes T units of time to add, the vector will also be added in T units as all the PEs work simultaneously. Such an organization of PEs is known as an *array processor*. An array processor uses data parallelism. In this example a single instruction, namely ADD, is performed simultaneously on a set of data.

This is an example of a Single Instruction Multiple Data (SIMD) architecture [Hord, 1990].

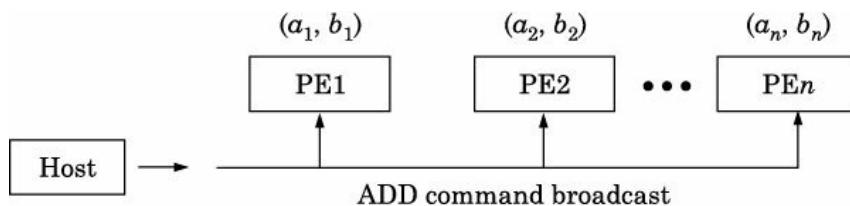


Figure 4.15 An array processor to add a vector.

An array processor will have in general, capability to perform all the arithmetic operations. It is attached to a host computer. The data to be processed is stored in the local memory of each of the PEs in the array. A sequence of instructions to process the data is broadcast to all the PEs by the host. All the PEs independently and simultaneously execute the instructions and process the data stored in each of them. If instead of broadcasting the instructions from the host, these instructions are stored in the private instruction memory of each of the PEs (making them full fledged computing elements), the host has to merely issue a start command and all the CEs would start computing simultaneously and asynchronously leading to faster computation. This model of parallel processing is called Single Program Multiple Data (SPMD) processing.

The SPMD architecture is similar to method 2 of organizing teachers to grade papers explained in Chapter 2. The variations of this method (methods 4, 5 and 6) may also be used with this architecture.

Many special purpose SIMD array processors were built during 70s and 80s for solving many numerically intensive problems which had regular structure. Currently they are extensively used in graphics processing. We will describe them in the next chapter.

4.6 SYSTOLIC ARRAY PROCESSORS

The general idea of pipeline processing is used in building special purpose high speed Very Large Scale Integrated (VLSI) circuits to perform various mathematical operations such as matrix multiplications and Fourier transformation. Such circuits are known as *systolic arrays* [Kung, 1982]. The word systolic is derived from systole, which means rhythmical contraction, especially of the heart. The primary characteristics of a systolic array processor are:

1. The system consists of a set of interconnected cells, each capable of performing a specified operation. The cells and operations performed by them are usually identical. The time taken for processing by each of the cells is identical.
2. Individual cells are connected only to their nearest neighbours.
3. Flow of data is rhythmic and regular.
4. Data can flow in more than one dimension.
5. The communication between cells is serial and periodic.
6. The cells, except those at the boundary of the array, do not communicate with the outside world.

In Fig. 4.16, we have given an example of a systolic array which replaces a sequence of values $(x_{n+2}, x_{n+1}, \dots, x_3)$ in a memory with a new sequence of values $(y_n, y_{n-2}, y_{n-1}, \dots, y_3)$ where

$$\begin{aligned} y_1 &\leftarrow w_1 x_3 + w_2 x_2 + w_3 x_1 \\ y_2 &\leftarrow w_1 x_4 + w_2 x_3 + w_3 x_2 \\ y_3 &\leftarrow w_1 x_5 + w_2 x_4 + w_3 x_3 \\ y_2 &\leftarrow w_1 x_4 + w_2 x_3 + w_3 x_2 \\ &\dots && \dots && \dots \\ y_n &\leftarrow w_1 x_{n+2} + w_2 x_{n+1} + w_3 x_n \end{aligned}$$

Observe that the contents of the serial memory are shifted one position to the right and pushed into the cells at each clock interval. The most significant bit position of the serial memory which is vacated when the contents are shifted right is filled up by the value transformed by the cells.

The details of the transformation rules are shown in Fig. 4.16. Complex systolic arrays have been built for applications primarily in signal processing.

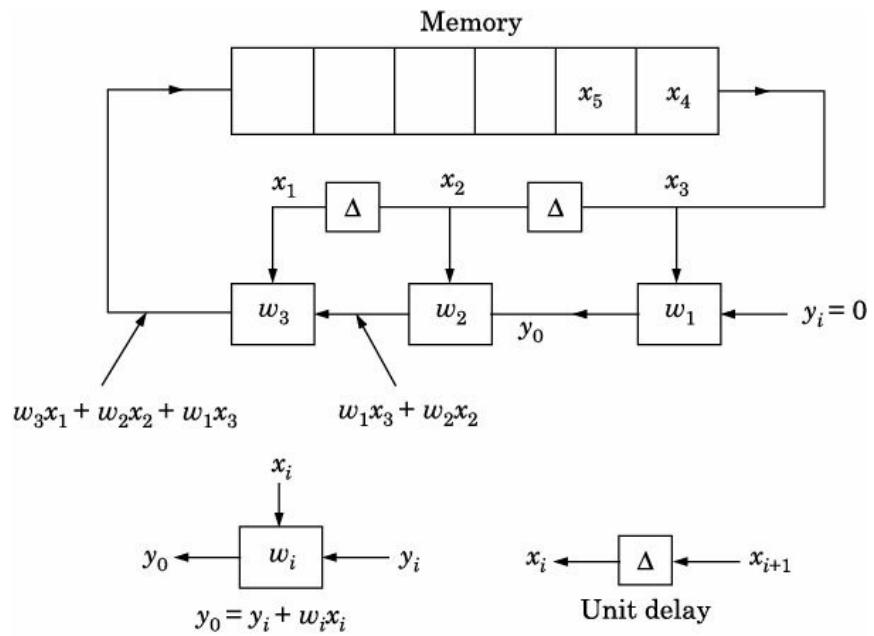


Figure 4.16 A systolic sequence generator.

4.7 SHARED MEMORY PARALLEL COMPUTERS

One of the most common parallel architectures using moderate number of processors (4 to 32) is a shared memory multiprocessor [Culler, Singh and Gupta, 1999]. This architecture provides a global address space for writing parallel programs. Each processor has a private cache. The processors are connected to the main memory either using a shared bus or using an interconnection network. In both these cases the average access time to the main memory from any processor is same. Thus, this architecture is also known as *Symmetric Multiprocessor* abbreviated SMP. This architecture is popular as it is easy to program it due to the availability of globally addressed main memory where the program and all data are stored. A shared bus parallel machine is also inexpensive and easy to expand by adding more processors. Thus, many systems use this architecture now and many future systems are expected to use this architecture. In this section we will first discuss this architecture.

4.7.1 Synchronization of Processes in Shared Memory Computers

In a shared memory parallel computer, a common program and data are stored in the main memory shared by all PEs. Each PE can, however, be assigned a different part of the program stored in the memory to execute with data also stored in specified locations. The main program creates separate processes for each PE and allocates them along with the information on the locations where data are stored for each process. Each PE computes independently. When all PEs finish their assigned tasks, they have to rejoin the main program. The main program will execute after *all* processes created by it have been finished. Statements are added in a programming language to enable creation of processes and for waiting for them to complete. Two statements used for this purpose are:

1. *fork*: to create a process and
2. *join*: when the invoking process needs the results of the invoked process(es) to continue processing.

For example, consider the following statements:

Process X;	Process Y;
:	:
fork Y;	:
:	:
join Y;	end Y;

Process X when it encounters *fork Y* invokes another Process Y. After invoking Process Y, it continues doing its work. The invoked Process Y starts executing concurrently in another processor. When Process X reaches *join Y* statement, it waits till Process Y terminates. If Y terminates earlier, then X does not have to wait and will continue after executing *join Y*.

When multiple processes work concurrently and update data stored in a common memory shared by them, special care should be taken to ensure that a shared variable value is not initialized or updated independently and simultaneously by these processes. The following example illustrates this problem. Assume that $\text{Sum} \leftarrow \text{Sum} + f(A) + f(B)$ is to be computed and the following program is written:

Process A : : fork B : : $\text{Sum} \leftarrow \text{Sum} + f(A)$: join B : end A	Process B : $\text{Sum} \leftarrow \text{Sum} + f(B)$: end B
---	---

Suppose Process A loads Sum in its register to add $f(A)$ to it. Before the result is stored back in main memory by Process A, if Process B also loads Sum in its local register to add $f(B)$ to it. Process A will have $\text{Sum} + f(A)$ and Process B will have $\text{Sum} + f(B)$ in their respective local registers. Now, both Processes A and B will store the result back in Sum. Depending on which process stores Sum last, the value in the main memory will be either $\text{Sum} + f(A)$ or $\text{Sum} + f(B)$ whereas what was intended was to store $\text{Sum} + f(A) + f(B)$ as the result in the main memory. If Process A stores $\text{Sum} + f(A)$ first in Sum and then Process B takes this result and adds $f(B)$ to it then the answer will be corrected. Thus, we have to ensure that only one process updates a shared variable at a time. This is done by using a statement called *lock* <variable name>. If a process locks a variable name, no other process can access it till it is *unlocked* by the process which locked it. This method ensures that only one process is able to update a variable at a time.

The updating program can thus be rewritten as:

Process A : fork B; : lock Sum; $\text{Sum} \leftarrow \text{Sum} + f(A);$ unlock Sum; : join B; : end A.	Process B : lock Sum; $\text{Sum} \leftarrow \text{Sum} + f(B);$ unlock Sum; : end B.
---	---

In the above case whichever process reaches *lock* Sum statement first will lock the variable Sum disallowing any other process from accessing it. It will *unlock* Sum after updating it. Any other process wanting to update Sum will now have access to it. That process will lock Sum and then update it. Thus, updating a shared variable is serialized. This ensures that the correct value is stored in the shared variable.

In order to correctly implement lock and unlock operations used by the software, we require hardware assistance. Let us first assume there is no hardware assistance. The two assembly language programs given below attempt to implement *lock* and *unlock* respectively in SMAC2P, the hypothetical computer we described in the last chapter [Culler, Singh and Gupta, 1999].

lock:	LD	R1, 0, L	/* C(R1) \leftarrow C(L) */
	CMP	R1, #0	/* Compare R1 with 0 */
	BNZ	lock	/* If R1 \neq 0 try again */
	ST	L, # 1	/* L \leftarrow 1 */
	RET		/* RETURN */
unlock:	ST	L, # 0	/* Store 0 in L */
	RET		

In the above two programs L is the lock variable and #0 and #1 immediate operands. A locking program locks by setting the variable L = 1 and unlocks by setting L = 0. A process trying to obtain the lock should check if L = 0 (i.e., if it is unlocked), enter the critical section and immediately lock the critical section by setting L = 1 thereby making the lock busy. If a process finds L = 1, it knows that the lock is closed and will wait for it to become 0 (i.e., unlocked). The first program above implements a “busy-wait” loop which goes on looping until the value of L is changed by some other process to 0. It now knows it is unlocked, enters a critical section locking it to other processes. After completing its work, it will use the unlock routine to allow another process to capture the lock variable. The lock program given above looks fine but will not work correctly in actual practice due to the following reason. Suppose L = 0 originally and two processes P₀ and P₁ execute the above lock code. P₀ reads L and finds it to be 0 and passes BNZ statement and in the next instruction will set L = 1. However, if before P₀ sets L = 1, if P₁ reads L it will also think L = 0 and assume the lock is open! This problem has arisen because the sequence of instructions: reading L, testing it to see if it is 0 and changing it to 1 are not *atomic*. In other words, they are separate instructions and another process is free to carry out any instruction in between these instructions. What is required is an *atomic instruction* which will load L into register and store 1 in it. With this instruction we can rewrite the assembly codes for lock and unlock as follows:

lock:	TST	R1, 0, L	/* C(R1) \leftarrow L; L \leftarrow 1 */
	CMP	R1, #0	/* Compare R1 with 0 */
	BNZ	lock	/* If R1 \neq 0 try again */
unlock:	ST	L, # 0	/* L \leftarrow 0 */
	RET		

The new instruction TST is called *Test and Set* in the literature even though a better name for it would be load and set. It loads the value found in L in the register R1 and sets the value of L to 1.

Observe that if L = 1, the first program will go on looping keeping L at 1. If L at sometime is set to 0 by another process, R1 will be set to 0, capturing the value of L and the procedure will return after making L = 1, thereby implementing locking. Observe that another process cannot come in between and disturb locking as test and set (TST) is an atomic instruction and is carried out as a single indivisible operation.

Test and set is a representative of an *atomic read-modify-write* instruction in the instruction set of a processor which one intends to use as a processor of a parallel computer. This instruction stores the contents of a location “L” (in memory) in a processor register and stores another value in “L”. Another important primitive called *barrier* which ensures that all processes complete specified jobs before proceeding further also requires such atomic read-modify-write instruction to be implemented correctly. Implementing barrier synchronization using an atomic read-modify write instruction is left as an exercise to the reader.

An important concept in executing multiple processes concurrently is known as *sequential consistency*. Lamport [1979] defines sequential consistency as:

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operation of all processors were executed in some sequential order and the operations of each individual processor occurs in this sequence in the order specified by its program”.

In order to ensure this in hardware each processor must appear to *issue* and *complete* memory operations one at a time *atomically* in program order. We will see in succeeding section that in a shared memory computer with cache coherence protocol, sequential consistency is ensured.

4.7.2 Shared Bus Architecture

A block diagram of a shared bus parallel computer is shown in Fig. 4.17. Observe that each PE has a private cache memory. The private cache memory has two functions. One is to reduce the access time to data and program. The second function is to reduce the need for all processors to access the main memory using the bus. Thus, bus traffic is reduced during program execution. We will illustrate this with a simple example.

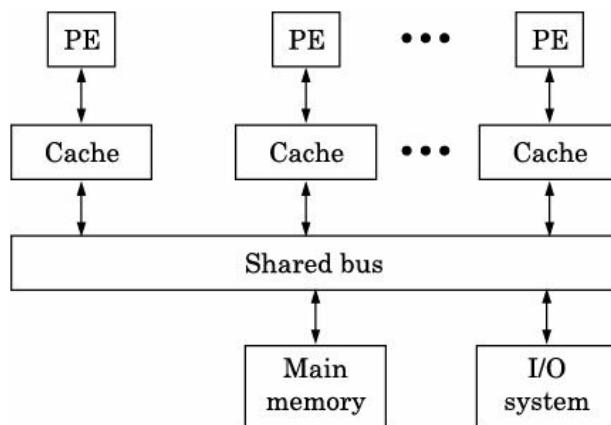


Figure 4.17 A shared memory parallel computer (number of PEs between 4 and 32).

EXAMPLE 4.1

Consider a shared bus parallel computer built using 32-bit RISC processors running at 2 GHz which carry out one instruction per clock cycle. Assume that 15% of the instructions are loads and 10% are stores. Assume 0.95 hit rate to cache for read and write through caches. The bandwidth of the bus is given as 20 GB/s.

1. How many processors can the bus support without getting saturated?
2. If caches are not there how many processors can the bus support assuming the main memory is as fast as the cache?

Solution

Assuming each processor has a cache.

Number of transactions to main memory/s = Number of read transactions + Number of write transactions

$$\begin{aligned}
 &= 0.15 \times (0.05) \times 2 \times 10^9 + 0.10 \times 2 \times 10^9 \\
 &= (0.015 + 0.2)10^9 = 0.215 \times 10^9
 \end{aligned}$$

Bus bandwidth = 20×10^9 bytes/s.

Average number of bytes traffic to memory = $4 \times 0.215 \times 10^9 = 0.86 \times 10^9$

\therefore Number of processors which can be supported

$$= (20 \times 10^9) / (0.86 \times 10^9) = 23$$

If no caches are present, then every load and store instruction requires main memory access.

Average number of bytes traffic to memory

$$= 2 \times 4 \times 0.25 \times 10^9 = 150 \text{ MB/s}$$

\therefore Number of processors which can be supported

$$= (20 \times 10^9) / (2 \times 10^9) = 10$$

This example illustrates the fact that use of caches allows more processors on the bus even if the main memory is fast because the shared bus becomes a bottleneck.

The use of caches is essential but they bring with them a problem known as *cache coherence* problem. The problem arises because when a PE writes a data, say 8, into its private cache in address x , for example, it is not known to the caches of other PEs. If another PE reads data from the address x of its cache, it will read a data stored in x earlier which may be, say 6. It is essential to keep the data in a given address x same in all the caches to avoid errors in computation. There are many protocols (or rules) to ensure coherence. We will describe in the next sub-section a simple protocol for bus based systems. Before describing it, we quickly review the protocol used to read and write in a single processor system.

4.7.3 Cache Coherence in Shared Bus Multiprocessor

In a single processor system, the following actions take place when read/write requests are initiated by the processor. If there is a read request and if the block containing the data is in the cache (called a Read-hit), it is delivered to the processor. If the block with the data is not in the cache (called a Read-miss), the block containing the data in the main memory replaces a block in the cache and the requested data is supplied. The block to be replaced is based on the block replacement policy.

If it is a write request and the block with the data is in the cache (Write-hit), it overwrites the existing data in the relevant block in the cache. If the protocol is *write-now* protocol (known in the literature as *write-through*), the data in the main memory block is also updated. If the protocol is *write-later* (known in the literature as *write-back*) then the data is written back in the memory only when the cache block in which the data is contained is to be replaced by another block from the main memory. The cache block contains, besides the data, a tag which indicates the cache block address, and data address within the block. It also has a bit (called a *dirty bit*) which is set to 1 when a new data overwrites the data originally read from the memory. Only when a cache block is “dirty” it should be written back in the main memory when the block is replaced. When a write instruction (request) is issued and the cache block with the desired data is not in the cache then it is a Write-miss. In this case the desired cache block is retrieved from the main memory replacing one of the blocks in the cache and the data is written in the desired word. If the block which is replaced is dirty then it is written in the main memory.

Why does a cache Read, Write-miss happen? There are three possible reasons. They are:

1. *Cold miss* which happens when cache is not yet loaded with the blocks in the working set. This happens at the start of operation and is inevitable.
2. *Capacity miss* which happens when the cache is too small to accommodate the working set. The only way to prevent this is to have a large cache which may not

- always be possible.
3. *Conflict miss* which happens when multiple cache blocks contend for the same set in a set associative cache. This is difficult to prevent.

These three types of misses are commonly called the 3C misses in single processor systems.

The situation in a multiprocessor is more complicated because there are many caches, one per processor, and one has to know the status of each block of the cache in each PE to ensure coherence. A bus based system has the advantage that a transaction involving a cache block may be broadcast on the bus and other caches can listen to the broadcast. Thus, cache coherence protocols are based on the cache controller of each cache receiving Read/Write instructions from the PE to which it is connected and also listening (called snooping which means secretly listening) to the broadcast on the bus initiated when there is a Read/Write request by any one of the PEs and taking appropriate action. Thus, these protocols are known as *snoopy cache protocols*. In this subsection, we will describe one simple protocol which uses only the information whether a cache block is valid or not. Details of cache coherence protocols are discussed by Sorin, Mill and Wood [2011]. A cache block will be invalid if that block has new data written in it by any other PE. Many protocols are possible as there are trade offs between speed of reading/writing the requested data, bandwidth available on the bus, and the complexity of the cache and the memory controllers. In the protocol, we describe below the cache controller is very simple.

The actions to be taken to maintain cache coherence depend on whether the command is to read data from memory to a register in the processor or to write the contents of a register in the processor to a location in memory. Let us first consider a Read instruction (request).

Read request for a variable stored in the cache of PE_k

The following cases can occur:

Case 1: If the block containing the address of the variable is in PE_k 's cache and it is valid (a bit associated with the block should indicate this), it is retrieved from the cache and sent to PE_k .

Case 2: If the block containing the requested address is in PE_k 's cache but is not valid then it cannot be taken from PE_k 's cache and there is a Read-miss. (This is a Read-miss which is due to lack of cache coherence and is called *Coherence miss*). The request is broadcast on the bus and if the block is found in some other PE's cache and is valid, the block containing the data is moved to PE_k 's cache. The data is read from PE_k 's cache. (It should be remembered that at least one PE must have a valid block).

Case 3: If the block containing the requested address is not in PE_k 's cache. The request is broadcast on the bus and if a block containing the requested data is found in any other PE's cache and is valid, the block is moved to PE_k 's cache and the data is read from there. A block in PE_k is replaced and if it is dirty, it is written in the main memory.

Case 4: If the block containing the requested address is not in PE_k 's cache. The request is broadcast on the bus and if the block is not found in any other PE's cache. The block with the required data is read from the main memory and placed in PE_k 's cache replacing a block and the data is read from there. If the block replaced is dirty, it is written in the main memory.

Table 4.1(a) summarises the above cases.

TABLE 4.1(a) Decision Table for Maintaining Cache Coherence in a Bus Based Multiprocessor:
Reading from cache

Conditions	Cases →			
	1	2	3	4
Block with data in own cache?	Y	Y	N	N
Is block valid?	Y	N	—	—
Block with valid data in any other cache?	—	Y	Y	N
Actions				
Move valid block to own cache overwriting own block	—	X	—	—
Move valid block to own cache replacing a block	—	—	X	—
Move valid block from main memory replacing a block in cache	—	—	—	X
If replaced cache block is in modified state write it in main memory	—	—	X	X
Read from own cache block	X	X	X	X
Actions with “—” against them ignored				
Actions with “X” against them carried out				

Write request to cache block in PE_k

Case 1: Write request from PE_k to write data (from specified register) to an address in memory. If the block with this address is in PE_k’s cache, the data overwrites the existing one. A broadcast is sent to all other caches to invalidate their copies if they have the specified block. This protocol is called *write invalidate*.

PE_k can also broadcast the block containing the new value and it can be copied into all caches which have this block. This protocol is called *write-update* protocol in the literature. This is, however, very slow and not normally used.

If write through policy is used then the updated block is written in the main memory, else the new value is written only when a block is replaced in the cache. We have assumed write back policy in our simple protocol.

Case 2: Write request to PE_k’s cache but the block is not in PE_k’s cache. In this case, the block in the main memory is updated and retrieved from the main memory into PE_k’s cache. This block replaces an appropriate cache block based on block replacement policy. If the block being replaced is dirty, it is written in the main memory. An invalidate signal is broadcast to enable all other caches containing this block to invalidate their copies.

This protocol is summarized in Table 4.1(b).

TABLE 4.1(b) Decision Table for Write Operation

Is the block in own cache?	Y	N
Actions		
Move block from main memory to the cache replacing a block	—	X
Write data in block	X	X

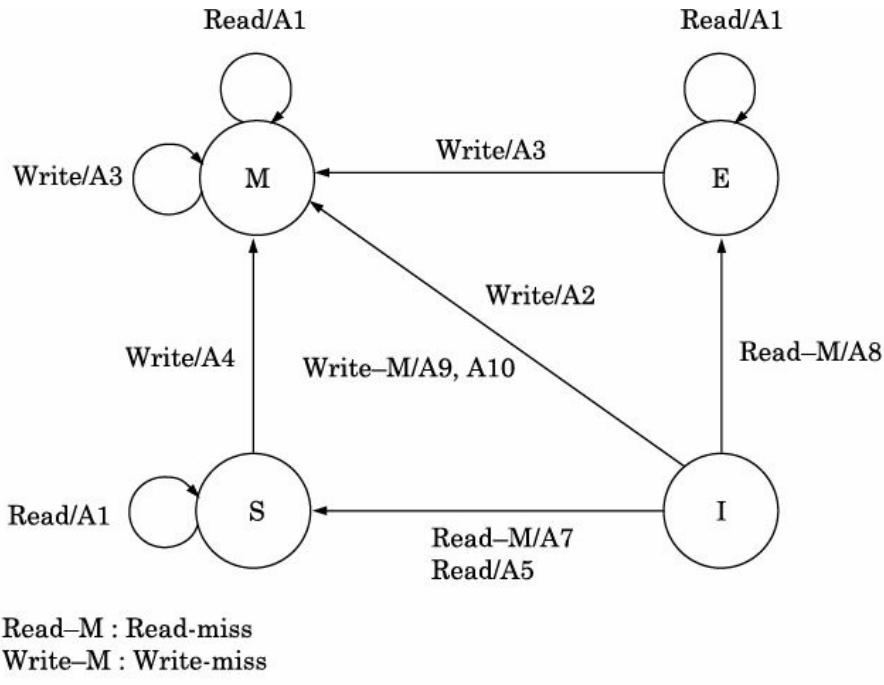
Send invalid signal to blocks shared with it in other caches	X	X
If block replaced is in modified state write it in main memory	—	X

As we pointed out earlier, many variations to this protocol are possible. For instance in case 2 (see Table 4.1(a)), this protocol moves a valid cache block and replaces the cache block in PE_k with it. Instead the protocol may have moved the valid block to the main memory and then moved it to PE_k 's cache. This would increase the read delay and may be inevitable if blocks cannot be moved from one cache to another via the bus.

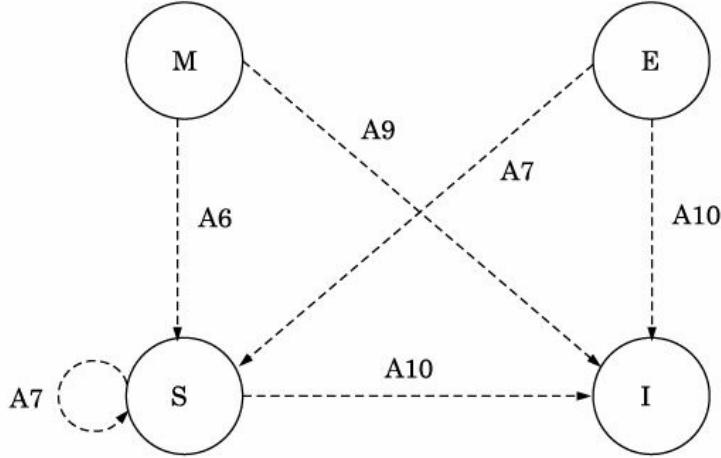
4.7.4 MESI Cache Coherence Protocol

In the last subsection we presented a cache coherence protocol in the form of a decision table. In the literature a state diagram representation is used and we present it for completeness taking as an example, the method proposed by Intel [Hennessy and Patterson, 2012]. This cache coherence protocol is known as the MESI protocol. The expansion of MESI is given in Table 4.2. A state transition diagram for this protocol is given in Fig. 4.18. This protocol invalidates the shared blocks in caches when new data is written in that block by any PE. (It is a write-invalidate protocol). When new data is written in any cache block, it is not written immediately in the main memory. In other words it is a write-back protocol to main memory. The protocol defines a set of states in which cache blocks may be found. These are defined in Table 4.2.

TABLE 4.2 MESI Protocol States	
<i>Cache block state</i>	<i>Explanation of state</i>
M = Modified	The data in cache block has been modified and it is the only valid copy. Main memory copy is an old copy. (Dirty bit set in cache block)
E = Exclusive	The data in cache block is valid and is the same as in the main memory. No other cache has this copy.
S = Shared	The data in cache block is valid and is the same as in the main memory. Some other caches may also have valid copies.
I = Invalid	The data in cache block has been invalidated as another PE has the same cache block with a newly written value.



(a) State transition diagram of the cache block caused by a PE initiating a Read/Write transaction



(b) State transition diagram of other PEs' cache block state caused by broadcast on the bus by the initiating PE

Figure 4.18 State transition diagram for MESI protocol.

Referring to Fig. 4.18, observe that the protocol is explained using two state transition diagrams marked (a) and (b). In Fig. 4.18, the circles represent the state of a cache block in the processor initiating action to Read or Write. The solid line shows the transition of the cache block state after Read/Write. The action taken by the processor is also shown as A1, A2 etc. Referring to Fig. 4.18(a) when a cache block is in state S and a write command is given by a PE, the cache block transitions, to state M and A4 is the control action initiated by the system.

In Fig. 4.18(b) we show the state transitions of the corresponding cache blocks in other PEs which respond to the broadcast by the initiating PE. We show these transitions using dashed lines and label them with the actions shown in Fig. 4.18(a).

A1: Read data into processor register from the block in own cache.

A2: Copy modified, shared or exclusive copy into own cache block. Write new data into own cache block. Invalidate copies of block in other caches.

A3: Write data into cache block.

A4: Broadcast intent to write on bus. Mark all blocks in caches with shared copy of block invalid. Write data in own cache block.

A5: Broadcast request for valid copy of block from another cache. Replace copy of block in cache with valid copy. Read data from own cache block.

The above actions are specified when the required data is found in the cache. If it is not found in the cache, namely, Read-miss or Write-miss, the following actions are taken:

Read-miss actions:

A6: A command to read the required block is broadcast. If only other cache has this block and it is in modified state, that copy is stored in the main memory. This block replaces a block in the cache. The initiating and supplying caches go to shared state. Data is read from the supplied block. If the replaced block is in modified state, it is written in the main memory.

A7: A command to read the required block is broadcast. If any other processor's cache has this block in exclusive or shared state, it supplies the block to the requesting processor's cache. The cache block status of requesting and supplying processors go to shared state. Data is read from the supplied block. If the replaced block is in modified state, it is written in the main memory.

A8: A command to read the required block is broadcast. If no processor's cache has this block, it is read from the main memory to the requesting processor's cache replacing a block. If the block which is replaced is in modified state, it is written in the main memory. The processor's cache block goes to the exclusive state. Data is read from the replaced block.

Write-miss actions:

A9: Intent to write is broadcast. If any cache has the requested block in modified state, it is written in the main memory and supplied to the requesting processor. Data is written in the supplied block. The state of requesting cache block is set as modified. The other processor's cache block is invalidated. If the block which is replaced is in modified state, it is written in main memory.

A10: Intent to write is broadcast. If another cache has the requested block in shared or exclusive state, the block is read from it and replaces a block. Data is written in it. Other copies are invalidated. The requesting cache block will be in modified state. If the block which is replaced is in modified state, it is written in main memory.

A11: Intent to write is broadcast. If no cache has the requested block, it is read from the main memory and replaces a block in the requesting cache. Data is written in this block and it is placed in exclusive state. If the block replaced is in modified state, it is written in the main memory.

We have described the MESI protocol using a state diagram. We may also use decision tables to explain the protocol. We have decision tables for the MESI protocol in Table 4.3.

TABLE 4.3 Decision Tables for MESI Protocol

Case 1: Read data. The required block with data is in own cache or in some other cache				
Current state	M	E	S	I
Actions				
Broadcast request for valid block from another cache	—	—	—	X

Replace block in cache with valid block	—	—	—	X
Read data from block in own cache	X	X	X	X
Next state	M	E	S	S
	A1	A1	A1	A5

Case 2: Write data in block. The required block is in own cache or in some other cache

Current state	M	E	S	I
Copy block in M, S or E state into own cache	—	—	—	X
Invalidate blocks in other caches which have this block	—	—	X	X
Write data in own cache block	X	X	X	X
Next state	M	M	M	M
Actions	A3	A3	A4	A2

Case 3: Read data. The required block is not in own cache (Read Miss)

Is block with data in any other cache?	Y	Y	Y	N
Current state of block in supplying cache	E	S	M	—
Actions				
Broadcast for block if in any other cache and replace a block.	X	X	X	—
Read block from main memory replacing a block.	—	—	—	X
Read data from block supplied by the other cache or memory	X	X	X	X
If replaced block is in modified state write it in main memory	X	X	X	X
Next state of supplying cache block	S	S	S	—
Next state of receiving cache block	S	S	S	E
	A7	A7	A6	A8

Case 4: Write data. The required block is not in own cache (Write-miss)

Is block in any other cache?	Y	Y	Y	N
Current state of supplying cache block	M	S	E	—
Replace cache block with that supplied.	X	X	X	—
As supplied block is in modified state write it in main memory	X	—	—	—
If replaced block is in modified state write it in main memory	X	X	X	—
Read data block from main memory and place it in requesting cache replacing a block.	—	—	—	X
If the replaced block in modified state write it in main memory				
Write data in cache block	X	X	X	X

Next state of supplying cache block	I	I	I	—
State of cache receiving cache block	M	M	M	E
Actions	A9	A10	A10	A11

EXAMPLE 4.2

A 4-processor system shares a common memory connected to a bus. Each PE has a cache with block size of 64 bytes. The main memory size is 16 MB and the cache size is 16 KB. Word size of the machine is 4 bytes/word. Assume that cache invalidate command is 1 byte long. The following sequence of instructions are carried out. Assume blocks containing these addresses are initially in all 4 caches. We use hexadecimal notation to represent addresses.

P1: Store R1 in AFFFFF

P2: Store R2 in AFFEFF

P3: Store R3 in BFFFFF

P4: Store R4 in BFFEFF

P1: Load R1 from BFFEFFF

P2: Load R2 from BFFFFF

P3: Load R3 from AFFEFF

P4: Load R4 from AFFFFF

(i) If write invalidate protocol is used, what is the bus traffic?

(ii) If write update protocol is used, what is the bus traffic?

Solution

(i) *Write-invalidate protocol*

Writes: 4 invalidate messages for the 4 write transactions. Bus traffic = $(1 + 3) \times 4$
 $= 16$ bytes
 (1 byte invalidate command, 3 byte address)

Reads: For all 4 reads valid data is in another cache

$$\begin{aligned} \text{Bus traffic} &= 4 \times (\text{Request} + \text{Response}) \\ &= 4 \times (4 + 64) = 272 \text{ bytes} \end{aligned}$$

4 bytes for request transaction and 64 bytes for response as the whole block is retrieved. Total bus traffic = $(16 + 272) = 288$ bytes

(ii) *Write-update protocol*

Writes: $4 \times (\text{update command} + \text{address} + \text{data} + \text{block replacement in main memory})$
 $= 4 \times (1 + 3 + 4 + 64) = 288$

Reads: No bus traffic as all caches have current copy.

Total bus traffic = 288 bytes.

This example *does not* illustrate that write invalidate protocol and write-update protocol are identical. A different pattern of read/write will make write-update protocol better, for instance if one processor writes many times to a location in memory and it is read only once by another processor (see Exercise 4.23).

4.7.5 MOESI Protocol

Another cache coherence protocol called MOESI protocol was proposed by AMD, another microprocessor manufacturer and a rival of Intel. This protocol introduced a new state called the *owned state* [Hennessy and Patterson, 2012]. The main motivation for introducing the “owned” state is to delay writing a modified block in the main memory which is beneficial in

the new generation of on chip multiprocessors where the time to write data in main memory is much larger than the time to transfer blocks among caches on chip. The explanation of the states of the cache blocks is given in Table 4.4.

Observe the change in the definition of the shared state. As opposed to MESI protocol definition, a block in the Shared state may be different from that in the main memory. We leave the development of state transition diagram for this protocol as an exercise to the reader.

TABLE 4.4 MOESI Protocol States	
<i>Cache block state</i>	<i>Explanation of state</i>
M = Modified	The data in the cache block has been modified and it is the only valid copy. The main memory copy is an old copy. (Dirty bit set in cache block).
O = Owned	The data in the cache block is valid. Some other caches may also have valid copies. However, the block in this state has exclusive right to make changes to it and must broadcast the change to all other caches sharing this block.
E = Exclusive	The data in cache block is valid and is the same as in the main memory. No other cache has this copy.
S = Shared	The data in the cache block is valid and may not be the same as in the main memory. Some other caches may also have valid copies.
I = Invalid	The data in cache block has been invalidated as another cache block has a newly written value.

Shared Bus Parallel Computers

Many multiprocessors used a four processor shared memory parallel computer manufactured by Intel Pentium pro as the building block in the late 90s. These are now obsolete as single chip shared memory processors are now available. We will describe these in Chapter 5.

4.7.6 Memory Consistency Models

In a shared memory multiprocessor, each PE may independently read or write to a single shared address space. A memory consistency model for such a system defines the behaviour of the memory system visible to a programmer which is guaranteed by the architect of the computer system. Consistency definition specifies the rules about reads and writes and how they alter the state of the memory. As part of supporting a memory consistency model, many multiprocessors also specify cache coherence protocols that ensure that multiple cached copies of data in a specified location in memory are all up to date. There are several memory consistency models. A detailed discussion of these models is outside the scope of this book. A tutorial by Adve and Gharachorloo [1996] and a book by Sorin, Hill and Wood [2011] give a detailed account of memory consistency models and the interested reader should refer to them. In this subsection we describe some of the commonly used consistency models.

The semantics of memory operations in a shared memory system should ensure ease of use of the system similar to serial semantics in single processor systems. In single processor systems, a programmer can assume that the sequential order of read and write to memory which he/she wrote in his/her program is observed even though the hardware may re-order instructions. Can such an assumption be made in a multiprocessor? The answer to this

question was formalized by Lamport [1979] by defining *sequential consistency* as:

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order and the operation of each individual processor occurs in this sequence in the order specified by its program”.

In other words, Lamport’s definition implies that the order of execution of statements in a program implicitly assumed by a programmer is maintained. Consider the following parallel program executed by PE1 and PE2.

PE1	PE2
X:=0	X:=0
F:=0	F:=0
:	:
X:=1	p:=F
F:=1	q:=X

This parallel program has the possible results: $(p,q) := (0,0), (0,1), (1,1)$ whereas the result $(p,q) := (1,0)$ is not possible as $p:=1$ implies that $F:=1$ which cannot happen unless $X:=1$ which will make $q:=1$ and q cannot be 0. Observe that even though this parallel program is sequentially consistent as per Lamport’s definition, it can give three possible results. This indeterminacy is inherent to programs running independently in two processors due to what is known as *data races*. In the above example, $(p,q) := (0,0)$ will occur if PE1 is slow and PE2 is fast so that it reaches statements $p:=F$ and $q:=X$ before PE1 gets to the statements $X:=1$, and $F:=1$. The result $(p,q) := (1,1)$ will occur if PE1 is fast and wins the race and makes $X:=1$ and $F:=1$ before PE2 reaches the statements $p:=F$ and $q:=X$. To get determinate results, a programmer must make the programs *data race free*. For example if the result $(p,q) := (1,1)$ is required the programs should be rewritten as:

PE1	PE2
X:=0	X:=0
F:=0	F:=0
B:=0	
:	:
X:=1	until ($B == 1$) wait
F:=1	$p:=F$
B:= 1	$q:=X$

This program will always give $p:=1$, $q:=1$ regardless of the speeds of PE1 and PE2. The program is now data race free.

Sequential consistency of a shared memory parallel computer is guaranteed if the following conditions are satisfied [Adve and Gharachorloo, 1996].

1. Cache coherence of the parallel computer is ensured (if the PEs have caches).
2. All PEs observe writes to a specified location in memory in the same program order.
3. For each PE delay access to a location in memory until all previous Read/Write operations to memory are completed. A read is completed if a value is returned to PE and a write is complete when all PEs see the written value in the main memory. (It is assumed that writes are atomic).

It is important to note that the above conditions are sufficient conditions for maintaining sequential consistency.

In a bus based shared memory multiprocessor system whose caches are coherent, sequential consistency is maintained as:

1. Writes to a specified location in the shared memory is serialized by the bus and observed by all PEs in the order in which they are written. Writes are atomic because all PEs observe the write at the same time.
2. Read by a PE is completed in program order. If there is a cache miss, the cache is busy and delays any subsequent read requests.

A data race free program should be the aim of a programmer writing a parallel program as it will give predictable results. Sequentially consistent memory model is easy to understand and what programmers expect as the behavior of shared memory parallel computers. Many commercial shared memory parallel computer architects have, however, used relaxed memory models as they felt it would improve the performance of the computer. For example, the multiprocessor architecture of Fig. 4.19 does not guarantee sequential consistency.

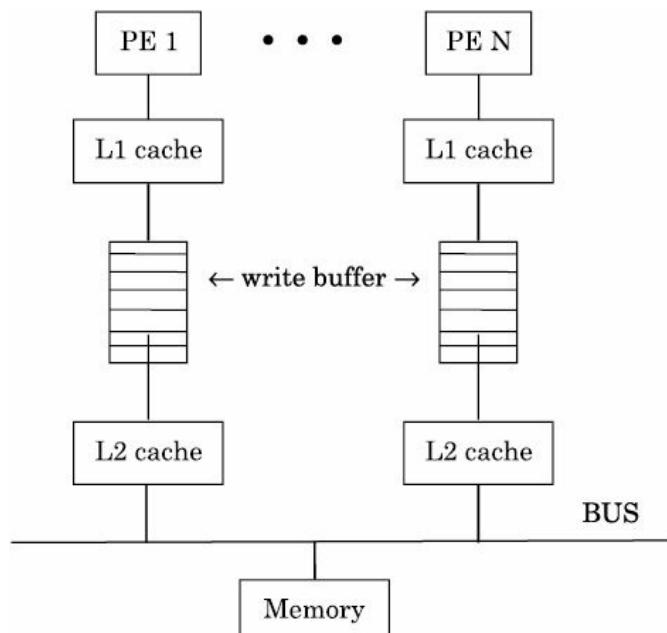


Figure 4.19 A shared bus multiprocessor with write buffers.

Assume that the first level cache is write-through and the second level cache is write-back. Assume also that the write buffers have no forwarding mechanism. In other words, reads to the second level cache are delayed until the write buffer is empty. If there are no hits to L1 cache then sequential consistency is maintained as in shared bus architecture. If, however, there are hits to L1 cache, the access from the cache should be delayed until the write buffers are empty to maintain sequential consistency as sequential consistency demands Reads and Writes to memory to be completed in program order. This requirement is relaxed by using the write buffers to improve performance.

An order is specified as $X \rightarrow Y$ which is interpreted as the operation X must complete before Y is executed. Sequential consistency requires that Read(R) and Write(W) operations to memory must maintain the four possible orderings $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$ and $W \rightarrow W$. In other words, a Read must complete before Write. Relaxed models relax one or more of the above ordering requirements of the sequential consistency model.

Relaxed Consistency Models

There are three major relaxed consistency models. They are:

1. Total Store Order (TSO) or Processor Consistency

In this model, the following orderings are required:

$$R \rightarrow W, R \rightarrow R, \text{ and } W \rightarrow W$$

The ordering $W \rightarrow R$ is not required. As this model requires $W \rightarrow W$, many programs which are sequentially consistent work without any need for additional synchronization statements in the program.

2. Partial Store Order

In this model, the following orderings are maintained:

$$R \rightarrow W, R \rightarrow R, \text{ and } W \rightarrow R$$

The ordering $W \rightarrow W$ is not required.

This model needs a programmer to modify a sequentially consistent program and introduce appropriate synchronization operations.

3. Weak Ordering or Release Consistency

In this modelling only two orderings are required. They are:

$$W \rightarrow W \text{ and } W \rightarrow R$$

The orderings $R \rightarrow W$ and $R \rightarrow R$ are relaxed.

As was pointed out earlier in this section, architects use these models in many commercial shared memory computers as they are convinced that performance gains can be obtained. However, these models are difficult to understand from a programmer's point of view. As synchronization of programs are specific to a multiprocessor using such relaxed model and are prone to error, it is recommended that programmers use synchronization libraries provided by the manufacturer of a multiprocessor which effectively hide the intricacies of the relaxed model.

Hill [1998] argues convincingly that with the advent of speculative execution, both sequential consistent and relaxed memory model implementations by hardware can execute instructions out of order aggressively. Sometimes instructions may have to be undone when speculation is incorrect. A processor will commit an instruction only when it is sure that it need not be undone. An instruction commits only when all previous instructions are already committed and the operation performed by the instruction is itself committed. A load or store operation will commit only when it is sure to read or write a correct value from/to memory. Even though relaxed and sequential implementations can both perform the same speculation, relaxed implementation may often commit to memory earlier. The actual difference in performance depends on the type of program being executed. Benchmark programs have been run [Hill, 1998] and it has been found that relaxed models' superiority is only around 10 to 20%. Thus, Hill recommends that even though many current systems have implemented various relaxed consistency models, it is advisable for future systems to be designed with sequential consistency as the hardware memory model. This is due to the fact that programming relaxed models add complexity and consequently increased programming cost. If a relaxed model is implemented, relaxing $W \rightarrow R$ order (processor consistency or total store order model) leads to least extra programming problems.

4.7.7 Shared Memory Parallel Computer Using an Interconnection Network

Shared bus architecture has limited scalability as the bandwidth of the bus limits the number

of processors which can share the main memory. For shared memory computers with large number of processors, specially designed interconnection networks are used which connect a set of memories to a set of processors (see Fig. 4.20). Each processor has a private cache to reduce load/store delay. Many types of interconnection networks have been used, some of which we will describe in the next section. The main advantage of such a system is higher bandwidth availability which can also be increased as more processors are added. The main disadvantage, however, is the difficulty in ensuring cache coherence. These networks do not have a convenient broadcasting mechanism required by snoopy cache protocols. Thus, a scheme known as *directory scheme* [Censier and Feautrier, 1978] is used to ensure cache coherence in these systems. The main purpose of the directory, which is kept in the main memory, is to know which blocks are in caches and their status. Suppose a multiprocessor has M blocks in main memory and there are N processors and each processor has a cache. Each memory block has N -bit directory entry as shown in Fig. 4.21. If the k^{th} processor's cache has this block, then the k^{th} bit in the directory is set to 1. In addition to the N bits for directory entries, two bits are allocated to indicate status bits as explained in Table 4.5. Besides this directory in the main memory, each block in the cache also has a 2-bit status indicator. The role of these 2 bits is explained in Table 4.6.

MEMORY MODULES

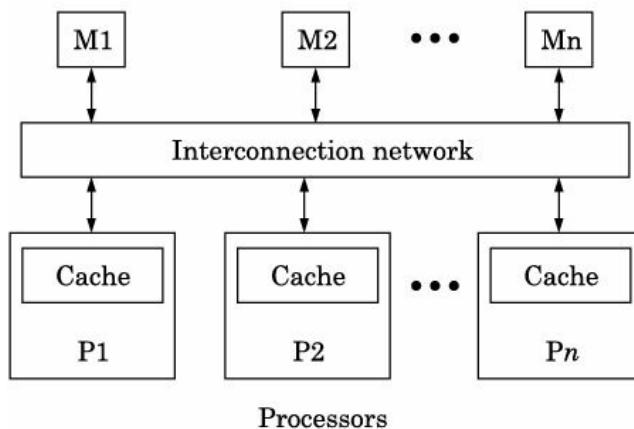


Figure 4.20 A shared memory multiprocessor using an interconnection network.

One bit per processor cache

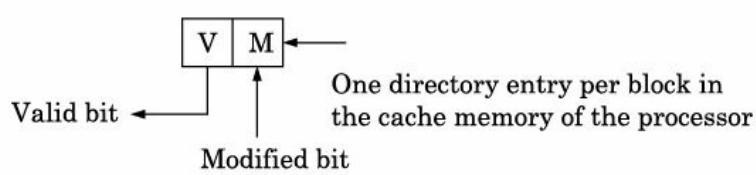
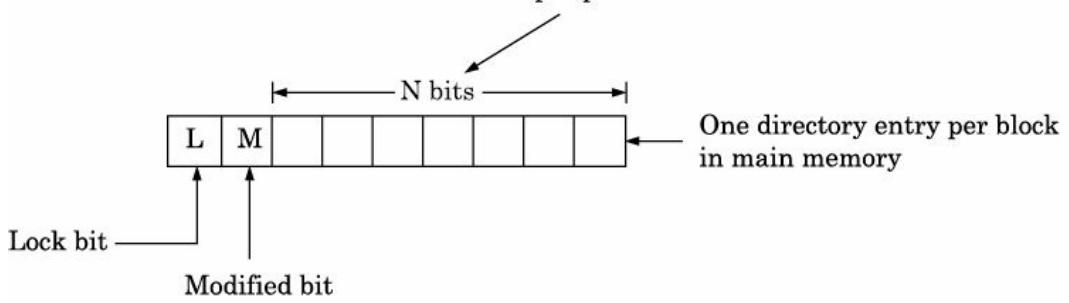


Figure 4.21 Directory entries in multiprocessor.

TABLE 4.5 States of Main Memory Blocks

State	Status bits		Explanation
	Lock bit	Modified bit	
Absent (A)	0	0	All processors' cache bits in directory 0. (No cache holds a copy of this block)
Present (P)	0	0	One or more processor cache bit is 1 (i.e., one or more caches has this block)
Present exclusive (PM)	0	1	Exactly one cache bit is 1 (i.e., exactly one cache holds a copy)
Locked (L)	1	—	An operation is going on involving the block and it is locked to avoid races

TABLE 4.6 States of Blocks in Cache (There is one 2-bit entry per block in the cache)

State	Status bits		Explanation
	Valid bit	Modified bit	
Invalid (I)	0	—	Contents of block in cache is invalid
Valid (V)	1	0	Contents of block in cache is valid and is unmodified
Valid M (VM)	1	1	Contents of block in cache is valid, it is modified and it is the only copy

The cache consistency is maintained by actions taken by the memory controllers. The actions depend on whether it is a read or a write request from processors, state of appropriate cache block and the state of the requested blocks in main memory. These actions constitute the *cache coherence protocols* for directory based multiprocessors. We will explain now a protocol which uses a write invalidate policy. It also updates the main memory whenever a new value is written in the cache of a processor.

Thus, the main memory will always have the most recently written value. We look at various cases which arise when we try to read a word from the cache.

Read from memory to processor register

k^{th} processor executes load word command.

Case 1: Read-hit, i.e., block containing word is in k^{th} processor's cache.

Cases 1.1,1.2: If the status of the block in cache is V or VM, read the word. No other action.

Case 1.3: If the status of the block in cache is I (i.e., the block is invalid), read the valid copy from the cache of a PE holding a valid copy (Remember at least one other PE must have a valid copy).

Case 2: Read-miss; i.e., block containing word not in cache. Check status of block in main memory.

Case 2.1: If it is A, i.e., no processor cache has this block.

Al: Copy block from main memory to k^{th} processor's cache using block replacement policy.

A2: Fulfill read request from cache.

A3: Set 1 in the k^{th} bit position of directory entry of this block.

A4.1: Change status bits of directory entry of this block to PM.

A5: Change status bits of k^{th} processor's cache block to V.

Case 2.2: If status of block in main memory is P (i.e., one or more caches has this block) then carry out actions A1, A2, A3 and A5 as in Case 2.1.

Case 2.3: If status of block in main memory is PM (exactly one cache has a copy and it is modified) then carry out actions A1, A2, A3 as in Case 2.1.

A4.2: Change status bits of directory entry of this block to P.

A5: Change status bits of k^{th} processor's cache block to V. These rules are summarized in Table 4.7.

TABLE 4.7 Read Request Processing in Multiprocessor with Directory Based Cache Coherence Protocol

(Read request by k^{th} processor)

Conditions	Cases →			
	1.1	1.2	1.3	2
State of block in cache	V	VM	I	—
Block requested in cache?	Y	Y	Y	N
Actions				
Read from cache	X	X	—	—
Read block state in directory	—	—	X	—
Set R =	—	—	1	2
Go to Decision Table R	—	—	X	X
Exit	X	X	—	—

Decision Table R

Conditions	Cases →			
	Case 1.3	Case 2.1	Case 2.2	Case 2.3
	1	2	2	2
R =	P	A	P	PM
Directory block state				
Actions				
A1 Copy block from main memory to k^{th} processor's cache using replacement policy	—	X	X	X
A2 Fulfill read request	X	X	X	X
A3 Set 1 in the k^{th} bit position of directory	—	X	X	X
A4 Change status bits of directory entry	—	PM	—	P
A5 Change status of k^{th} processor cache block to V	—	X	X	X

Note: Write-invalidate protocol used and write through policy to main memory.

Write from processor register to memory

Store from k^{th} processor's register to its cache. (Assume write-invalidate policy and write through to main memory).

Case 1: Write hit, i.e., k^{th} processor's cache has the block where the word is to be stored.

Case 1.1: If the block's status in cache is VM (i.e., it is an exclusive valid copy).

A1: Write contents of register in cache.

A2: Write updated cache block in main memory.

Case 1.2: If the block's status is V carry out actions A1, A2 of Case 1.1.

A3: Change the status of this block in other processors to I (The identity of other processors having this block is found from the directory in main memory). (Note: This is write-invalidate policy).

Case 1.3: If the block's status is I.

A0: Copy block from main memory to cache and set its status to V. Carry out actions A1, A2 of Case 1.1 and action A3 of Case 1.2.

Case 2: Write miss, i.e., block containing word is not in k^{th} processor's cache. Read directory of block from main memory.

Case 2.1: If its status is A do following:

A2.1: Copy block from memory into $k^{\text{'s}}$ cache using block replacement policy.

A2.2: Write register value in block.

A2.3.1: Change status of cache block bits to VM.

A2.4: Write back block to main memory.

A2.5: Put 1 in k^{th} bit of directory.

A2.6.1: Set status bits of directory entry to PM.

Case 2.2: If its status is P do following:

Perform actions A2.1, A2.2, A2.4, A2.5 and the following:

A2.3.2: Change status of cache block bits to V.

A2.8: Change status bits of cache blocks storing this block to I.

Case 2.3: If its status is PM, perform actions A2.1, A2.2, A2.4, A2.5, A2.3.2, A2.8.

A2.6.2: Change status of directory entry to P.

The write protocol is summarized in Table 4.8.

One of the main disadvantages of the directory scheme is the large amount of memory required to maintain the directory. This is illustrated by the following example:

EXAMPLE 4.3

A shared memory multiprocessor with 256 processors uses directory based cache coherence and has 2 TB of main memory with block size of 256 bytes. What is the size of the directory?

Solution

There are $(2 \text{ TB}/256\text{B})$ blocks = 8G blocks.

Each block has one directory entry of size $(256 + 2) = 258$ bits.

\therefore Number of bits in the directory = 8×258 Gbits

$$= 258 \text{ GB.}$$

This is more than one eighth of the total main memory!

Number of bits for directory in each cache = $2 \times$ Number of blocks/cache. If each cache is 1 MB, number of blocks in each cache = 4096.

\therefore Number of bits for directory in each cache = 8192 bits.

TABLE 4.8 Write Request Processing in Multiprocessor with Directory Based Cache Coherence Protocol

(Write request by k^{th} processor, Write invalidate policy for cache blocks, Write through policy for main memory)		Cases →			
<i>Conditions</i>		1.1 1.2 1.3 2			
Cache block status of k^{th} processor		VM V I —			
Requested block in cache		Y Y Y N			
<i>Actions</i>					
A0	Copy block from main memory to cache and set cache block state to V	— — X —			
A1	Write contents of register in cache	X X X —			
A2	Write updated cache block in main memory	X X X —			
A3	Change status of cache block in other processors to I Go to Table W Exit	— X X — — — — X X X X —			

Decision Table W					
Read directory of requested block from main memory. Do following actions as per conditions.					
<i>Conditions</i>		Cases →			
Status of block in main memory		2.1 2.2 2.3			
A P PM					
<i>Actions</i>					
A2.1	Copy block from main memory to k^{th} 's cache	X X X			
A2.2	Write register value in cache block	X X X			
A2.3	Change status bits of cache block to	VM V V			
A2.4	Write back block in main memory	X X X			
A2.5	Set 1 in k^{th} bit of directory	X X X			
A2.6	Set status bits of directory to	PM — P			
A2.7	Change status of cache block in other processors to Exit	— I I X X X			

The directory scheme we have described was proposed by Censier and Feautrier [1978] and is a simple scheme. The main attempts to improve this scheme try to reduce the size of the directory by increasing the block size. For example, in Example 4.3 if the block size is increased to 1 KB, the directory size will be around 64 GB. Larger blocks will, however, increase the number of invalid cached copies and consequent delay in moving large blocks from the main memory to the processors' caches even when only a small part of it is changed. One method which has been proposed is a limited directory scheme in which the number of simultaneously cached copies of a particular block is restricted. Suppose not more than 7

simultaneous copies of a cache block are allowed in Example 4.3 then the directory entry can be organized as shown in Fig. 4.22.

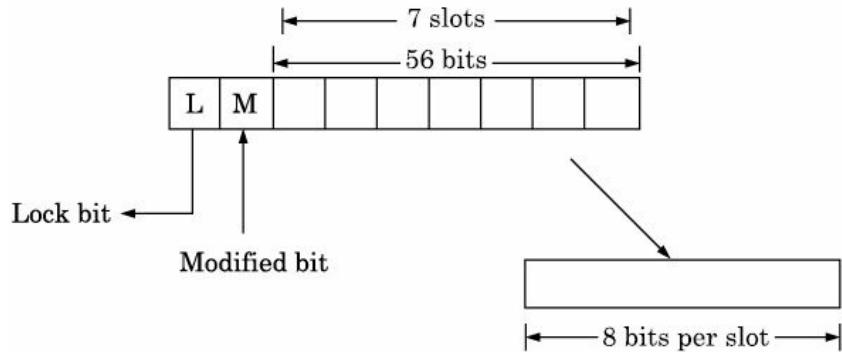


Figure 4.22 Directory entry for limited simultaneous caching.

Each 8-bit entry indicates the address of the PE whose block is cached and 7 addresses are allowed for each block at a time. The directory size will now be:

$$\text{Total number of blocks} = 8 \times 10^9$$

$$\text{Directory size} = (2 + 56) \times 8 \text{ Gbits} = 58 \text{ GB}$$

In general if i simultaneous cached copies are allowed and there are N processors, the size of each directory entry is $(i \log_2 N + 2)$. This scheme will work if there is some locality of references to blocks in a program.

There are many variations of the Censier and Feautrier [1978] scheme. One of them is called a sectored scheme in which larger block sizes are used with sub-blocks to reduce size of data movement to main memory. Another scheme is a chained directory scheme. We will not discuss them here and an interested reader can read a survey of cache coherence protocols for multiprocessors [Stenstrom, 1990].

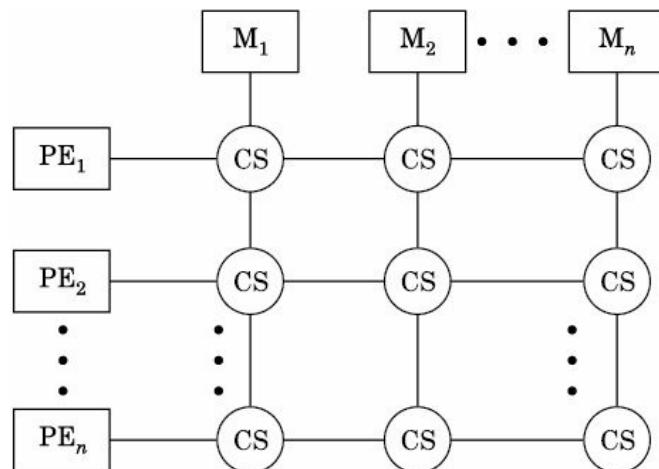
4.8 INTERCONNECTION NETWORKS

In the last section, we saw that memory modules can be connected to processors by an interconnection network. The main advantage of an interconnection network is the possibility of designing the network in such a way that there are several independent paths between two modules being connected which increase the available bandwidth. Interconnection networks may be used not only to connect memory modules to processors but also to interconnect computers. Such an interconnected set of computers is a parallel computer, often called distributed memory parallel computer. We will discuss distributed memory parallel computers in the next section. In this section, we will describe various interconnection networks which have been described in the literature and also used in building commercial parallel computers.

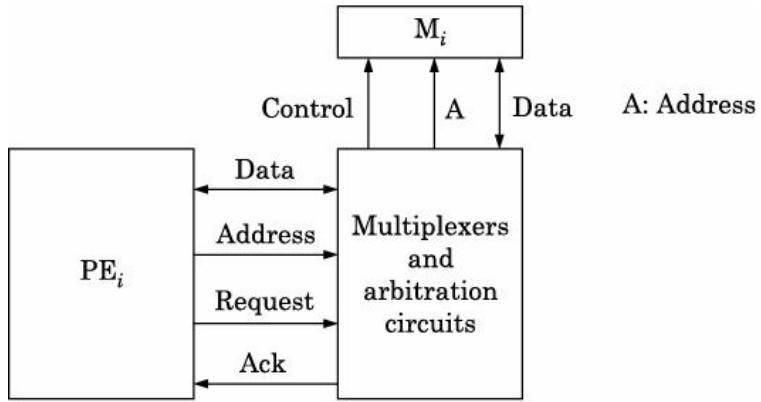
Interconnection networks may be classified into two categories called direct and indirect. In *direct networks*, each computing node is connected to a neighbour by dedicated communication lines. In *indirect networks*, on the other hand, computing nodes are connected to many other nodes using one or more switching elements. In the next subsection, we will discuss indirect network. Direct networks are discussed in Section 4.8.2.

4.8.1 Networks to Interconnect Processors to Memory or Computers to Computers

One of the best known interconnection systems is called the *crossbar switch* (see Fig. 4.23). The crossbar switch allows any processor PE_i to connect to any memory M_i . It also allows simultaneous connection of (P_i, M_i) , $i = 1$ to n . The routing mechanism is called a *crosspoint switch*. This switch consists of multiplexers and arbitrators and is a complicated logic circuit. When more than one processor requests the same memory module, arbitration logic has to set up priority queues. A crossbar switch based system is very versatile but very expensive. If there are n processors and n memory modules, n^2 switches are needed. Some computers with small number of processors have been built using this idea.



(a) Crossbar connection of PEs to memories



(b) Structure of a Crosspoint Switch (CS)

Figure 4.23 Crossbar interconnection network.

A more commonly used interconnection network is called a *multistage inter-connection network* or a *generalized cube network*. A basic component of these networks is a two-input, two-output interchange switch as shown in Fig. 4.24. A simple interchange switch has a control input C . If $C = 0$, the inputs are connected straight to the corresponding outputs and if $C = 1$, they are cross-connected (see Fig. 4.24).

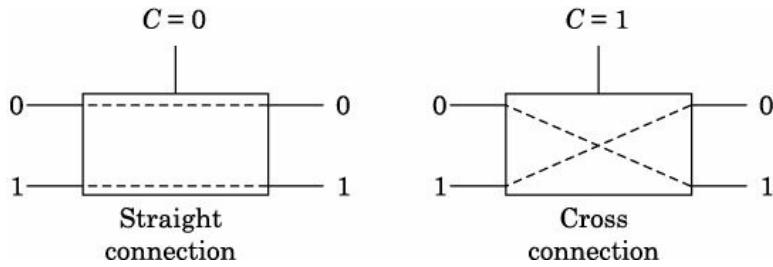


Figure 4.24 An interchange switch.

A general multistage network has N inputs and N outputs with $N = 2^m$, where m is called the number of stages. Each stage uses $(N/2)$ interchange switches for a total of $m(N/2) = (N/2)\log_2 N$ switches. A general multistage network with $N = 8$ is shown in Fig. 4.25. The inputs and outputs are labelled from 0 to 7.

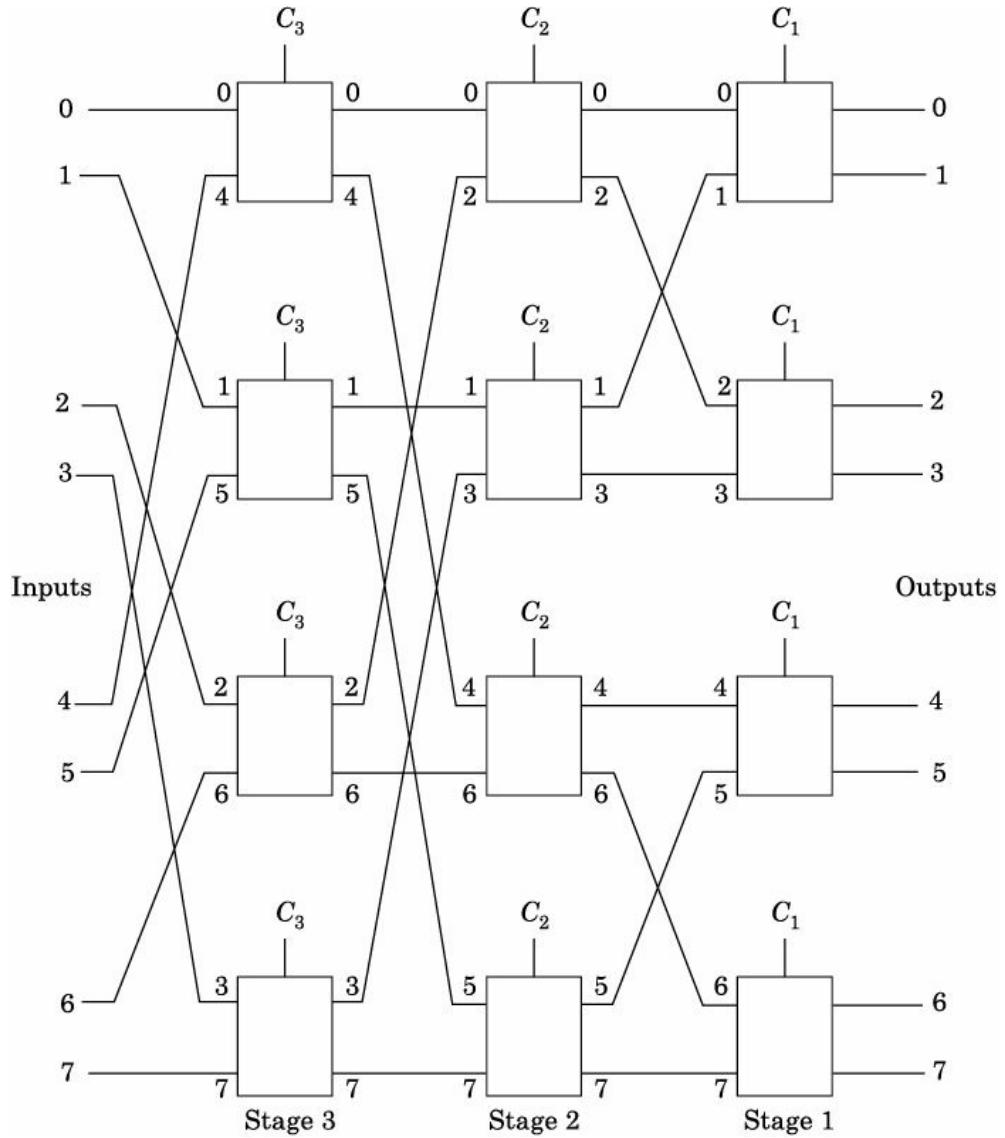


Figure 4.25 A three stage generalized network.

Observe that stages are numbered 1 to 3 from output to input. In stage 1, the labels of the input of each switch differ in the least significant or first bit position. In stage 2, the labels of input to switches differ in the second bit position and in the third stage in the third bit position.

In general if the binary representation of the upper input of k^{th} stage is $(a_m, a_{m-1}, \dots, a_k, \dots a_0)$, the label of the lower input is $(a_m, a_{m-1}, \dots, \bar{a}_k \dots a_0)$ where \bar{a}_k is the complement of a_k . Observe that outputs of a switch labelled p at level k are connected to the inputs with the same label p at the $(k+1)^{\text{th}}$ stage. Assume processors labelled P_0, P_1, \dots, P_7 are connected to inputs with the labels 0, 1, 2, ..., 7 respectively and memory modules M_0, M_1, \dots, M_7 are connected to the outputs with labels 0, 1, 2, ..., 7, respectively. If say P_2 is to be connected to M_6 , exclusive OR (\oplus) the binary representations of their labels, namely, 2 and 6. The result is $(010 \oplus 110) = 100$. These 3 bits are used to set the controls of the switches in the three stages. Thus C_3 is set to 1, C_2 to 0 and C_1 to 0. These control bits set up the switches to create a path to connect P_2 and M_6 . Simultaneously paths are also set up to connect P_0 to M_4 , P_1 to M_5 , P_3 to M_7 , P_4 to M_0 , P_5 to M_1 , P_6 to M_2 and P_7 to M_3 .

Observe that a given P_i can always be connected to any M_j but at the same time any arbitrary P_r cannot be connected to a memory module, say, M_q . Only if $|i - j| = |r - q|$, P_r and M_q will be connected. In general in an m stage interconnection network of the type shown in Fig. 4.25, P_i and M_j will be connected if $C_m C_{m-1} \dots C_1 = i \oplus j$ where i and j are represented by their binary equivalents. Observe that the complexity (in terms of number of switches) is much lower for a multistage interconnection network as compared to a crossbar switch network. The number of switches of a $(N \times N)$ crossbar is N^2 whereas that of multistage network is $(N/2) \log_2 N$. Thus if $N = 64$, $(N \times N) = 4096$ whereas $(N/2) \log_2 N = 32 \times 6 = 192$. The delay or latency of the multistage network is of the order of $\log_2 N$ whereas it is constant for a crossbar. A crossbar is called *nonblocking* as any P_i can be connected to any M_j and at the same time another arbitrary P_r may be connected to M_q ($r \neq i, j; q \neq i, j$). This is not possible in the multistage network we considered. This multistage network is called a *blocking network*.

Another popular multistage network uses what is known as a *shuffle* transformation. Given a binary string $(a_n a_{n-1} \dots a_2 a_1)$

$$\text{Shuffle } (a_n a_{n-1} \dots a_2 a_1) = a_{n-1} a_{n-2} \dots a_2 a_1 a_n$$

For 3 bit numbers, we get shuffle $(001) = 010$ and shuffle $(101) = 011$

In other words, shuffle transformations of a binary string is the same as left circular shifting it by 1 bit.

A network built using $\log_2 N$ cascaded switches using shuffle connection is called an *omega* network. In Fig. 4.26, we depict a 3-stage omega network. Observe that the input labels of a switch at i^{th} stage is obtained by shuffle transforming the output labels of the corresponding $(i - 1)^{th}$ stage switch. For example the input labels of switch 23 (see Fig. 4.26), namely 001 and 011 are obtained by left circular shifting (i.e., shuffle transforming), the labels 100 and 101 which are the output labels of switch 13. If a PE attached to input i is to be connected to a memory module attached to output j , exclusive OR (\oplus) of the binary equivalents of i and j is found. If $(i \oplus j) = p$, the least significant bit of p is used as control bit C_1 (see Fig. 4.26), the next significant bit as C_2 and the most significant bit as C_3 . For example if input 7 is to be connected to output 3, $(111 \oplus 011 = 100)$. Thus, $C_1 = 0$, $C_2 = 0$ and $C_3 = 1$. Remember that $C = 0$ direct connects input to output and $C = 1$ cross connects them. By examining, Fig. 4.26, it is clear that $C_1 = 0$, $C_2 = 0$, $C_3 = 1$ connects 7 to 3. Observe that several connections between input and output can be made at the same time. The omega network was used in some experimental multicomputers [De Legama, 1989].

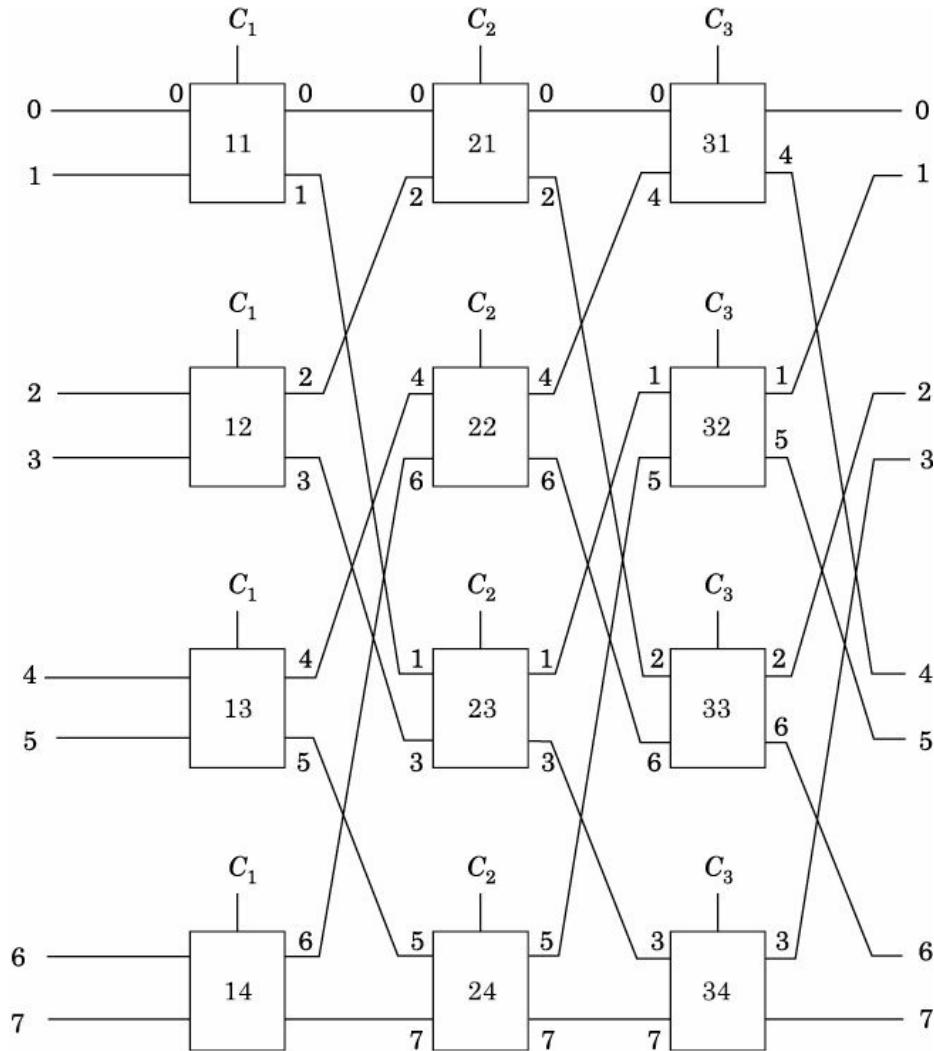


Figure 4.26 A three stage omega network.

Another multistage network which has been used in many commercial computers is called a *butterfly* network. A (4×4) switch using 4 butterfly switch boxes is shown in Fig. 4.27. Observe that the butterfly connection is not very different from the generalized network of Fig. 4.25. The output switches of Fig. 4.25 are butterfly switches. Alternate switches (seen from top to bottom) of the previous stage are also butterfly connected switches.

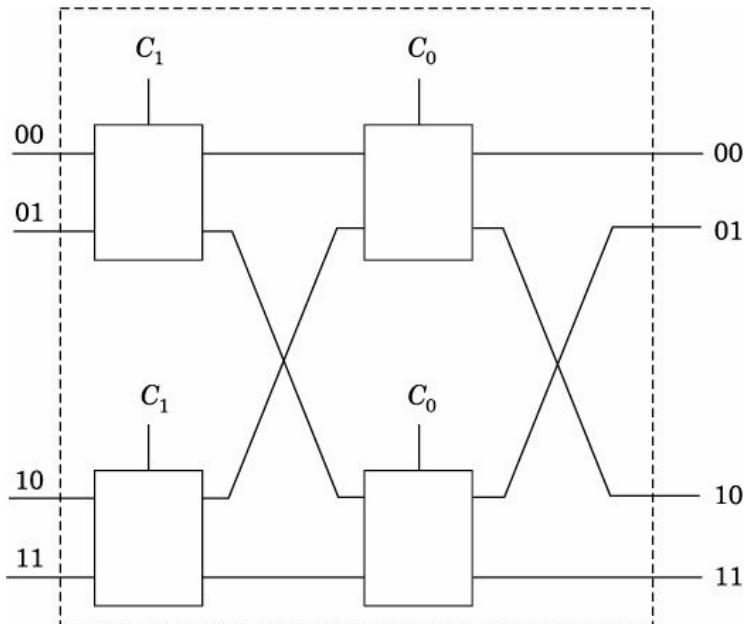


Figure 4.27 A (4×4) butterfly switch building block.

A butterfly switch network also does not allow an arbitrary connection of N inputs to N outputs without conflicts. It has been shown that if two butterfly networks are laid back to back so that data flows forward through one and in reverse through the other, then it is possible to have a conflict free routing from any input port to any output port. This back-to-back butterfly is known as a *Benes* network [Leighton, 1992]. Benes networks have been extensively studied due to their elegant mathematical properties but not widely used in practice.

4.8.2 Direct Interconnection of Computers

The interconnection networks we considered so far, namely, crossbar and multistage interconnection networks can be used either to connect PEs to memories or CEs to CEs (CE is a PE with memory). There are many other interconnection networks which have been used to only interconnect CEs to build parallel computers. In early parallel computers algorithms were tailored to run optimally on a particular interconnection of CEs. This had a negative impact of making programs written for a machine using a particular interconnection network not portable to another machine. Thus, the trend now is not to closely couple the application program to a particular interconnection structure. However, these interconnection methods have many other advantages such as regularity, fault tolerance, and good bandwidth, and are thus used in several machines. The simplest of such an interconnection structure is a ring shown in Fig. 4.28. Observe that each CE is connected through a Network Interface Unit (NIU) to a bidirectional switches. Data can flow in either direction through the switch. For example, CE2 can send data to CE3 or CE3 may send data to CE1. Unlike a bus, pairs of CEs can be communicating with one another simultaneously. For example, while CE2 is sending data to CE3, CE1 can be receiving data from CE0. CEs connected using such an interconnection network solve problems by exchanging messages using the NIUs and associated switches. Interconnection networks are characterized by four parameters. They are:

1. The total network bandwidth, i.e., the total bandwidth, namely, bytes/second the network can support. This is the best case when all the CEs simultaneously

- transmit data. For a ring of n CEs, with individual *link bandwidth* B , the total bandwidth is nB .
2. The *bisection bandwidth* is determined by imagining a cut which divides the network into two parts and finding the bandwidth of the links across the cut. In the case of a ring of n CEs it is $2B$ (independent of n).
 3. The number of ports each network switch has. In the case of a ring it is 3.
 4. The total number of links between the switches and between the switches and the CEs. For a ring it is $2n$.

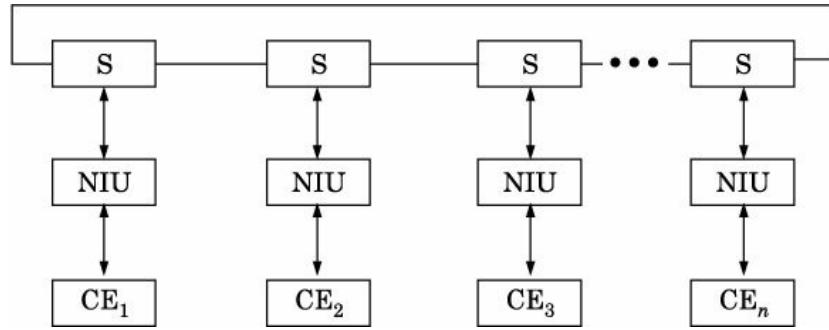
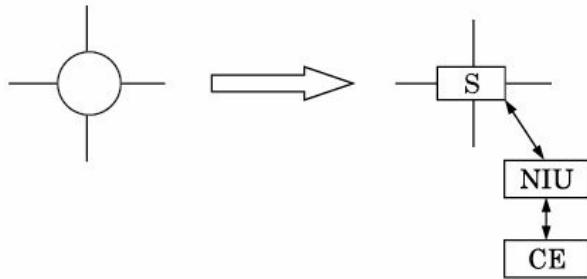
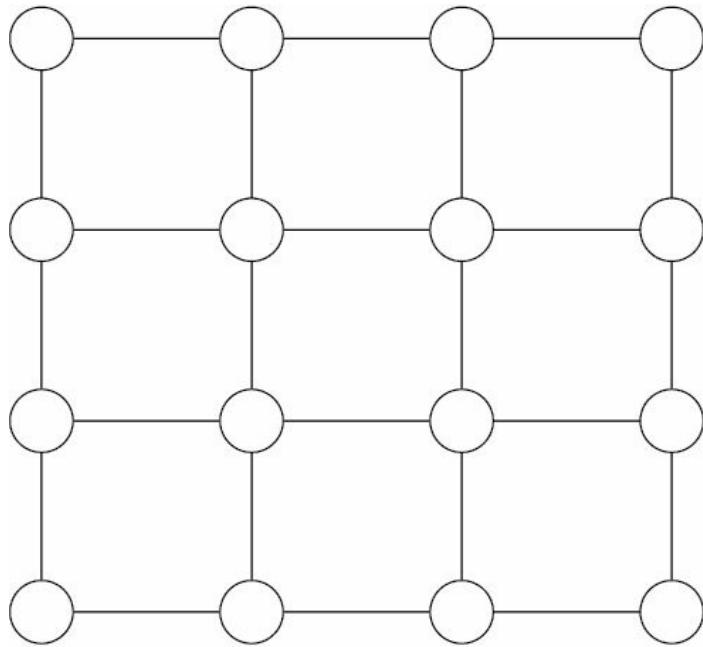


Figure 4.28 A ring interconnection of computing elements.

The first two parameters give an indication of the performance of the inter-connection network as they tell how fast data can be transferred between CEs. The last two parameters are proportional to the cost of switches and links which make up the interconnection system.

There are a large variety of interconnection networks that have been proposed. We will consider two of them which are quite popular. The first one is called a 2D grid. It is shown in Fig. 4.29. Observe that CEs are all in a plane and nearest neighbours are interconnected. This network is easy to construct. For a 2D grid, the total bandwidth is computed by observing that in n processor system there are \sqrt{n} rows and \sqrt{n} columns. Each row has $(\sqrt{n} - 1)$ links connecting switches and there are \sqrt{n} rows. Similarly there are \sqrt{n} columns and $(\sqrt{n} - 1)$ links per column. The bandwidth of each connection is B . Thus the total bandwidth is $2\sqrt{n}(\sqrt{n} - 1)B$. The bisection bandwidth is $\sqrt{n}B$. The number of ports per switch is 5, 4 to the 4 nearest neighbours and 1 to the network interface unit. The total number of links is $(2\sqrt{n}(\sqrt{n} - 1) + n)$ as one link is to be provided from each switch to the CE.



Each circle in the figure consists of a switch connected to a Network Interface Unit (NIU) and a Computing Element (CE).

Figure 4.29 2D grid of 16 CEs.

A variation of 2D grid is a 2D toroid (Fig. 4.30) in which the switches at the edges of the grid are also connected by links. Observe that it is a more symmetric network. The disadvantage is that the network is non-planar. The toroid with n CEs has a higher total bandwidth of $2nB$, the bisection bandwidth is $2\sqrt{n}$ B and the number of links is $3n$. The ports per switch is uniformly 5 for all switches in the toroid.

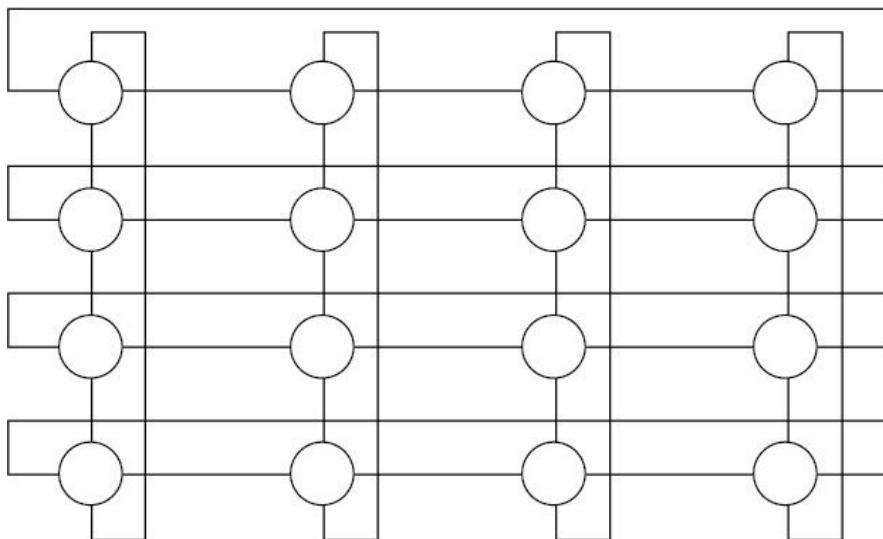
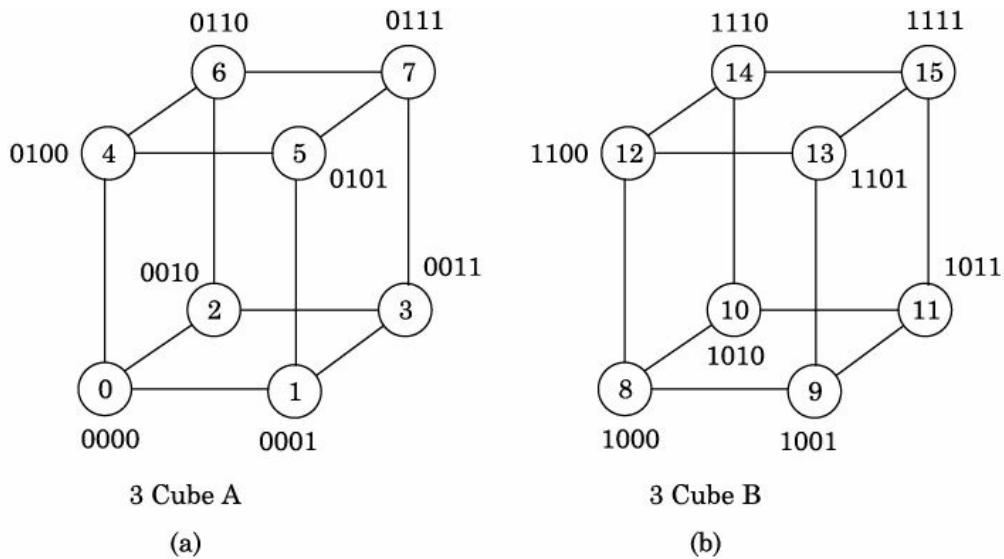


Figure 4.30 2D toroid of 16 CEs.

Another popular interconnection network is called *N cube* or hypercube. An *N* cube interconnects 2^N CEs, one at each corner of the cube. A 3-dimensional cube (or 3-cube) has 8 CEs and is shown in Fig. 4.31. When corresponding corners of two 3 cubes are interconnected we get a 16CE, 4-cube. Observe the address labels of the switches. The address labels of neighbours (i.e., switches connected by a single link) differ only in one bit position. Thus, routing of messages between CEs can be done methodically. Another interesting aspect of a hypercube interconnection is that the maximum number of links to be traversed from any CE to any other CE in n CE hypercube is $\log_2 n$ where $n = 2^N$. Thus in a 4-cube the maximum number of hops (i.e., number of links to be traversed) is 4. A hypercube is a rich interconnection structure with many good theoretical properties and thus allows many other interconnection structures to be mapped on to it. Hence, its popularity is with computer architects.



Corresponding vertices of Cube A and Cube B are interconnected to get a 4 cube of 16 CEs

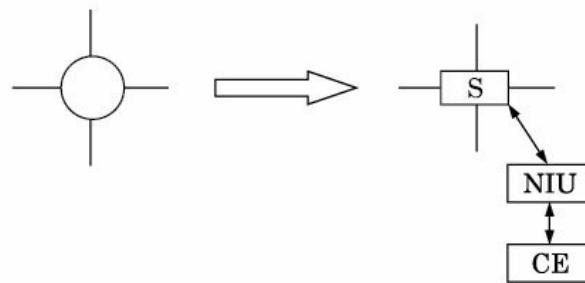


Figure 4.31 A hypercube interconnection network.

The total aggregate bandwidth of an N -dimensional hypercube which has $n = 2^N$ CEs is $B(n \log_2 n)/2$. Thus for a 16 CE hypercube, the total bandwidth is 32B. The bisection bandwidth of a N cube is $Bn/2$ as when a hypercube is cut in half ($n/2$) links are cut. The total number of ports per switch is $(\log_2 n + 1)$. The total number of links is $n + (n \log_2 n/2)$ as we have to provide 1 link at each switch to connect it to the network interface unit.

In Table 4.9 we summarize the properties of different types of networks interconnecting n processors. If we compute these values for $n = 256$, for example we find that the network bandwidth of hypercube is 1024 B and that of 2D toroid is 512 B. The bisection bandwidths

are respectively 128B and 32B for the hypercube and 2D toroid. The total number of links for hypercube is 1280 whereas it is 768 for 2D toroid. Finally the ports/switches of hypercube is 9 and it is 5 for the toroid. The number of links and ports are proportional to the cost whereas the total bandwidth and bisection bandwidth are proportional to performance. We see that the higher performance of the hypercube is obtained at a higher cost (Is it proportionately higher? What happens when n increases?).

TABLE 4.9 Summary of Parameters of Interconnection Networks					
Parameter	Bus	Ring	2D grid	2D toroid	N cube
Total network bandwidth	B	nB	$2\sqrt{n}(\sqrt{n}-1)B$	$2nB$	$\frac{Bn \log_2 n}{2}$
Bisection bandwidth	B	$2B$	$\sqrt{n}B$	$2\sqrt{n}B$	$Bn/2$
Total number of links	—	$2n$	$3n - 2\sqrt{n}$	$3n$	$n + \frac{n \log_2 n}{2}$
Ports per switch	1	3	5	5	$\log_2 n + 1$

n : Number of CEs; B: Bandwidth per link; $n = 2^N$

4.8.3 Routing Techniques for Directly Connected Multicomputer Systems

In the last section, we saw that each node of a multicomputer system is connected to an interconnection network through a switch. In all modern systems the switch is made more sophisticated into what is called a *router*. This has been facilitated by advances in large scale integrated circuits which have allowed fabrication of router chips. Routers control transmission of messages between nodes and allow overlapping of computation with communication. The main job of a router is to examine an incoming message and decide whether it is intended for the CE connected to it or it is to be forwarded to the next logical neighbour. Normally long messages are broken down into a number of smaller packets. Each packet has a header followed by data. The header has information on the destination of the packet and its serial number in the message. Messages are transmitted between nodes by using what is known as a *packet switching* method. The packets leave the source node and each packet takes an available route to reach the destination. At the destination the packets are re-assembled in the right order. In order to route packets, the router at each node requires storage to buffer packets. Further the time taken by a message to reach the destination from a source depends on the time taken by the packet which took the longest time. The storage needed to buffer packets at a node can be quite large. To reduce both storage needed by a router and time for routing a method called *virtual cut through routing* is used in some parallel computers. This method buffers a packet at a node only if the next link it is to take is occupied. This expedites message routing time and reduces the buffer space needed. If links are busy with earlier packets, this method will reduce to a normal packet switched routing.

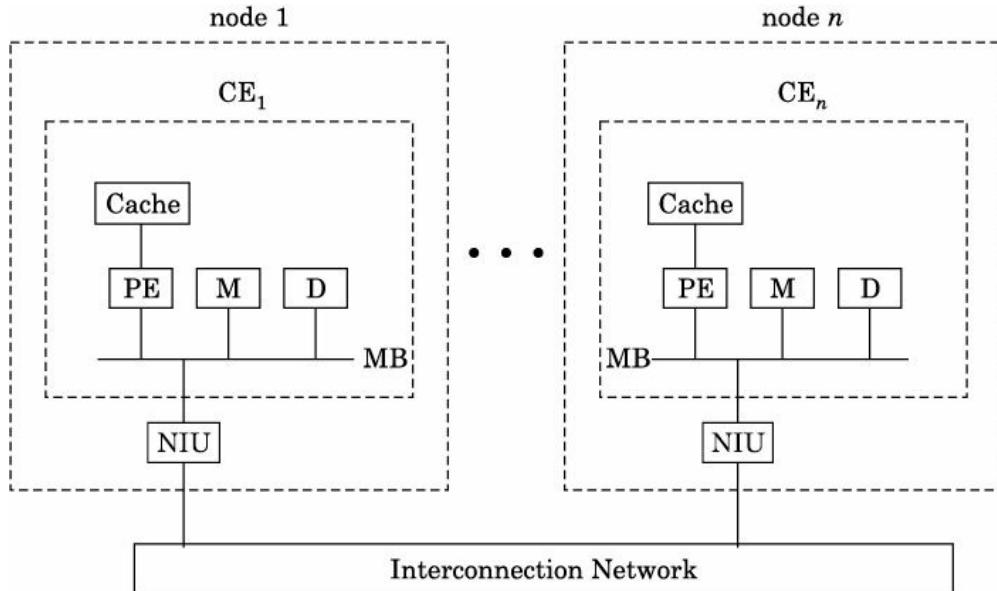
Another method called *wormhole routing* [Mahapatra, 1998] is now employed in many systems. In wormhole routing a packet is further subdivided into what are known as *flits* (flow control digits). The first flit has the header information and the rest are data. All flits of a packet follow the same route as the header flit. This is similar to a train with one engine (header flit) and several passenger compartments (data flits). Flits belonging to different packets cannot be interleaved (because the compartments may go to wrong destinations). Successive flits in a packet are pipelined asynchronously in hardware using a handshaking protocol. When the header flit is blocked, the following flits are buffered in the earlier nodes

in their path. Wormhole routing is quite fast and needs much smaller buffer space as compared to packet switching. Many algorithms have been developed to ensure deadlock free transmission of packets in wormhole routing. Wormhole routing may also be used in multistage interconnection networks.

4.9 DISTRIBUTED SHARED MEMORY PARALLEL COMPUTERS

A shared memory parallel computer is easy to program as it gives a user a uniform global address space. Scalability is, however, a problem. As we saw, the number of PEs in a shared memory computer using a shared bus is limited to around 16 due to the saturation of the bus. A shared memory parallel computer using interconnection network to connect PEs to memory alleviates this problem to some extent by providing multiple paths between PEs and memory. In this case also it is not possible to increase the number of PEs beyond 64 due to high cost of designing, high bandwidth interconnection networks, and difficulty in maintaining cache coherence. This has led to the development of Distributed Shared Memory (DSM) parallel computer in which a number of PEs with their own private memories (i.e., CEs) are interconnected. In this system, even though the memory is physically distributed, the programming model is a shared memory model which is easier to program. The system, however allows a much larger number of CEs to be interconnected as a parallel computer at reasonable cost without severely degrading performance. In other words, DSM parallel computers are scalable while providing a logically shared memory programming model.

A general block diagram of a Distributed Shared Memory (DSM) parallel computer is shown in Fig. 4.32. In a DSM parallel computer each node is a full fledged computer, i.e., it has a processor, cache and memory. Each computer in the system is connected to an interconnection network using a Network Interface Unit (NIU) (also called by some authors Communication Assist, as it often has processing capability of its own). Even though each CE has its own private memory and cache, the system has hardware and software assists to give a user the facility to program the machine as though it is a shared memory parallel computer. In other words, a user can program assuming that there is a single global address space. It is clear when we see the structure of the machine that accessing data in the memory attached to a processor in a given node is much faster as compared to accessing a data in the memory belonging to a CE in another node connected to it via the interconnection network. Thus, this type of parallel computer is also known as Non Uniform Memory Access (NUMA) parallel computer. When each processor has its own cache memory, it is also necessary to maintain cache coherence between the individual processors. This is done by using a directory scheme similar to the method described in the Section 4.7.7. Thus a more accurate nomenclature for a Distributed Shared Memory Computer in use today is Cache Coherent Non Uniform Memory Access parallel computer or CC-NUMA parallel computer for short.



PE : Processing Element

D : Directory

MB : Internal Memory Bus

NIU : Network Interface Unit

CE : Computing Element

(The entire multicomputer has a single address space)

Figure 4.32 Distributed shared memory multicomputer.

The programming model for CC-NUMA computer is the same as for shared memory parallel computer. Let us examine first how the computers cooperate to solve a problem. If a processor issues a read request (load data from memory), it will specify the address from where data is to be retrieved. The address is split by the system software into two parts; a node number and the address of the location within the node. Thus if the node number is that of the processor issuing the read command, the data is retrieved from the memory attached to it. Otherwise the node number is used to send a message via the interconnection network to the appropriate node's memory from where the data is retrieved and delivered via the interconnection network to the requesting processor. The procedure is summarized as follows:

Procedure 4.1 Command: Load from specified address into a register

Step 1: Translate address:

Address \leftarrow CE number + memory address in CE.

Step 2: Is CE local?

Step 3: If yes, load from local memory of CE, else send request via network to remote CE.

Step 4: Retrieve data from remote CE's memory.

Step 5: Transport data via network to requesting CE.

Step 6: Load data in specified CE's processor register.

Observe that if the required data is in the requesting CE's memory, the work is done in Step 3. Otherwise we have to carry out Steps 4, 5 and 6. Typically the time to retrieve data from the local memory is around 10 cycles and from the local cache 1 cycle whereas the time to obtain it from the memory of remote CE is of the order of 1000 cycles. Thus, it is extremely important for a programmer (or a compiler which compiles a program for NUMA machine) to ensure that data needed by a CE during computation is available in the local memory. If

care is not taken to do this, the execution time of a program will be very large and the advantage of having multiple processors will be lost.

Procedure 4.2 gives the operations to be carried out to store (i.e., write register contents in a specified location in memory).

Procedure 4.2 Command: Store register in address

Step 1: Translate address:

Address \leftarrow CE number + memory address in CE.

Step 2: Is CE local?

Step 3: If yes, store contents of register in specified location in memory of local CE, else send contents of register via network to remote CE.

Step 4: Store contents of register in the specified location in the memory of the remote CE.

Step 5: Send message to CE which issued the store command that the task has been completed.

Observe that storing in memory requires one less step.

We have not discussed in the above procedures the format of the request packet to be sent over the network for loads and stores. A load request is transmitted over the network as a packet whose format is:

Source CE address	Destination CE address	Address in memory from where data is to be loaded
-------------------	------------------------	---

Retrieved data is sent over the network using the following format to the requesting CE.

Source CE address	Destination CE address	Data
-------------------	------------------------	------

In the case of store instruction, the store request has the format

Source CE address	Destination CE address	Address in memory where data is to be stored	Data
-------------------	------------------------	--	------

After storing an acknowledgement packet is sent back to the CE originating request in the format

Source CE address	Destination CE address	Store successful
-------------------	------------------------	------------------

Error detecting bits may be added to the above packets.

When a request is sent over the interconnection network for retrieving data from a remote memory, the time required consists of the following components:

1. A fixed time T needed by the system program at the host node to issue a command over the network. This will include the time to decode (remote node to which the request is to be sent) and format a packet.
2. Time taken by the load request packet to travel via the interconnection network. This depends on the bandwidth of the network and the packet size. If B is the network bandwidth (in bytes/s) and the packet size in n bytes, the time is (n/B) s.
3. Time taken to retrieve a word from the remote memory, q .
4. Fixed time T needed by the destination CE system program to access the network.
5. Time taken to transport the reply packet (which contains the data retrieved) over the network. If the size of this packet is m bytes, the time needed is (m/B) s.

Thus, the total time is $2T + q + [(n + m)/B]$.

The time taken to store is similar. The values of n and m may, however, be different from the above case. The above model is a simplified model and does not claim to be an exact model. A basic question which will arise in an actual system is “Can the CE requesting service from a remote CE continue with processing while awaiting the arrival of the required data or acknowledgement?” In other words, can computation and communication be overlapped? The answer depends on the program being executed. Another issue is how frequently service requests to a remote processor are issued by a CE. Yet another question is, “Whether multiple transactions by different CEs can be supported by the network”. We will now consider an example to get an idea of penalty paid if services of remote CEs are needed in solving a problem.

EXAMPLE 4.4

A NUMA parallel computer has 256 CEs. Each CE has 16 MB main memory. In a set of programs, 10% of instructions are loads and 15% are stores. The memory access time for local load/store is 5 clock cycles. An overhead of 20 clock cycles is needed to initiate transmission of a request to a remote CE. The bandwidth of the interconnection network is 100 MB/s. Assume 32-bit words and a clock cycle time of 5 ns. If 400,000 instructions are executed, compute:

1. Load/store time if all accesses are to local CEs.
2. Repeat 1 if 25% of the accesses are to a remote CE.

Solution

Case 1: Number of load/store instructions = $400,000/4 = 100,000$.

Time to execute load/store locally = $100,000 \times 5 \times 5 \text{ ns} = 2500 \mu\text{s}$.

Case 2: Number of load instructions = 40,000.

Number of local loads = $40000 \times 3/4 = 30,000$.

Time taken for local loads = $30000 \times 25 \text{ ns} = 750 \mu\text{s}$ (Each load takes 25 ns).

Number of remote loads = 10,000

Request packet format is:

Source address	Destination address	Address in memory
8 bits	8 bits	24 bits

∴ Request packet length 5 bytes

Response packet length = 6 bytes (as word length is 32 bits).

Time taken for remote load of one word = Fixed overhead to initiate request over network + Time to transmit request packet over the network + Data retrieval time from memory at remote CE + Fixed overhead to initiate request to transmit over network by remote CE + Time to transport packet over network

$$\begin{aligned}
 &= (20 \times 5 \times 10^{-9} + \frac{5}{100} \times 10^{-6} + 5 \times 5 \times 10^{-9} + 20 \times 5 \times 10^{-9} + \frac{6}{100} \times 10^{-6}) \\
 &= 335 \text{ ns}
 \end{aligned}$$

(Remember that clock time is 5 ns and $T = 20$ clock cycles and memory access time is 5 clock cycles).

Number of requests to remote CE = 10,000.

\therefore Total time for remote load = 3350 μ s

Time for local load = 750 μ s

Thus, total time taken for loads = 4100 μ s

Number of store instructions = $400000 \times 0.15 = 60,000$.

Number of local stores = $60000 \times 0.75 = 45000$.

Number of remote stores = 15,000.

Time taken for local stores = $45000 \times 25 \text{ ns} = 1125 \mu\text{s}$.

Time taken for 1 remote store of 1 word = (Fixed overhead to initiate request over network + Time to transmit remote store packet + Data store time + Fixed overhead to initiate request for acknowledgement + Time to transmit acknowledgement packet)

$$= \left(20 \times 5 \times 10^{-9} + \frac{9}{100} \times 10^{-6} + 5 \times 5 \times 10^{-9} + 20 \times 5 \times 10^{-9} + \frac{3}{100} \times 10^{-6} \right) \\ = 345 \text{ ns}$$

Time to store 15000 words = $345 \times 15000 \text{ ns} = 5175 \mu\text{s}$

Total time for stores = $1125 + 5175 = 6300 \mu\text{s}$

Total time for loads and stores = $4100 + 6300 = 10400 \mu\text{s}$

Total time for loads and stores if entirely in local CE = 2500 μ s (Case 1)

$$\frac{\text{Total time for load and store (local and remote)}}{\text{Total time for load and store if entirely local}} = \frac{10400}{2500} = 4.16$$

Observe that due to remote access, the total time taken is over 4 times the time if all access were local. It is thus clear that it is important for the programmer/compiler of the parallel program for a NUMA parallel computer to (if possible) eliminate remote accesses to reduce parallel processing time.

A question which may arise is, "What is the effect of increasing the speed of the interconnection network? In the example we have considered, if the bandwidth of the interconnection network is increased to 1 GB/s from 100 MB/s, let us compute the time taken for remote load and store.

The time for remote load =

$$= 10,000 \times \left(100 \times 10^{-9} + \frac{5}{1000} \times 10^{-6} + 25 \times 10^{-9} + 100 \times 10^{-9} + \frac{6}{1000} \times 10^{-6} \right) \\ = 2360 \mu\text{s}$$

Total time for load (local + remote) = $(750 + 2360) = 3110 \mu\text{s}$

Similarly the time for remote store

$$= 15,000 \times \left(100 \times 10^{-9} + \frac{9}{1000} \times 10^{-6} + 25 \times 10^{-9} + 100 \times 10^{-9} + \frac{3}{1000} \times 10^{-6} \right) \\ = 3555 \mu\text{s}$$

Total time for stores = $1125 + 3555 = 4680 \mu\text{s}$

Total time for loads and stores = $(3110 + 4680) = 7790 \mu\text{s}$

Total time taken if all loads and stores are local = 2500 μs

Ratio of load and store time remote/local = $7790/2500 = 3.11$.

Observe that the ratio is still high in spite of increasing the speed of the interconnection network to ten fold. The reason is the fixed overhead for each transaction over the network. Unless it is also reduced, merely, increasing the network speed will not help. It further

reiterates the point that remote accesses should be eliminated, if possible, by appropriate distribution of program and data.

As was pointed out at the beginning of this section, the programming model for NUMA machine is the same as for a shared memory parallel machine. The protocols used to maintain both cache coherence and inter-process communication must ensure sequential consistency of programs.

4.9.1 Cache Coherence in DSM

Maintaining cache coherence is important for the correctness of processing. CC-NUMA machines ensure cache coherence by adapting directory based cache coherence protocol described in Section 4.7.7. The major difference is the distribution of the directory to various nodes of the parallel computer. The directory is built by maintaining one directory per node. In this directory, there is one entry per cache block of the memory belonging to that node. This is called the *home directory*. The number of bits in the home directory entry for a cache block equals the total number of nodes in the machine. Thus if there are 128 nodes, there will be 128-bit entries for each cache block. If the memory size is 16 MB and block size is 64 bytes, the number of entries in the directory will be 256 K and each entry will have 128 one-bit entries. Thus, the directory size in each node will be 4 MB which is 25% of main memory size. One of the important questions is, “How to reduce the size of the directory?” We will return to this question later in this section. Before that we ask how cache coherence is ensured by using distributed directory. The algorithm is very similar to the one used in Section 4.7.7 for a shared memory system using an interconnection network. The major difference is the need to read from a remote memory if a cache block is not in the local main memory of the CE. For instance in the Read protocol given in Table 4.7, the action A1 of Decision Table R is: “Copy block from main memory to k^{th} processor’s cache using replacement policy”. In the case of CC-NUMA machine, the main memory is distributed. If the required block is in the local memory of CE, it is copied. If the block is not in the local main memory a message has to be sent via the interconnection network to the CE holding this block. Besides this, when a memory block is held by multiple caches and an invalidation has to be done, messages have to be sent over the network to all the CEs whose caches hold this block. All the messages can be sent simultaneously if the network allows it. Any message sent using the network will considerably slow down computation which again reiterates the importance of careful data distribution. Similar need arises in the write protocol. Modifying the protocols for read and write given in Section 4.7.7 for implementation in a CC-NUMA parallel computer is left as an exercise to the reader.

The main problem with the directory scheme is the large overhead in terms of storage needed for the directory. As the number of processors increase, the size of the directory increases linearly. Thus, other methods have been proposed to reduce the storage requirements. One of them is based on the fact that it is quite improbable that all CEs will share the same memory block. There will be some locality of access to memory and we can use a limited directory scheme explained in Section 4.7.7 (Fig. 4.22) in this case also.

Another method is also based on the idea that the number of shared cache blocks would not be very large. Thus, the caches holding a block are linked together as a linked list. This method is known as cache based directory scheme and has been standardized by IEEE in a standard known as the *Scalable Coherent Interface* (SCI) standard. This standard is now being used in many commercial CC-NUMA parallel computers. We will briefly sketch the basic ideas used in this scheme. For more details the reader is referred to the IEEE standard [Gustavson, 1992]. In this scheme each cache block has a “home” main memory for the

block. The directory entry at the home node's main memory contains only the pointer (i.e., address) of the first node whose cache has this block. This pointer is called the *head pointer* for the block. The other nodes which have this block are linked together as a *doubly linked list* (see Fig. 4.33). The cache block at the head contains address of the next node holding this block. Let us say the next node is x . The cache block at x holds the identity of the previous cache which has this block using a backward pointer to it. It also has a pointer to point to the next node which has this cache block. Thus, all the nodes with the specified cache block are doubly linked as shown in Fig. 4.33.

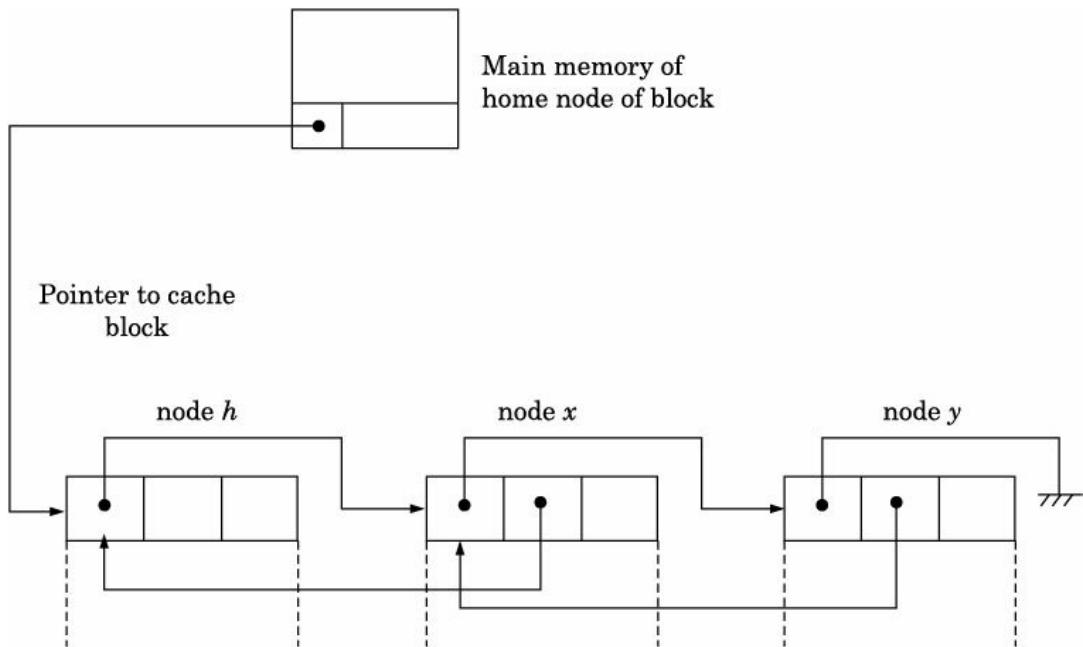


Figure 4.33 A distributed cache block directory using a doubly linked list.

Let us now examine how cache coherence is maintained. This is given in the following procedure:

Read Procedure-maintenance of cache coherence

Read command issued by processor p

1. Is word in p 's cache?
2. If yes, is it valid?
3. If valid read from local cache, else use link to traverse previous nodes till a node with a valid copy of the block is found. Read it and send to requesting node.
4. If word is not in the cache, send a message to home node. Does home node directory show any cached copy? If no, copy the block from the memory into requesting node's cache. Make it the head of the list and enter its address in the memory directory of home node.
5. If directory has address of cached copy, go to the address, retrieve block and copy it into the requesting cache (node p) block. Place the requesting cache at the head of the list and link it to the old head of list. Link old head backwards to node p 's cache block.

Write Procedure-maintenance of cache coherence

Write command issued by processor p

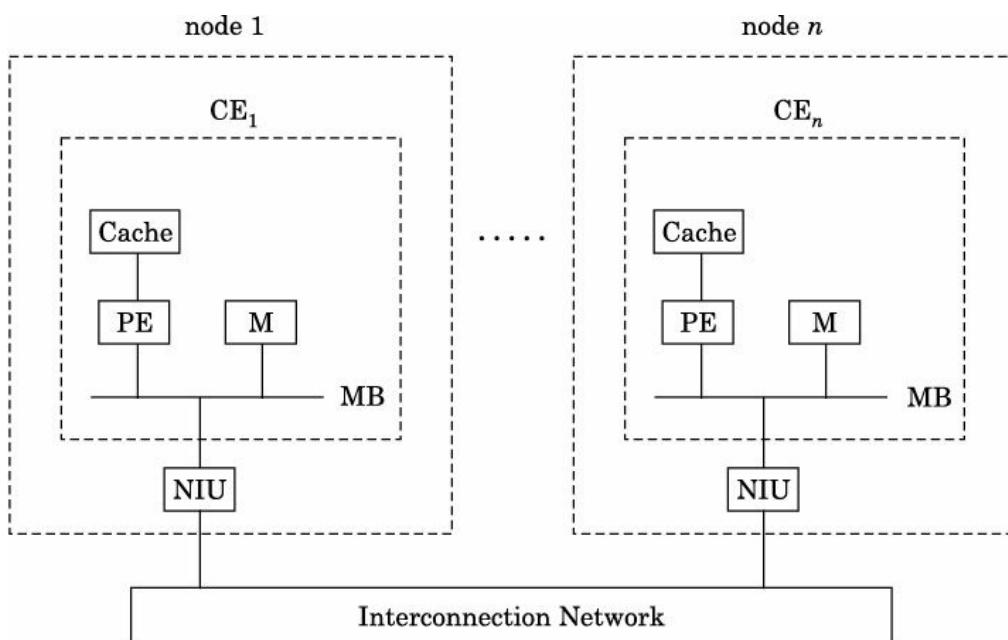
1. Is block in p 's cache?
2. If yes, update it and update home node's main memory block. Invalidate cache block held in all other nodes by traversing the linked list starting with p .
3. If block not in p 's cache, go to the main memory of home node. Read block and write into p 's cache by replacing some other cache block from it using specified replacement policy.
4. Find if home node has directory entry for this cache block. If yes, go to it and link it backwards to p 's cache block.
5. Make p 's cache block the head of the list.
6. Enter its address in home node directory.
7. Invalidate all copies of the cache block in the linked list by traversing the list starting at the head of the list.

The major disadvantage of this method is the large number of messages which have to be sent using the interconnection network while traversing the linked list. This is very slow. The main advantage is that the directory size is small. It holds only address of the node at the head of the list. The second advantage is that the latest transacting node is nearer to the head of the list. This ensures some fairness to the protocol. Lastly, invalidation messages are distributed rather than centralized thereby balancing the load. There are many other subtle problems which arise particularly when multiple nodes attempt to update a cache block simultaneously. Protocol for resolving such conflicts is given in detail in the IEEE standard [Gustavson, 1992]. It is outside the scope of this book.

4.10 MESSAGE PASSING PARALLEL COMPUTERS

A general block diagram of a message passing parallel computer (also called loosely coupled distributed memory parallel computer) is shown in Fig. 4.34. On comparing Fig. 4.32 and Fig. 4.34, they look identical except for the absence of a directory block in each node of Fig. 4.34. In this structure, each node is a full fledged computer. The main difference between distributed shared memory machine and this machine is the method used to program this computer. In DSM computer a programmer assumes a single flat memory address space and programs it using *fork* and *join* commands. A message passing computer, on the other hand, is programmed using *send* and *receive* primitives. There are several types of send-receive used in practice. We discuss a fairly general one in which there are two types of send, synchronous send and asynchronous send. There is one command to receive a message. A *synchronous send* command has the general structure:

Synch-Send (*source address, n, destination process, tag*)



PE : Processing Element

MB : Memory Bus

M : Local Memory

NIU : Network Interface Unit

CE : Computing Element

(Observe that each CE has its own private address space)

Figure 4.34 Message passing multicomputer.

A synchronous send is initiated by a source node. It is a command to send n bytes of data starting at *source address* to a destination process using tag as identifier of the message. After sending the message, the sender waits until it gets confirmation from the specified receiver (i.e., destination process). In other words, the sender suspends operation and waits. This is also called *blocked send*.

Another send command is:

Asynch-Send (*source address, n, destination process, tag*)

This is called *asynchronous send* instruction. It is initiated by a source node. It is a

command to send n bytes of data starting at *source address* to the specified destination process using tag as the identifier of the message. After sending the message, the sender continues with its processing and does not wait.

A receive command to receive and store a message is also required. The command is:

Receive (buffer-address, n, source process, tag)

Receives a message n bytes long from the specified source process with the tag and stores it in a buffer storage starting at *buffer-address*.

A blocked receive may also be implemented. In this case, the receiving process will wait until the data from the specified source process is received and stored.

If a programmer wants to make sure that an asynchronous send command has succeeded, a receiving process may issue a command *receive-probe (tag, source process)* which checks whether a message with the identifier tag has been sent by the source process. Similarly to check whether a message has been sent to a receiver and received by it a command

send-probe (tag, destination process)

may be issued which checks whether a message with identifier tag has been received by the destination process. These probe commands may be used with asynchronous send/receive.

Hardware support is provided in message passing systems to implement both synchronous and asynchronous send commands. The support is through appropriate communication protocols. The protocol used for a synchronous send is given in Fig. 4.35. This is called a *three phase protocol* as the communication is used thrice.

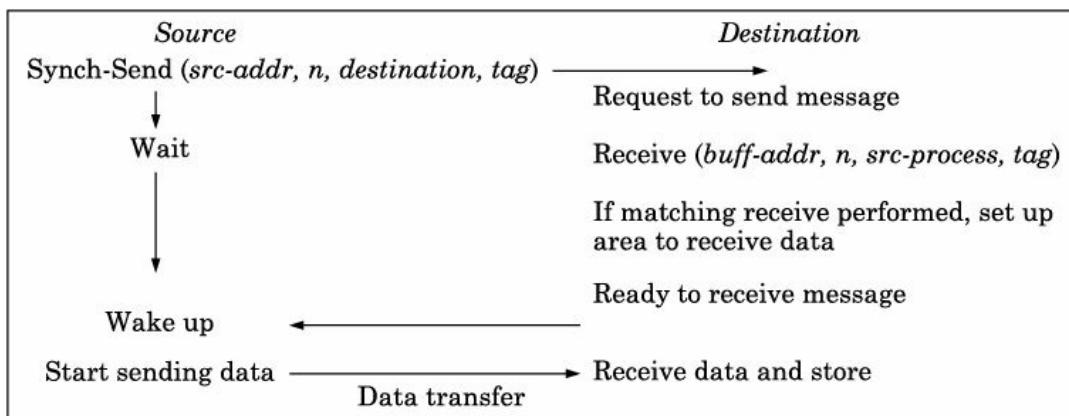


Figure 4.35 Synchronous message passing protocol.

In this protocol as soon as a source process executes a synch-send command, it sends a request to the destination computer (via the communication network) for permission to send the data. The request message is an “envelope” containing source address, number of bytes to be transmitted and a tag. After sending this request the sending process suspends its work and waits. The receiving process waits till a receive command matching the source process and tag is executed by the destination process. As soon as this command is executed, memory to receive data from the source is allocated. A message is sent back to the requesting source computer that the destination is ready to receive the data. When this message is received by the source, it “wakes up” and starts sending n bytes of data to the destination computer.

The major problem with synchronous send is that the sending process is forced to wait. There could also be a deadlock when two processes send messages to one another and each waits for the other to complete.

Asynchronous send is more popular for programming message passing systems. Hardware support is provided for implementing two communication protocols. One of them is called an

optimistic protocol and the other *pessimistic*. In the optimistic protocol, the source sends a message and assumes correct delivery. This protocol works as follows. The send transaction sends information on the source identity, length of data and tag followed by the data itself. The destination process (which must have a matching receive instruction) strips off the tag and source identity and matches it with its own internal table to see if a matching receive has been executed. If yes, the data is delivered to the address specified in the receive command. If no matching receive has been executed yet, the destination processor receives the message into a temporary buffer. When the matching receive is executed, the data from the temporary buffer is stored in the destination address specified by the receive command. The temporary buffer is then cleared.

This simple protocol has several problems. A major problem will occur if several processes choose to send data to the same receiver. Several buffers may have to be allocated and one may soon run out of buffer space. In theory an infinite buffer space is assumed. Another problem is the need to match tag and source process in order to find the address where the message is to be stored. This is a slow operation, typically performed by the software.

A more robust protocol is used by many systems which alleviates this problem at the receiver. This protocol is explained using Fig. 4.36. When asynchronous send is executed, it sends an “envelope” containing source process name, tag and number of bytes to be transmitted to the destination processor. After sending this envelope, the sending process continues processing. When the destination processor receives the envelope, it checks whether the matching receive has been executed. If yes, it sends a ready to receive message to sender. If matching receive has not been executed, then the receiving processor checks if enough buffer space is available to temporarily store the received data. If yes, it sends a ready to receive message to the sender. Otherwise it waits till such time that either buffer space is available or receive command is executed at which time a ready to receive message is sent. When “the ready to receive message” is received by the sender, it invokes the procedure to send data and transmits the data to the destination. The destination stores the data at the allocated space.

The main problem with this three phase protocol is the long delays which may be encountered in sending three messages using the communication network. This overhead may be particularly unacceptable if the data to be sent is a few bytes. In such a case the single phase protocol (explained earlier) may be used with some safety feature. One of them is to allocate a buffer space to each processor to receive short messages and subtract from this amount a quantity equal to the number of bytes stored. When the allocated space is exhausted, revert to three phase protocol.

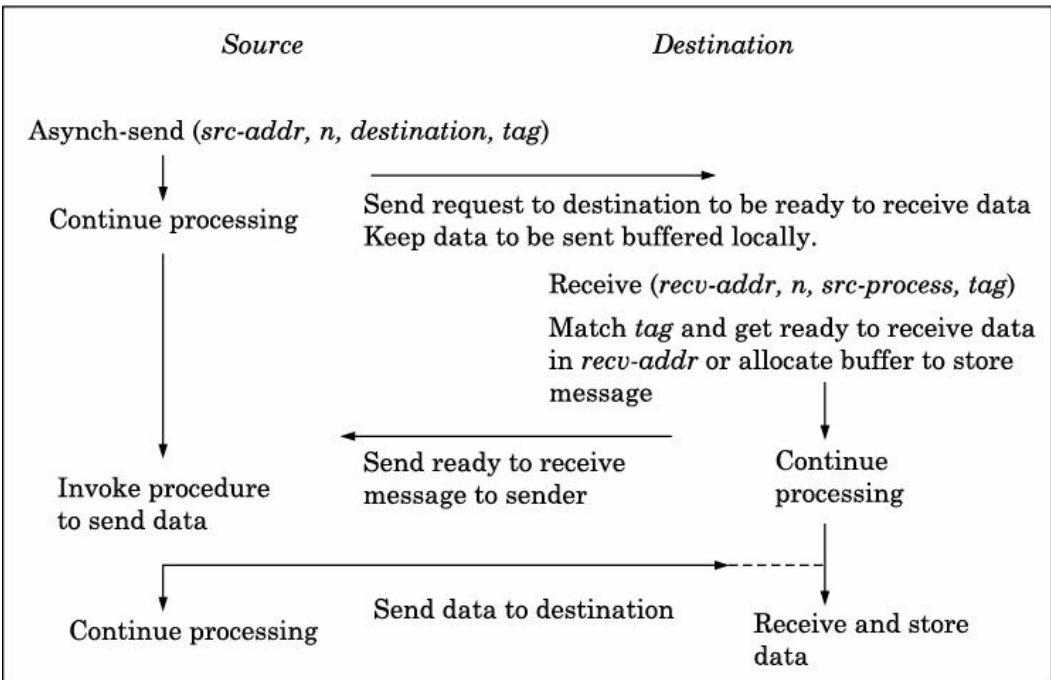


Figure 4.36 Asynchronous message passing protocol—A three phase conservative protocol.

We will write some message passing parallel programs using pseudo code in Chapter 8. The message passing instructions will be used are:

Blocked Send (*destination address*, *variable*, *size*)

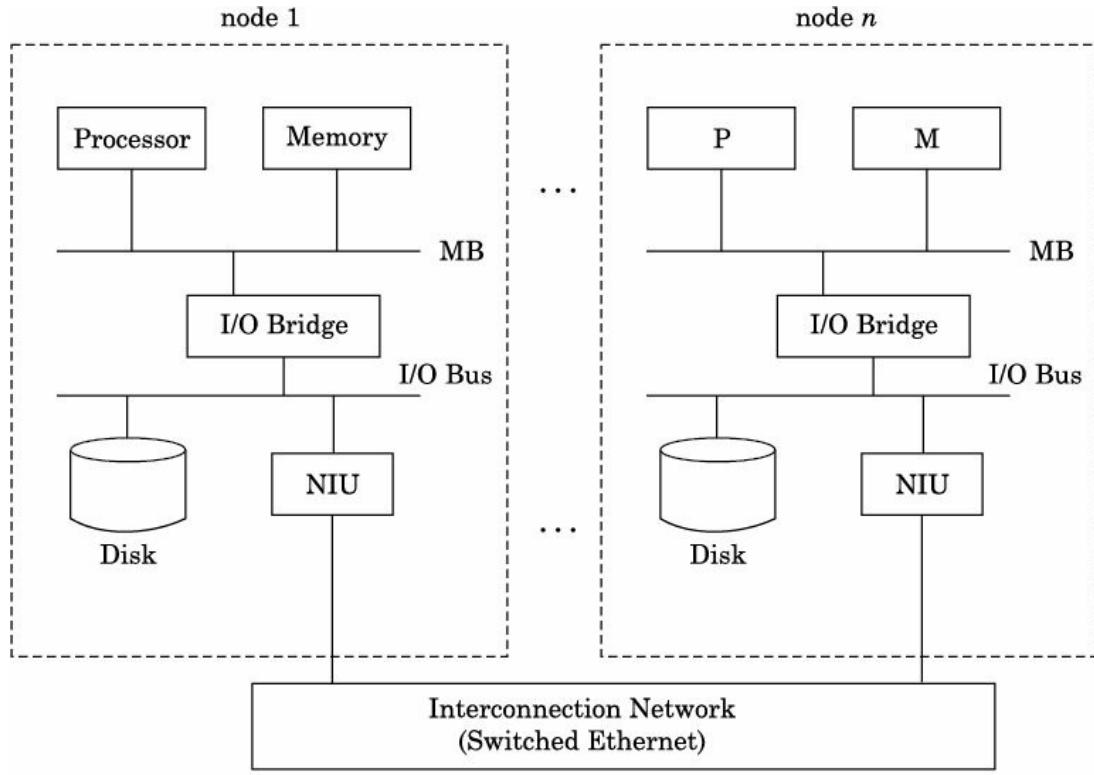
Send (*destination address*, *variable*, *size*)

In the above, *variable* is the same as *src-address*. Tag is not a part of the instruction as it could be supplied by the translator. However, in a message passing library called MPI (Message Passing Interface) which we will discuss in Chapter 8, tag is explicitly specified by the programmer to ensure that each send has a matching receive.

A large number of message passing systems use direct interconnection networks such as hypercube or toroid. The routers at each node manage the Send/Receive instructions and reduce time taken by these operations.

4.11 COMPUTER CLUSTER

The reduction in cost of microprocessors (which use several processors integrated in a chip) and consequent increase in their speed has drastically reduced the cost of high performance PCs/Servers. High performance PCs typically use several processors integrated in a chip, work at clock speeds of around 3 GHz, and have built-in network interface to a gigabit Ethernet. Many organizations have hundreds of high performance PCs. The questions which have been asked are: Is it possible to use an interconnected set of high performance PCs as a high speed parallel computer? Can such an interconnected set of high performance PCs be running compute intensive application as a parallel program in the background while allowing interactive use in the foreground? The answers to both the above questions are yes. A block diagram of such a system, called a Computer Cluster, is given in Fig. 4.37. Observe that the major difference between Fig. 4.37 and Fig. 4.34 of a message passing computer is that the NIU is connected to the I/O bus in the computer cluster whereas it is connected to the memory bus in the message passing multicomputer. Thus, it is clear that a computer cluster may be programmed using message passing method. However, the time taken to pass messages between nodes will be at least thousand times higher than in message passing parallel computers. The main reason for using the I/O bus, which is much slower than the memory bus to transmit messages is that it is standardized and can be used with several manufacturer's computers. When an I/O bus is used, a message to another computer will be taken as an I/O transaction requiring assistance from the operating system. Any transaction requiring operating system support will require several thousand clock cycles. Thus, running a parallel program which requires too many send/receive commands on computers connected by a Local Area Network (LAN) will not give good speedup. In other words, this structure will only be effective for SPMD style programming or programs in which the grain size is very coarse. As it is a very inexpensive method of building a parallel computer with no special hardware, many laboratories have developed software systems and parallel application programs to use such a system. They typically use Linux Operating System, a stripped down kernel based on Linux for managing inter-computer communication, a gigabit Ethernet connection and Message Passing Interface Library (see Chapter 8) for parallel programming. On some programs, this computer cluster is able to give gigaflops speed. Software called CONDOR [Litzkow, 1988] is widely used to "steal" the processing power of idle computers of the cluster to solve compute intensive problems.



MB : Memory Bus
P : Processor

NIU : Network Interface Unit
M : Memory

Figure 4.37 A computer cluster.

4.11.1 Computer Cluster Using System Area Networks

In the previous subsection, we described one low cost approach to parallel computing. One of the main problems with using a LAN as an interconnection network is the long communication delays in the network. LANs are primarily intended to connect large number of heterogeneous computers located far apart. LANs are designed for reliable communication over long distances in the presence of transmission errors. Multicomputer interconnection networks we discussed in Section 4.8, on the other hand, are for tightly connected computers packaged in a “box”. They are high bandwidth, high cost systems. A question which has been recently investigated is whether we can design a network connection in-between the two. The purpose of such a network is to be able to connect a small number of standard “off the shelf” computer boards located close together (a few metres) ensuring high speed, highly reliable communication with reduced communication overhead. In other words, the communication protocol should have few microseconds overhead to send and receive messages as opposed to milliseconds which are normal in LANs. Such a network is called a System Area Network (SAN) which is being increasingly used to build medium cost high performance parallel computers. This approach is being used by several computer manufacturers who have designed their own proprietary SANs.

4.11.2 Computer Cluster Applications

Apart from their use as high performance parallel computer, computer clusters are also used in the following applications.

1. Load distribution: A batch of programs may be allocated to a cluster computer with the primary objective of maximizing the throughput of the system. In such a case a scheduler is

designed to assess the requirements of the jobs in the batch and allocate them to computers in the cluster so that all computers are optimally utilized and the batch is completed in minimum time.

2. High availability system: In this mode if one of the nodes of the cluster fails, the program being executed on it may be shifted to another healthy computer. Another possible use is to run the same program in three or most systems and if their results do not match, declare as the result the one given by the majority of the systems. This ensures fault tolerance.

3. Web farm: Some web sites such as those maintained by the Indian Railways have a large number of customers logging on simultaneously. In such cases, web sites may be duplicated in many computers of the cluster and customers may be routed to an idle computer. Such farms may be elastic, in other words when there is a holiday season with a large number of travelers more computers may be added to the cluster. This is one example of a general class of parallel processing called *request level parallel computing*. In these problems, a system with a large number of independent computers is accessed by several users with independent jobs to be done. The jobs will be unrelated. Some may be enquiry on train schedules and others may be reservation requests. If the jobs are unrelated, there is no synchronization problem and the jobs can be allocated to individual computers in the system and a large number of users can be serviced simultaneously.

4. Rendering graphics: Displaying complex three-dimensional graphic images is compute intensive. The operations required for rendering may be partitioned into a sequence of operations which may be performed in several computers in a cluster.

4.12 WAREHOUSE SCALE COMPUTING

In the cluster computers we described in the last section, the number of computers is in the order of hundreds. In most cases the individual computers will be very powerful. Warehouse scale computers [Hennessy and Patterson, 2012] are also clusters but the number of computers is of the order of 50,000 to 100,000. Individual systems need not be very powerful. Such systems are primarily used for Internet based applications such as web search (e.g., Google), email, e-commerce (e.g., Amazon). They use request level parallelism. Most of these applications are interactive and the number of users is unpredictable. For example, e-commerce sites have to cater to a large number of customers just before festival seasons such as Deepawali or Christmas. Thus, the system should cater for the maximum demand and are overdesigned for average load. Another important requirement of Warehouse Scale Computers (WSC) is extremely high availability of the system (both hardware and software) of the order of 99.99%. A failure may adversely affect the reputation of the service and should be prevented. Other important requirements are:

1. In these systems each server is self-contained with on-board disk memory besides the main memory. As such systems have a large number of servers and the total cost of the system may be of the order of \$100 million, a small saving in the cost of each server will add up to several million dollars. Thus, the servers must be of low cost without sacrificing reliability to make the system cost effective.
2. The power dissipation of a server must be minimal commensurate with its capability to reduce the overall power consumption. If each server consumes 300 W, the power consumption of 100,000 servers will be 30 MW. A 50 W reduction in power consumption by a server will result in a saving of 5 MW. The system design involves proper cooling to dissipate the heat. Without proper cooling the performance of the system will be adversely affected.
3. The systems are accessed using the Internet. Thus, very high bandwidth network connection and redundant communication paths are essential to access these systems.
4. Even though many of the services are interactive good high performance parallel processing is also an important requirement. Web search involves searching many web sites in parallel, getting information appropriate for the search terms from them and combining them.
5. The architecture of WSC normally consists of around 48 server boards mounted in a standard rack inter-connected using low cost Ethernet switches. The racks are in turn connected by a hierarchy of high bandwidth switches which can support a higher bandwidth communication among the computers in the racks.
6. Single Program Multiple Data (SPMD) model of parallel computing is eminently suitable for execution on a WSC infrastructure as a WSC has thousands of loosely coupled computers available on demand. This model is used, for example, by search engines to find the URLs of websites in which a specified set of keywords occur. A function called Map is used to distribute sets of URLs to computers in the WSC and each computer asked to select the URLs in which specified keywords occur. All the computers search simultaneously and find the relevant URLs. The results are combined by a function called Reduce to report all the URLs where the specified keywords occur. This speeds up the search in proportion to the number of

computers in the WSC allocated for the job. An open source software suite called Hadoop MapReduce [Dean and Ghemawat, 2004] has been developed for writing applications on WSCs that process peta bytes of data in parallel, in a reliable fault tolerant manner which in essence uses the above idea.

4.13 SUMMARY AND RECAPITULATION

In this chapter we saw that there are seven basic types of parallel computers. They are vector computers, array processors, shared memory parallel computers, distributed shared memory parallel computers, message passing parallel computers, computer clusters, and warehouse scale parallel computers. Vector computers use temporal parallelism whereas array processors use fine grain data parallelism. Shared memory multiprocessors, message passing multicomputers, and computer clusters may use any type of parallelism. Warehouse scale computers are well suited for SPMD model of programming. Individual processors in a shared memory computer or message passing computer, for instance, can be vector processors. Shared memory computers use either a bus or a multistage interconnection network to connect a common global memory to the processors. Individual processors have their own cache memory and it is essential to maintain cache coherence. Maintaining cache coherence in a shared bus-shared memory parallel computer is relatively easy as bus transactions can be monitored by the cache controllers of all the processors [Dubois, et. al., 1988]. Increasing the processors in this system beyond 16 is difficult as a single bus will saturate degrading performance. Maintaining cache coherence is more difficult in a shared memory parallel computer using an interconnection network to connect a global memory to processors. A shared memory parallel computer using an interconnection network, however, allows a larger number of processors to be used as the interconnection network provides multiple paths from processors to memory. As shared memory computers have single global address space which can be uniformly accessed, parallel programs are easy to write for such systems. They are, however, not scalable to larger number of processors. This led to the development of Distributed Shared Memory (DSM) parallel computers in which a number of CEs are interconnected by a high speed interconnection network. In this system, even though the memory is physically distributed, the programming model is a shared memory model. DSM parallel computers, also known as CC-NUMA machines are scalable while providing a logically shared memory programming model.

It is expensive and difficult to design massively parallel computers (i.e., parallel machines using hundreds of CEs) using CC-NUMA architecture due to the difficulty of maintaining cache coherence. This led to the development of message passing distributed memory parallel computers. These machines do not require cache coherence as each CE runs programs using its own memory systems and cooperate with other CEs by exchanging messages. Message passing distributed memory parallel computers require good task allocation to CEs which reduce the need to send messages to remote CEs. There are two major types of message passing multicomputers. In one of them, the interconnection network connects the memory buses of CEs. This allows faster exchange of messages. The other type called a computer cluster interconnects CEs via their I/O buses. Computer clusters are cheaper. Heterogeneous high performance PCs can be interconnected as a cluster because I/O buses are standardized. Message passing is, however, slower as messages are taken as I/O transactions requiring assistance from the operating system. Computer clusters are, however, scalable. Thousands of CEs distributed on a LAN may be interconnected. With the increase in speed of high performance PCs and LANs, the penalty suffered due to message passing is coming down making computer clusters very popular as low cost, high performance, parallel computers. Another recent development driven by widely used services such as email, web search, and e-commerce is the emergence of warehouse scale computers which use tens of thousands of servers interconnected by Ethernet switches connected to the computers' network interface

cards. They are used to cater to request level parallel jobs. They are also used to solve SPMD style of parallel programming. In Table 4.10 we give a chart comparing various parallel computer architectures we have discussed in this chapter.

TABLE 4.10 Comparison of Parallel Computer Architectures				
<i>Type of parallel computer</i>	<i>Criteria for comparison</i>			
	<i>Type of parallelism exploited</i>	<i>Suitable task size</i>	<i>Programming ease</i>	<i>Scalability</i>
Vector computers	Temporal	Fine grain	Easy	Very good
Array processors	Data parallelism (SIMD)	Fine grain	Easy	Very good
Shared Memory using bus	MIMD	Medium grain	Relatively easy due to availability of global address space	Poor
Shared memory using interconnection network	MIMD	Medium grain	Relatively easy	Moderate
Distributed shared memory CC-NUMA	MIMD	Medium grain	Same as above	Good
Message passing multi-computer	MIMD	Coarse	Needs good task allocation and reduced inter-CE communication	Very good
Cluster computer	MIMD	Very coarse as O.S. assistance needed	Same as above	Very good (low cost system)
Warehouse scale computer (WSC)	SPMD	Very coarse	Same as above	Excellent (Uses tens of thousands of low cost servers)

EXERCISES

- 4.1 We saw in Section 4.1 that a parallel computer is made up of interconnected processing elements. Can you explain how PEs can be logically inter-connected using a storage device? How can they cooperate with this logical connection and solve problems?
- 4.2 Parallel computers can be made using a small number of very powerful processors (e.g., 8-processor Cray) or a large number of low speed micro-processors (say 10,000 Intel 8086 based microcomputers). What are the advantages and disadvantages of these two approaches?
- 4.3 We gave a number of alternatives for each of the components of a parallel computer in Section 4.1. List as many different combinations of these components which will be viable parallel computer architecture (Hint: One viable combination is 100 full fledged microprocessors with a cache and main memory connected by a fixed interconnection network and cooperating by exchanging intermediate results).
- 4.4 For each of the alternative parallel computer structure you gave as solution to Exercise 4.3, classify them using Flynn's classification.
- 4.5 Give some representative applications in which SIMD processing will be very effective.
- 4.6 Give some representative applications in which MISD processing will be very effective.
- 4.7 What is the difference between loosely coupled and tightly coupled parallel computers? Give one example of each of these parallel computer structures.
- 4.8 We have given classifications based on 4 characteristics. Are they orthogonal? If not what are the main objectives of this classification method?
- 4.9 When a parallel computer has a single global address space is it necessarily a uniform memory access computer? If not explain why it is not necessarily so?
- 4.10 What do you understand by grain size of computation? A 4-processor computer with shared memory carries out 100,000 instructions. The time to access shared main memory is 10 clock cycles and the processors are capable of carrying out 1 instruction every clock cycle. The main memory need to be accessed only for inter-processor communication. If the loss of speedup due to communication is to be kept below 15%, what should be the grain size of computation?
- 4.11 Repeat Exercise 4.10 assuming that the processors cooperate by exchanging messages and each message transaction takes 100 cycles.
- 4.12 A pipelined floating point adder is to be designed. Assume that exponent matching takes 0.1 ns, mantissa alignment 0.2 ns, adding mantissas 1 ns and normalizing result 0.2 ns. What is the highest clock speed which can be used to drive the adder? If two vectors of 100 components each are to be added using this adder, what will be the addition time?
- 4.13 Develop a block diagram for a pipelined multiplier to multiply two floating point numbers. Assuming times for each stage similar to the ones used in Exercise 4.12, determine the time to multiply two 100 component vectors.
- 4.14 What is vector chaining? Give an example of an application where it is useful.
- 4.15 If the access time to a memory bank is 8 clock cycles, how many memory banks are needed to retrieve one component of a 64-component vector each cycle?
- 4.16 A vector machine has a 6-stage pipelined arithmetic unit and 10 ns clock. The time required to interpret and start executing a vector instruction is 60 ns. What should be the

length of vectors to obtain 95% efficiency on vector processing?

4.17 What are the similarities and differences between

- (i) Vector processing,
- (ii) Array processing, and
- (iii) Systolic processing?

Give an application of each of these modes of computation in which their unique characteristics are essential.

4.18 Obtain a systolic array to compute the following:

```
for i := 5 to 1 with step size of -1 do
    for j := 1 to 4 with step size of 1 do
        Yi ← wi ×(i+j)
    end for
end for
```

4.19 A shared bus parallel computer with 16 PEs is to be designed using 32-bit RISC processor running at 2 GHz. Assume average 1.5 clocks per instruction. Each PE has a cache and the hit ratio to cache is 98%. The system is to use write-invalidate protocol. If 10% of the instructions are loads and 15% are stores, what should be the bandwidth of the bus to ensure processing without bus saturation? Assume reasonable values of other necessary parameters (if any).

4.20 In Section 4.7.1, we stated that an atomic read-modify-write machine level instruction would be useful to implement barrier synchronization. Normally a counter is decremented by each process which reaches a barrier and when the count becomes 0, all processes cross the barrier. Until then processors wait in busy wait loops. Write an assembly language program with a read-modify-write atomic instruction to implement barrier synchronization.

4.21 Assume a 4-PE shared memory, shared bus, cache coherent parallel computer. Explain how lock and unlock primitives work with write-invalidate cache coherence protocol.

4.22 Develop a state diagram or a decision table to explain MOESI protocol for maintaining cache coherence in a share memory parallel computer.

4.23 A 4-PE shared bus computer executes the streams of instructions given below:

Stream 1 : (R PE1)(R PE2)(W PE1)(W PE2)(R PE3)(W PE3)(R PE4)(W PE4) Stream 2 : (R PE1)(R PE2)(R PE3)(R PE4)(W PE1)(W PE2)(W PE3)(W PE4) Stream 3 : (R PE1)(W PE1)(R PE2)(W PE2)(R PE3)(W PE3)(R PE4)(W PE4) Stream 4 : (R PE2)(W PE2)(R PE1)(R PE3)(W PE1)(R PE4)(R PE3)(W PE4)

Where (R PE1) means read from PE1 and (W PE1) means write into PE1 memory. Assume that all caches are initially empty. Assume that if there is a cache hit the read/write time is 1 cycle and that all the read/writes are to the same location in memory. Assume that 100 cycles are needed to replace a cache block and 50 cycles for any transaction on the bus. Estimate the number of cycles required to execute the streams above for the following two protocols.

- (i) MESI protocol
- (ii) Write update protocol

4.24 List the advantages and disadvantages of shared memory parallel computers using a bus for sharing memory and an interconnection network for sharing memory.

4.25 We have given detailed protocols to ensure cache coherence in directory based shared memory systems assuming a write-invalidate policy and write-through to main memory.

Develop a detailed protocol for a write-update policy and write-back to main memory. Compare and contrast the two schemes. Specify the parameters you have used for the comparison.

- 4.26 A shared memory parallel computer has 128 PEs and shares a main memory of 1 TB using an interconnection network. The cache block size is 128B. What is the size of the directory used to ensure cache coherence? Discuss methods of reducing directory size.
- 4.27 Solve Exercise 4.19 for a directory based shared memory computer. In the case of directory scheme the solution required is the allowed latency in clock cycles of the interconnection network for “reasonable functioning” of the parallel computer.
- 4.28 A parallel computer has 16 processors which are connected to independent memory modules by a crossbar interconnection network. If each processor generates one memory access request every 2 clock cycles, how many memory modules are required to ensure that no processor is idle?
- 4.29 What is the difference between a blocking switch and a non-blocking switch? Is an omega network blocking or non-blocking?
- 4.30 If switch delay is 2 clock cycles, what is the network latency of an n -stage omega network?
- 4.31 Obtain a block diagram of a 16-input 16-output general interconnection network similar to the one shown in Fig. 4.25.
- 4.32 What is the difference between a direct and an indirect network?
- 4.33 What is the significance of bisection bandwidth? What is the bisection bandwidth of a 3-stage omega network?
- 4.34 Draw a diagram of a 32-node hypercube. Number the nodes using systematic binary coding so that processors which are physical neighbours are also logical neighbours. What is the bisection bandwidth? How many links are there in this hypercube? What is the maximum number of hops in this network? How many alternate paths are there between any two nodes?
- 4.35 What is packet switching? Why is it used in CE to CE communication in direct connected networks?
- 4.36 What do you understand by routing? What is the difference between direct packet routing, virtual cut through routing and wormhole routing?
- 4.37 Distinguish between UMA, NUMA and CC-NUMA parallel computer architectures. Give one example block diagram of each of these architectures. What are the advantages and disadvantages of each of these architectures? Do all of these architectures have a global addressable shared memory? What is the programming model used by these parallel computers?
- 4.38 Take the data of Example 4.4 given in the text except the bandwidth of the interconnection network. If 15% of the accesses are to remote CEs, what should be the bandwidth of the interconnection network to keep the penalty due to remote access below 2?
- 4.39 A NUMA parallel computer has 64 CEs. Each CE has 1 GB memory. The word length is 64 bits. A program has 500,000 instructions out of which 20% are loads and 15% stores. The time to read/write in local memory is 2 cycles. An overhead of 10 clock cycles is incurred for every remote transaction. If the network bandwidth is 500 MB/s and 30% of the accesses are to remote computers, what is the total load/store time? How much will it be if all accesses are to local memory?

- 4.40 Obtain an algorithm to ensure cache coherence in a CC-NUMA parallel computer. Assume write-invalidate protocol and write through caches.
- 4.41 A CC-NUMA parallel computer has 128 CEs connected as a hypercube. Each CE has 256 MB memory with 32 bits/word. Link bandwidth is 500 MB/s. Local load/store take 1 clock cycle. Fixed overhead for remote access is 50 cycles. Each hop from link to next link costs extra 5 cycles. Answer the following questions:
- (i) Describe the directory organization in this machine.
 - (ii) If a load from the memory of PE 114 to PE 99's register is to be performed, how many cycles will it take?
- 4.42 Compare and contrast directory based cache coherence scheme and scalable coherent interface standard based scheme.
- 4.43 A 64-PE hypercube computer uses SCI protocol for maintaining cache coherence. Statistically it is found that not more than 4 caches hold the value of a specified variable. Memory size at each node is 256 MB with 32-bit words. Cache line size is 128 bytes. Link bandwidth is 500 MB/s. If a read from Node x requires reading from Node $(x + 4)$'s cache as it is not in x 's cache, how many cycles does it take? Access time to main memory at each node is 10 cycles and to local cache 1 cycle. Assume reasonable value(s) for any other essential parameter(s).
- 4.44 What are the major differences between a message passing parallel computer and a NUMA parallel computer?
- 4.45 A series of 10 messages of 64 bytes each are to be sent from node 5 to 15 of a 32-CE hypercube. Write synchronous send commands to do this. If start up to send a message is 10 cycles, link latency 15 cycles and memory cycle is 5 cycles, how much time (in cycles) will be taken by the system to deliver these 10 messages?
- 4.46 Repeat Exercise 4.45 if it is asynchronous send. What are the advantages (if any) of using an asynchronous send instead of a synchronous send? What are the disadvantages?
- 4.47 What are the advantages and disadvantages of cluster computer when compared with a message passing parallel computer connected by a hypercube?
- 4.48 A program is parallelized to be solved on a computer cluster. There are 16 CEs in the cluster. The program is 800,000 instructions long and each CE carries out 500,000 instructions. Each CE has a 2 GHz clock and an average of 0.75 instructions are carried out each cycle. A message is sent by each CE once every 20,000 instructions on the average. Assume that all messages do not occur at the same time but are staggered. The fixed overhead for each message is 50,000 cycles. The average message size is 1,000 bytes. The bandwidth of the LAN is 1 GB/s. Calculate the speedup of this parallel machine.
- 4.49 What are the advantages of using a NIC (which has a small microprocessor) in designing a computer cluster? Make appropriate assumptions about NIC and solve Exercise 4.48. Assume that the network bandwidth remains the same.
- 4.50 Repeat Exercise 4.48 assuming that the LAN bandwidth is 10 GB/s. Compare the results of Exercises 4.48 and 4.50 and comment.
- 4.51 What are the differences between computer clusters and warehouse scale computers? In what applications are warehouse scale computers used?
- 4.52 Differentiate between request level parallelism and data level parallelism. Give some examples of request level parallelism.

4.53 A warehouse scale computer has 10,000 computers. Calculate the availability of the system in a year if there is a hardware failure once a month and software failure once in fifteen days. Each hardware failure takes 15 minutes to rectify (assuming a simple replacement policy) and a software failure takes 5 minutes to rectify by just rebooting. If an availability is to be 99.99%, what would be the allowed failure rates.

BIBLIOGRAPHY

- Adve, S.N., and Gharachorloo, K., "Shared Memory Consistency Models: A Tutorial", *IEEE Computer*, Vol. 29, No. 12, Dec. 1996, pp. 66–76.
- Censier, L. and Feautrier, P., "A New Solution to Cache Coherence Problems in Multiprocessor Systems", *IEEE Trans. on Computers*, Vol. 27, No. 12, Dec. 1978, pp. 1112–1118.
- Culler, D.E., Singh, J.P. and Gupta, A., *Parallel Computer Architecture: A Hardware Software Approach*, Morgan Kaufmann, San Francisco, USA, 1999.
- Dean, J., and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters", *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2004, pp. 137–150, <http://research.google.com/archive/mapreduce.html>
- DeCegama, A.L., *Parallel Processing Architectures and VLSI Hardware*, Vol. 1, Prentice-Hall Inc., Englewood Cliffs, New Jersey, USA, 1989.
- Dubois, M., et al., "Synchronization, Coherence, and Event Ordering", *IEEE Computer*, Vol. 21, No. 2, Feb. 1988, pp. 9–21.
- Flynn, M.J., "Some Computer Organizations and their Effectiveness", *IEEE Trans. on Computers*, Vol. 21, No. 9, Sept. 1972, pp. 948–960.
- Gustavson, D., "The Scalable Coherence Interface and Related Standards", *IEEE Micro*, Vol. 12, No. 1, Jan. 1992, pp. 10–12.
- Hennessy, J.L., and Patterson, D.A., *Computer Architecture—A Quantitative Approach*, 5th ed., Morgan Kauffman, Waltham, MA, USA, 2012.
- Hill, M.D., "Multiprocessors should Support Simple Memory Consistency Models", *IEEE Computer*, Vol. 31, No. 8, Aug. 1998, pp. 28–34.
- Hord, M.R., *Parallel Supercomputing in SIMD Architectures*, CRC Press, Boston, USA, 1990.
- Kung, H.T., "Why Systolic Architectures"? *IEEE Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37–46.
- Lamport, L., "How to Make Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Trans. on Computers*, Vol. 28, No. 9, Sept. 1979, pp. 690–691.
- Leighton, F.T., *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Francisco, USA, 1992.
- Litzkow, M., Livny, M. and Mutka, M.W., "CONDOR-A Hunter of Idle Workstations", *Proceedings of IEEE International Conference of Distributed Computing Systems*, June 1988, pp. 104–111.
- Mahapatra, P., "Wormhole Routing Techniques for Directly Connected Multicomputer Networks", *ACM Computing Surveys*, Vol. 30, No. 3, Sept. 1998, pp. 374–410.
- Nakazato, S., et al., "Hardware Technology of the SX-9", *NEC Technical Journal*, Vol. 3, No. 4, 2008.
- Rajaraman, V., *Supercomputers*, Universities Press, Hyderabad, 1999.
- Sorin, D.J., Hill, M.D., and Wood, D.A., *A Primer on Memory Consistency and Cache Coherence*, Morgan and Claypool, USA, 2011.
- Stenstrom, P., "A Survey of Cache Coherence Protocols for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, June 1990, pp. 12–29.

Core Level Parallel Processing

In Chapter 3, we saw how the speed of processing may be improved by perceiving instruction level parallelism in programs and exploiting this parallelism by appropriately designing the processor. Three methods were used to speedup processing. One was to use a large number of pipeline stages, the second was to initiate execution of multiple instructions simultaneously which was facilitated by out of order execution of instructions in a program, and the third was to execute multiple threads in a program in such a way that all the sub units of the processor are always kept busy. (A thread was defined as a small sequence of instructions that shares the same resources). All this was done by the processor hardware automatically and was hidden from the programmer. If we define the performance of a processor as the number of instructions it can carry out per second then it may be increased (i) by pipelining which reduces the average cycle time needed to execute instructions in a program, (ii) by increasing the number of instructions executed in each cycle by employing instruction level parallelism and thread level parallelism, and (iii) by increasing the number of clock cycles per second by using higher clock speed. Performance continually increased up to 2005 by increasing the clock speed, exploiting to the maximum extent possible pipelining, instruction and thread level parallelism.

While research and consequent implementation of thread level parallelism to increase the speed of processors was in progress, researchers were examining how several processors could be interconnected to work in parallel. Research in parallel computing started in the late 70s. Many groups and start up companies were active in building parallel machines during the 80s and 90s upto early 2000. We discussed the varieties of parallel computers that were built and their structure in Chapter 4. The parallel computers we described in Chapter 4 were not used as general purpose computers such as desktop computers and servers due to the fact that single processor computers were steadily increasing in speed with no increase in cost. This was due to the increase in clock speed as well as the use of multithreading in single processor computers which led to their steady increase in speed. Parallel computer programming was not easy and their performance was not commensurate with their cost for routine computing. They were only used where high performance was needed in numeric intensive computing in spite of their many advantages which we pointed out in Chapter 1. However, as we will explain in the next section, the performance of single chip processors reached saturation around 2005. Chip designers started examining integrating many processors on a single chip to continue to build processors with better performance. When this happened many of the ideas we presented in Chapter 4 were resurrected to build parallel computers using many processors or *cores* integrated on a single silicon chip. The objective of this chapter is to describe parallel computers built on a single integrated circuit chip called Chip Multiprocessors (CMP) which use what we call *core level parallelism* to speedup execution of programs. While many of the architectural ideas described in the previous chapter are applicable to chip multiprocessors, there are important differences which we discuss in the next section.

5.1 CONSEQUENCES OF MOORE'S LAW AND THE ADVENT OF CHIP MULTIPROCESSORS

In 1965 Moore had predicted, based on the data available at that time, that the number of transistors in an integrated circuit chip will double every two years. This is an empirical prediction (called Moore's law) which has been surprisingly accurate. Starting with about 1000 transistors in a chip in the 70s by the year 2014 there were almost 10 billion transistors in a chip. The increase in the number of transistors allowed processor designers to implement many innovations culminating in complex pipelined, superscalar, multithreaded processor which exploited to the maximum extent the available instruction level parallelism. Simultaneously the clock frequency was being increased. The increase in clock speed was sustainable till 2003. Beyond a clock frequency of around 4 GHz, the heat dissipation in the processors increased. The faster the transistors are switched the higher is the heat dissipated. There are three components of heat dissipation. They are:

1. When transistors switch states, their equivalent capacitances charge and discharge leading to power dissipation. This is the *dynamic power dissipation* which is proportional to $CV_d^2 f$, where, C is the equivalent capacitance of the transistor, V_d the voltage at which the transistor operates and f the clock frequency.
2. Small currents called *leakage current* flow continuously between different doped parts of a transistor. This current increases as the transistor becomes smaller and the ambient temperature goes up. This leakage current I_L also leads to higher power consumption proportional to $V_d I_L$ and a resultant increase of heat dissipation.
3. Lastly during the finite time a logic gate toggles, there is a direct path between the voltage source and ground. This leads to *short circuit current* I_{st} and consequent power dissipation proportional to $V_d I_{st}$. This in turn heats up the gates.

These three sources of power dissipation and the consequent heat generation increases when transistor size decreases. However, the dynamic power loss due to the increase of the frequency of switching causes the greatest increase in heat generation. This puts a limit to the frequency at which processors operate. This is called *frequency wall* in the literature. As was pointed out earlier when the clock frequency reached 4 GHz, processor designers had to look for other means of increasing the speed of processors, rather than brute force increase in clock speed which required special cooling systems. As Moore's law was still valid and more transistors could be packed in a chip, architects had to find ways of using these transistors.

Simultaneous multithreading uses the extra transistors. However, instruction level parallelism had its limitations. Even if many arithmetic elements are put in one processor, there was not enough instruction level parallelism available in threads to use them. This upper bound on the number of instructions which can be executed simultaneously by a processor is called the *ILP wall* in the literature. Thus, instead of using the extra transistors to increase the number of integer units, floating point units, and registers which reached a point of diminishing returns, architects had to explore other ways of using the extra transistors. Further as the complexity of the processor increased design errors crept in, debugging complex processors on a single chip with too many transistors was difficult and expensive.

There are two other ways of using transistors. One is to increase the size of the on-chip memory called the L1 cache and the other is to put several processors in a single chip and

make them cooperate to execute a program. There is a limit beyond which increasing the L1 cache size has no advantage as there is an upper limit to locality of reference to memory. Putting n processors in a chip, however, can very often lead to n -fold speedup and in some special cases even super-linear speedup.

Starting in the late 70s, many research groups had been working on inter-connecting independent processors to architect parallel computers which we discussed in the previous chapter. Much of the research could now be used to design single chip parallel computers called multicore processors or chip multiprocessors. However, there are important differences between the architectural issues in designing parallel computers using independent processing units and designing *Chip Multiprocessors* (CMP). These primarily relate to the fact that individual cores heat up at high clock frequency and may affect the performance of physically adjoining cores. Heat sensors are often built as part of a core to switch off the core, or reduce its operating clock frequency, or switch the thread being processed to another core which may be idle. Another problem arises due to threads or processes running on different cores sharing an on-chip cache memory leading to contention and performance degradation unless special care is taken both architecturally and in programming. The total power budget of a chip has to be apportioned to several cores. Thus, the power requirement of each core has to be examined carefully. There are, however, many advantages of CMPs compared to parallel computers using several computers. As the processors are closely packed, the delay in transferring data between cooperating processors is small. Thus, very light weight threads (i.e., threads with very small number of instructions) can run in the processors and communicate more frequently without degrading performance. This allows the software designers to re-think appropriate algorithms for CMPs. There is also a downside. Whereas in parallel computers wires interconnecting computers for inter-computer communication were outside the computer and there was no constraint on the number of wires. Even though the wires inside a chip are short, they occupy chip area and thus their number has to be reduced. The switches used for inter-processor communication also use chip space and their complexity has to be reduced. The trade-offs are different in CMPs as compared to parallel computers and this has to be remembered by architects.

As cores are duplicated, the design of a single core may be replicated without incurring extra design cost. Besides this, depending on the requirement of the application, the power budget, and the market segment, processors may be tailor made with 2, 4, 6, 8 or many processors as duplicating processors on a die for special chips do not incur high cost.

The terminology *processing core* is used to describe an independent Processing Element (PE) in an integrated circuit chip. When many PEs are integrated in a chip, it is called a multicore chip or a chip multiprocessor as we pointed out earlier. The independent processing cores normally share common on-chip cache memory and use parallelism available in programs to solve problems in parallel. You may recall that we called it *core level parallelism*. In this chapter, we will describe single chip parallel computers which use core level parallelism.

Unlike multithreaded processors in which the hardware executes multiple threads without a programmer having to do anything, multicore processors require careful programming. As the number of cores per processor is increasing rapidly programming becomes more difficult. We discuss programming multicore processors in Chapter 8. In Table 5.1, we summarise the discussions of this section.

TABLE 5.1 Motivation for Multicore Parallelism

Bounds/Walls	Reasons	Solutions
--------------	---------	-----------

Instruction level parallelism (ILP wall)	<ul style="list-style-type: none"> Non availability of parallelism beyond an upper bound on the instructions that can be issued simultaneously Design complexity of chips and compilers 	<ul style="list-style-type: none"> Use multiple threads with multiple cores. Application such as pixel processing amenable to SIMD processing and can support an enormous number of parallel threads
Power dissipated (Power wall)	<ul style="list-style-type: none"> Heating due to increase in frequency of clock Increase in leakage current as transistor size becomes smaller Short circuit power loss during switching 	<ul style="list-style-type: none"> Reduce frequency Reduce V_d Switch off unused cores/subsystems Transistor technology improvement
Clock frequency	<ul style="list-style-type: none"> Heating as frequency increases Power consumption increases as frequency increases Transmission line effects at higher frequencies Electromagnetic radiation at higher frequencies 	<ul style="list-style-type: none"> Use multiple slower cores in a chip Use heterogeneous cores
Memory	<ul style="list-style-type: none"> Working set size does not increase beyond a point based on locality of reference Memory bandwidth lower than CPU bandwidth On-chip memory requires large area and dissipates power 	<ul style="list-style-type: none"> Use distributed cache hierarchy L1, L2, L3 caches Shared memory multiprocessor with caches in each processor Multiple processors can use private caches better
Design complexity	<ul style="list-style-type: none"> Design verification of complex processors Architectural innovation Hardware software co-design 	<ul style="list-style-type: none"> Reuse earlier design by replication facilitated by duplicating well designed cores
Improvement in performance	<ul style="list-style-type: none"> Increase in performance of processors proportional to square root of increase in complexity. Doubling logic gates gives 40% increase in performance 	<ul style="list-style-type: none"> Use multiple cores with simple logic in each core. Promise of linear increase in performance

5.2 A GENERALIZED STRUCTURE OF CHIP MULTIPROCESSORS

Following the model we used in Chapter 4, we define a chip multiprocessor (CMP) as:

“An interconnected set of processing cores integrated on a single silicon chip which communicate and cooperate with one another to execute one or more programs fast”.

We see from the above definition that the key words are: *processing cores*, *single silicon chip*, *communication*, and *cooperation*. It should also be noted that a CMP may be used to execute a number of independent programs simultaneously.

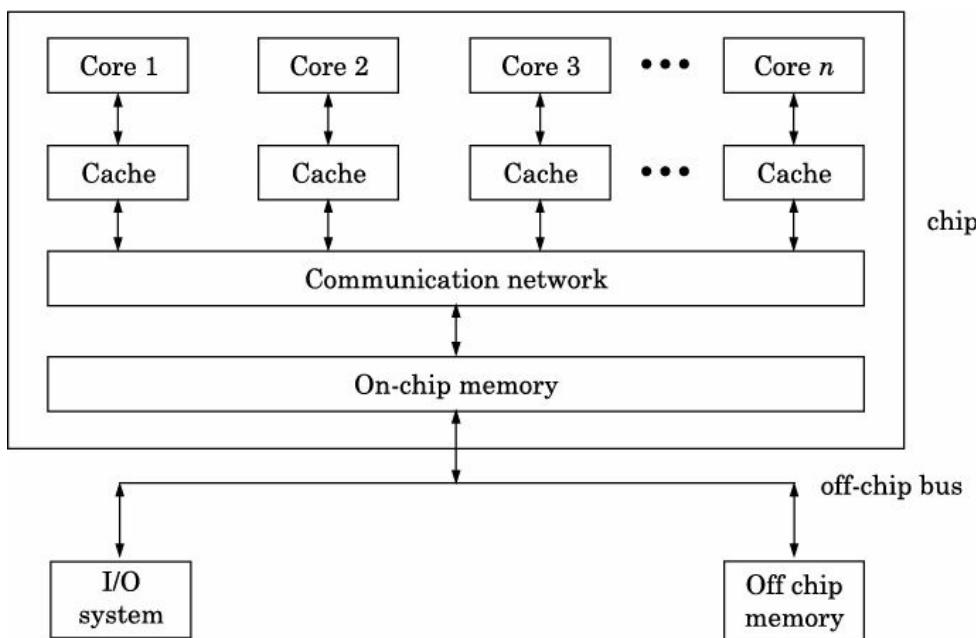


Figure 5.1 A generalized structure of a chip multiprocessor.

In Fig. 5.1, we show a generalized structure of a CMP. The heart of a CMP is a set of processing cores, abbreviated cores, interconnected by a communication network. The general structure given in Fig. 5.1 can have many variations (similar to parallel computers) based on the type of cores, the memory system used by the cores, the type of communication system interconnecting the cores, and how the cores communicate and cooperate. The variations (just like in parallel computers) of each of these lead to a variety of CMPs. The possibilities for each of these crucial components of a CMP are:

Types of Processing Cores

1. A single threaded simple processor.
2. A multithreaded, superscalar processor.
3. A mixture of simple and multithreaded processors.
4. A processor which is an ALU which adds and multiplies.
5. Some general purpose simple processors and processor(s) suitable for digital signal processing.

Number of Processing Cores

1. A small number of identical cores (<10).
2. A medium number of identical cores (>10 and <100).

3. A large number of identical tiny cores (> 1000).

Memory System

1. Each core has an L1 cache. All cores share an L2 cache.
2. Each core has both an L1 and L2 cache. All cores share an on-chip L3 cache.
3. All cores access an external main memory in addition to local caches.

Communication Network

1. A bus, or a ring bus interconnecting cores (with caches).
2. A fixed interconnection network such as a ring, grid, or crossbar switch with appropriate routers.

Mode of Cooperation

1. Each core is assigned an independent task. All the cores work simultaneously on assigned tasks. This is called *request level parallel processing*.
2. All cores have a single program running but on different data sets (called *data level parallelism*, or SPMD).
3. All cores execute a single instruction on multiple data streams or threads (called *single instruction multiple data/thread parallelism*, or SIMD/SIMT).
4. All threads and data to the processor are stored in the memory shared by all the cores. A free core selects a thread to execute and deposits the results in the memory for use by other cores (called multiple instructions multiple data style of computing, or MIMD).

From the above description, we see that a rich variety of CMPs can be built. We will describe some of the chip multiprocessors which have been built.

The main considerations which have to be remembered while designing CMPs and many core processors (when processors exceed 30) are:

1. The applications for which the system is built.
2. The total power budget for the chip (which is normally not more than around 200 W) if the chip is to be air cooled.
3. Avoiding hot spots and spreading the heat dissipation evenly.
4. Dynamic voltage and frequency scaling to avoid heating of cores.
5. On-chip interconnection networks should preferably be planar as the interconnecting wires are normally put on top of the processors on a plane. Thus bus, ring, crossbar switch, and grid are suitable whereas hypercube and other 3D interconnection systems are not suitable.
6. Design cost of CMP is reduced if a single module (a core, its cache, and connection system) is replicated. In other words, a system which replicates a module uniformly is easier to design and verify.
7. Interconnecting wires between cores are normally short as compared to wires connecting separate computers in multicomputers. Thus, latency is small. An interconnection system requiring long lines will have reduced bandwidth as resistance and capacitance values will increase.
8. The processor cores should be modified to include instructions to fork, lock,

unlock, and join, as described in Section 4.7.1. In message passing systems including send/receive primitives in the instruction set of the cores will be useful.

5.3 MULTICORE PROCESSORS OR CHIP MULTIPROCESSORS (CMPs)

A block diagram of a single core processor is shown in Fig. 5.2. This single core processor can range from a simple RISC processor to a complex multithreaded superscalar processor. This processor has besides register files, multiple execution units and appropriate controlling circuits, a built-in cache memory of the order of 32 KB, called level 1 (L1) cache. Besides these the chip has a bus interface to connect to an external system bus. As we pointed out earlier, the clock speed has to be of the order of 2 to 4 GHz to avoid overheating of the core.

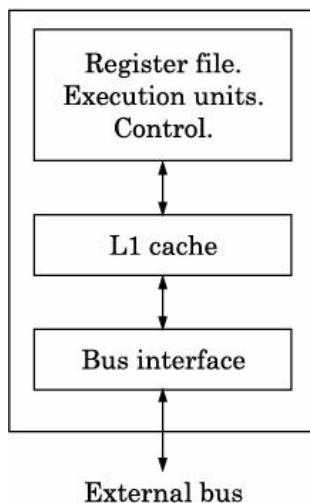


Figure 5.2 A single core processor.

As the number of transistors which can be integrated in a chip has been steadily increasing and as a single processor's speed cannot be increased, we saw that chip architects placed several processors in a chip which can cooperate to solve problems in parallel. In Fig. 5.3, we have a multicore processor which has n cores.

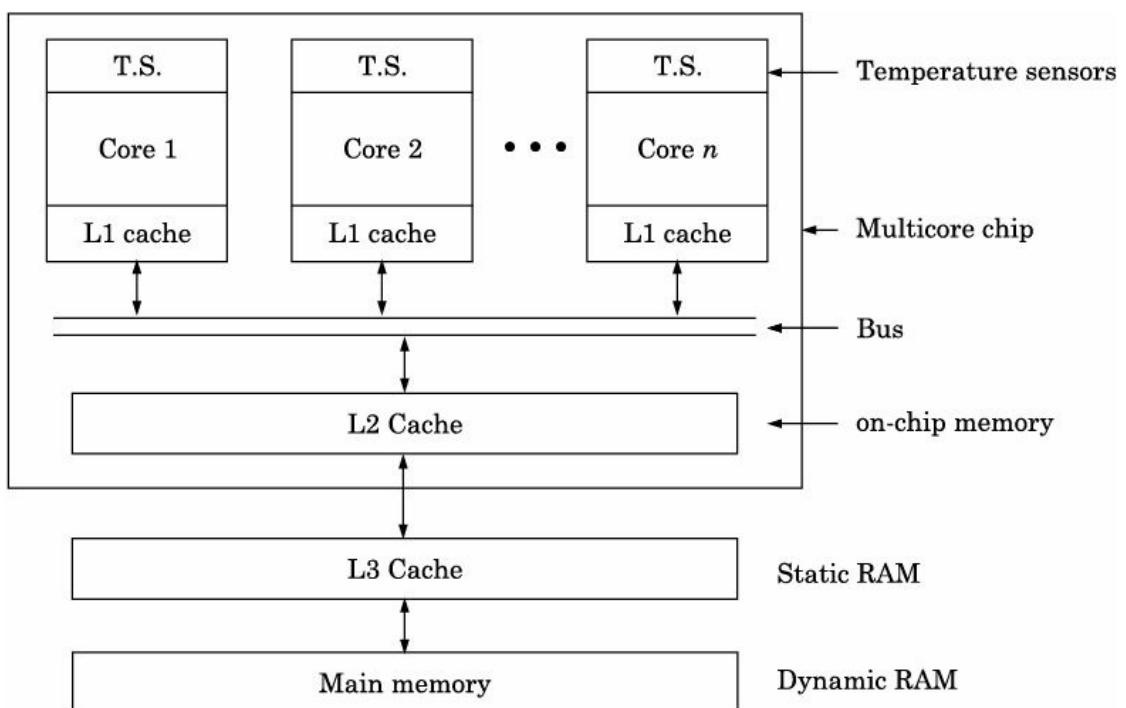


Figure 5.3 A bus connected multicore processor.

The value of n has been changing over the years. Starting with 2 cores in 2003, it has been doubling every 2 years. Observe the difference between Fig. 5.2 and Fig. 5.3. In Fig. 5.3 each core has in addition to registers, execution units, and L1 cache, a temperature sensor [Stallings, 2010]. This sensor is required particularly if the individual cores are complex superscalar multithreaded processors which have a tendency to heat up at high frequencies. If neighbouring cores heat up, a hotspot can develop in the chip and leads to failure of transistors and consequently unreliable results. The temperature sensors may be used to prevent such problems by taking preemptive action. The multiple processing cores in CMPs may be used in several ways. A common characteristic is that each core runs a thread of a program. Another difference one may observe between Figs. 5.2 and 5.3 is the on-chip L2 cache within the chip in Fig. 5.3. In this architecture we have shown that each core has an L1 cache memory which is a part of the core and another shared cache memory L2 on chip. As the number of transistors which can be integrated in a chip has been steadily increasing, it has become feasible to put a large L2 cache in the chip itself. Architects put this memory in the chip as they occupy less area and consume less power as compared to processors. Usually the L2 cache's size is of the order of 1 MB with cache block size of 128 bytes and that of L1 around 64 KB with a cache block size of 64 bytes. The L1 cache is faster to access than the L2 cache.

This is not the only type of CMP architecture. In Fig. 5.3, the L2 cache is shared by all the cores whereas in Fig. 5.4 each core has a private L2 cache. Observe that in this architecture, we have shown a level 3 cache (L3 cache) which is placed outside the chip. Typically in this type of organization L3 would be quite large of the order of 64 to 128 MB.

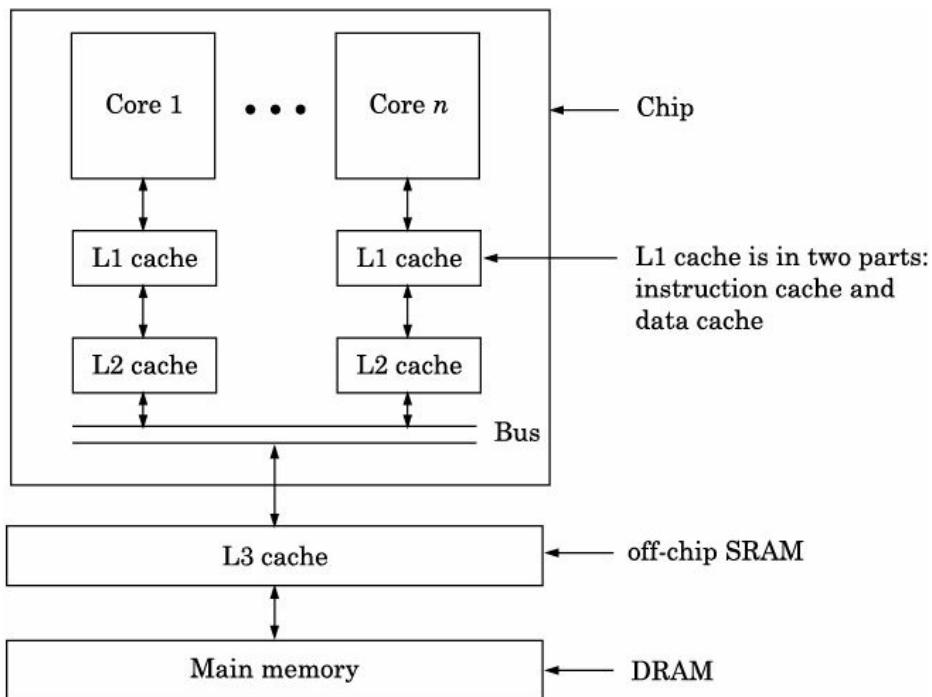


Figure 5.4 A multicore processor with each core having an L2 cache.

We have yet another CMP architecture in Fig. 5.5. Observe that in this architecture L3 is also integrated within the chip. This has become feasible as the number of transistors integrated in a chip has been doubling every 2 years. In this architecture L1 and L2 caches are private to each core whereas L3 cache is shared by all the cores. The size of L3 cache could range from 4 to 16 MB. (Normally the size of L3 cache is 2 MB per core in the chip.) A

common characteristic of all these architectures is that L1 cache is not a unified cache. It is divided into two parts: one stores data and the other part stores instructions. Normally the sizes of these two parts are equal.

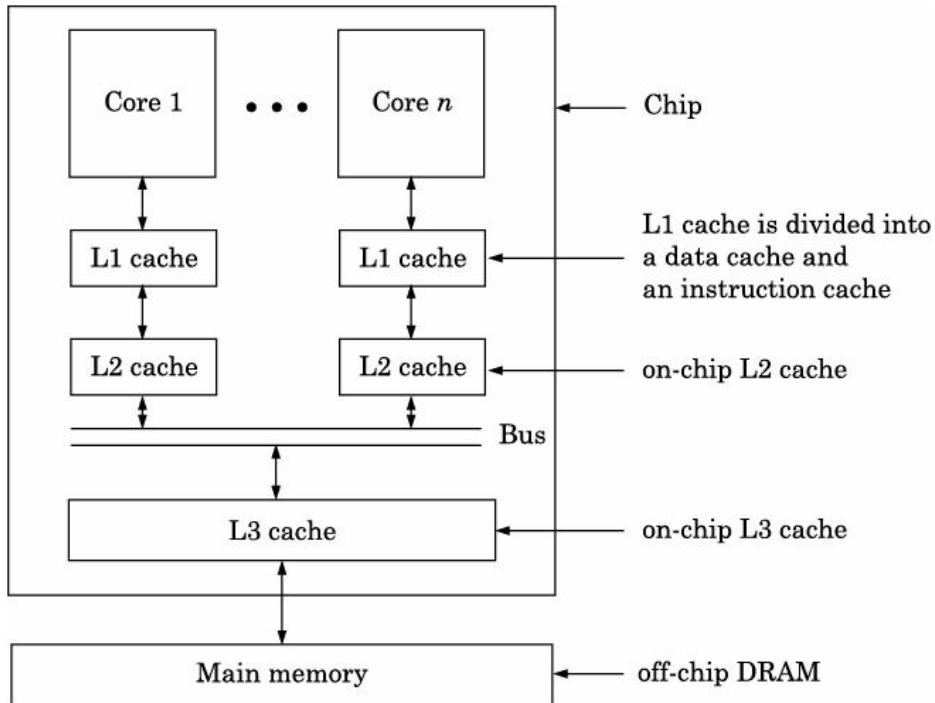


Figure 5.5 A multicore processor with separate L2 and a shared on-chip L3 cache.

The choice of architecture depends on the nature of the application of the multicore chip and also the area available in the chip to integrate memory. For example, Intel core duo an early CMP had the architecture shown in Fig. 5.3 whereas a later CMP, AMD Opteron used the architecture of Fig. 5.4. In the architecture of Fig. 5.5 which uses Intel i7 cores, the processor is a general purpose processor and having an L3 cache on-chip is a great advantage. The choice of the architecture also depends on the application of the CMP. A CMP may be used as:

1. The main processor of a tablet computer
2. The processor of a desktop computer
3. A server
4. A processor for compute intensive tasks such as numerical simulations
5. A co-processor (e.g., graphics processor) of a general purpose processor

In the first two cases, the cores may be easily used to run separate tasks of a multitasking system. In other words, when one core is being used as a word processor, another may be used to play music from an Internet radio station and a third core may be used for running a virus scanner. In such cases, there is hardly any interaction or cooperation between the processor cores. The architecture of Fig. 5.3 would be appropriate as a fast L1 cache will be sufficient as no communication among the cores is required. As a server is normally used to run programs such as query processing of large databases and running independent tasks of several users, the architecture of Fig. 5.4 which has an L2 cache with each core would be advantageous to speedup processing. In this case also in many situations the cores may be working on independent tasks. In case 4, the processor will be required to solve a single problem which needs cooperation of all the cores. The architecture of Fig. 5.5 with a built-in

L3 cache would facilitate cooperation of all the cores in solving a single numeric intensive problem. In this case also different cores may be assigned independent jobs which will increase the efficiency of the multicore processor. The architecture of co-processors is problem specific. We discuss graphic processors later in this chapter.

5.3.1 Cache Coherence in Chip Multiprocessor

The use of independent caches by the cores of CMP leads to cache coherence problem we discussed in detail in Chapter 4. If the number of cores sharing a bus is up to 8 then snoopy bus protocols we described in Sections 4.7.3 and 4.7.4 are used. The cache controllers in these cases are simple and easily integrated in the chip. If the number of cores exceeds 8, directory based cache coherence protocol (Section 4.7.7) is appropriate. The memory consistency model and the need to write race free programs explained in Section 4.7.6 are also applicable to CMPs. The main point is that instead of a bus or an interconnection network being outside the processor chips in a conventional parallel computer, the bus or an interconnection network in a CMP is an integral part of a single microprocessor chip.

5.4 SOME COMMERCIAL CMPs

Many types of multicore processors may be designed as was pointed out in Section 5.2. The type of architecture chosen depends on the application. If a processor is to be used in a mobile device such as a smart phone or a tablet computer, one of the major constraints is power consumption. A power consumption of around one watt is appropriate to reduce the interval between recharging the battery. In this case one or two simple processors and a digital signal processor would be suitable. For a powerful server, up to 8 superscalar multithreaded processors will be appropriate. A large number of tiny processors used in SIMD mode is suitable for graphics processors which we will describe later in this chapter.

5.4.1 ARM Cortex A9 Multicore Processor

In Chapter 3 we described ARM Cortex A9 RISC core. It is a soft core which can be custom tailored as part of a chip for manufacturing. A low power multicore version which is used primarily in tablet computers and mobile phones is shown in Fig. 5.6. A special feature of this processor is its low power consumption, around 1 W. As we saw the ARM Cortex A9 is a superscalar RISC whose performance is quite good. Each core has separate L1 instruction and data caches each of which can store 16 to 64 KB. The chip has up to 4 cores which are interconnected by a bus. The cache coherence protocol is snoopy bus protocol explained in the previous chapter. It has an on-chip 2 MB, L2 cache. A 4-core system can carry out 12 operations per clock cycle.

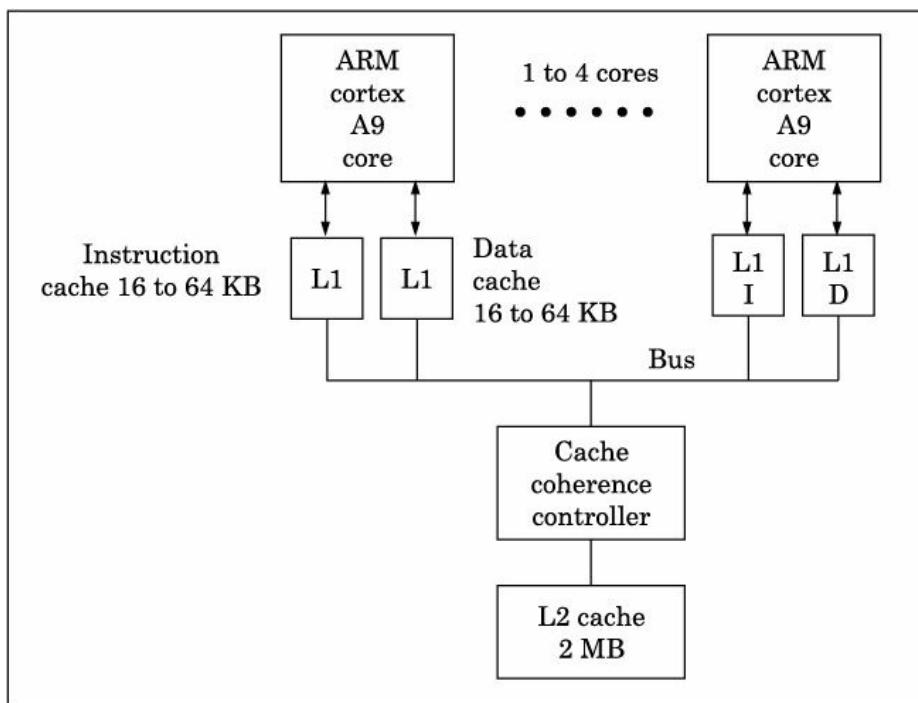


Figure 5.6 ARM Cortex A9 multicore processor.

5.4.2 Intel i7 Multicore Processor

We described a single core Intel i7 processor in Chapter 3. We saw that i7 is a high performance processor which supports x86 instruction set architecture—a CISC. Each i7 core supports simultaneous multithreading and can run four threads. An i7 based multicore processor has between 2 and 8 cores. A block diagram of the multicore processor is shown in

Fig. 5.7. As may be seen from the figure, each core has separate 32 KB, L1 instruction and data caches and a unified 256 KB, L2 cache. The cores are interconnected by a bus and the cache coherence protocol used is a slightly modified MESI. The bus is connected to an 8 MB on-chip L3 cache. It consumes 130 W and the clock frequency is 2.66 to 3.33 GHz depending on the model. The eight core i7 can support 32 simultaneous threads and perform 128 operations per clock cycle. In Fig. 5.7, we show an i7 based CMP using a bus interconnection. It has been superseded by two other types of interconnection. Four i7's have been interconnected by a crossbar switch. This provides faster communication. It is not economical to build a switched CMP beyond 4 processors. Thus, Intel used a ring bus based interconnect which is much more flexible to interconnect 8 or more processors. We will describe the ring interconnected CMP using i7 cores later in this chapter.

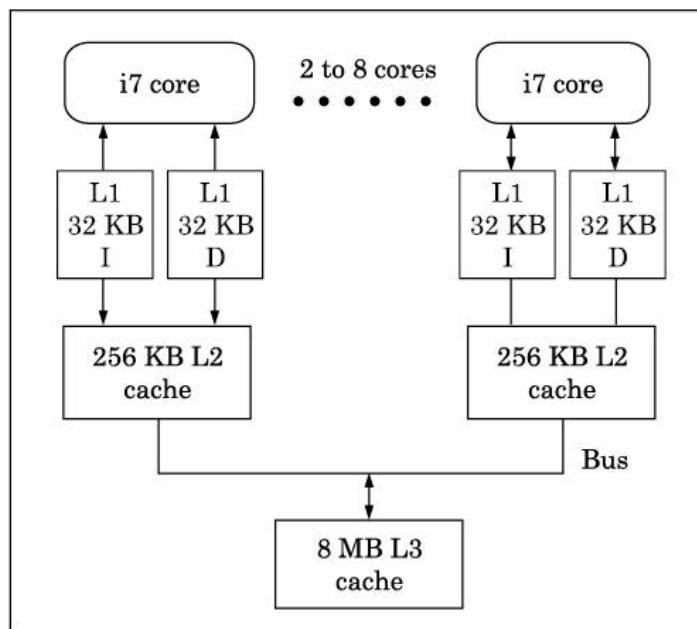


Figure 5.7 Intel i7 based multicore processor in a chip.

5.5 CHIP MULTIPROCESSORS USING INTERCONNECTION NETWORKS

In the last two sections, we described chip multiprocessors in which the individual processing cores were interconnected using a bus. A bus interconnection is simple to fabricate. The cache coherence controller is also easy to design. However, a bus can support only around 8 processors as it will become a bottleneck when many processors simultaneously attempt to access the L3 on-chip cache. As we pointed out in the introduction, the number of transistors which can be integrated in a chip is rapidly increasing. Thus, when a chip multiprocessor is to be built with more than 8 processors, an interconnection network is normally used. Some authors call such a multicore processor with a large number of processors that use an interconnection network as *Many Core Processors* (MCPs) as opposed to chip multiprocessors. In a parallel computer using interconnection network cache coherence is not an issue as the processors work independently and cooperate by passing messages between them as we saw in Chapter 4. Even though message passing many core processors are popular, there are shared memory architectures which use a ring bus and directory based cache coherence protocol.

In Section 4.8.2, we described a number of methods of directly interconnecting computers. In the context of that chapter, these networks consisted of wires which interconnect a number of independent computers. The same type of interconnections may be used to interconnect processors integrated in one chip. There are, however, important differences listed below:

1. As the wires are inside one chip, their lengths are smaller, resulting in lower delay in communication between processors. This in turn allows the units of communication between processors to be light-weight threads (a few hundred bytes).
2. Planar wiring interconnections are easier to fabricate on a chip. Thus, a ring interconnection and a grid interconnection described in Section 4.8.2 are the ones which are normally used.
3. On-chip routers are normally used to interconnect processors as they are more versatile than switches. It is not very expensive to fabricate routers on a chip. They, however, dissipate heat. Power efficiency in the design of routers is of great importance.
4. Routing of data between processors has to be designed to avoid deadlocks. Simple routing methods which guarantee deadlock prevention even if they take a longer route are preferable.
5. The router connecting the processing cores need not be connected to the I/O system of the core. It may be connected to registers or caches in the core. This significantly reduces the delay in data communication.
6. The logical unit of communication we used in Chapter 4 was a message. Most on-chip networks communication is in units of *packets*. Thus when a message is sent on a network, it is first broken up into packets. The packets are then divided into fixed length *flits*, an abbreviation for flow control digits. This is done as storage space for buffering messages is at a premium in on-chip routers and it is economical to buffer smaller size flits. For example, a 64-byte cache block sent from a sender to a receiver will be divided into a sequence of fixed size flits. If a maximum packet size is 64 bytes, the cache block will be divided into 16 flits. In

addition to the data flits, an additional header flit will specify the address of the destination processor followed by data flits, and a flit with flow control information and end of data packet information. The flits that follow do not have the destination address and follow the same route as the header flit. This is called *wormhole routing* in which flits are sent in a pipeline fashion through the network (See Exercise 7.5). Details of router design are outside the scope of this book and the readers are referred to the book On-chip networks for multicore systems [Peh, Keckler and Vangal, 2009] for details.

7. As opposed to a communication system in which TCP/IP protocol is used and packets are sent as serial string of bits on a single cable or a fibre optic link, the flits sent between cores in a many core processor are sent on a bus whose width is determined by the distance between routers, interline coupling, line capacitance, resistance of the line, etc. The bus width may vary from 32 bits to as much as 256 bits depending on the semiconductor technology when a chip is fabricated. Accordingly the flit size is adjusted.

5.5.1 Ring Interconnection of Processors

We pointed out in the last section that ring and grid (also known as mesh) inter-connection between processors is normally used in many core multiprocessors. In Section 4.8.2, we described a switched ring interconnecting processors. We give a modified ring interconnection in Fig. 5.8. This is a bidirectional ring in which data can travel clockwise and/or anti-clockwise. The main difference between Fig. 4.28 and Fig. 5.8 is that the router (switch) is connected to the NIU (which is like an I/O unit) in Fig. 4.28 whereas it is connected directly to the core in Fig. 5.8. In on-chip interconnection network this is possible as both the router and the core are on the same silicon chip. The number of bits carried by the inter-router path can be packets whose size may vary between 32 and 256 bits. The characteristics of this switched ring connection are:

1. The number of links to each router is fixed and equals 3.
2. As it is a switched ring, any pair of cores can communicate with one another simultaneously as long as there is no conflict. For example, core n can send a packet to core $n - 1$ and simultaneously core k can send a packet to core $k + 3$. Observe that the packet from core n to core $n - 1$ travels clockwise whereas that from core k to core $k + 3$ travels anti-clockwise. This is possible as it is a bidirectional ring.
3. If there are independent set of wires for the clockwise ring and the anti-clock-wise ring, the number of possible communication paths double. This also increases the possible conflict free independent packet communication among cores.
4. The ring requires one router per core. Routers are fabricated in silicon and dissipate heat. Routers, however, are very simple in a ring interconnection.
5. This type of switched ring is normally used for message passing many core processors with no shared memory.

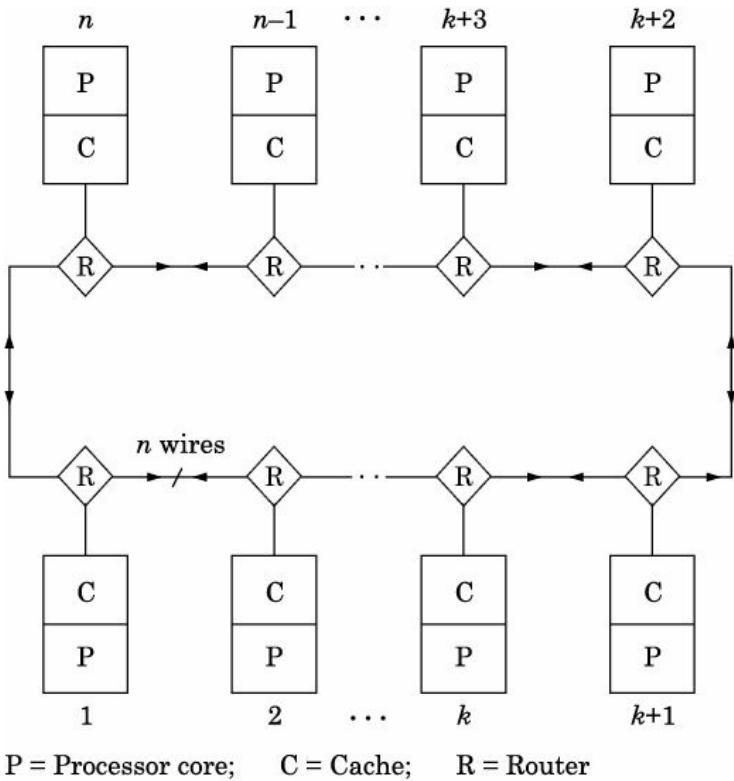


Figure 5.8 A switched bidirectional ring interconnection.

Ring Bus—Token Ring

There is another type of ring interconnection which is called a *ring bus* as it is similar to a bus but is configured as a ring interconnecting cores. The multicore processor using a ring bus interconnection normally uses a shared memory programming model. In such a structure, the ring consists of a set of parallel wires and the processor cores are connected to these wires. Usually a bidirectional ring is used (see Fig. 5.9). Thus, there is a bus in which bits travel in parallel clockwise and another in which bits travel in parallel anti-clockwise. When a processor core accesses its cache and misses, it has to see if it is in the cache of any other processor core on the ring. If it is not in any cache, it has to access the next level memory (which may be off-chip). If data is to be written by a processor core in its cache, it has to invalidate the cache blocks with this address in other caches. Data is normally in units of a cache block. Thus, a ring in this case consists of three independent rings. One is the data ring which carries a cache block. The second ring is a ring on which read/write requests and the corresponding addresses are sent. The third is an acknowledgement ring which is used for sending coherence messages and flow control information. It should be remembered that at a given time only one cache block can be carried by a ring bus. One way in which a core may access the ring to read/write data is similar to the token ring protocol used in local area ring networks. A distinct set of bits, e.g., (1111.....1) called a *free token* is sent on the flow control ring. This token signifies that the data ring is free and is available to carry data. A processor wanting to send/receive data on the ring continuously monitors the token ring. If it captures a free token, it changes to (say) (0000.....0) which indicates that it now owns the data ring to send data. The processor puts the data to be sent on the data ring and the address of the destination processor on the address ring. The address ring is monitored by all the processors and the processor whose address matches captures the data and appends acknowledgement bits to the token and puts it on the ring. The token continues to travel on the ring and when it reaches the sender, the sender captures it and inspects whether acknowledgement bits are

there. If the acknowledgement is OK, then it frees the token by changing the bits to (1111....1) indicating that the ring is free to carry some other data. Thus, the token has to make a full round before the ring can be used again. As there are two rings, a processor will monitor both the rings and use the one whose distance (i.e. number of hops) to the intended destination is smaller. Thus, on the average the number of hops is halved.

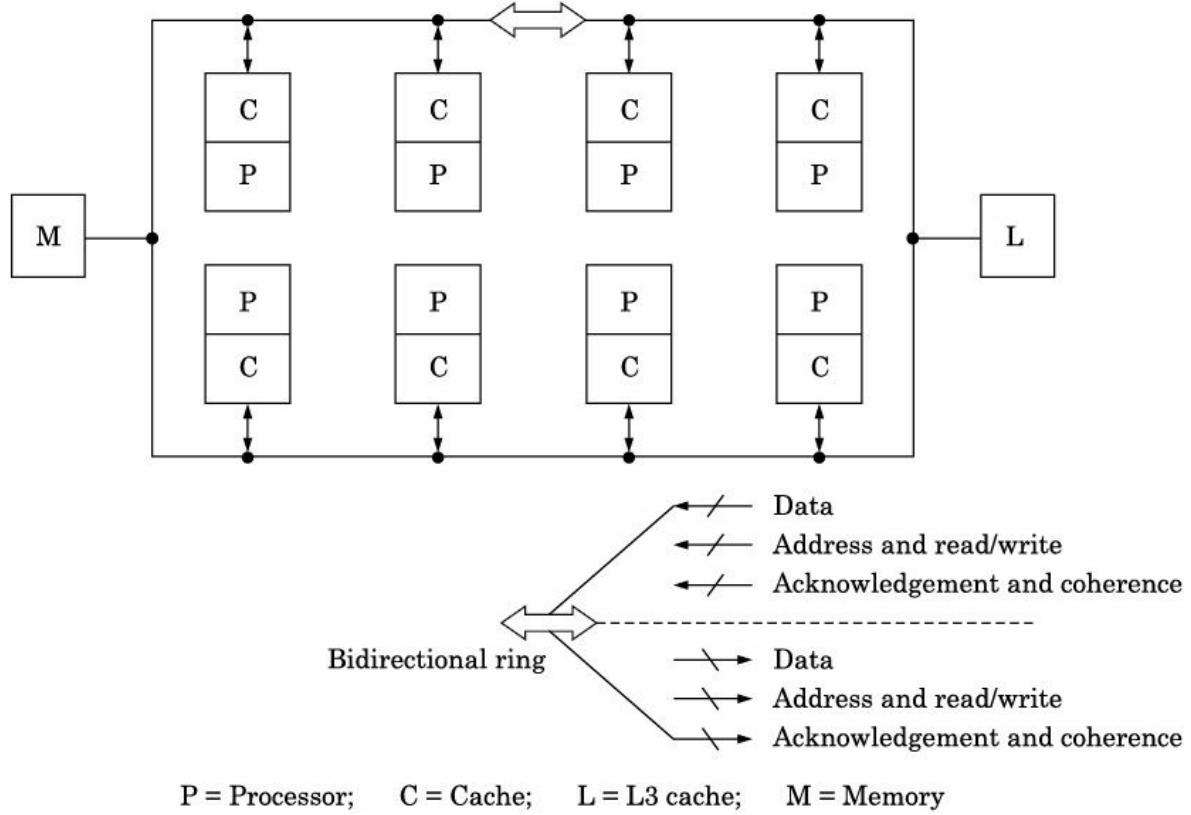


Figure 5.9 A bidirectional ring bus interconnection.

Ring Bus—Slotted Ring

The architecture of a unidirectional single wire ring is shown in Fig. 5.10. The ring interface of a processor core consists of one input link, a set of latches, and an output link. The number of latches constitutes a *slot*. Data is transmitted, slot by slot, on the ring in a pipeline mode driven by a clock. At each clock cycle, the contents of a latch is copied to the next latch so that the ring behaves like a circular pipeline. The primary function of the latches is to hold an incoming data and wait for a control signal to shift it to the next latch. In a slotted ring with uniform slot sizes, a processor ready to transmit data waits for an empty slot. When one arrives, it captures and transmits the data. If data to be transmitted is a cache block of 32 bytes, there will be 256 latches per slot and slots for source and destination addresses. The latches will be filled and clocked round the ring till it is captured by the intended destination processor. A single wire ring will be very slow.

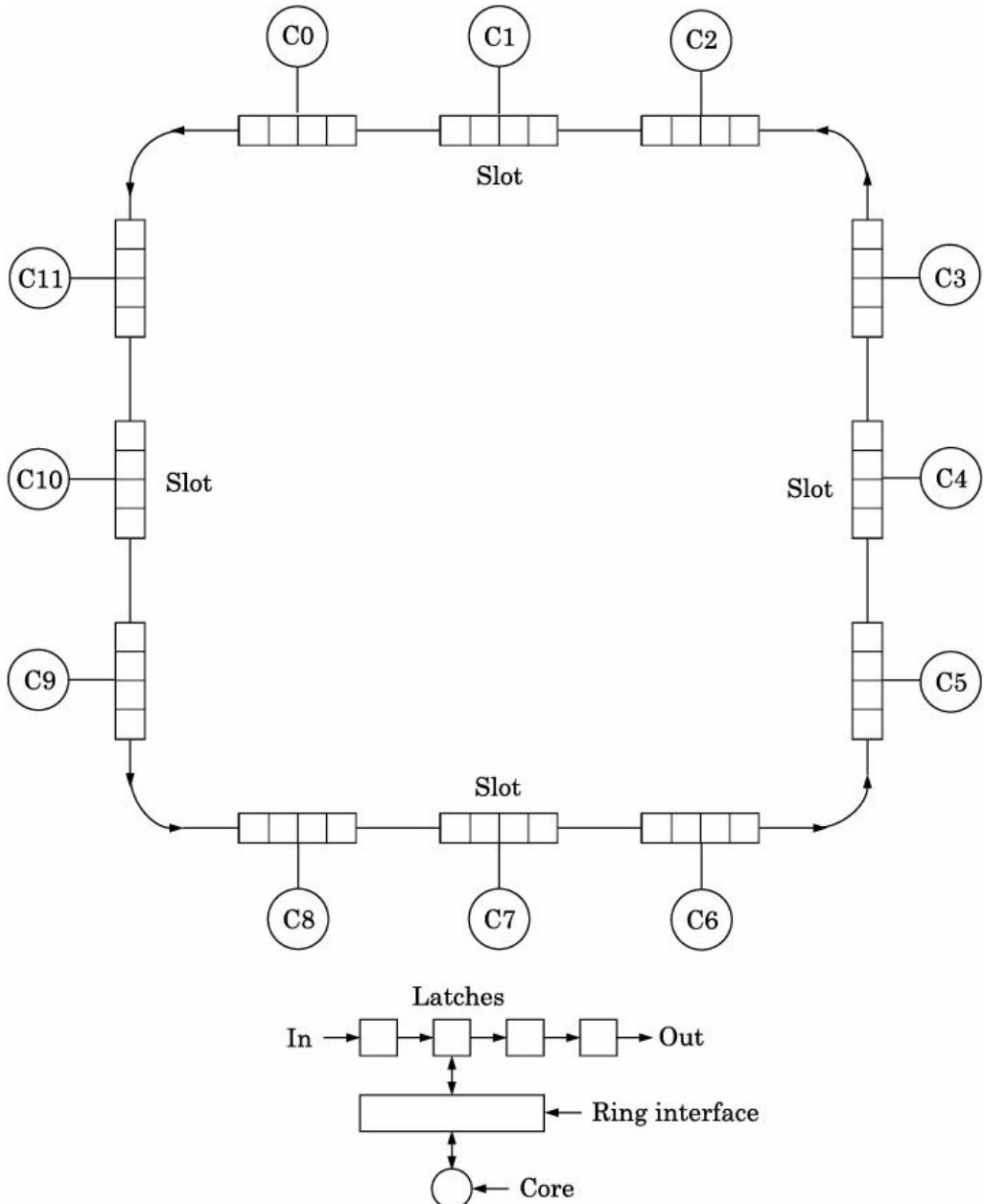


Figure 5.10 A slotted ring (single wire).

In multicore processors instead of a single wire ring, a number of parallel wires constituting a *ring bus* is used. In a ring bus each slot will be a latch and for 32 bytes data, the bus will be 256 wires wide with one latch per wire. If the latches are clocked at the clock rate of the core, the transmission of the cache block between adjoining cores will take only 3 clock cycles; one to put data on the latch, one to transfer it to the next latch, and one to read the data by the destination core. If the number of hops of a core from the source core is n , the time taken will be $(n + 2)$ clock cycles. Apart from the data ring bus, many more ring buses are needed in the architecture of a multicore system. One ring bus will be required to carry the source and destination addresses, another ring bus for read/write command, a third ring bus to acknowledge the receipt of the data, and one more for ensuring cache coherence. The number of wires in the other ring buses will depend on the number of cores in the multicore system, the type of cache coherence algorithm etc. This type of ring bus interconnection of cores has been adapted appropriately by Intel in some of the multicore systems designed by it. If there is only one data ring and the data can travel in only one direction, the time taken to

send data from processor 0 to processor n will be $(n + 2)$ clock cycles as we saw earlier. To reduce the time to transmit data, two counter rotating ring buses may be used. A core desiring to transmit data will pick the ring closest to the destination and reduce the time to send data. The maximum time to send data from core 0 to core n will be $(n/2 + 2)$ clock cycles.

5.5.2 Ring Bus Connected Chip Multiprocessors

In Section 5.4.2 we gave bus connected architecture of a CMP using Intel i7 cores. It has been superseded by a ring bus based interconnection between i7 processors which is shown in Fig. 5.11. This is known as Intel's Sandy Bridge architecture for servers [Kanter, 2010]. The L3 cache is divided into 8 slices; the size of each slice is 2.5 MB. This architecture uses ring buses to interconnect the cores, L3 caches, other agents such as memory controller, Quick Path Interconnect (QPI) interface to connect other multicores, and the I/O system (see Fig. 5.11). (We use the term ring to mean ring bus in what follows.)

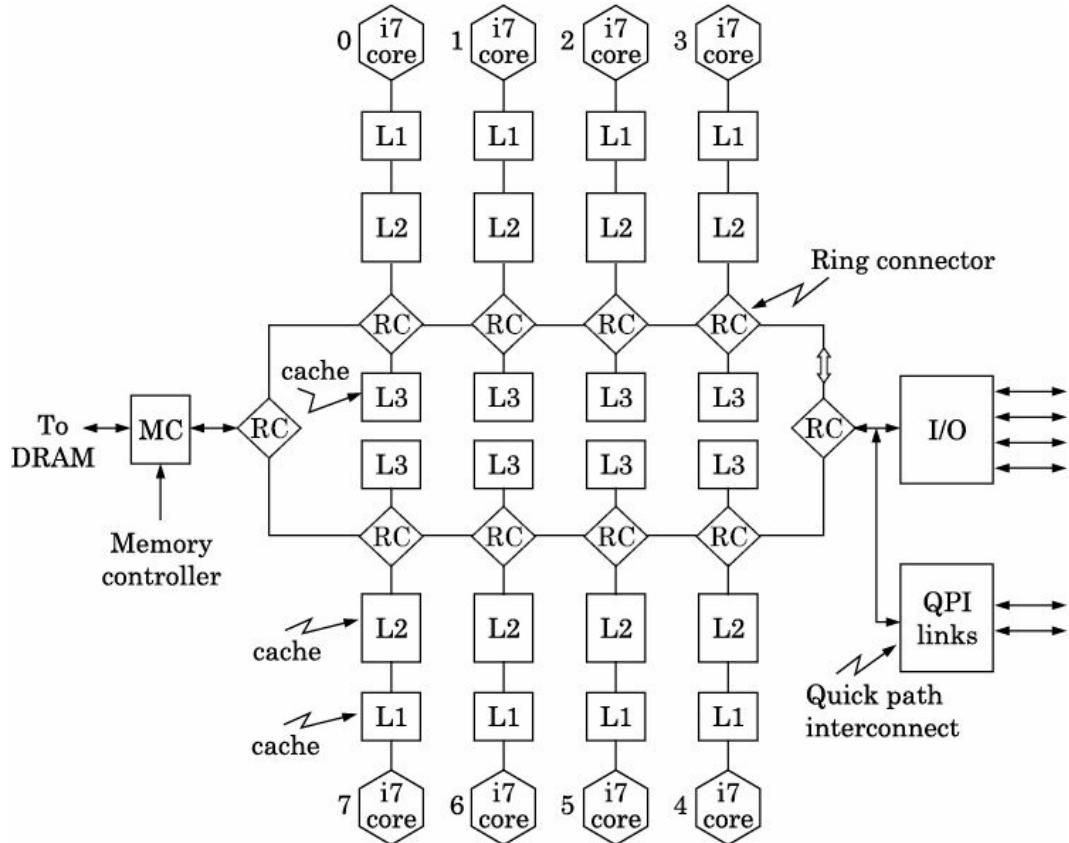


Figure 5.11 A ring bus interconnected Intel i7 cores. (also called Sandy Bridge architecture for servers).

The ring is composed of four different rings. A 32-byte data ring (half of a cache block), a request ring, a snoop ring, and an acknowledgement ring. These four rings implement a distributed communication protocol that enforces coherency and ordering. The communication on the rings is fully pipelined and runs at the same clock speed as the core. The rings are *duplicated*. One set runs in the clockwise direction and the other in the anti-clockwise direction. Each agent interfacing a core with the rings interleaves accesses to the two rings on a cycle by cycle basis. A full 64-byte cache block takes two cycles to deliver to the requesting core. All routing decisions are made at the source of a message and the ring is not buffered which simplifies the design of the ring system. The agent on the ring includes an interface block whose responsibility is to arbitrate access to the ring. The protocol informs each interface agent one cycle in advance whether the next scheduling slot is available and if

so allocates it. The cache coherence protocol used is a modified MESI protocol discussed in section 4.7.4.

A question which would arise is why a ring interconnect was chosen by the architects. The number of wires used by the ring is quite large as 32 bytes of data to be sent in parallel will require 256 lines. The total number of wires in the rings is around 1000. The delay in sending a cache block from one core to another is variable. In spite of these shortcomings, ring interconnection is used due to the flexibility which it provides. The design of the ring will be the same for a 4, 8, or 16 processor system. If the processors exceed 8, instead of a MESI protocol a directory protocol will be used to ensure cache coherence. It is also easy to connect to the ring agents to control off-chip memory and off-chip I/O systems.

As we pointed out earlier, temperature sensors are essential in CMPs. In the Sandy Bridge architecture, Intel has used better sensors. Further the voltage used by the on-chip memory, the processor cores, and dynamic memories are different. There is also a provision to dynamically vary the voltage of the power supply and the clock of different parts of the chip.

5.5.3 Intel Xeon Phi Coprocessor Architecture [2012]

Intel has designed a ring connected many core chip multiprocessor to be used as a coprocessor to increase the speed of i7 based host processors used in servers. This is code named Knights Corner and was released in 2012 as Xeon Phi Coprocessor. This coprocessor uses Linux OS, runs FORTRAN, C, and C++, supports x86 instruction set, and uses IEEE floating point standard for arithmetic. The coprocessor is connected to the host processor through PCI express bus of Intel. The coprocessor may also be used independently. The version announced in 2012 has 60 cores with a provision to increase the number of cores. Each core uses a short in-order pipeline capable of supporting 4 threads in hardware. It can support x86 legacy programs as the hardware supports this instruction set. A block diagram of a core is shown in Fig. 5.12. As may be seen it has 32 KB L1 cache and 512 KB L2 cache. Besides the scalar processing unit it has 512 bit SIMD Vector Processing Unit (VPU) which can add two vectors followed by a multiply, i.e., it can compute $(\mathbf{A} + \mathbf{B}) * \mathbf{C}$ where \mathbf{A} , \mathbf{B} , and \mathbf{C} are vectors. The vectors \mathbf{A} , \mathbf{B} , and \mathbf{C} may either have 16 components each 32 bits long or 8 components each 64 bits long. A vector multiply add may be done in 1 clock cycle. The components may be either floating point numbers or integers. The VPU also supports gather-scatter operations in hardware (see Section 4.4 for details how vector processors function). The VPU is very power efficient as a single operation can encode a lot of work.

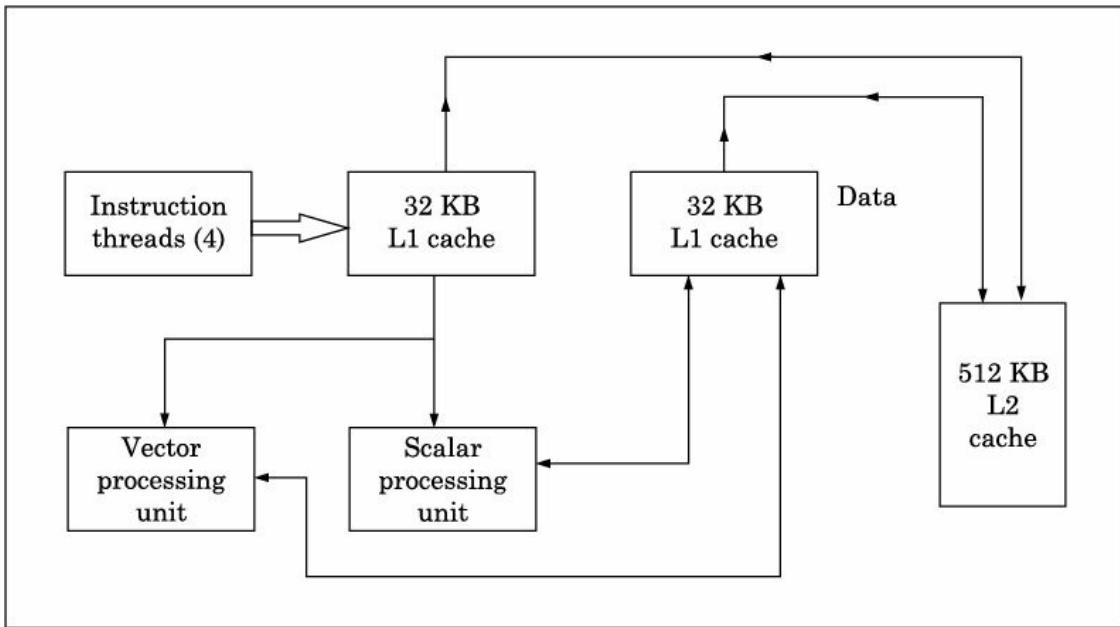


Figure 5.12 Intel Xeon Phi Coprocessor core.

The cores are connected by a very high bandwidth bidirectional ring bus. Each direction of the ring contains three independent rings. They are a 64 byte data ring, an address ring which is used to send read/write commands and memory addresses, and an acknowledgement ring which is used to send flow control and cache coherence messages. A block diagram of the system is shown in Fig. 5.13. As may be seen the ring connects the cores, a set of Tag Directories (TD) used for maintaining a global directory based cache coherence, PCIe (Peripheral Component Interconnect express) client logic to connect to I/O bus, a set of memory controllers to connect to off-chip main memory (usually a dynamic memory). It is logical to assume that the ring architecture is similar to the one used by Sandy Bridge architecture of Intel.

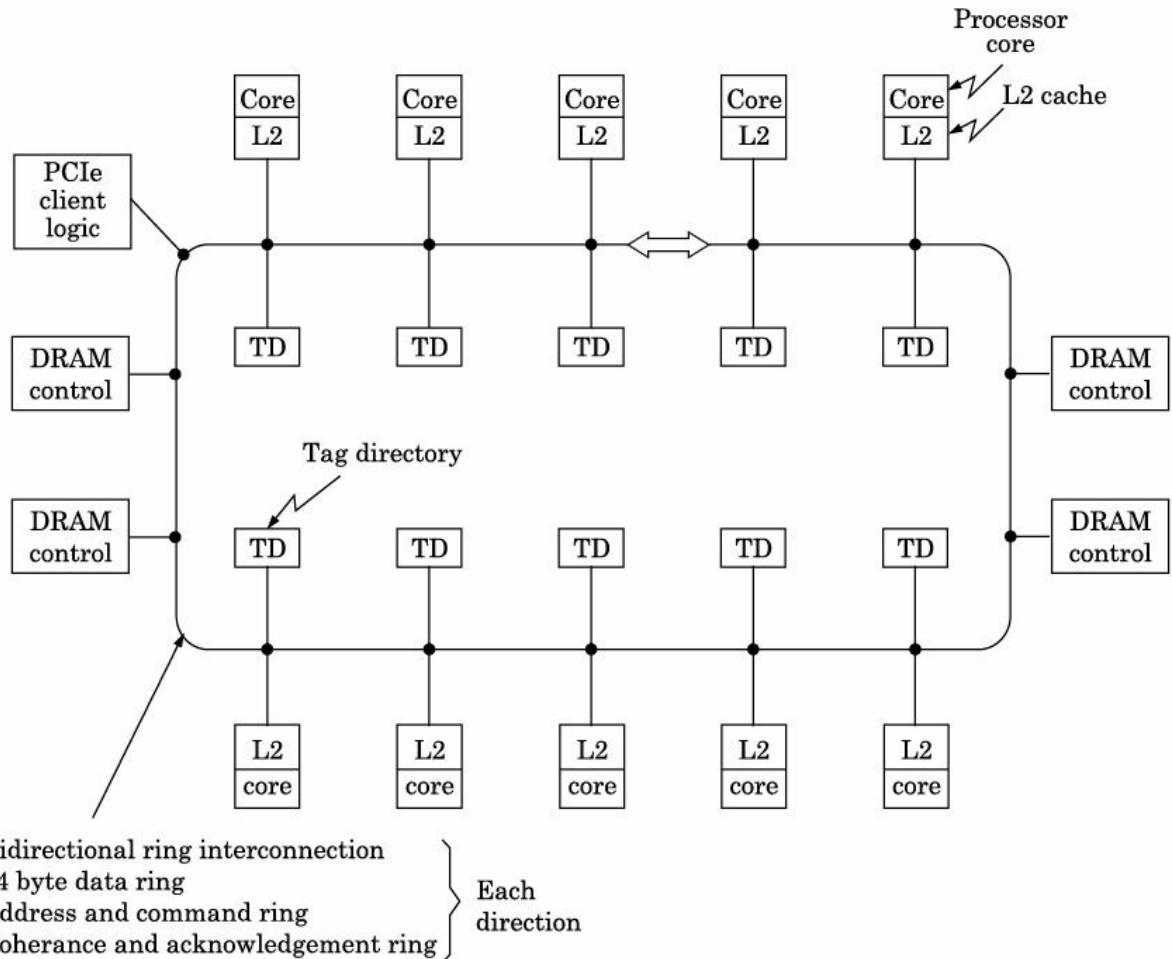
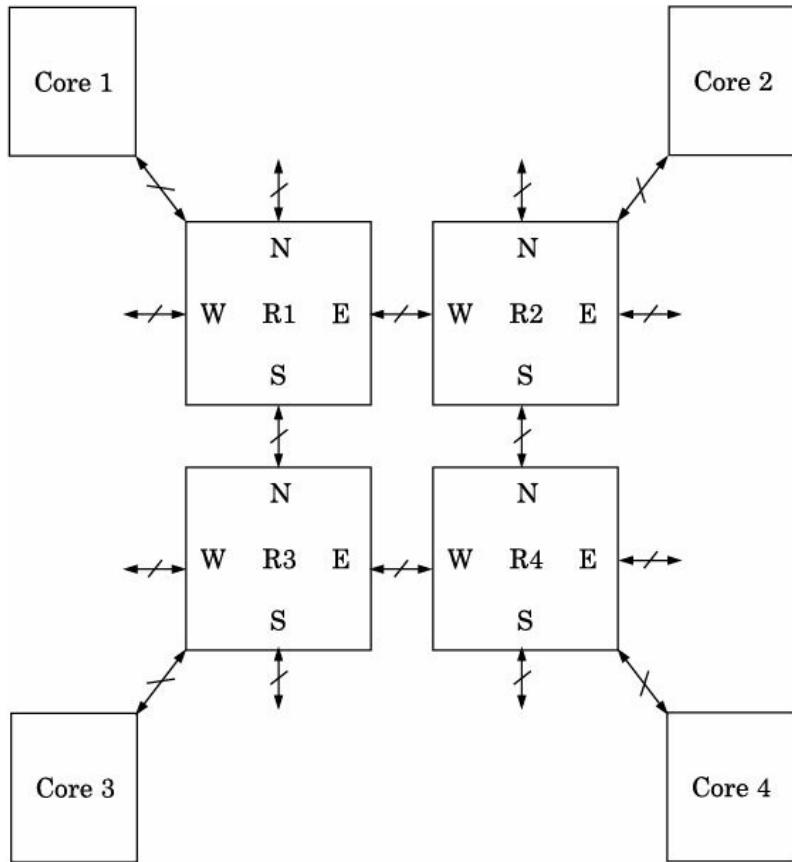


Figure 5.13 Intel Xeon Phi Coprocessor ring based architecture.

By inspecting Fig. 5.13 we see that the memory controllers are symmetrically interleaved on the ring. The tag directories track the cache blocks in all L2 caches. They are evenly distributed to ensure fast response. During memory access by a core when an L2 cache miss occurs, the core generates an address request which is sent on the address ring which queries the tag directories. If the data is not found in the tag directories, a request is sent to the main memory by the core. The memory controller fetches the data from the main memory and sends it to the core on the data ring. Observe that the number of wires required by the ring is quite large (over 1100) and occupies a lot of space on the chip. The ring, however, is versatile and it is easy to add more cores as the technology improves and more transistors can be integrated in a single chip.

5.5.4 Mesh Connected Many Core Processors

A very popular many core processor architecture is a set of cores arranged as a rectangle with n cores in the X direction and m cores in the Y direction. The cores are normally simple identical cores. Identical cores are used to minimize design cost of the chip. The cores are interconnected by routers. Each core is connected to a router. The router consists of 5 ports. One port connects it to the processor. The other four ports marked N, E, S, W are connected to the four neighbouring processors (See Fig. 5.14).

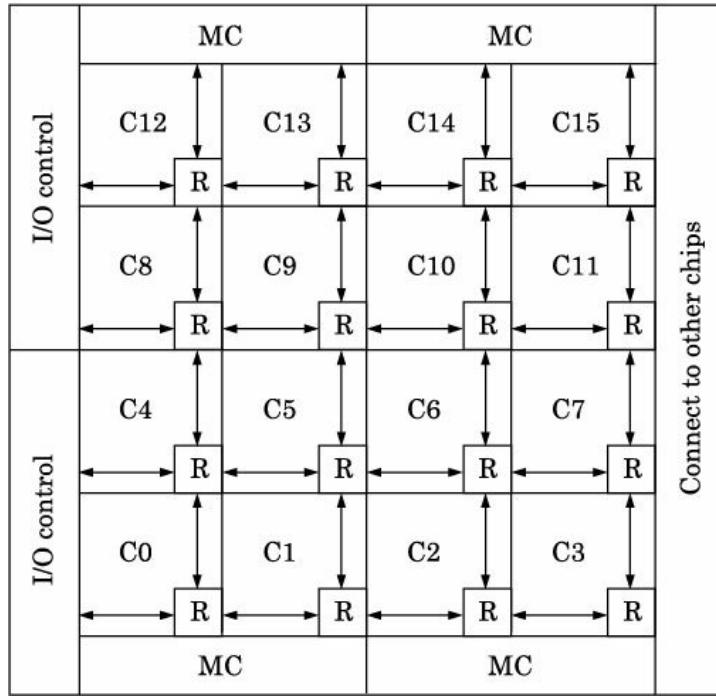


Each router has five ports, one connecting to a core and the other four to four other neighbouring routers.

Figure 5.14 Four processor cores connected by four routers.

The instruction set of the processor core will have send/receive primitives as the core is part of a message passing multicomputer. As the cores are connected to the routers integrated on the same chip, the routers will normally be connected to the register file of the core. The message would normally be a cache block which is divided into packets. Each packet is segmented into fixed length flits (flow control digits) with a header flit that has the destination address bits, several body flits, and a tail flit that indicates the end of the packet. The flits will contain information to control the flow of flits on the network and some bits indicating the action to be performed by the core receiving the packet. As only the head flit has the destination address, all flits belonging to a packet follow the same path as the head flit. A routing algorithm will be used to minimise latency and avoid deadlock. Unlike routing in a TCP/IP network, on-chip networks are designed not to drop packets if there is congestion.

A 16-core network layout is shown in Fig. 5.15. As may be seen the structure is modular. If a core C4 wants to send a message to C15, the message will be packetized as flits. The head flit will route the message via routers on C5, C6, C7, C11, and C15. This East to North routing will be indicated by the header flit. This type of routing avoids deadlocks. More than one message may be sent at the same time as the routers will have buffers to keep the packets waiting till a route is clear. For details of routing algorithms the reader is referred to on-chip networks for multicore systems [Peh, Keckler and Vangal, 2009].



C0 to C15 are processor cores + cache, R are routers. The connection of a router to a chip is hidden. Observe that one router per core connects four neighbouring routers. MC are memory controllers for off-chip memory.

Figure 5.15 Sixteen core mesh connected chip.

In real chips there is provision to switch off inactive parts of the system. Some many core chips even provide fine grain power control at the level of groups of transistors.

5.5.5 Intel Tera flop Chip [Peh, Keckler and Vangal, 2009]

The Intel Tera flop chip is capable of performing 10^{12} (i.e., 1 Tera) floating point operations per second while dissipating less than 100 W. This chip has 80 tiles on-chip arranged as a (8×10) 2D array and connected as a mesh. The network fabric works at 5 GHz. A tile consists of a processor connected to a five port router which forwards packets between the tiles. The processor in each tile contains two independent fully pipelined single precision floating point multiply accumulate units, 3 KB single cycle instruction memory, and 2 KB data memory. A 96-bit very long instruction word is capable of encoding up to 8 operations per clock cycle. The processor architecture allows scheduling both the floating point multiply accumulate units simultaneously, loading and storing in data memory, packet send/receive from the mesh network, and synchronization of data transfer between processors. A block in the processor called Router Interface Block (RIB) handles encapsulation of packets between the processor and the router. The architecture is fully symmetric and allows any processor to send/receive data packets to or from any other processor. This architecture is meant to be used as a coprocessor for performing fast floating point operations. The design is modular and may be expanded as transistor technology improves—details of the architecture may be found in on-chip networks for multicore systems [Peh, Keckler and Vangal, 2009].

5.6 GENERAL PURPOSE GRAPHICS PROCESSING UNIT (GPGPU)

Graphics processors evolved from graphics accelerators which were used in personal computers. As semiconductor technology evolved and the requirements of realistic graphics processing was felt, companies specializing in graphics processing such as NVIDIA started designing single chip many processor graphics engines. The special feature of graphic programs is that operations on each pixel of an array of pixels can be done independently. As the resolution of graphics increased, the number of pixels to be processed became of the order of millions. Thus, there is plenty of parallelism. The type of parallelism is data parallelism as the same computation is performed on all data records and there is no dependency among data elements. The computation performed is simple multiply, add, and accumulate on pairs of integers and on pairs of single precision floating point numbers. Thus, the processing units are tiny. Further as graphics processors do not just process static pictures but also videos where the pictures are repeated, data parallel processing is performed on streams of pixels. Thus, pipelined set of simple multiply accumulate units is the natural configuration appropriate for GPUs. The graphics processors use a single instruction multiple data style of parallel processing which we described in Section 4.2.1. Figure 4.2 gave the structure of an SIMD computer. Graphics processors have a similar structure but the nature of the PE is a pipelined array of simple arithmetic units as we pointed out earlier. Even though Graphics Processing Units (GPUs) were originally designed for processing images, it was soon realized that the type of architecture used is appropriate for *stream processing*. A stream is defined as a set of records that require identical set of instructions to be carried out on each record. This style of processing is appropriate not only for records (or tuples) representing pictures but also in many applications in science and engineering. Some of these are:

- Operations on a large set of vectors and matrices common in solving sets of simultaneous algebraic equations
- Fast Fourier transforms
- Protein folding
- Data base search, particularly big data arrays

Thus, the GPUs were soon used with appropriate software for general purpose computing and were renamed GPGPU (General Purpose GPUs).

As the origin of GPGPUs is from graphics, a number of technical terms relevant to graphics processing have been traditionally used which confuses students who want to understand the basic architecture of GPGPUs [Hennessy and Patterson, 2011]. We avoid reference to graphics oriented terminology and attempt to simplify the basic idea of processing threads in parallel. Consider the following small program segment.

```
for i := 1 to 8192 do
    p(i) = k * a(i) + b(i)
end for
```

This loop can be unrolled and a *thread* created for each value of i . Thus there are 8192 threads which can be carried out in parallel. In other words, the computations $p(1) = k * a(1) + b(1)$, $p(2) = k * a(2) + b(2)$, ... $p(8192) = k * a(8192) + b(8192)$ can all be carried out simultaneously as these computations do not depend on one another. If we organize a parallel computer with 8192 processing units with each processor computing $k * a(i) + b(i)$ in say 10 clock cycles, the *for* loop will be completed in 10 cycles. Observe that in this case each

processing unit is a simple arithmetic logic unit which can carry out a multiply operation and add operation. If $a(i)$ and $b(i)$ are floating point numbers and k an integer, we need only a floating point adder and a multiplier. If we do not have 8192 processing units but only 1024 processing units, then we can employ a thread scheduler to schedule 1024 threads and after they are computed, schedule the next 1024 threads. Assuming 10 cycles for each thread execution, $(8192/1024) \times 10 = 80$ cycles will be required to complete the operation.

This example illustrates the essence of the idea in designing GPGPUs. In general GPGPUs may be classified as *Single Instruction Multiple Thread* (SIMT) architecture. They are usually implemented as a multiprocessor composed of a very large number of multithreaded SIMD processors. If a program consists of a huge number of loops that can be unrolled into a sequence of independent threads and can be executed in parallel, this architecture is appropriate. A simplified model of architecture for executing such program is shown in Fig. 5.16.

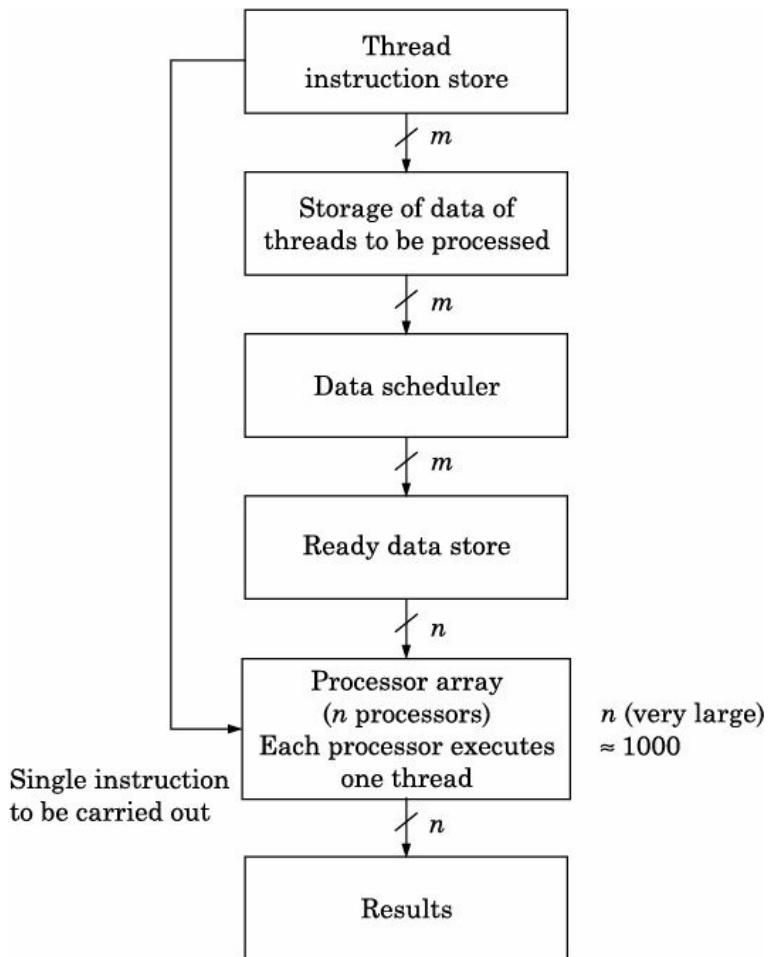
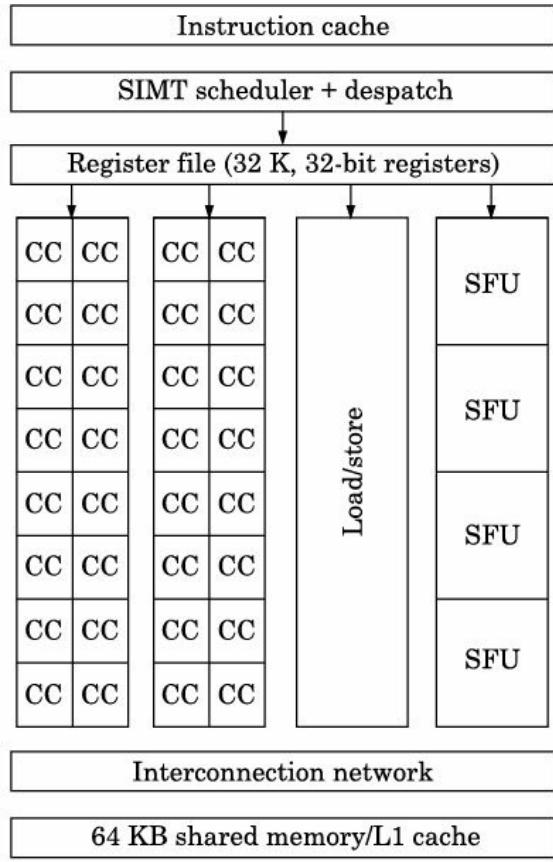


Figure 5.16 A simplified model of single instruction multiple thread architecture.

Having described the general idea of SIMT processing, we will take a concrete example of a GPGPU designed by NVIDIA, called Fermi. Fermi consists of a number of what are called Streaming Multiprocessors (SM). A block diagram of a Fermi SM is given in Fig. 5.17.



CC → CUDA core SFU → Special function unit

There are 32 CUDA cores. Each core is very simple.
There are also 4 special functional units (SFU).

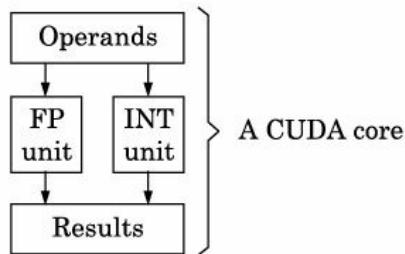
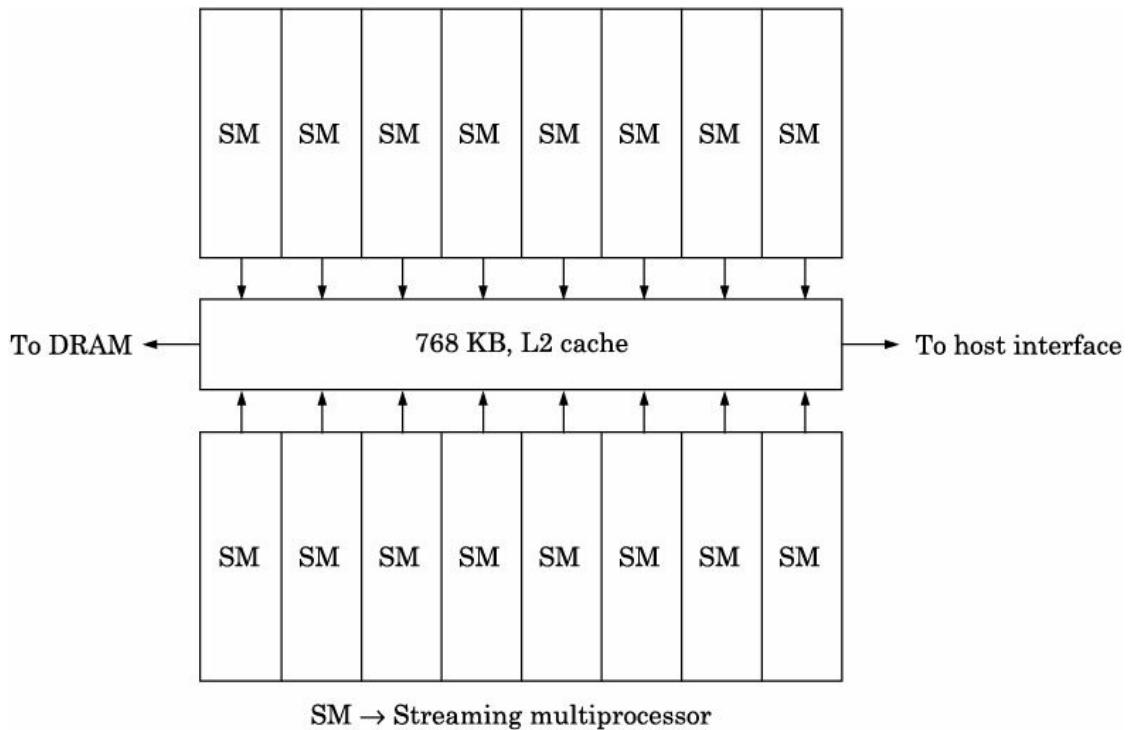


Figure 5.17 A Streaming Multiprocessor (SM) of Fermi GPGPU.

As may be seen from this figure, each Fermi SM has 32 very simple pipelined arithmetic units along with units to send instructions and operands to these units and a storage unit to hold the results. The pipelined arithmetic unit is called a CUDA (Computer Unified Device Architecture) core by NVIDIA. Besides the CUDA cores, SM has four special function units to calculate square roots, reciprocals, sines, cosines, etc. Observe the huge register file which is used in a manner similar to that in simultaneous multithreading in a traditional CPU design (Section 3.7.2), namely, fast context switching. Besides this an SM has 64 KB shared memory which is an L1 cache. Fermi GPGPU has 16 SMs that share a 768 KB, L2 cache (see Fig. 5.18).



Each SM has 32 CUDA cores. There are 512 CUDA cores in one GPGPU chip.

Figure 5.18 Fermi GPGPU with 16 streaming multiprocessors (SM).

The L2 cache is connected via a multi-ported interface to an off-chip DRAM. There is an interface to the host processor (remember that GPGPU is a coprocessor to a host) which is normally a PCI express interface. Observe that the SMs are capable of executing 32 identical operations at a time. Thus, each SM fetches and decodes only a single instruction. Sixteen SMs with two CUDA cores per SM can be put into a single chip only because all the 512 CUDA cores are identical and instruction decoding and issue are shared by all the 512 cores. If a programmer can perform 512 identical operations in one “instruction”, the speed of the Fermi GPGPU is very high. To facilitate programming for the Fermi like architecture, NVIDIA developed a proprietary language called CUDA. An open source equivalent is called OpenCL. The terminologies used by the two programming languages are different but the general philosophy of design is same. CUDA language specifies the parallelism in the program using threads. Threads are grouped into blocks which are assigned to SMs. As long as every thread in a block executes identical sequence of code (including taking the same paths in branches), 32 operations are executed simultaneously by an SM. When threads on an SM take different branches in a code, performance degrades. Thus, one has to select appropriate code and scheduling to get the most out of the large number of cores. Significant memory bandwidth is required to support 512 cores to process simultaneously. This is achieved by a hierarchy of memories. Each SM has 64 KB shared memory or L1 cache. Fermi has a mode bit which allows the 64 KB of SRAM to be partitioned as 16 KB, L1 cache + 48 KB local shared memory or as 48 KB, L1 cache with 16 KB local shared memory. Observe also the huge register file in SM (128 KB) much larger than L1 cache, which is a peculiarity of Fermi architecture.

Observe that the 16 SMs share a 768 KB L2 cache. The L2 cache allows SMs to share data among them. It is used to temporarily store data stored in DRAM. If data is to be fetched from the off-chip DRAM for computation, it will enormously slow down the system and should be avoided.

Observe that in order to get good performance from Fermi GPGPU, a programmer has to understand the way the architecture processes data. Scheduling threads to keep all the CUDA cores busy is crucial. The clock speed used by Fermi is 772 MHz. With 512 cores it can potentially run at 1.5 Tera flops as the ALUs in CUDA cores are pipelined.

As the technology of integrated circuits improves, more cores can be packed in a chip. Usually chip designers keep the architecture same while increasing the number of cores. The successor of Fermi is Kepler designed by NVIDIA which has a similar architecture but the performance is three times that of Fermi per watt. It has 1 Tera flop double precision arithmetic throughput.

The discussion given in this section primarily emphasizes the important concepts without delving into many details which are important if one wants to be an expert user of GPGPUs. For a detailed discussion of NVIDIA architecture students may refer Computer architecture: A quantitative approach [Hennessy and Patterson, 2011]. An interesting discussion on the differences between SIMD, SIMT, and simultaneous multithreading a student may refer SIMD-SMT parallelism [Kreinin, 2011]. The web site www.gpgpu.org has a number of definitive articles on the topic of GPGPU.

EXERCISES

- 5.1 Enumerate the problems encountered when the clock speed used in single chip processors is increased beyond 4 GHz.
- 5.2 The value of V_d of transistors used in microprocessor chips has continuously been reduced from 5 V to around 1 V. What are the pros and cons of reducing V_d .
- 5.3 The number of transistors which are packed in a microprocessor chip has been doubling every two years. Will this trend stop? Explain the reasoning for your answer.
- 5.4 List the major reasons which led to the emergence of multicore chips.
- 5.5 Multicore processors have been designed with a small number of very powerful cores or a large number of tiny cores. Discuss the type of problems for which each of these architectures is appropriate.
- 5.6 In Chapter 4, we described how parallel computers were built using a number of independent computers. Discuss to what extent those ideas may be used to design multicore parallel computers on a chip. Enumerate the significant differences between the two which require a different approach in designing chip multiprocessors.
- 5.7 Enumerate the differences between multithreaded parallelism and multicore parallelism.
- 5.8 What are the major design costs in designing multicore processors? How can this cost be optimized?
- 5.9 Enumerate the major constraints or walls which motivated the trend away from single core processor chips to many core processor chips. Among these which walls do you consider to be difficult to surmount. Justify your answer.
- 5.10 Give a generalized structure of chip multiprocessors. Enumerate various parts of a chip multiprocessor. What are combinations of these parts practical in designing chip multiprocessors.
- 5.11 What are the major aspects which a designer should keep in mind while designing multicore processors?
- 5.12 What modifications to instruction set architecture of a processor core are essential for it to be integrated in a chip multiprocessor?
- 5.13 Discuss the issues which arise while designing the interconnection system between cores in a multicore processor.
- 5.14 Two designs of multicore processors were described in Section 5.3. One had an L1 cache in each core and the L2 cache was shared. In the other, each core had both L1 and L2 caches and a shared L3 cache. Discuss the pros and cons of these two styles of design and the applications for which these designs are appropriate.
- 5.15 Give appropriate sizes for cache blocks of L1 and L2 caches in the two architectures alluded to in Exercise 5.14. Justify your answer.
- 5.16 Why are temperature sensors used in the cores of multicore processors? How are they used?
- 5.17 What are the major types of computers (e.g. desktop, tablet, smart phone etc.) in which multicore processors used? What is the nature of applications of these types of computers? What is the appropriate architecture of multicore chips for use in each type of computer?
- 5.18 What cache coherence algorithms are appropriate for the chip multiprocessor

architectures given in Figure 5.1.

- 5.19 In Sandy Bridge ring connected multicore shared memory computer system, a cache coherence protocol called MESIF by Intel is used. Find out about this protocol by searching the Internet. Obtain a decision table or a state diagram to show how this protocol is implemented.
- 5.20 What are the differences between a chip multiprocessor and a many core processor?
- 5.21 Enumerate the main differences between interconnection networks connecting independent computers (discussed in Chapter 4) and that used in many core processors.
- 5.22 Why are messages passed between cores in many core processors broken into packets and flits?
- 5.23 A many core processor has a cache block size of 128 bytes. If cache blocks are to be sent from a core to another core in a 64 core system interconnected by an 8×8 mesh explain how the cache block will be routed from a core (2, 4) to core (4, 8) where the first number is the x -coordinate and the second is the y -coordinate of the core. What will be the appropriate flit size?
- 5.24 What is the difference between TCP/IP routing of packets between computers and the routing used in a many core mesh connected parallel computer chip?
- 5.25 Explain how a switched ring interconnection works in a many core processor. Is it appropriate to use counter rotating rings in a switched ring system? If not explain why.
- 5.26 What is a token ring interconnection of cores in a multiprocessor. What are the advantages of token ring interconnection?
- 5.27 What is a ring bus? What are the advantages of a ring bus as compared to a bus?
- 5.28 Explain how MESI cache coherence protocol can be used with a ring bus connected multicore processor system.
- 5.29 Why is the ring bus in Intel Sandy Bridge architecture divided into four rings? What are the functions of each of these rings?
- 5.30 Why are counter rotating rings used? Enumerate their advantages and disadvantages.
- 5.31 The core clock speed is 3 GHz in a ring bus connected shared memory multicore processor with 8 cores. What is the time to transmit a cache block from core 0 to core 3? Assume cache block size of 64 bytes and data ring bus width of 256 bits.
- 5.32 What is a slotted ring? How is it different from a token ring? Explain how data can be transmitted in a pipeline mode in a slotted ring.
- 5.33 How many latches will be used to connect the Sandy Bridge rings to each core? How much time will be taken to transfer a byte from the core to the ring bus assuming the core is clocked at 3.5 GHz?
- 5.34 Look up Intel white paper on Intel Xeon Phi Coprocessor available in the web. Describe the ring bus configuration used to interconnect multiple cores. How is it different from the ones used in Sandy Bridge architecture?
- 5.35 Explain the cache coherence protocol used by Xeon Phi architecture.
- 5.36 What are the types of interconnection networks used to interconnect cores in a many core processors? List the advantages and disadvantages of each of these.
- 5.37 Discuss the issues in designing interconnection systems of many core processor after reading the paper by Jayasimha, Zafar and Hoskote, listed in the references at the end of this chapter.
- 5.38 Explain how routers are used to interconnect cores in a mesh connected many core

processors.

- 5.39 Watts/teraflop is a figure of merit used in evaluating many core coprocessors. What is the significance of this figure of merit? Find out from the Internet watts per teraflop for Xeon Phi, Intel Tera flop chip, NVIDIA's Fermi, and NVIDIA's Kepler.
- 5.40 A many core processor with 60 cores is to be designed using a mesh interconnection of cores. The MCP uses message passing. Draw a sketch of the MCP. How many routers will the MCP use? If the cache block size is 128 bytes, what will be the packet size? How will the routers be configured to interconnect the cores? Compare this MCP with Intel's Xeon Phi MCP. State all the parameters used to compare the systems.
- 5.41 Intel announced a "cloud many core chip". Explore the literature and write an article giving the architectural details of this chip.
- 5.42 What are the basic architectural differences between a GPGPU and many core processors such as Intel's Tera Flop chip?
- 5.43 What is stream processing? Give some applications in which such a model of processing is useful.
- 5.44 Describe how the *for* loop given in the text will be scheduled in NVIDIA's Fermi architecture GPGPU.
- 5.45 What is the reason, the register file used in Fermi GPGPU is larger than the L1 cache?
- 5.46 Compare the architectures of NVIDIA Fermi and Kepler GPGPUs.
- 5.47 Suppose you are asked to design a GPGPU with 24 SMs with each SM having 16 cores which will perform single precision floating point add/multiply. Assume that the system has a 1 GHz clock. What is the peak flop rating of this GPGPU assuming that all memory latencies could be hidden?
- 5.48 Compare SPMD, SIMD, SIMT, and array processors. Pick the criteria for comparison.

BIBLIOGRAPHY

- Barroso, L.A. and Dubois, M., "The Performance of Cache-Coherent Ring-based Multiprocessors", *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 1993, pp. 268–277. (barroso.org/publications/ring-isca.pdf).
- Blake, G., Drestrinski, R.G. and Mudge, T., "A Survey of Multicore Processors", *IEEE Signal Processing Magazine*, pp. 26–37, November 2009.
- Dubois, M., Annavaram, M. and Stenstrom, P., *Parallel Computer Organization and Design*, Cambridge University Press, UK, 2012.
- Hennessy, J.L. and Peterson, D.A., *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kauffman, Waltham, MA, USA, 2011.
- Hyman Jr., R., "Performance Issues in Multi-core Processors", www.csee.usf.edu/~rhyman/Projects/multicoreperformance.pdf.
- Intel Xeon Phi Coprocessor, "the architecture", Nov. 2012. (<https://software.intel.com/en-us/intel-xeon-phi-coprocessor>).
- Jayasimha, D.N., Zafar, B. and Hoskote, Y., *On-chip Interconnection Networks: Why They are Different and How to Compare Them*, Intel Publication, 2006.
- Kanter, D., "Intel's, Sandy Bridge for Servers", 2010. (www.realworldtech.com/Sandy-bridge-servers).
- Keckler, S.W., et al. (Ed.), *Multicore Processors and Systems*, Springer, USA, 2009.
- Kreinin, Y., SIMD-SMT Parallelism, 2011. (yosefk.com/blog/simd-simt-parallelism-in-Nvidia-gpus.html)
- Lipasti, M., "Multithreading Vs Multicore", 2005 (ece757.ece.missouri.edu/oct03-cores-multithreaded.tex)
- Marowka, A., "Back to Thin-core Massively Parallel Processors", *IEEE Computer*, Dec. 2011, pp. 49–54.
- Peh, L., Keckler, S.W. and Vangal, S., *On-chip Networks for Multicore Systems*, (in *Multicore Processors and Systems*, pp. 35–71), Springer, USA, 2009.
- Sodan, A.C., et al., "Parallelism via Multithreaded and Multicore CPUs", *IEEE Computer*, Vol. 43, March 2010, pp. 24–32.
- Stallings, W., *Computer Organization and Architecture*, 8th ed., Pearson, NJ, USA, 2010.
- Sutter, Herb, "Welcome to the Jungle", 2010. (www.herb-sutter.com/welcome-to-the-jungle).

Grid and Cloud Computing

There are three basic systems which are used to build parallel and distributed computer systems. They are:

1. Processor with built-in cache memory and external main memory
2. Secondary memory such as disks
3. Communication system to interconnect computers

There has been immense progress in the design and fabrication of all these three systems. Processors and memory systems which depend on semiconductor technology have seen very rapid development. The number of transistors which can be integrated in a microprocessor chip has been doubling almost every two years with consequent improvement of processing power at constant cost. This development has led to single chip multiprocessors as we saw in Chapter 5. Storage systems have also kept pace with the development of processors. Disk sizes double almost every eighteen months at constant cost. The density of packing bits on a disk's surface was 200 bits/sq.inch in 1956 and it increased to 1.34 Tbits/sq.inch in 2015, nearly 600 billion fold increase in 59 years. The cost of hard disk storage is less than \$0.03 per GB in 2015. Communication speed increase in recent years has been dramatic. The number of bits being transmitted on fibre optic cables is doubling every 9 months at constant cost. The improvement of communication is not only in fibre optics but also in wireless communication. A report published by the International Telecommunication Union [www.itu.int/ict] in 2011 states that between 2006 and 2011, the communication bandwidth availability in the world has increased eight fold and the cost of using bandwidth has halved between 2008 and 2011. It is predicted that by 2016 wireless bandwidth will overtake wired bandwidth.

The improvement in computer hardware technology has led to the development of servers at competitive cost. The increase of communication bandwidth availability at low cost has a profound impact. It has made the Internet all pervasive and led to many new computer environments and applications. Among the most important applications which have changed our lifestyle are:

- Email and social networks
- E-Commerce
- Search engines and availability of vast amount of information on the World Wide Web

Two important new computing environments which have emerged due to improvements in computing and communication technologies are:

- Grid computing
- Cloud computing

In this chapter, we will describe the emergence of these two environments and their impact on computing. We will also compare the two environments and their respective roles in the

computing scenario in general and parallel computing in particular.

6.1 GRID COMPUTING

We saw that the all pervasive Internet and the improvement of bandwidth of communication networks have spawned two new computing environments, namely, grid computing and cloud computing. In this section, we will describe grid computing which was historically the first attempt to harness the power of geographically distributed computers. A major consequence of the increase of speed of the Internet is that distance between a client and a server has become less relevant. In other words, computing resources may be spread across the world in several organizations, and if the organizations decide to cooperate, a member of an organization will be able to use a computer of a cooperating organization whenever he or she needs it, provided it is not busy. Even if it is busy, the organization needing the resource may reserve it for later use. High performance computers, popularly known as supercomputers have become an indispensable tool for scientists and engineers especially those performing research. High performance computer based realistic simulation models are now considered as important in science and engineering research as formulating hypotheses and performing experiments. “Big Science” is now a cooperative enterprise with scientists all over the world collaborating to solve highly complex problems. The need for collaboration, sharing data, scientific instruments (which invariably are driven by computers), and High Performance Computers led to the idea of grid computing [Berman, Fox and Hey, 2003]. Organizations may cooperate and share not only high performance computers but also scientific instruments, databases, and specialized software. The interconnected resources of cooperating organizations constitute what is known as a *computing grid*. The idea of grid computing was proposed by Ian Foster and his coworkers in the late 1990s, and has spawned several projects around the globe. A project called *Globus* [Foster and Kesselman, 1997] was started as a cooperative effort of several organizations to make the idea of grid computing a reality. Globus has developed standard tools to implement grid computing systems.

The Globus project (www.globus.org) defined a computer grid as: *an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases and scientific instruments owned and managed by multiple organizations*. It later refined the definition as: *a system that coordinates resources that are not subject to centralized control using standards, open general purpose protocols, and interfaces to deliver non trivial quality of service*.

Comparing these two definitions, we see that both of them emphasize the fact that the *resources are owned and operated by independent organizations, not subject to centralized control* and that the resources may be combined to deliver a requested service. The first definition emphasizes collaborative use of high performance computers and other facilities. The second definition emphasizes, in addition, the use of open standards and protocols as well as the importance of ensuring a specified Quality of Service (QoS). A number of important aspects related to grid computing which we may infer from these definitions are:

1. Remote facilities are used by an organization only when such facilities are not locally available.
2. Remote facilities are owned by other organizations which will have their own policies for use and permitting other organizations to use their facilities.
3. There may not be any uniformity in the system software, languages, etc., used by remote facilities.
4. While using remote facilities, an organization should not compromise its own

- security and that of the host.
5. There must be some mechanisms to discover what facilities are available in a grid, where they are available, policies on their use, priority, etc.
 6. As the grid belongs to a “virtual organization”, namely, separate organizations with agreed policies on collaboration, there must also be a policy defining Quality of Service (QoS). QoS may include a specified time by which the requested resource should be provided and its availability for a specified time.

In order to support a grid infrastructure a number of functions should be performed by software components. The important ones are:

1. Define the communication protocol to be used. Authenticate a user and determine his/her access rights to resources and limits on use of resources.
2. Interpret user’s request for resources and forward it in an appropriate form to a resource manager. Quote cost of resources requested where appropriate.
3. Maintain a directory of resources and the “cost” for their use. Very often the cost may not be real money but specified hours based on project approval.
4. Provide time estimate for job completion. This may normally be one of the QoS parameters.
5. Schedule users’ jobs and monitor progress of each job.
6. Encrypt data being sent to the grid and in general provide security of jobs.
7. Guard against threats such as viruses, worms etc.
8. Ensure security and privacy of users’ data as well as the data resident in the grid infrastructure.
9. Log diagnostic messages and forward them to the user.
10. Send results to the user when the job is completed.

These software components are typically organized as a layered architecture. Each layer depends on the services provided by the lower layers. Each layer in turn supports number of components which cooperate to provide the necessary service. The layered architecture is shown in Table 6.1.

TABLE 6.1 Layered Grid Architecture	
Layer name	Components in layer
Application layer	Scientific and engineering applications (e.g., TeraGrid Science Gateway). Collaborative computing environment (e.g., National Virtual Observatory). Software environment.
Collective layer	Directory of available services and resource discovery. Brokering (resource management, selection, aggregation), diagnostics, monitoring, and usage policies.
Resource layer	Resource availability, allocation, monitoring use, and payment/barter.
Connectivity layer	Communication and authentication protocols. Secure access to resources and services. Quality of Service monitoring.
Fabric layer	Computers, storage media, instruments, networks, databases, OS, software libraries.

The necessary software requirements to support a grid infrastructure are so huge that it is impractical for a single organization to develop it. Several organizations including computer industry participants have developed an open-source Globus toolkit consisting of a large suite

of programs. They are all standards-based and the components are designed to create appropriate software environment to use the grid. There is also a global *open grid forum* (www.ogf.org) which meets to evolve standards.

We will now examine how the grid infrastructure is invoked by a user and how it carries out a user's request. The steps are:

1. A user develops a program as a distributed program using appropriate tools available in the application layer software.
2. A user specifies computing, software, and data resources needed and QoS requirement and submits the task to a broker. (The broker is part of the collective layer). The broker has access to resource directory, equivalent resources, their availability and cost.
3. Using the above data, the broker finds the optimal set of resources which can be scheduled to carry out the user's request while meeting QoS requirement.
4. Before scheduling a user's job, the broker estimates the cost and verifies customer's capacity to pay based on its available credit. The broker may send the cost estimation to the user and get the user's concurrence before proceeding further.
5. If the user agrees, the broker schedules the job on the resources which it had identified. It monitors progress of the job and in case of any problem notifies the user.
6. When the job is successfully completed, the results along with the cost of running the job is sent to the user.

Very often the programs which are executed using the grid infrastructure are parallel programs which use the high performance computers and appropriate programming systems (such as MPI) we discuss in Chapter 8 of this book.

6.1.1 Enterprise Grid

The grid computing environment we described so far assumes that several organizations cooperate to form a virtual organization. Large enterprises often have computing facilities distributed in different locations. A multinational enterprise may, for instance, have computing facilities at branches spread in several countries. Each branch would have a local view. From an enterprise point of view, it is desirable to have a global view and optimize the configuration and the use of all the computers, particularly high performance servers located in the branches. The research done by the grid computer group and the availability of Globus software tools motivated large enterprises to examine creation of an enterprise grid. An enterprise grid would interconnect all computers and other resources, particularly databases to create a computing fabric accessible to all the members of the enterprise. Several advantages accrue to an enterprise when they use grid architecture. Some of them are as follows:

1. Savings which accrue due to the reduced idle time of computers. Idle computers in one location may be used by users in other locations of the enterprise.
2. Access to high performance computing to all employees who request them from their desktop.
3. Improvement in reliability and QoS.
4. Reduced hardware and software costs, reduced operational cost and improvement

- in the productivity of resources.
- 5. Better utilization of the enterprise's intranet.

Other benefits which accrue when an enterprise grid is used:

- 1. Promotion of collaboration between employees and formation of (virtual) teams spread across the enterprise.
- 2. Easy, seamless, and secure access to the remote instruments and sensors besides software and hardware.
- 3. In case of failure of a system at one location, the grid allows migration of applications to other locations of the enterprise so that there is minimal disruption.

Overall enterprise grid infrastructure benefits the enterprises by enabling innovations and making better business decisions as an enterprise wide view is now possible. It also reduces the overall cost of computing infrastructure.

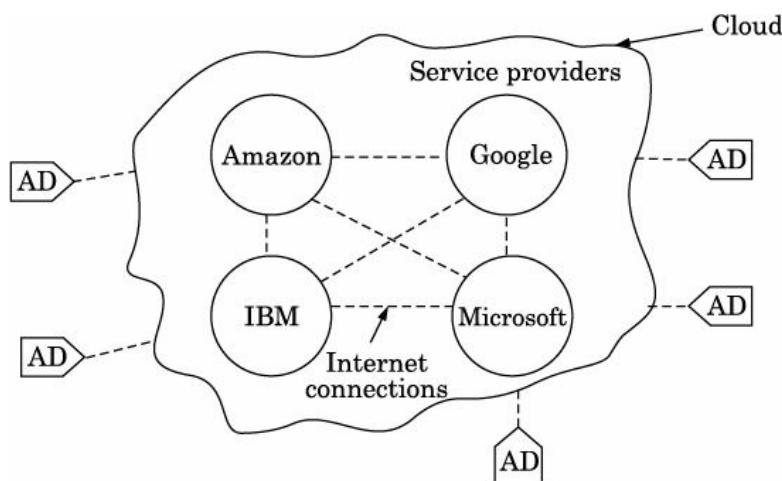
Several vendors have cooperated to establish the *enterprise grid alliance* (www.ega.org). The objectives of this alliance are to adopt and deploy grid standards including interoperability. They work closely with Globus consortium. An important point to be observed is that an enterprise grid need not be confined to a single enterprise. It may encompass several independent enterprises which agree to collaborate on some projects sharing resources and data.

There are also examples of enterprises collaborating with universities to solve difficult problems. One often cited industrial global grid project is distributed aircraft maintenance environment (www.wrgrid.org.uk/leaflets/DAME.pdf). In this project Rolls Royce aircraft engine design group collaborated with several UK universities to implement a decision support system for maintenance application. The system used huge amount of data which were logged by temperature sensors, noise sensors and other relevant sensors in real-time from thousands of aircrafts and analyzed them using a computing grid to detect problems and suggest solutions to reduce down time of aircraft and to avert accidents.

There are several such projects in the anvil and enterprise grids will be deployed widely in the near future.

6.2 CLOUD COMPUTING

There are many definitions of *cloud computing*. We give a definition adapted from the definition given by the National Institute of Standards and Technology, U.S.A. [Mell and Grance, 2011]. *Cloud computing is a method of availing computing resources from a provider, on demand, by a customer using computers connected to a network (usually the Internet)*. The name “cloud” in cloud computing is used to depict remote resources available on the Internet [which is often depicted as enclosed by a diffuse cloud in diagrams (see Fig. 6.1)] over which the services are provided. In Fig. 6.1 Amazon, Google, Microsoft, and IBM are providers of cloud services.



AD: Access Device to cloud (Desktop laptop, tablet, thin client, etc.)

Figure 6.1 Cloud computing system.

A cloud computing service has seven distinct characteristics:

1. The service is hosted on the Internet. It is sold by a service provider on demand to customers without any human intervention. The investment in infrastructure is made by the provider who also maintains it. Some of the service providers have massive hardware resources (more than 100,000 servers) which they can provide at short notice.
2. A customer has the flexibility to use any one or several services, namely computing infrastructure, data storage, compute servers with program development environment, or software on demand for any contracted period.
3. The service is fully managed by the provider and the customer does not have to invest in any infrastructure except in a device to access the Internet. The access device may be a desktop, a laptop, a tablet, a thin client, or a smart phone depending on the service required. The service may be accessed from any place and at any time by a customer.
4. The service is elastic. In other words, a customer can either increase or decrease the demand for disk space, server capacity, communication bandwidth, or any other service available from a provider. From a customer's point of view, the resources are unlimited. The customer pays only for the resources used.
5. The computing resources of a provider are geographically distributed and pooled. Several customers use the resources simultaneously. The servers of a provider are thus said to be “multi-tenanted”.

6. The system is adaptive. In other words, load is distributed automatically to the servers in such a way that the overall system utilization is optimal. A customer pays only for the resources utilized. A customer is permitted to monitor and control resource usage to minimize cost. The billing is thus transparent.
7. The service provider normally signs a Service Level Agreement (SLA) with customers assuring a specified Quality of Service (QoS).

Even though the cloud computing model is very similar to a utility such as a power utility, it is different in three important respects due to the distinct nature of computing. They are:

1. Cloud infrastructure is “virtualized”. In other words, even though the hardware servers are installed by the providers which may be provider dependent, from the customer’s perspective they may demand from the provider a specific computing environment they need. The provider employs virtualization software to meet customers’ demand.
2. As customers need to debug their programs, there is a two way interaction between customers and providers.
3. As the locations of computers and data storage of several major providers are distributed all over the world, there is lack of clarity on what laws apply if there is a dispute between the provider and a customer.

6.2.1 Virtualization

An important software idea that has enabled the development of cloud computing is *virtualization*. The dictionary meaning of virtual is “almost or nearly the thing described, but not completely”. In computer science the term virtual is used to mean: “a system which is made to behave as though it is a physical system with specified characteristics by employing a software layer on it”. For example, in a virtual memory system, a programmer is under the illusion that a main random access memory is available with a large address space. The actual physical system, however, consists of a combination of a small random access semiconductor memory and a large disk memory. A software layer is used to mimic a larger addressable main memory. This fact is hidden (i.e. transparent) to the programmer. This is one of the early uses of the term “virtual” in computing.

The origin of virtualization in servers was driven by the proliferation of many servers, one for each service, in organizations. For example, organizations use a server as a database server, one to manage email, another as a compute server and yet another as a print server. Each of these servers was used sub optimally. It was found that sometimes not more than 10% of the available server capacity was used as managers normally bought servers to accommodate the maximum load. Each of these applications used the facilities provided by the respective Operating Systems (OS) to access the hardware. The question arose whether all the applications can be executed on one server not interfering with one another, each one using its own native OS. This led to the idea of a software layer called *hypervisor* which runs on one server hardware and manages all the applications which were running on several servers. Each application “thinks” that it has access to an independent server with all the hardware resources available to it. Each application runs on its own “virtual machine”. By virtual machine we mean an application using the OS for which it is written. (See Fig. 6.2).

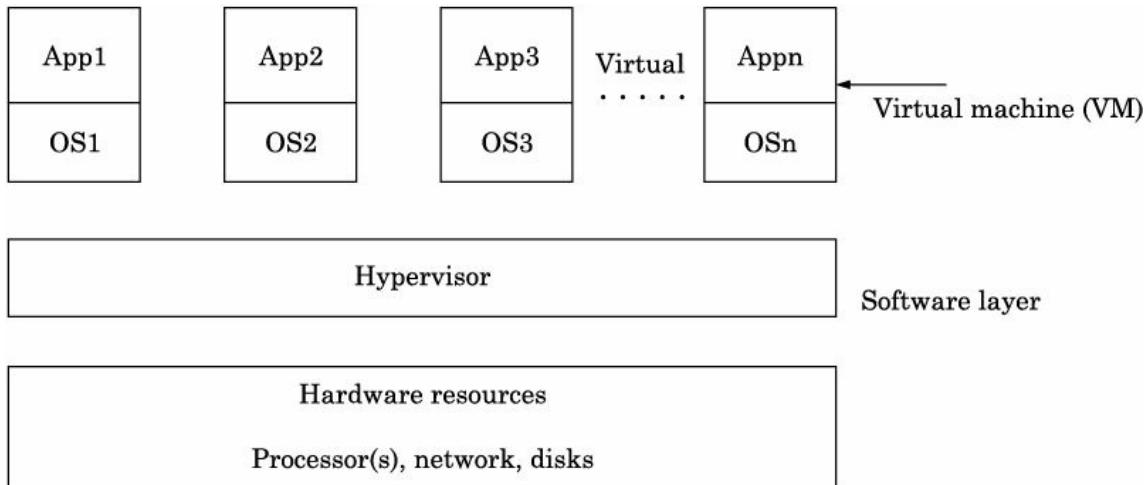


Figure 6.2 Virtual machines using a server.

There are several advantages which accrue when Virtual Machines (VM) are used. They are:

- The hardware is utilized better. As the probability of all the applications needing the entire hardware resources simultaneously is rare, the hypervisor can balance the load dynamically.
- If one of the applications encounters a problem, it can be fixed without affecting other applications.
- Virtual machines maybe added or removed dynamically.
- If a set of servers called a “server farm” is used, they can all be consolidated and a hypervisor can move VMs between the physical servers. If a physical server fails the VM running on it will not be affected as it can be moved to another server. The entire operation of shifting VMs among servers is hidden from the user. In other words, where a VM runs is not known to a user.

Virtualization software (namely hypervisor) is marketed by many companies; VMware and Microsoft are the ones with a large slice of the market. Two popular open source hypervisors are Xen and KVM.

6.2.2 Cloud Types

A cloud is normally classified as a *public cloud*, a *private cloud*, a *hybrid cloud*, or a *community cloud*.

A *public cloud* sells services to anyone who is willing to pay the charges. At any given time there may be several customers simultaneously using the service. This is said to be a *multi-tenanted* service. Amazon web service is an example of a public cloud provider. Amazon provides server time and disk space to store data. Amazon.com started as an e-commerce company which initially sold books and later expanded to sell other goods. In order to support its e-commerce business, Amazon had to maintain distributed infrastructure of servers, disks, etc. The utilization of the infrastructure was sub-optimal as the system had to be designed for peak load, for example, during “gift giving seasons” such as Christmas or Deepawali, and it was idle at other times. Amazon got an excellent idea of selling the slack time to customers requiring computing resources. Amazon marketed the computing service as Amazon Elastic Cloud Computing (EC2) [aws.amazon.com/ec2] and the storage service as S3 (Simple Storage Service). Other companies such as Google also maintained large server

farms to support their primary function (search engine) and found it is profitable to provide services, some of them free such as Gmail and Google drive to customers using this infrastructure. Many companies such as IBM and Microsoft also entered the market internationally. Companies such as Wipro, NetMagic, and Ramco started public cloud services in India.

In the case of a public cloud, a company (such as Amazon) provides, on demand, (virtual) servers, i.e., servers which mimic the software environment demanded by a customer, with unique IP addresses and blocks of disk. Virtualization is essential, as was explained in Section 6.2.1, to allow many customers with varying application requirements to use a group of servers maintained by a provider. Complex virtualization software and software to manage multiple virtual machines, called a *hypervisor*, are installed on the provider's servers. Customers use the provider's API (Application Programmer Interface) to start, access, configure their virtual servers and disk space, and terminate use. More resources are brought online by the vendor whenever demanded (sometimes quite huge, that is, over 10,000) by a customer. The service is also *multi-tenanted*, i.e., the same physical server may be allocated to several users in a time shared mode. This kind of service where the primary resources sold are compute servers and storage is called *Infrastructure as a Service* (IaaS).

A *private cloud* uses a proprietary network and computing infrastructure that is provided to a single customer. The infrastructure may be owned by the customer in which case it is similar to an enterprise grid. An enterprise may also outsource the service to a provider with the condition that no one shares the infrastructure.

An open source cloud implementation which provides an interface that is compatible with Amazon's EC2 is called *Eucalyptus* (<http://eucalyptus.cs.ucsb.edu>). It allows an organization to create a cloud infrastructure by connecting geographically distributed computing resources of their organization and experiment with it.

A *hybrid cloud* is a combination of a private cloud and a public cloud. An organization may normally run all its important applications on a private cloud and those which are accessible to the public such as its website on a public cloud. Sometimes when there is an unforeseen increase in the demand for some services, it may migrate some of these services to a public cloud provider. This is called *cloud bursting* in the literature.

A *community cloud* is one in which a set of cooperating institutions with similar requirements may request a provider to create a cloud infrastructure for their exclusive use. This is superficially similar to a grid.

6.2.3 Cloud Services

There are three major types of cloud services. They are called *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS).

In all types of cloud services (IaaS, PaaS, and SaaS), a provider maintains compute servers, storage servers, and communication infrastructure. In IaaS, customers may deploy and execute their own systems and application software. For instance, a customer may deploy an OS of its choice to execute its application. The IaaS provider may also install Application Programmer Interface (API) to access the servers. APIs allow monitoring the execution of applications and enable transparency in billing. In addition IaaS Provider would install *middleware* which is a software layer that supports complex distributed business applications running on a network of computers. Some examples of IaaS are Amazon's Elastic Cloud (EC2) and S3 storage service. The computing infrastructure provided are warehouse scale computers which we described in Chapter 4. The huge computing infrastructure needs secure buildings, massive air-conditioning, access control, uninterrupted power supply,

uninterrupted communication facility, and virtualization software.

We may define PaaS as a set of software and product development tools hosted on the provider's infrastructure. Different type of software development platforms such as UNIX, and .NET of Microsoft are provided. PaaS providers would normally install Application Programmer Interfaces (APIs) on customer's computer to permit them to access the service. Some providers do not permit software created in their platform to be moved elsewhere. Some examples of PaaS are: Windows Azure of Microsoft, Smart Cloud of IBM, and Google's App Engine.

In SaaS, a provider supplies software product(s) installed on its hardware infrastructure and interacts with the customer through a front-end portal. As both software and hardware are provided by the vendor, a customer may use the service from anywhere. Gmail is a free SaaS. In general, SaaS (not only Cloud SaaS) is a system whereby a vendor licenses software application to a customer either hosted on vendor's website or downloaded to a customer's computer on demand for a fixed period. A widely used SaaS is provided by "salesforce.com" to manage all the software requirements of marketing groups of companies. Figure 6.3 shows at a glance the differences between IaaS, PaaS, and SaaS.

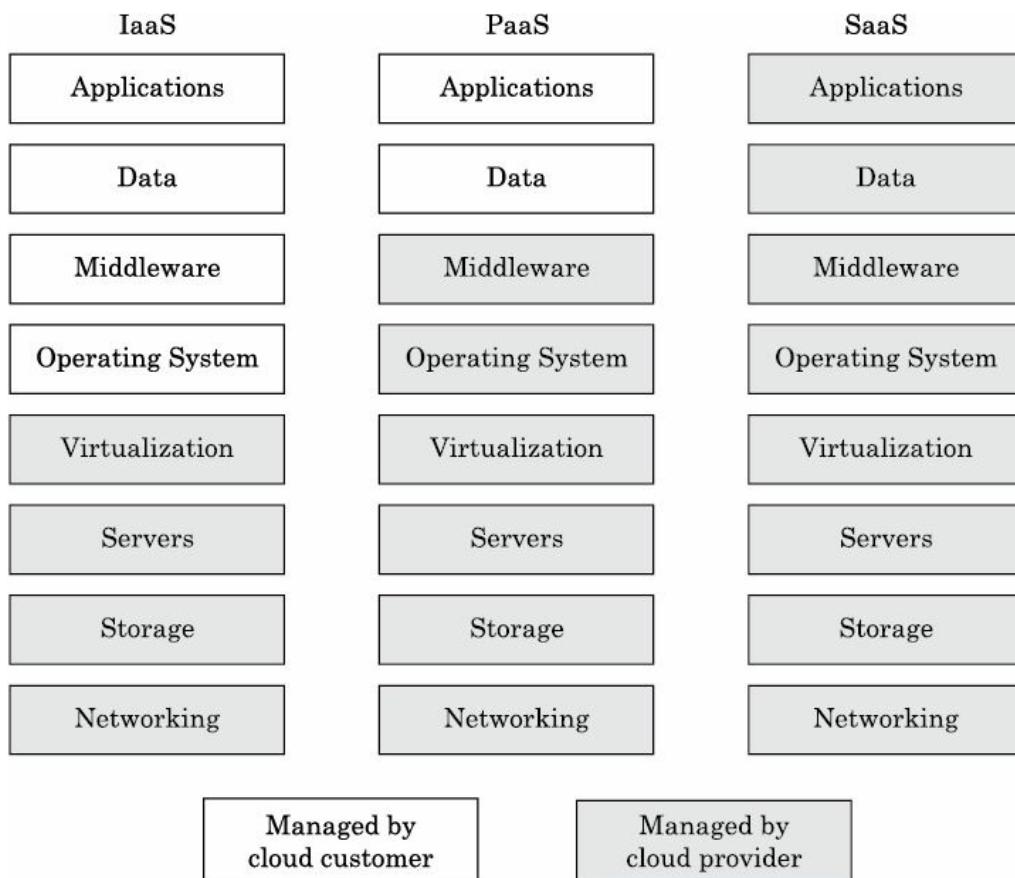


Figure 6.3 Types of cloud services showing how they differ.

6.2.4 Advantages of Cloud Computing

The primary advantages of cloud computing from users' perspective are:

1. An enterprise need not invest huge amounts to buy, maintain, and upgrade expensive computers.
2. The service may be hired for even short periods, e.g., one server for one hour. As

and when more resources such as more powerful servers or disk space is needed, a cloud service provider may be requested to provide it. As we pointed out earlier some providers may offer over 10,000 servers on demand. There is no need to plan ahead. This flexibility is of great value to small and medium size enterprises and also to entrepreneurs starting a software business. The billing is based on the resources and the period they are used.

3. System administration is simplified.
4. Several copies of commonly used software need not be licensed as payment is based on software used and time for which it is used.
5. A large variety of software is available on the cloud.
6. Migration of applications from one's own data centre to a cloud provider's infrastructure is easy due to virtualization and a variety of services available from a large number of cloud services providers.
7. Quality of service is assured based on service level agreements with the service provider.
8. The system is designed to be fault tolerant as the provider would be able to shift the application running on a server to another server if the server being used fails.
9. Organizations can automatically back up important data. This allows quick recovery if data is corrupted. Data archiving can be scheduled regularly.
10. Disaster recovery will be easy if important applications are running simultaneously on servers in different geographic areas.

The advantages from providers' perspective are [Armtrust, et al., 2009]:

1. Normally a provider would already have invested considerable amount in creating the infrastructure to support its primary service. For example, Amazon had already invested in large computing infrastructure to support its e-commerce service as we pointed out earlier. As it had excess capacity starting a cloud service required very little extra capital. Cloud services provided an additional revenue stream.
2. It is cheaper to buy a large number of servers as volume discounts would be available. Manpower cost to maintain both the software and hardware will be low per unit. The business can be very profitable if a good reputation is built on its quality of service.
3. The cloud provider can place the infrastructure in a location where electricity is cheap and uninterrupted power is available. The cost of power to run warehouse level computers to support a cloud service is significant due to cooling requirements as well as power needed to run servers. With the reduction in cost of solar power, location may also depend on the number of sunny days in the location. A cooler, low tax, and low real estate cost location is an advantage.
4. A company may decide to start a cloud service to prevent its existing customers whose services are outsourced to it from migrating to other providers. This was probably the motivation of IBM to start a cloud service.
5. Another motivation of a company to enter cloud business may be to protect its software investments and allow easy migration of its clients to its cloud. Microsoft Azure was probably started to allow its .NET platform and software systems to be relevant.
6. A company may find a business opportunity in starting a cloud service with a specialized platform for running some of the new web based services created by it.

6.2.5 Risks in Using Cloud Computing

The primary risks of using cloud computing are:

1. Managers of enterprises are afraid of storing sensitive data on disks with a provider due to security concerns especially due to the fact the cloud resources are shared by many organizations which may include their competitors. To mitigate this, strong encryption such as AES 256-bit should be used. Users are also afraid of losing data due to hardware malfunction. To mitigate this concern, a backup copy should be normally kept with the organization.
2. Failure of communication link to the cloud from an enterprise would seriously disrupt an enterprise's function. To mitigate this, duplicate communication links to the cloud should be maintained.
3. Failure of servers of a provider will disrupt an enterprise. Thus, Service Level Agreements (SLA) to ensure a specified QoS should be assured by the provider.
4. If the service of a cloud service provider deteriorates, managers may not be able to go to another provider as there are no standards for inter-operability or data portability in the cloud. Besides this, moving large volumes of data out of a cloud may take a long time (a few hundred hours) due to the small bandwidth available on the Internet.
5. If a provider's servers are in a foreign country and the data of a customer are corrupted or stolen, complex legal problems will arise to get appropriate compensation.
6. There is a threat of clandestine snooping of sensitive data transmitted on the Internet by some foreign intelligence agencies that use very powerful computers to decode encrypted data [Hoboken, Arnbak and Eijk, 2013].

Cloud computing is provided by several vendors. A customer signs a contract with one or more of these vendors. As we pointed out earlier, if a customer is dissatisfied with a vendor, it is difficult to migrate to another vendor as there is no standardization and thus no inter-operability. There is no service provided by a group of cooperating vendors.

6.2.6 What has Led to the Acceptance of Cloud Computing

During the last decade management philosophy has been changing. Managers have been concerned about the mounting cost of the computing infrastructure in their organizations. Almost all clerical operations are now computerized. Desktop computers have replaced typewriters and centralized compute servers are required to store databases of organizations and to process data. Each desktop computer requires licensed software such as office software whose cost adds up. A more disturbing aspect is the rapid obsolescence of computers requiring organizations to budget recurring capital expense almost every three years. The cost of specialized IT staff to maintain the hardware and software of the computing infrastructure has been another irritant. It was realized by the top management of organizations that computing is not the “core activity” of their organizations; it is only a means to an end. Management trend currently is to “outsource” non-core activities to service providers. Thus, the availability of cloud computing allows organizations to pay service providers for what they use and eliminates the need to budget huge amounts to buy and maintain large computing infrastructure is a welcome development. Those organizations which are concerned about security issues contract with providers to establish a private cloud

for their organization.

6.2.7 Applications Appropriate for Cloud Computing

A major reservation that many organizations have about shifting all their application to a cloud provider is due to their fear pertaining to the security of programs and data. This is not as serious as it is made out to be. Security breaches are most often due to improper controls within the organization. As cloud providers' sole business is selling data processing services, they employ very good professionals and follow best practices to ensure security. Organizations, as of now are cautious about handing over all their data processing to an outsider. Some of the applications which are appropriate for outsourcing to a cloud computing service provider are:

1. Routine data processing jobs such as pay roll processing, order processing, sales analysis, etc.
2. Hosting websites of organizations. The number of persons who may login to the websites of organizations is unpredictable. There may be surges in demand which can be easily accommodated by a cloud provider as the provider can allocate more servers and bandwidth to access the servers as the demand increases.
3. High performance computing or specialized software not available in the organization which may be occasionally required. Such resources may be obtained from a provider on demand.
4. Parallel computing, particularly Single Program Multiple Data (SPMD) model of parallel computing. This model is eminently suitable for execution on a cloud infrastructure as a cloud provider has several thousands of processors available on demand. This model is used, for example, by search engines to find the URLs of websites in which a specified set of keywords occur. A table of URLs and the keywords occurring in the web pages stored in each URL may be created. This table may be split into, say 1000 sets. Each set may be given to a computer with a program to search if the keywords occur in them and note the URLs, where they are found. All the 1000 computers search simultaneously and find the URLs. The results are combined to report all the URLs where the specified keywords occur. This can speed up the search almost thousand fold. Software called MapReduce [Dean and Ghemawat, 2004] was developed by Google which used this idea. An open source implementation of MapReduce called Hadoop [Turner, 2011] has been developed for writing applications on the cloud that process petabytes of data in parallel, in a reliable fault tolerant manner which in essence uses the above idea. We will discuss MapReduce in Chapter 8.
5. Seasonal demand such as processing results of an examination conducted during recruitment of staff once or twice a year.
6. Archiving data to fulfill legal requirements and which are not needed day to day. It is advisable to encrypt the data before storing it in a cloud provider's storage servers.

6.3 CONCLUSIONS

As of now (2015), cloud computing is provided by a large number of vendors (over 1000 worldwide) using the Internet. Each vendor owns a set of servers and operates them. There is no portability of programs or data from one vendor's facility to that of another. On the other hand, the aim of grid computing is to create a virtual organization by interconnecting the resources of a group of cooperating organizations. It has inter-operability standards and tools for portability of programs among the members of the group. It would be desirable to combine the basic idea of the grid, that of cooperation and open standards, with that of the cloud, namely, vendors maintaining infrastructure and providing services (IaaS, SaaS and PaaS) as a pay for use service to customers. This will result in what we term a *cooperative* or *federated cloud*. Such a cloud infrastructure is not yet available.

We conclude the chapter by comparing grid and cloud computing summarized in Table 6.2. This is based on a comparison given by Foster, et al. [2008]. As can be seen, these computing services provide a user a lot of flexibility. In fact, today it is possible for a user to dispense with desktop computers and use instead a low cost mobile thin client to access any amount of computing (even supercomputing), storage and sophisticated application programs and pay only for the facilities used. In fact Google has recently announced a device called Chromebook which is primarily a mobile thin client with a browser to access a cloud.

TABLE 6.2 Comparison of Grid and Cloud Computing		
Aspect compared	Grid computing	Cloud computing
Business model	<ul style="list-style-type: none"> • Cooperating organizations • Operation on project mode • Community owned infrastructure • No “real money” transaction • Driven originally by scientists working in universities • Cost shared. No profit/no loss model 	<ul style="list-style-type: none"> • Massive infrastructure with profit motive • Economics of sale • 100,000 + servers available on demand • Customers pay by use
Architecture	<ul style="list-style-type: none"> • Heterogeneous high performance computers • Federated infrastructure • Standard protocols allowing inter-operability • No virtualization 	<ul style="list-style-type: none"> • Uniform warehouse scale computers • Independent, competing providers • No standards for inter-operability • Iaas, Paas, Saas • Virtualized systems
Resource management	<ul style="list-style-type: none"> • Batch operation and scheduling • Resource reservation and allocation based on availability • Not multi-tenanted 	<ul style="list-style-type: none"> • Massive infrastructure allocation on demand • Multi-tenanted • Resource optimization

Programming model	<ul style="list-style-type: none"> • High performance computing parallel programming methods such as MPI 	<ul style="list-style-type: none"> • SPMD model • MapReduce, PACT etc.
Applications	<ul style="list-style-type: none"> • High performance computing • High throughput computing • Portal for application in scientific research 	<ul style="list-style-type: none"> • Loosely coupled transaction oriented • Mostly commercial • Big data analysis
Security model	<ul style="list-style-type: none"> • Uses public key based grid security infrastructure • Users need proper registration and verification of credentials 	<ul style="list-style-type: none"> • Customer credential verification lacks rigour • Multi-tenancy risk • Data location risk • Multinational legal issues

EXERCISES

- 6.1 What are the rates of increase of speed of computing, size of disk storage, and communication bandwidth as of now? What is the main implication of this?
- 6.2 The distance between a client and server is 500 m. The speed of the communication network connecting them is 10 Gbps. How much time will it take to transfer 10 MB of data from client to server? If the distance between a client and server is 100 km and the network speed is 1 Gbps, how much time will be required to transfer 10 MB data from client to server? How much time will be needed to transfer 1 TB data?
- 6.3 Define grid computing. We gave two definitions of grid computing in the text. Combine them and formulate a new definition.
- 6.4 Can a grid environment be used to cooperatively use several computers to solve a problem? If yes, suggest ways of achieving this.
- 6.5 What functions should be provided by the software system in a grid computing environment?
- 6.6 Why is grid software architecture specified as layers? Give the components in each layer and explain their functions.
- 6.7 Explain how a grid environment is invoked by a user and how it carries out a user's request for service.
- 6.8 What is an enterprise grid? In what way is it different from an academic grid?
- 6.9 What are the advantages which accrue to enterprises when they use an enterprise grid?
- 6.10 Define cloud computing. What are its distinct characteristics?
- 6.11 Define IaaS, SaaS, and PaaS.
- 6.12 In what way is cloud computing similar to grid computing?
- 6.13 Describe different types of clouds and their unique characteristics.
- 6.14 List the advantages of using cloud computing.
- 6.15 What are the risks of using cloud computing?
- 6.16 What are the applications which are appropriate for outsourcing to a cloud computing provider?
- 6.17 What are the major differences between cloud computing and grid computing?
- 6.18 What are the major differences between a community cloud and grid computing?
- 6.19 What are the major differences between a private cloud and a public cloud?

BIBLIOGRAPHY

- Amazon Elastic Compute Cloud (Amazon EC2), 2008, (aws.amazon.com/ec2).
- Armbrust, M., et al., “Above the Clouds: A Berkeley View of Cloud Computing”, Technical report EECS—2009–20, UC, Berkeley, 2009, (<http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>).
- Berman, F., Fox, G. and Hey, A. (Eds.), *Grid Computing: Making Global Infrastructure a Reality*, Wiley, New York, USA, 2003.
- Dean, J., and Ghemawat, S., “MapReduce: Simplified Data Processing in Large Clusters”, <http://research.google.com/archive/mapreduce.html>
- Foster, I. and Kesselman, C., “Globus: A Metacomputing Infrastructure Toolkit”, *Intl. Journal of Supercomputer Applications*, Vol. 11, No. 2, 1997, pp. 115–128.
- Foster, Ian, et al., Cloud Computing and Grid Computing 360 Degree Compared, 2008. ([www.arxiv.org/pdf/0901.0131](http://www.arxiv.org/pdf/0901.0131.pdf)).
- Hoboken, J. Van, Arnbak, A., and Van Eijk, N., Van, “Obscured by Clouds or How to Address Government Access to Cloud Data from Abroad”, 2013. <http://ssrn.com/abstract=2276103>.
- Hypervisor, Wikipedia, Oct., 2013.
- International Telecommunication Union, “ICT Facts and Figures”, Switzerland, 2011. (www.itu.int/ict).
- Mell, P. and Grance, T., “The NIST Definition of Cloud Computing”, National Institute of Standards and Technology, USA, Special Publication 800–145, Sept. 2011 (Accessible on the World Wide Web).
- Rajaraman, V. and Adabala, N., *Fundamentals of Computers*, (Chapter 17), 6th ed., PHI Learning, Delhi, 2014.
- Rajaraman, V., Cloud Computing, *Resonance*, Indian Academy of Sciences, Vol. 19, No. 3, 2014, pp. 242–258.
- Turner, J., “Hadoop: What it is, how it works, and what it can do”, 2011. <http://strata.oreilly.com/2011/01/what-is-hadoop.html>.

Parallel Algorithms

In order to solve a problem on a sequential computer we need to design an algorithm for the problem. The algorithm gives the sequence of steps which the sequential computer has to execute in order to solve the problem. The algorithms designed for sequential computers are known as *sequential algorithms*. Similarly, we need an algorithm to solve a problem on a parallel computer. This algorithm, naturally, is quite different from the sequential algorithm and is known as *parallel algorithm*. We gave some ideas on how parallel algorithms are evolved in Chapter 2. A parallel algorithm defines how a given problem can be solved on a given parallel computer, i.e., how the problem is divided into subproblems, how the processors communicate, and how the partial solutions are combined to produce the final result. Parallel algorithms depend on the kind of parallel computer they are designed for. Thus even for a single problem we need to design different parallel algorithms for different parallel architectures.

In order to simplify the design and analysis of parallel algorithms, parallel computers are represented by various abstract machine models. These models try to capture the important features of a parallel computer. The models do, however, make simplifying assumptions about the parallel computer. Even though some of the assumptions are not very practical, they are justified in the following sense: (i) In designing algorithms for these models one often learns about the inherent parallelism in the given problem. (ii) The models help us compare the relative computational powers of the various parallel computers. They also help us in determining the kind of parallel architecture that is best suited for a given problem.

In this chapter we study the design and analysis of parallel algorithms for the computational problems: (i) prefix computation, (ii) sorting, (iii) searching, (iv) matrix operations. These problems are chosen as they are widely used and illustrate most of the fundamental issues in designing parallel algorithms. Finally, we look at some recent models of parallel computation. Throughout this chapter we use Pascal-like language to express parallel algorithms.

7.1 MODELS OF COMPUTATION

In this section, we present various abstract machine models for parallel computers. These models are useful in the design and analysis of parallel algorithms. We first present the abstract machine model of a sequential computer for the sake of comparison.

7.1.1 The Random Access Machine (RAM)

This model of computing abstracts the sequential computer. A schematic diagram of the RAM is shown in Fig. 7.1. The basic functional units of the RAM are:

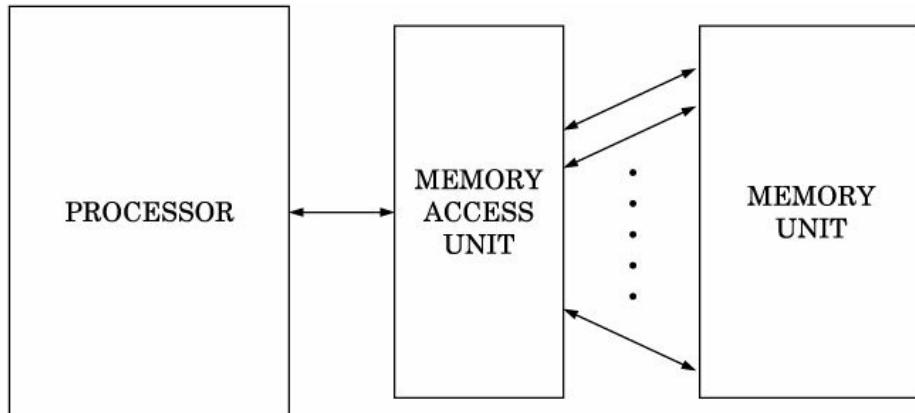


Figure 7.1 RAM model.

1. A memory unit with M locations. Theoretically speaking, M can be unbounded.
2. A processor that operates under the control of a sequential algorithm. The processor can read data from a memory location, write to a memory location, and can perform basic arithmetic and logical operations.
3. A Memory Access Unit (MAU), which creates a path from the processor to an arbitrary location in the memory. The processor provides the MAU with the address of the location that it wishes to access and the operation (read or write) that it wishes to perform; the MAU uses this address to establish a direct connection between the processor and the memory location.

Any step of an algorithm for the RAM model consists of (up to) three basic phases namely:

1. **Read:** The processor reads a datum from the memory. This datum is usually stored in one of its local registers.
2. **Execute:** The processor performs a basic arithmetic or logic operation on the contents of one or two of its registers.
3. **Write:** The processor writes the contents of one register into an arbitrary memory location.

For the purpose of analysis, we assume that each of these phases takes constant, i.e., $O(1)$ time.

7.1.2 The Parallel Random Access Machine (PRAM)

The PRAM is one of the popular models for designing parallel algorithms. As shown in Fig.

7.2, the PRAM consists of the following:

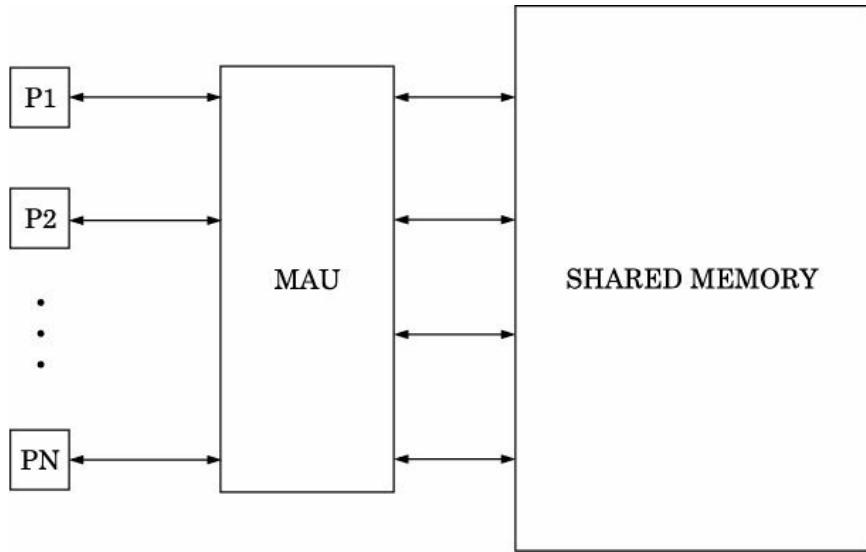


Figure 7.2 PRAM model.

1. A set of $N(P_1, P_2, \dots, P_N)$ identical processors. In principle, N is unbounded.
2. A memory with M locations which is shared by all the N processors. Again, in principle, M is unbounded.
3. An MAU which allows the processors to access the shared memory.

It is important to note that the shared memory also functions as the communication medium for the processors. A PRAM can be used to model both SIMD and MIMD machines. In this chapter, we see that an interesting and useful application of the PRAM occurs when it models an SIMD machine. When PRAM models an SIMD machine all the processors execute the same algorithm synchronously. As in the case of RAM, each step of an algorithm here consists of the following phases:

1. **Read:** (Up to) N processors read simultaneously (in parallel) from (up to) N memory locations (in the common memory) and store the values in their local registers.
2. **Compute:** (Up to) N processors perform basic arithmetic or logical operations on the values in their registers.
3. **Write:** (Up to) N processors write simultaneously into (up to) N memory locations from their registers.

Each of the phases, READ, COMPUTE, WRITE, is assumed to take $O(1)$ time as in the case of RAM. Notice that not all processors need to execute a given step of the algorithm. When a subset of processors execute a step, the other processors remain idle during that time. The algorithm for a PRAM has to specify which subset of processors should be active during the execution of a step.

In the above model, a problem might arise when more than one processor tries to access the same memory location at the same time. The PRAM model can be subdivided into four categories based on the way simultaneous memory accesses are handled.

Exclusive Read Exclusive Write (EREW) PRAM

In this model, every access to a memory location (read or write) has to be exclusive. This

model provides the least amount of memory concurrency and is therefore the weakest.

Concurrent Read Exclusive Write (CREW) PRAM

In this model, only write operations to a memory location are exclusive. Two or more processors can concurrently read from the same memory location. This is one of the most commonly used models.

Exclusive Read Concurrent Write (ERCW) PRAM

This model allows multiple processors to concurrently write into the same memory location. The read operations are exclusive. This model is not frequently used and is defined here only for the sake of completeness.

Concurrent Read Concurrent Write (CRCW) PRAM

This model allows both multiple read and multiple write operations to a memory location. It provides the maximum amount of concurrency in memory access and is the most powerful of the four models.

The semantics of the concurrent read operation is easy to visualize. All the processors reading a particular memory location read the same value. The semantics of the concurrent write operation on the other hand is not obvious. If many processors try to write different values to a memory location, the model has to specify precisely, the value that is written to the memory location. There are several protocols that are used to specify the value that is written to a memory in such a situation. They are:

1. **Priority CW:** Here the processors are assigned certain priorities. When more than one processor tries to write a value to a memory location, only the processor with the highest priority (among those contending to write) succeeds in writing its value to the memory location.
2. **Common CW:** Here the processors that are trying to write to a memory location are allowed to do so only when they write the *same* value.
3. **Arbitrary CW:** Here one of the processors that is trying to write to the memory location succeeds and the rest fail. The processor that succeeds is selected arbitrarily without affecting the correctness of the algorithm. However, the algorithm must specify exactly how the successful processor is to be selected.
4. **Combining CW:** Here there is a function that maps the multiple values that the processors try to write to a single value that is actually written into the memory location. For instance, the function could be a summing function in which the sum of the multiple values is written to the memory location.

7.1.3 Interconnection Networks

In the PRAM, all exchanges of data among processors take place through the shared memory. Another way for processors to communicate is via direct links connecting them. There is no shared memory. Instead, the M locations of memory are distributed among the N processors. The local memory of each processor now consists of M/N locations. In Chapter 4 we saw several interconnection networks (topologies) used in parallel computers. A network topology (which specifies the way the processors are interconnected) is important to specify the interconnection network model of a parallel computer as it plays a vital role in determining the computational power of the parallel computer.

7.1.4 Combinational Circuits

Combinational circuit is another model of parallel computers. As with the inter-connection

network model of parallel computers, the term *combinational circuit* refers to a family of models of computation. A combinational circuit can be viewed as a device that has a set of input lines at one end and a set of output lines at the other. Such a circuit is made up of a number of interconnected *components* arranged in columns called *stages*. Each component, which can be viewed as a simple processor, has a fixed number of input lines called *fan-in* and a fixed number of output lines called *fan-out*. And each component, having received its inputs, does a simple arithmetic or logical operation in *one time unit* and produces the result as output. A component is active only when it receives all the inputs necessary for its computation.

A schematic diagram of a combinational circuit is shown in Fig. 7.3. For convenience, the fan-in and the fan-out are assumed to be 2 in this figure. An important feature of a combinational circuit is that it has no feedback, i.e., no component can be used more than once while computing the circuit's output for a given input. The important parameters that are used to analyze a combinational circuit are:

1. **Size:** It refers to the number of components used in the combinational circuit.
2. **Depth:** It is the number of stages in the combinational circuit, i.e., the maximum number of components on a path from input to output. Depth, therefore, represents the worst case running time in solving a problem, assuming that all inputs arrive at the same time and all outputs exit at the same time.
3. **Width:** It is the maximum number of components in a given stage.

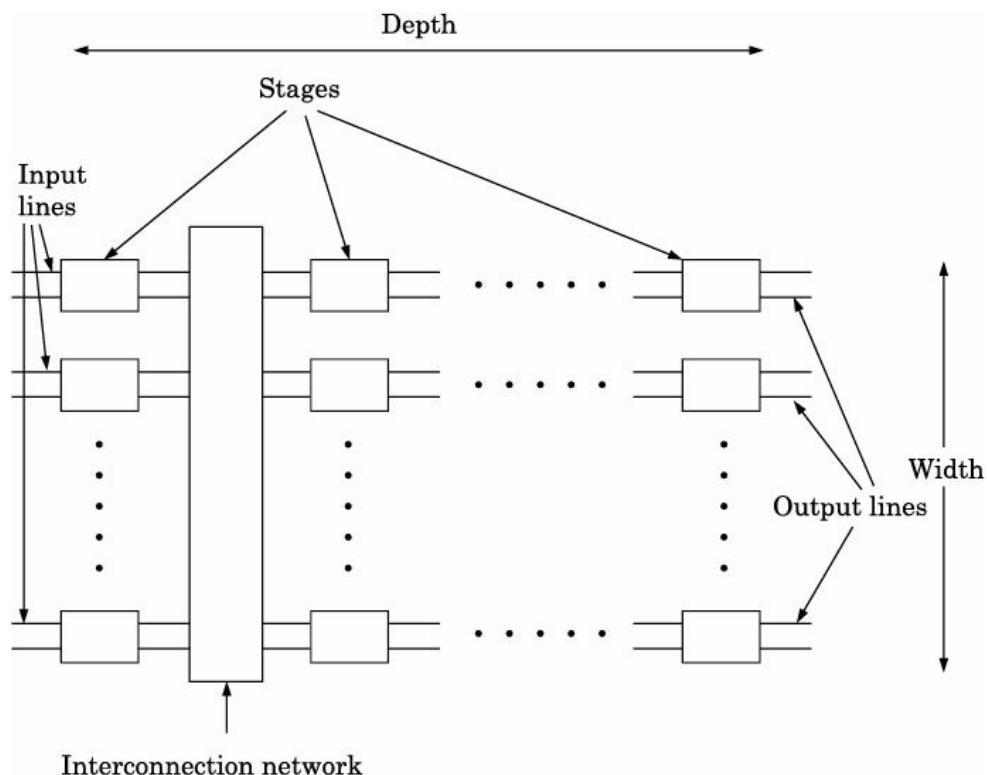


Figure 7.3 Combinational circuit.

Note that the product of the depth and width gives an upper bound on the size of the circuit. Now consider the circuit of Fig. 7.4, known as *butterfly circuit*, which is used in designing a combinational circuit for the efficient computation of Fast Fourier Transform (FFT). The butterfly circuit has n inputs and n outputs. The depth of the circuit is $(1 + \log n)$ and the width of the circuit is n . The size in this case is $(n + n \log n)$. Note that in Fig. 7.4, $n = 8$.

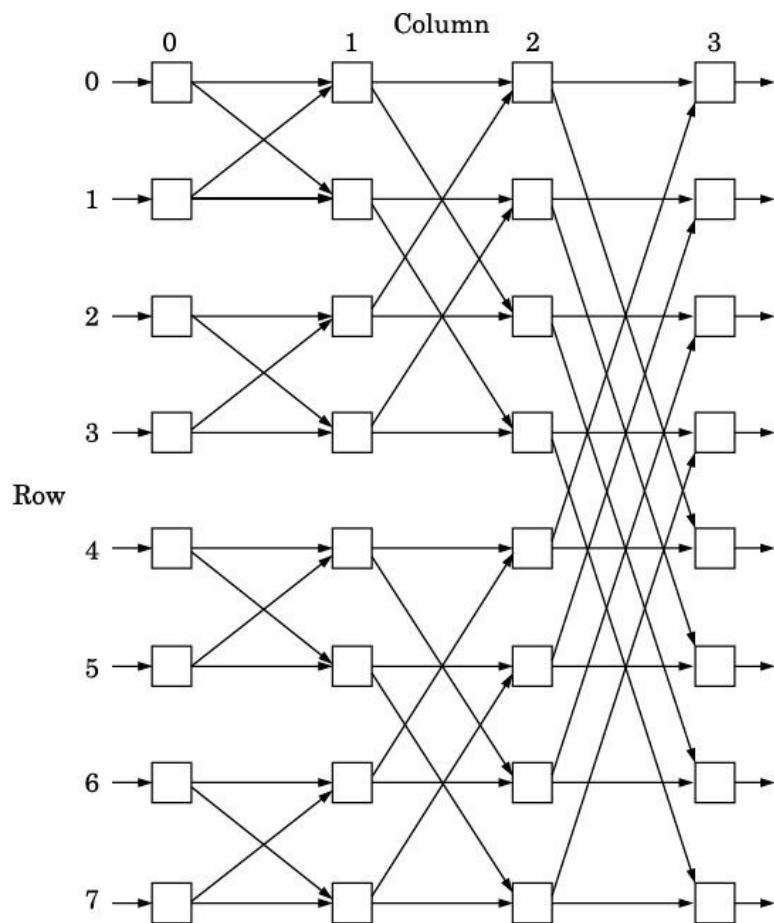


Figure 7.4 Butterfly circuit.

7.2 ANALYSIS OF PARALLEL ALGORITHMS

Any sequential algorithm is evaluated in terms of two parameters or criteria, viz, the running (execution) time complexity and the space complexity. A “good” sequential algorithm has a small running time and uses as little space as possible. In this section, we will look at the criteria that are used in evaluating parallel algorithms. There are three principal criteria that are typically used for evaluating parallel algorithms:

1. Running time
2. Number of processors
3. Cost

7.2.1 Running Time

Since speeding up solution of a problem is the main reason for building parallel computers, an important measure in evaluating a parallel algorithm is its *running time*. This is defined as the time taken by the algorithm to solve a problem on a parallel computer. A parallel algorithm is made up of two kinds of steps. In the *computation step*, a processor performs a local arithmetic or logic operation. In the *communication step* data is exchanged between the processors via the shared memory or through the interconnection network. Thus the running time of a parallel algorithm includes the time spent in both the computation and the communication steps.

The running time of an algorithm is a function of the input given to the algorithm. The algorithm may perform well for certain inputs and relatively badly for certain others. The *worst case running time* is defined as the maximum running time of the algorithm taken over all the inputs. The *average case running time* is the average running time of the algorithm over all the inputs.

The running time as described depends not only on the algorithm but also on the machine on which the algorithm is executed. Moreover, the running time is a function of the size of the input. To compare two algorithms, we need a machine independent notion of the running time of an algorithm. To this end, we describe the running time of the algorithm using the *order* notation.

Notion of Order

Consider two functions $f(n)$ and $g(n)$ defined from the set of positive integers to the set of positive reals.

1. The function $g(n)$ is said to be *of order at least* $f(n)$ denoted by $\Omega(f(n))$ if there exist positive constants c, n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$.
2. The function $g(n)$ is said to be *of order at most* $f(n)$ denoted by $O(f(n))$ if there exist positive constants c, n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$.
3. The function $g(n)$ is said to be *of the same order* as $f(n)$ if $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.

Consider the following examples: The function $g(n) = n^2$ is of order at least $f(n) = n \log n$. In this case, one can consider $c = 1, n_0 = 1$. The function $g(n) = 4n^2 + 3n + 1$ is of the same order as the function $f(n) = n^2 + 6n + 3$.

The order notation is used to compare two algorithms. For instance, an algorithm whose

running time is $O(n)$ is asymptotically better than an algorithm whose running time is $O(n^2)$ because as n is made larger n^2 grows much faster than n .

Lower and Upper Bounds

Every problem can be associated with a lower and an upper bound. The lower bound on a problem is indicative of the minimum number of steps required to solve the problem in the worst case. If the number of steps taken by an algorithm for execution in the worst case is equal to or of the same order as the lower bound, then the algorithm is said to be optimal. Consider the matrix multiplication problem. The best known algorithm for this particular problem takes $O(n^{2.38})$ time while the known lower bound for the problem is $\Omega(n^2)$. In the case of sorting, the lower bound is $\Omega(n \log n)$ and the upper bound is $O(n \log n)$. Hence probably optimal algorithms exist for the sorting problem while no such algorithm exists for matrix multiplication to date.

Our treatment of lower and upper bounds in this section has so far focused on sequential algorithms. Clearly, the same general ideas also apply to parallel algorithms while taking two additional factors into account: (i) the model of parallel computation used and (ii) the number of processors in the parallel computer.

Speedup

A good measure for analyzing the performance of a parallel algorithm is *speedup*. Speedup is defined as the ratio of the worst case running time of the best (fastest) known sequential algorithm and the worst case running time of the parallel algorithm.

$$\text{Speedup} = \frac{\text{Running time of best sequential algorithm}}{\text{Running time of parallel algorithm}}$$

Greater the value of speedup, better is the parallel algorithm. The speedup S of any algorithm A is less than or equal to N where N denotes the number of processors in the parallel computer. The lower bound for summing (adding) n numbers is $O(\log n)$ for any interconnection network parallel computer. The maximum speedup achievable for the summing problem is therefore $O(n/\log n)$, as the best sequential algorithm for this problem takes $O(n)$ time.

7.2.2 Number of Processors

Another important criterion for evaluating a parallel algorithm is the *number of processors* required to solve the problem. Given a problem of input size n , the number of processors required by an algorithm is a function of n denoted by $P(n)$. Sometimes the number of processors is a constant independent of n .

7.2.3 Cost

The *cost* of a parallel algorithm is defined as the product of the running time of the parallel algorithm and the number of processors used. It is indicative of the number of steps executed collectively by all the processors in solving a problem in the worst case. Note that in any step of a parallel algorithm a subset of processors could be idle. The above definition of cost includes all these *idle* steps as well.

$$\text{Cost} = \text{Running time} \times \text{Number of processors}$$

If the cost of the parallel algorithm matches the lower bound of the best known sequential algorithm to within a constant multiplicative factor, then the algorithm is said to be *cost-optimal*. The algorithm for adding (summing) n numbers takes $O(\log n)$ steps on an $(n - 1)$,

processor tree. The cost of this parallel algorithm is thus $O(n \log n)$. But the best sequential algorithm for this problem takes $O(n)$ time. Therefore, the parallel algorithm is not cost-optimal.

The *efficiency* of a parallel algorithm is defined as the ratio of the worst case running time of the fastest sequential algorithm and the cost of the parallel algorithm.

$$\text{Efficiency} = \frac{\text{Worst case running time of best sequential algorithm}}{\text{Cost of parallel algorithm}}$$

Usually, efficiency is less than or equal to 1; otherwise a faster sequential algorithm can be obtained from the parallel one!

7.3 PREFIX COMPUTATION

Prefix computation is a very useful suboperation in many parallel algorithms. The prefix computation problem is defined as follows.

A set χ is given, with an operation o on the set such that

1. o is a binary operation.
2. χ is closed under o .
3. o is associative.

Let $X = \{x_0, x_1, \dots, x_{n-1}\}$, where $x_i \in \chi$ ($0 \leq i \leq n - 1$). Define

$$s_0 = x_0$$

$$s_1 = x_0 o x_1$$

.

.

.

$$s_{n-1} = x_0 o x_1 o \dots o x_{n-1}$$

Obtaining the set $S = \{s_0, s_1, \dots, s_{n-1}\}$ given the set X is known as the *prefix computation*.

Note that the indices of the elements used to compute s_i form a string $012 \dots i$, which is a prefix of the string $012 \dots n - 1$, and hence the name prefix computation for this problem. Until explicitly mentioned otherwise, we assume that the binary operation o takes constant time. In this section, we look at some algorithms for the prefix computation on the PRAM models.

7.3.1 Prefix Computation on the PRAM

For simplifying the discussion, let us assume that the set χ is the set of natural numbers and the operation o is the $+$ operation. Let $X = \{x_0, x_1, \dots, x_{n-1}\}$ be the input. We need to find the partial sums s_i ($0 \leq i \leq n - 1$) where $s_i = x_0 + x_1 + \dots + x_i$. On a RAM this can be done easily in $O(n)$ time.

The PRAM algorithm given here requires n processors P_0, P_1, \dots, P_{n-1} , where n is a power of 2. There are n memory locations m_0, m_1, \dots, m_{n-1} . Initially the memory location m_i ($0 \leq i \leq n - 1$) contains the input x_i . When the algorithm terminates the location m_i ($0 \leq i \leq n - 1$) has the partial sum output s_i . The algorithm is given below. It consists of $(\log n)$ iterations; during each step, the binary operation $+$ is performed by pairs of processors whose indices are separated by a distance twice that in the previous iteration. This computation is illustrated in Fig. 7.5.

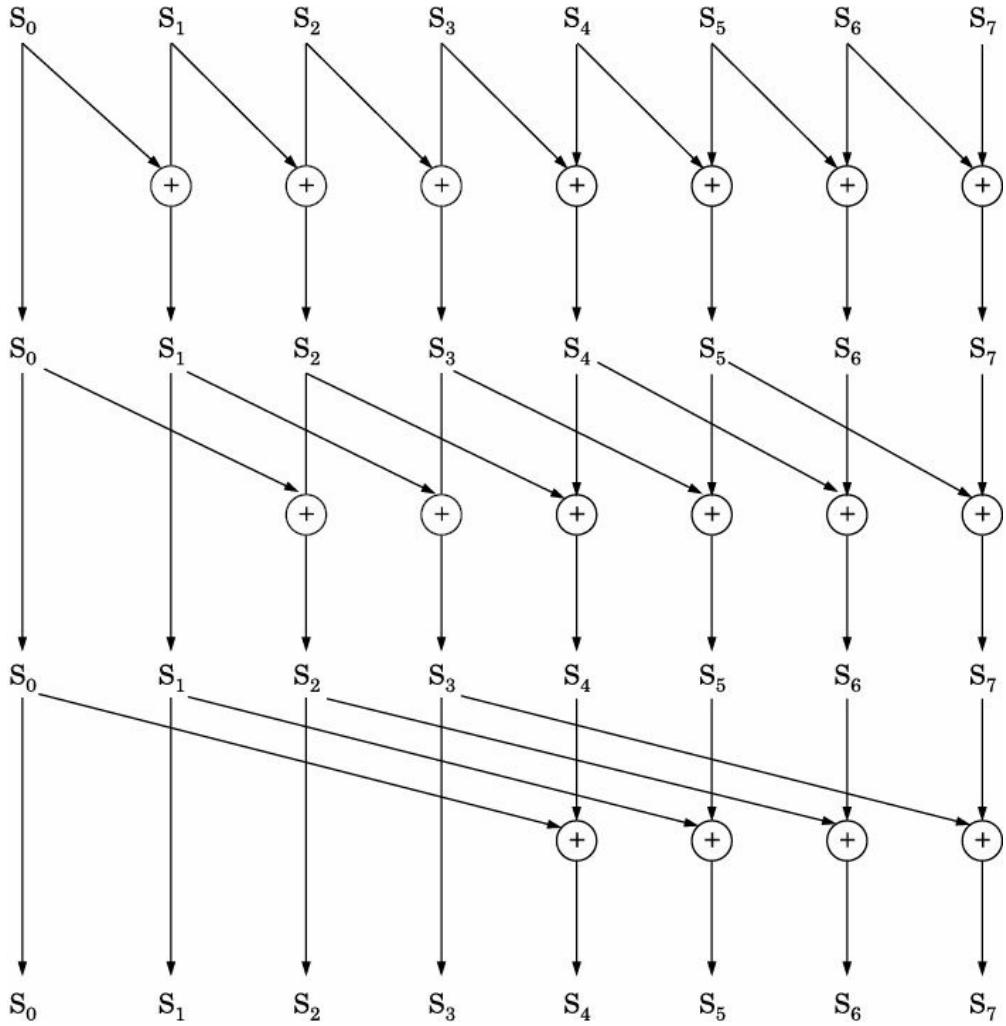


Figure 7.5 Prefix computation on the PRAM.

Procedure Prefix Computation

```

for  $j := 0$  to  $(\log n) - 1$  do
  for  $i := 2^j$  to  $n-1$  do in parallel
     $s_i := s_{(i - 2^j)} + s_i$ 
  end for
end for

```

Analysis

There are $\log n$ iterations in this algorithm. In each iteration one addition is done in parallel and hence takes $O(1)$ time. The time complexity of the algorithm is therefore $O(\log n)$. Since the algorithm uses n processors; the cost of the algorithm is $O(n \log n)$, which is clearly not cost optimal. Note that this algorithm has the optimal time complexity among the non-CRCW machines. This is because prefix computation of s_{n-1} requires $O(n)$ additions which has a lower bound of $O(\log n)$ on any non-CRCW parallel machine.

A Cost Optimal Algorithm for Prefix Computation

We now present a cost optimal parallel algorithm for the prefix sums computation. This algorithm also runs in $O(\log n)$ time but makes use of fewer processors to achieve this. Let $X = \{x_0, x_1, \dots, x_{n-1}\}$ be the input to the prefix computation. Let $k = \log n$ and $m = n/k$. The algorithm uses m processors P_0, P_1, \dots, P_{m-1} . Processors P_0, P_1, \dots, P_{m-1} . The input sequence

X is split into m subsequences, each of size k , namely

$$Y_0 = x_0, x_1, \dots, x_{k-1}$$

$$Y_1 = x_k, x_{k+1}, \dots, x_{2k-1}$$

.

.

.

$$Y_{m-1} = x_{n-k}, x_{n-k+1}, \dots, x_{n-1}$$

The algorithm given below proceeds in three steps.

In step 1, the processor P_i ($0 \leq i < m - 1$) works on the sequence Y_i , and computes the prefix sums of the sequence Y_i using a sequential algorithm. Thus the processor P_i computes $s_{ik}, s_{ik+1}, \dots, s_{(i+1)k-1}$, where

$$s_{ik+j} = x_{ik} + x_{ik+1} + \dots + x_{ik+j}$$

for $j = 0, 1, \dots, k-1$.

In step 2, the processors P_0, P_1, \dots, P_{m-1} perform a parallel prefix computation on the sequence $\{s_{k-1}, s_{2k-1}, \dots, s_{n-1}\}$. This parallel prefix computation is done using the algorithm described earlier. When this step is completed, s_{ik-1} will be replaced by $s_{k-1} + s_{2k-1} + \dots + s_{ik-1}$.

In the final step, each processor P_i ($1 < i < m - 1$) computes $s_{ik+j} = s_{ik-1} + s_{ik+j}$, for $j = 0, 1, \dots, k-2$. The working of this algorithm is illustrated in Fig. 7.6. Here $X = \{1, 2, \dots, 16\}$ and $m = 16/4 = 4$.

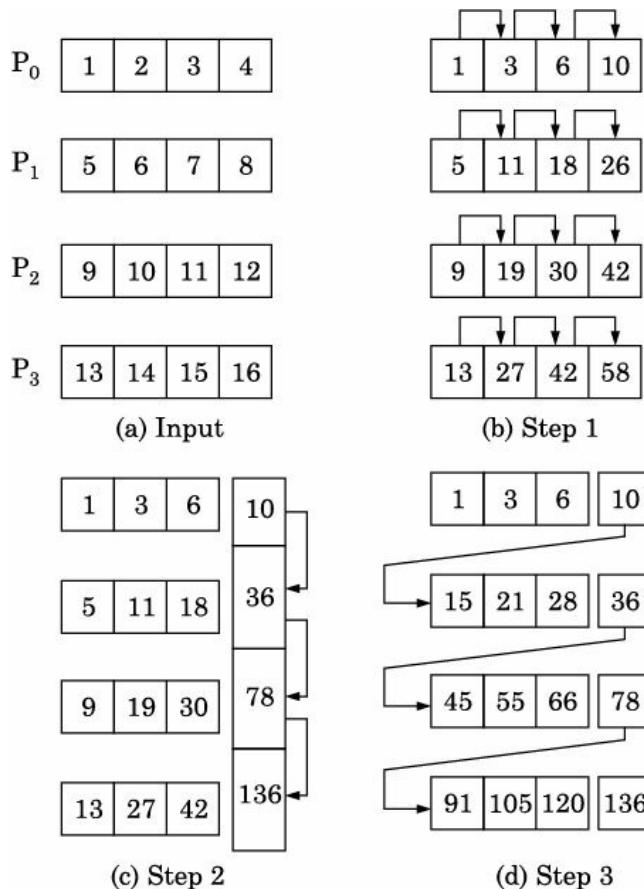


Figure 7.6 Optimal prefix computation on the PRAM.

Procedure Optimal Prefix Computation

```

Step 1: for  $i := 0$  to  $m-1$  do in parallel
     $s_{ik} := x_{ik}$ 
    for  $j := 1$  to  $k-1$  do
         $s_{ik+j} := s_{ik+j-1} + x_{ik+j}$ 
    end for
end for

Step 2: for  $j := 0$  to  $(\log m) - 1$  do
    for  $i := 2^j + 1$  to  $m$  do in parallel
         $s_{ik-1} := s_{(i-2^j)k-1} + s_{ik-1}$ 
    end for
end for

Step 3: for  $i := 1$  to  $m-1$  do in parallel
    for  $j := 0$  to  $k-2$  do
         $s_{ik+j} := s_{ik-1} + s_{ik+j}$ 
    end for
end for

```

Analysis

Steps 1 and 3 of the algorithm require $O(k)$ time. Step 2 requires $O(\log m)$ time. Since $k = (\log n)$ and $m = n/(\log n)$, the running time of the algorithm is $O(\log n) + O(\log(n/\log n))$ which is $O(\log n)$. The cost of the algorithm is $O(\log n) \times (n/\log n)$ which is $O(n)$. Hence this algorithm is cost optimal.

7.3.2 Prefix Computation on a Linked List

We will now look at prefix computation on linked lists. The algorithm for the prefix computation which we describe here illustrates a powerful technique, called **pointer jumping**. The idea of pointer jumping is used in solving problems whose data are stored in pointer-based data structures such as linked lists, trees, and general graphs. The exact linked list data structure that is used in the rest of this subsection is given below.

Linked List Data Structure

A linked list is composed of *nodes* that are linked by *pointers*. Each node i of the linked list consists of the following:

1. An information field, denoted by $\text{info}(i)$, containing some application dependent information.
2. A value field, denoted by $\text{val}(i)$, holding a number x_j .
3. A pointer field, denoted by $\text{succ}(i)$, pointing to the next node (successor) in the linked list.

The last node in the list, the *tail*, has no successor. Hence its pointer is equal to *nil*. In what follows, we omit the *info* field from the discussion.

Linked List Prefix Computation

In the linked list version of the prefix computation, the values of the input set X are given as elements of a linked list. Consider a linked list L shown in Fig. 7.7. Given the linked list L , we need to perform the prefix computation

$$x_0, x_0 o x_1, x_0 o x_1 o x_2, \dots$$

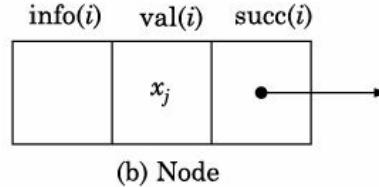
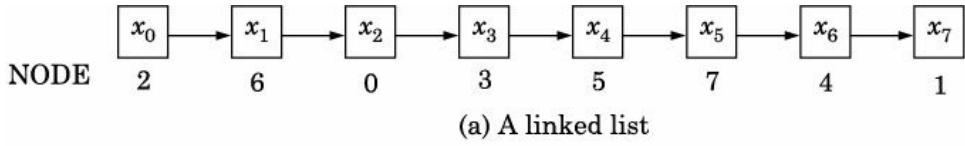


Figure 7.7 Linked list data structure.

for some operation o . The final required configuration of the linked list is shown in Fig. 7.8. The value x_{ij} denotes the sum $x_i o x_{i+1} o \dots o x_j$. There are certain issues that arise in the case of the linked list parallel prefix computation that do not arise in the usual prefix computation. These are:

- Each node of the linked list is assigned to a processor. However, the processor knows only the “location” of the node that is assigned to it. It does not know the relative position of the node in the linked list. It can however access the next node in the linked list by following the $\text{succ}(i)$ pointer. This is in contrast to the usual case of the prefix computation where each processor knows the position (or index) of the element it is working with. (Though the notion of a node’s index is

meaningless here, we still use it in order to be able to distinguish among nodes and their respective fields. Note that the index of a node denotes its position from the beginning of the list. For instance, in Fig. 7.7, node 2 has an index 0 and node 5 has an index 4.)

- None of the processors knows how many nodes are present in the linked list. In other words, each processor has only a “local” picture of the linked list, and not the “global” one. Hence the processors participating in the prefix computation should have some means of determining when the algorithm has terminated.

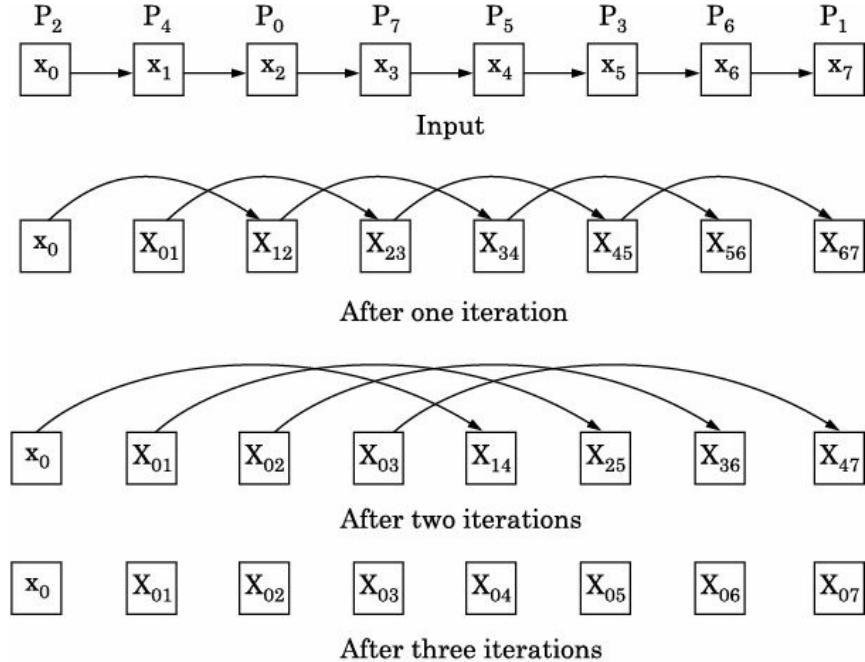


Figure 7.8 Prefix computation by pointer jumping.

The algorithm given here uses $N \geq |L|$ processors, to compute the linked list partial sums, where $|L|$ is the number of nodes in the linked list. At each iteration of the algorithm, all the processors execute the following steps:

1. Perform an o operation on the value of its node and the value of its successor node, and leave the resultant value in the successor node.
2. Update its $\text{succ}(i)$ pointer to the successor of the current successor node.

The iterations of the algorithm are shown in Fig. 7.8 for $N = |L| = 8$. Notice that this algorithm is quite similar to the prefix computation algorithm described earlier except that it uses pointer jumping to determine the nodes which perform the operation.

An important issue in this algorithm is the termination. The processors should know when the algorithm has terminated. For this purpose, the algorithm uses a COMMON CW memory location called *finished*. After each iteration, all the processors whose *succ* is not *nil* write a **false** to *finished*. Because of the COMMON CW property, the value in *finished* becomes **true** only when all the processors write **true** to it. Hence when *finished* contains the value **true** the algorithm terminates. At the beginning of each iteration, all processors read *finished* and if it is **false**, determine that another iteration is required. This way, all processors will “know” simultaneously when the algorithm has terminated.

Procedure PRAM Linked List Prefix Computation

```

Step 1: for all  $i$  do in parallel
         $\text{next}(i) := \text{succ}(i)$ 
    end for
Step 2:  $\text{finished} := \text{false}$ 
Step 3: while  $\text{not finished}$  do
         $\text{finished} := \text{true}$ 
        for all  $i$  do in parallel
            if  $\text{next}(i) \neq \text{nil}$  then
                 $\text{val}(\text{next}(i)) := \text{val}(i) \circ \text{val}(\text{next}(i))$ 
                 $\text{next}(i) := \text{next}(\text{next}(i))$ 
            end if
            if  $\text{next}(i) \neq \text{nil}$  then
                 $\text{finished} := \text{false}$ 
            end if
        end for
    end while

```

Analysis

Steps 1 and 2 and each iteration of Step 3 take constant time. Since the number of partial sums doubles after each iteration of Step 3, the number of iterations is $\theta(\log |L|)$. Assuming $|L| = n =$ number of processors, the algorithm runs in $O(\log n)$ time.

7.4 SORTING

The problem of sorting is important as sorting data is at the heart of many computations. In this section, we describe a variety of sorting algorithms for the various models of parallel computers discussed earlier. The problem of sorting is defined as follows: Given a sequence of n numbers $S = s_1, s_2, \dots, s_n$, where the elements of S are initially in arbitrary order, rearrange the numbers so that the resulting sequence is in non-decreasing (sorted) order.

7.4.1 Combinational Circuits for Sorting

Here we give two different combinational circuits for sorting. The input to the combinational circuits is n unsorted numbers and the output is the sorted permutation of the n numbers.

The basic processing unit used in the combinational circuits for sorting is the **comparator**. A comparator has two inputs x and y , and has two outputs x' and y' . There are two kinds of comparators. In an *increasing comparator*, $x' = \min(x, y)$ and $y' = \max(x, y)$. In a decreasing comparator, $x' = \max(x, y)$ and $y' = \min(x, y)$. Figure 7.9 shows the schematic representation of the two comparators.

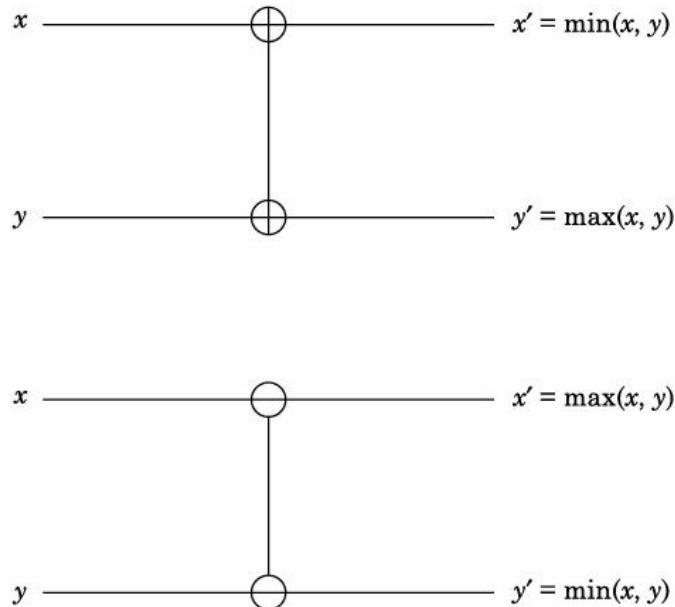


Figure 7.9 Increasing and decreasing comparator.

Bitonic Sorting Network

Bitonic sorting is based on an interesting property of the **bitonic sequence**. A bitonic sequence is a sequence of numbers $\langle a_0, a_1, \dots, a_{n-1} \rangle$ such that either (a) $\langle a_0, \dots, a_i \rangle$ is a monotonically increasing sequence and $\langle a_{i+1}, \dots, a_{n-1} \rangle$ is a monotonically decreasing sequence for some $0 \leq i \leq n-1$ or (b) there exists a cyclic shift of the sequence $\langle a_0, \dots, a_{n-1} \rangle$ such that the resulting sequence satisfies condition (a). For example both the sequences $\langle 5, 7, 8, 9, 6, 4, 2, 1 \rangle$ and $\langle 5, 7, 9, 6, 4, 2, 1, 3 \rangle$ are bitonic sequences while the sequence $\langle 5, 7, 9, 6, 4, 2, 3, 6 \rangle$ is not a bitonic sequence.

Let $S = \langle a_0, a_1, \dots, a_{n-1} \rangle$ be a bitonic sequence such that $a_0 \leq a_1 \leq \dots \leq a_{n/2-1}$ and $a_{n/2} \geq a_{n/2+1} \geq \dots \geq a_{n-1}$. Consider the two subsequences:

$$S_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \dots, \min(a_{n/2-1}, a_{n-1}) \rangle$$

$$S_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \dots, \max(a_{n/2-1}, a_{n-1}) \rangle$$

The following can be easily verified.

- All elements of the sequence S_1 have a value lesser than all elements of the sequence S_2 .
- Each sequence, S_1 and S_2 , is a bitonic sequence.

This way of splitting a bitonic sequence into two bitonic sequences is known as **bitonic split**. The idea of bitonic split can be used to sort a given bitonic sequence. A sequential recursive algorithm for sorting a bitonic sequence $S = \langle a_0, a_1, \dots, a_{n-1} \rangle$ is given below:

1. Split the sequence S into two sequences S_1 and S_2 using the bitonic split.
2. Recursively sort the two bitonic sequences S_1 and S_2 .
3. The sorted sequence of S is the sorted sequence of S_1 followed by the sorted sequence of S_2 .

This process of sorting a bitonic sequence is known as *bitonic merging*. This bitonic merging can easily be implemented as a combinational circuit. The bitonic merging circuit (network) for a bitonic sequence of length $n = 16$ is shown in Fig. 7.10. The bitonic merging network for a sequence of length n consists of $\log n$ layers each containing $n/2$ comparators. If all the comparators are increasing comparators, we get an ascending sequence at the output. Such a bitonic merging circuit is represented by $(+)BM(n)$. If all the comparators are decreasing comparators, we get a decreasing sequence at the output. Such a bitonic merging circuit is represented by $(-)BM(n)$. Note that the depth of both $(+)BM(n)$ and $(-)BM(n)$ is $\log n$.

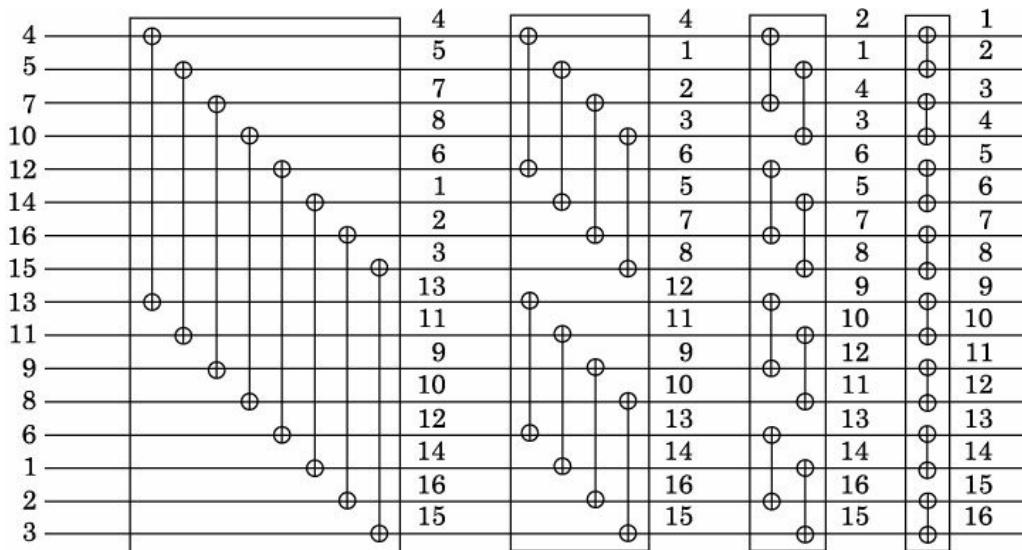


Figure 7.10 Bitonic merging network.

The bitonic merging networks $(+)BM(n)$ and $(-)BM(n)$ are used as the basic building blocks for the combinational circuit for sorting. The sorting network is based on the following observations.

- Any sequence of only two numbers forms a bitonic sequence, trivially.

- A sequence consisting of a monotonically increasing sequence and a monotonically decreasing sequence forms a bitonic sequence.

The basic idea of the bitonic sorting algorithm is to incrementally construct larger and larger bitonic sequences starting from bitonic sequences of length 2. Given two bitonic sequences of length $m/2$ each, we can obtain one ascending order sorted sequence using $(+BM(m/2)$ and one descending order sorted sequence using $(-BM(m/2)$. These two can be concatenated to get a single bitonic sequence of length m . Once a bitonic sequence of length n is formed, this is sorted using $(+BM(n)$ to get a sorted output sequence. The bitonic sorting network for length $n = 16$ is shown in Fig. 7.11.

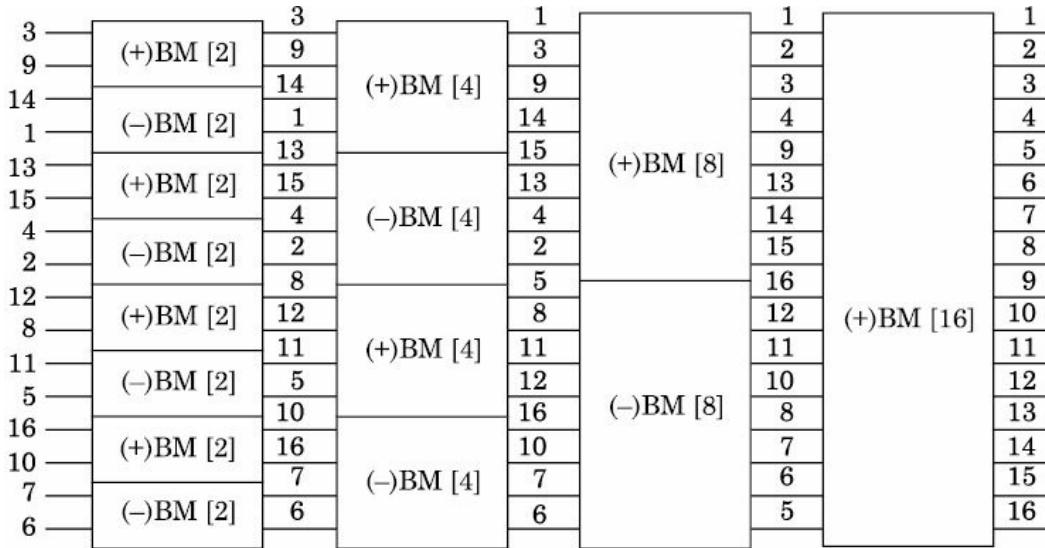


Figure 7.11 Bitonic sorting network.

Analysis

The bitonic sorting network for sorting a sequence of n numbers consists of $\log n$ stages. The last stage of the network uses a $(+BM(n)$ which has a depth of $\log n$. The remaining stages perform a complete sort of $n/2$ elements. Thus the depth of the sorting network (i.e., the running time of bitonic sorting) is given by the recurrence relation

$$d(n) = d(n/2) + \log n$$

Solving this we get $d(n) = ((\log^2 n) + \log n)/2 = O(\log^2 n)$. This idea of bitonic sort is used for sorting in other models of parallel computer as well. In a later section we see how bitonic sorting network can be used for sorting on a hypercube interconnection network.

Combinational Circuit for Sorting by Merging

The second combinational circuit for sorting is based on the merge sort algorithm for a sequential computer. The merge sort algorithm involves splitting a sequence into two halves, sorting each half recursively and merging the two sorted half sequences. To realize a combinational circuit for sorting using merge sort, we first develop a combinational circuit for merging, and then use this as a building block for the sorting network.

Odd-Even Merging Circuit

The merging problem is defined as follows. Given two sorted sequences $s_1 = \langle x_1, x_2, \dots, x_m \rangle$ and $s_2 = \langle y_1, y_2, \dots, y_m \rangle$ form a single sorted sequence $s = \langle z_1, z_2, \dots, z_{2m} \rangle$, where m is a power of 2, such that every element of s_1 and s_2 is represented exactly once in s . If $m = 1$,

then we can use one comparator to produce x_1 and y_1 in sorted order. If $m = 4$, it is easy to verify that the sorted sequences $\langle x_1, x_2, x_3, x_4 \rangle$ and $\langle y_1, y_2, y_3, y_4 \rangle$ are merged correctly by the circuit of Fig. 7.12. (From this figure, it is also easy to verify the correctness of the merging circuit when $m = 2$.)

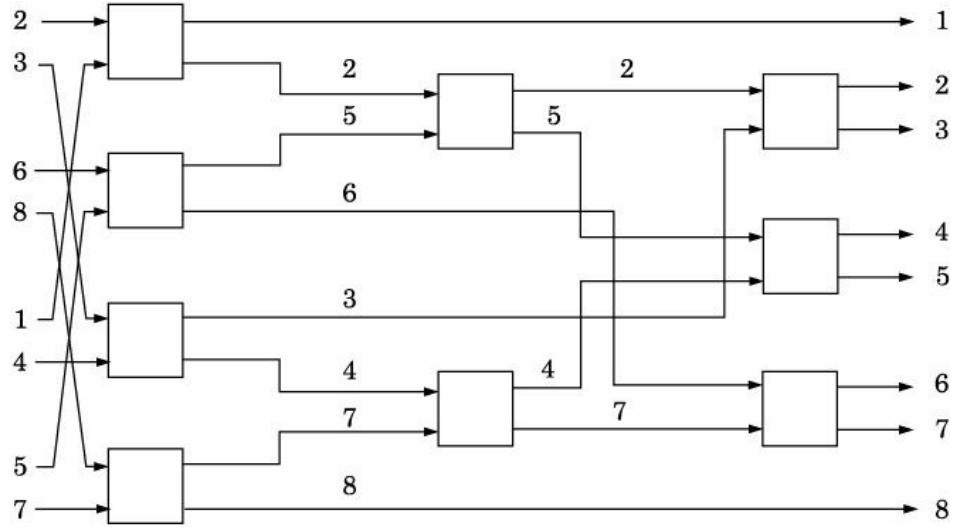


Figure 7.12 Merging circuit.

In general, a circuit for merging two sequences is obtained (recursively) as follows. Let $s_1 = \langle x_1, x_2, \dots, x_m \rangle$ and $s_2 = \langle y_1, y_2, \dots, y_m \rangle$ be the input sequences that need to be merged. (We refer to a circuit for merging two sorted sequences of length m each as an (m, m) odd-even merging circuit.)

1. Using an $(m/2, m/2)$ merging circuit, merge the odd-indexed elements of the two sequences: $\langle x_1, x_3, \dots, x_{m-1} \rangle$ and $\langle y_1, y_3, \dots, y_{m-1} \rangle$ to produce a sorted sequence $\langle u_1, u_2, \dots, u_m \rangle$.
2. Using an $(m/2, m/2)$ merging circuit, merge the even-indexed elements of the two sequences: $\langle x_2, x_4, \dots, x_m \rangle$ and $\langle y_2, y_4, \dots, y_m \rangle$ to produce a sorted sequence $\langle v_1, v_2, \dots, v_m \rangle$.
3. The output sequence $\langle z_1, z_2, \dots, z_{2m} \rangle$ is obtained as follows: $z_1 = u_1$, $z_{2m} = v_m$, $z_{2i} = \min(u_{i+1}, v_i)$, and $z_{2i+1} = \max(u_{i+1}, v_i)$, for $i = 1, 2, \dots, m - 1$.

The schematic diagram of an (m, m) odd-even merging circuit, which uses two $(m/2, m/2)$ odd-even merging circuits, is shown in Fig. 7.13.

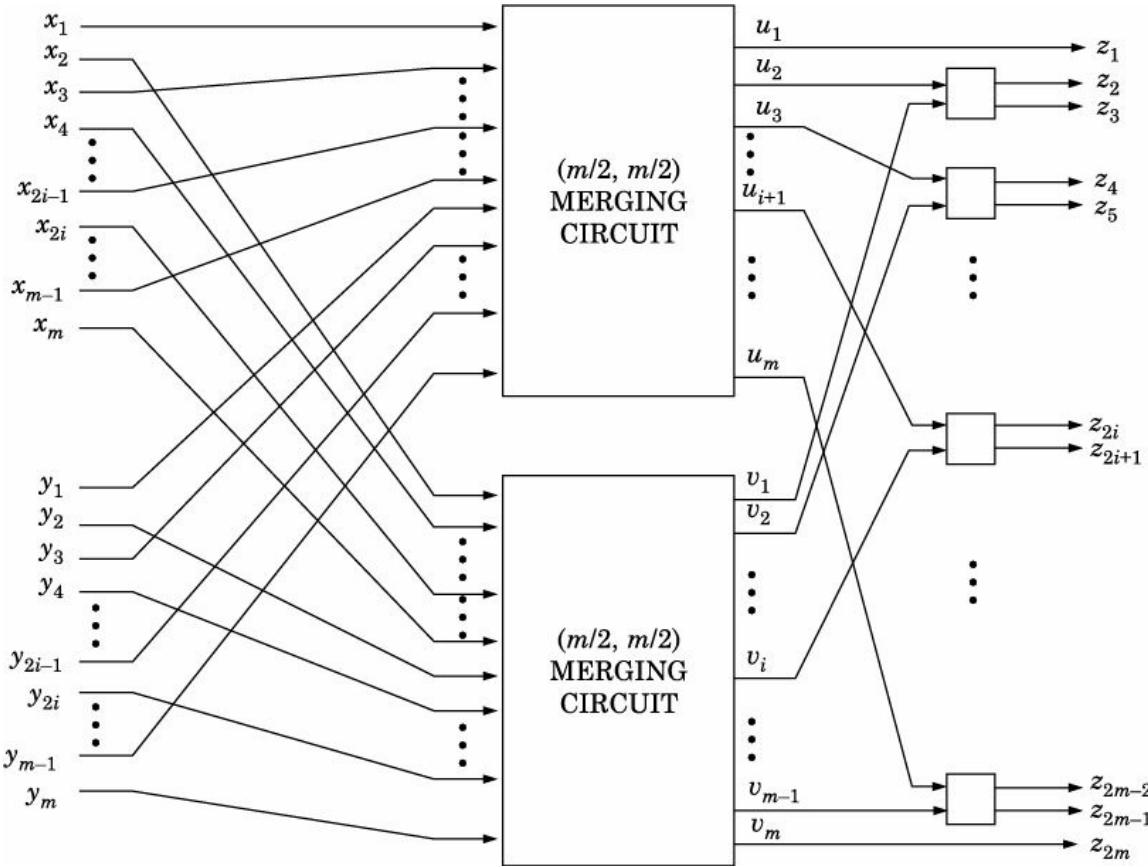


Figure 7.13 Odd-even merging circuit.

Analysis

1. *Width:* Each comparator has exactly two inputs and two outputs. The circuit takes $2m$ inputs and produces $2m$ outputs. Thus it has a width of m .
2. *Depth:* The merging circuit consists of two smaller circuits for merging sequences of length $m/2$, followed by one stage of comparators. Thus the depth $d(m)$ of the circuit (i.e., the running time) is given by the following recurrence relations.

$$d(m) = 1 \quad (m = 1)$$

$$d(m) = d(m/2) + 1 \quad (m > 1)$$

Solving this we get $d(m) = 1 + \log m$. Thus the time taken to merge two sequences is quite fast compared to the obvious lower bound of $\Omega(m)$ on the time required by the RAM to merge two sorted sequences of length m .

3. *Size:* The size $p(m)$ of the circuit (i.e., the number of comparators used) can be obtained by solving the following recurrence relations.

$$p(1) = 1 \quad (m = 1)$$

$$p(m) = 2p(m/2) + m - 1 \quad (m > 1)$$

Solving this we get $p(m) = 1 + m \log m$.

Odd-Even Merge Sorting Circuit

The sorting circuit can now be derived using the merging circuit, developed above, as the basic building block. The circuit for sorting an input sequence of length n is constructed as follows:

1. Split the input unsorted sequence $\langle a_1, a_2, \dots, a_n \rangle$ into two unsorted sequences $\langle a_1, a_2, \dots, a_{n/2} \rangle$ and $\langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$. Recursively sort these two sequences.
2. Merge the two sorted sequences using an $(n/2, n/2)$ merging circuit to get the sorted sequence.

Figure 7.14 shows an odd-even merge sorting circuit for $n = 8$.

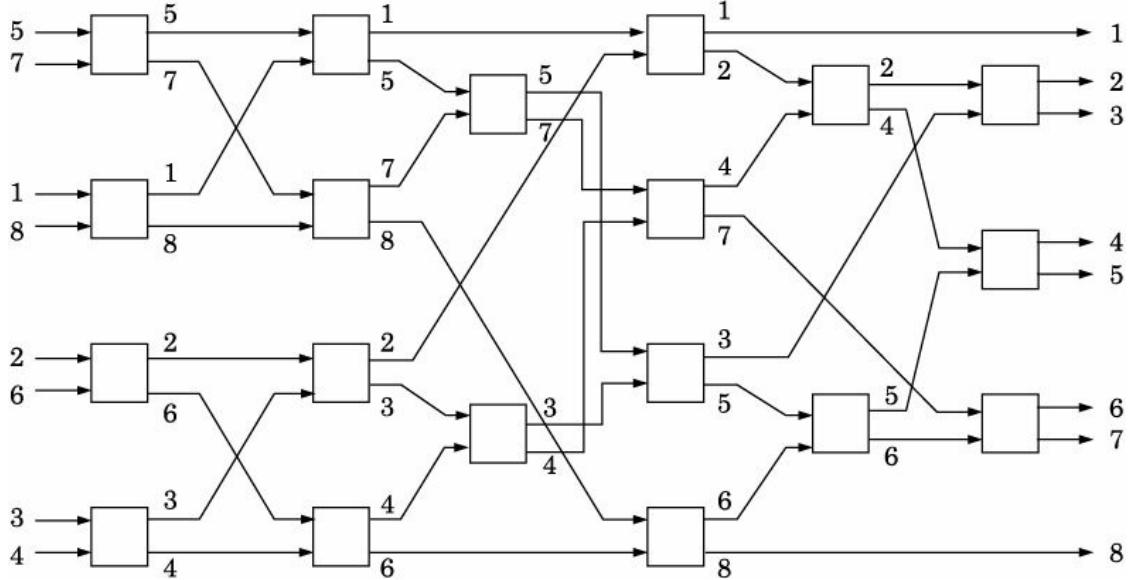


Figure 7.14 Sorting circuit.

Analysis

1. *Width:* The width of the circuit is easily seen to be $O(n/2)$.
2. *Depth:* The circuit consists of two sorting circuits for sorting sequences of length $n/2$ followed by one merging circuit to merge the two sorted sequences. Let $d_s()$ and $d_m()$ denote the depth of sorting circuit and merging circuit, respectively. Then, $d_s(n)$ is given by the following recurrence relation:

$$d_s(n) = d_s(n/2) + d_m(n/2) = d_s(n/2) + \log(n/2)$$

Solving this we get $d_s(n) = O(\log^2 n)$. This means that the time required for sorting is $O(\log^2 n)$.

3. *Size:* Let $P_s()$ and $P_m()$ denote the size of sorting network and merging circuit, respectively. Then, $p_s(n)$ is given by the recurrence relation:

$$p_s(n) = 2p_s(n/2) + p_m(n/2)$$

Solving this we get $p_s(n) = O(n \log^2 n)$. In other words, the circuit performs $O(n \log^2 n)$ comparisons.

7.4.2 Sorting on PRAM Models

We now present some sorting algorithms for the PRAM models of a parallel computer. All these algorithms are based on the idea of *sorting by enumeration*. Here each element finds its own position in the sorted sequence by comparing its value with that of all the other remaining elements.

Let $S = \langle s_1, s_2, \dots, s_n \rangle$ denote the unsorted input sequence. The position of each element s_i

of S in the sorted sequence, known as *rank* of s_i , is determined by computing r_i , the number of elements smaller than it. All the algorithms discussed here use an array $R = \langle r_1, r_2, \dots, r_n \rangle$ to keep track of the rank of the elements in the input sequence. The array R is initialized to 0 in all the algorithms. When the algorithms terminate, r_i will contain the number of elements in the input sequence that are smaller than s_i .

CRCW Sorting

We describe a sorting algorithm for the SUM CRCW model of a parallel computer which has n^2 processors. In this model, when more than one processor attempts to write to a common memory location, the sum of all the individual values is written onto the memory location. To help visualize the algorithm, the n^2 processors are logically organized in a two-dimensional $n \times n$ array. $P_{i,j}$ denotes the processor in the i^{th} row and j^{th} column. The processor $P_{i,j}$ tries to write the value 1 in r_i if $s_i > s_j$ or ($s_i = s_j$ and $i > j$). Thus, the actual value written onto the location r_i will be the number of elements that are smaller than s_i in the input sequence. Therefore, by placing the element s_i in the position $r_i + 1$ ($1 \leq i \leq n$) we get a completely sorted sequence.

Procedure CRCW Sorting

```

for  $i := 1$  to  $n$  do in parallel
    for  $j := 1$  to  $n$  do in parallel
        if  $s_i > s_j$  or ( $s_i = s_j$  and  $i > j$ ) then
             $P_{i,j}$  writes 1 to  $r_i$ 
        end if
    end for
end for
for  $i := 1$  to  $n$  do in parallel
     $P_{i,1}$  puts  $s_i$  in  $(r_i + 1)$  position of  $S$ 
end for

```

The above algorithm takes $O(1)$ time. And it uses $O(n^2)$ processors which is very large. Further the assumption made on the write conflict resolution process is quite unrealistic.

CREW Sorting

The sorting algorithm for the CREW model is very similar to the one described for the CRCW model. The algorithm uses n processors and has a time complexity of $O(n)$. Let the processors be denoted by P_1, P_2, \dots, P_n . The processor P_i computes the value r_i after n iterations. In the j^{th} iteration of the algorithm, all the processors read s_j ; the processor P_i increments r_i if $s_i > s_j$ or ($s_i = s_j$ and $i > j$). Therefore, after n iterations, r_i contains the number of elements smaller than s_i in the input sequence. As in the previous case, by writing the value s_i in the location $r_i + 1$ we get a completely sorted sequence.

Procedure CREW Sorting

```

for  $i := 1$  to  $n$  do in parallel
  for  $j := 1$  to  $n$  do
    if ( $s_i > s_j$ ) or ( $s_i = s_j$  and  $i > j$ ) then
       $P_i$  adds 1 to  $r_i$ 
    end if
  end for
   $P_{i,1}$  puts  $s_i$  in  $(r_i + 1)$  position of  $S$ 
end for

```

EREW Sorting

The above algorithm uses concurrent read operations at every iteration. Concurrent read conflicts can be avoided if we ensure that each processor reads a unique element in each iteration. This can be achieved by allowing the processors to read elements in a cyclic manner. In the first iteration the processors P_1, P_2, \dots, P_n read s_1, s_2, \dots, s_n respectively and in the second iteration the processors read $s_2, s_3, \dots, s_n, s_1$ respectively and so on. Thus in the j^{th} iteration the processors P_1, P_2, \dots, P_n read $s_j, s_{j+1}, \dots, s_{j-1}$ respectively and update the corresponding r values. This algorithm has a running time complexity of $O(n)$. Therefore we get an $O(n)$ running time algorithm for the EREW model using $O(n)$ processors.

Procedure EREW Sorting

```

for  $i := 1$  to  $n$  do in parallel
  for  $j := 0$  to  $n-1$  do
     $k := (i+j) \bmod n$ 
    if ( $s_i > s_k$ ) or ( $s_i = s_k$  and  $i > k$ ) then
       $P_i$  adds 1 to  $r_i$ 
    end if
  end for
   $P_{i,1}$  puts  $s_i$  in  $(r_i + 1)$  position of  $S$ 
end for

```

7.4.3 Sorting on Interconnection Networks

We now consider sorting on interconnection networks. Before going into the details of the parallel sorting algorithms, we first describe how comparisons are made in an interconnection network parallel computer. Consider two adjacent processors P_i and P_j in an interconnection network. The processor P_i has the element a_i and the processor P_j has the element a_j . The processor P_i sends the element a_i to the processor P_j , and the processor P_j sends the element a_j to the processor P_i . The processor P_i keeps the element $\min(a_i, a_j)$ and the processor P_j keeps the element $\max(a_i, a_j)$. This operation is referred to as **compare-exchange** (P_i, P_j) in the rest of this subsection (see Fig. 7.15).

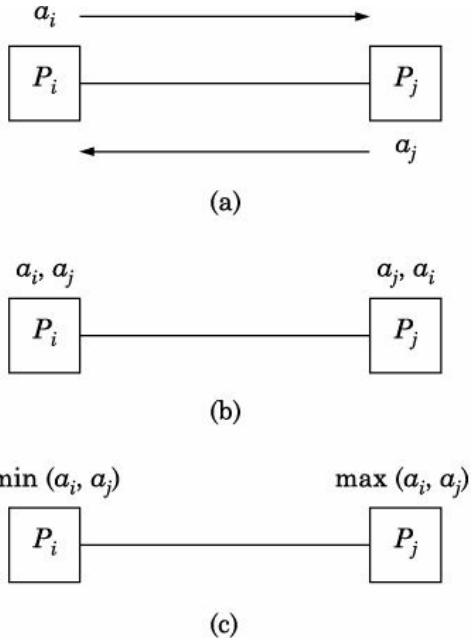


Figure 7.15 A parallel compare-exchange operation.

Sorting on a Linear Array

We now present a sorting algorithm called **odd-even transposition** sort for a parallel computer where the processors are connected to form a linear array. Odd-even transposition sort is based on the idea of bubble sort and lends itself to a neat implementation on the linear array.

Odd-Even Transposition Sort

Let $\langle a_1, a_2, \dots, a_n \rangle$ be the unsorted input sequence. The odd-even transposition sort alternates between two phases namely, *odd phase* and *even phase*. In the odd phase, the elements $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$, are compared and swapped if they are out of order. In the even phase, the elements $(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1})$ are compared and swapped if they are out of order. Note that the elements a_1 and a_n are not compared in the even phase. After n such phases ($n/2$ odd and $n/2$ even), the entire sequence will be correctly sorted. The proof of correctness of this algorithm is left as an exercise. The running time of this algorithm is obviously seen to be $O(n^2)$.

Implementing Odd-Even Transposition Sort on a Linear Array

It is straightforward to implement the odd-even transposition sort algorithm on a linear array interconnection network. In order to sort an input sequence $\langle a_1, a_2, \dots, a_n \rangle$ of size n , a linear array containing n processors P_1, P_2, \dots, P_n is used. Initially, the processor P_i holds the element a_i . In the odd phase, each processor with an odd index does a compare-exchange operation with its right neighbour. In the even phase, each processor with an even index does a compare-exchange operation with its right neighbour. All the compare-exchange operations of a particular phase are done in parallel. The parallel algorithm is given below.

Procedure Odd-Even Transposition Sort

```

for  $i := 1$  to  $n$  do
    if  $i$  is odd then
        for  $k := 1, 3, \dots, 2 \lfloor n/2 \rfloor - 1$  do in parallel
            compare-exchange ( $P_k, P_{k+1}$ )
        end for
    else
        for  $k := 2, 4, \dots, 2 \lfloor (n-1)/2 \rfloor$  do in parallel
            compare-exchange ( $P_k, P_{k+1}$ )
        end for
    end if
end for

```

The execution of this algorithm on a linear array of size $n = 8$ is shown in Fig. 7.16.

Analysis

There are n (parallel) phases in the algorithm and during each phase the odd-indexed or even-indexed processors perform a compare-exchange operation which takes $O(1)$ time. Thus the time complexity of the algorithm is $O(n)$. The number of processors used is n . Hence the cost of the algorithm is $O(n^2)$ which is not optimal.

Sorting on a Hypercube

There are several sorting algorithms for the hypercube network. Here we consider one simple algorithm—bitonic sort which has already been discussed in the context of combinational circuits for sorting.

Bitonic sort algorithm can be efficiently mapped onto the hypercube network without much communication overhead. The basic idea is to “simulate” the functioning of the combinational circuit for bitonic sorting using a hypercube network. Let the input wires (lines) in the combinational circuit shown in Fig. 7.11 be numbered (in binary) 0000, 0001, ..., 1110, 1111 (sequentially from top to bottom). Then observe that a comparison operation is performed between two wires whose indices differ in exactly one bit. In a hypercube, processors whose indices (labels) differ in only one bit are neighbours. Thus, an optimal mapping of input wires to hypercube processors is the one that maps an input wire with index l to a processor with index l where $l = 1, 2, \dots, n - 1$ and n is the number of processors. So whenever a comparator performs a comparison operation on two wires, a compare-exchange operation is performed between the corresponding processors in the hypercube. The algorithm, which captures this idea, is given below. Figure 7.17 illustrates the communication during the last stage of the bitonic sort algorithm. Note that the time complexity of the algorithm, $O(\log^2 n)$, remains unchanged.

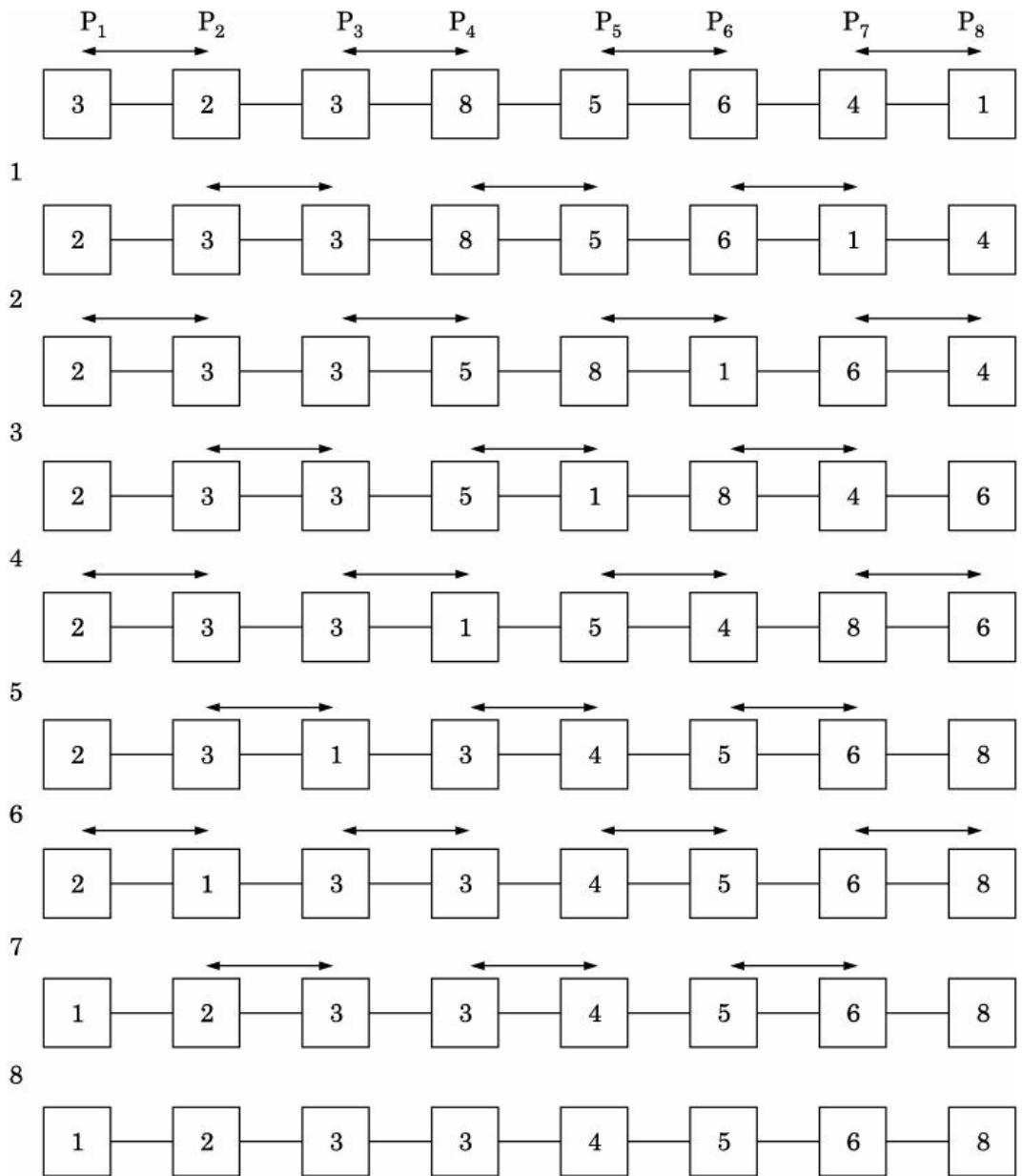


Figure 7.16 Odd-even transposition sort.

Procedure Bitonic Sort

```

for i := 0 to d-1 do {n = 2d; d is the dimension of hypercube}
  for j := i downto 0 do
    for each Pk such that (i + 1)st bit of k ≠ jth bit of k do in parallel
      compare-exchange (Pk, pk(j))
      {For any index p, p(l) denotes the index obtained by flipping
       the lth bit in the binary representation of p}
    end for
  end for
end for

```

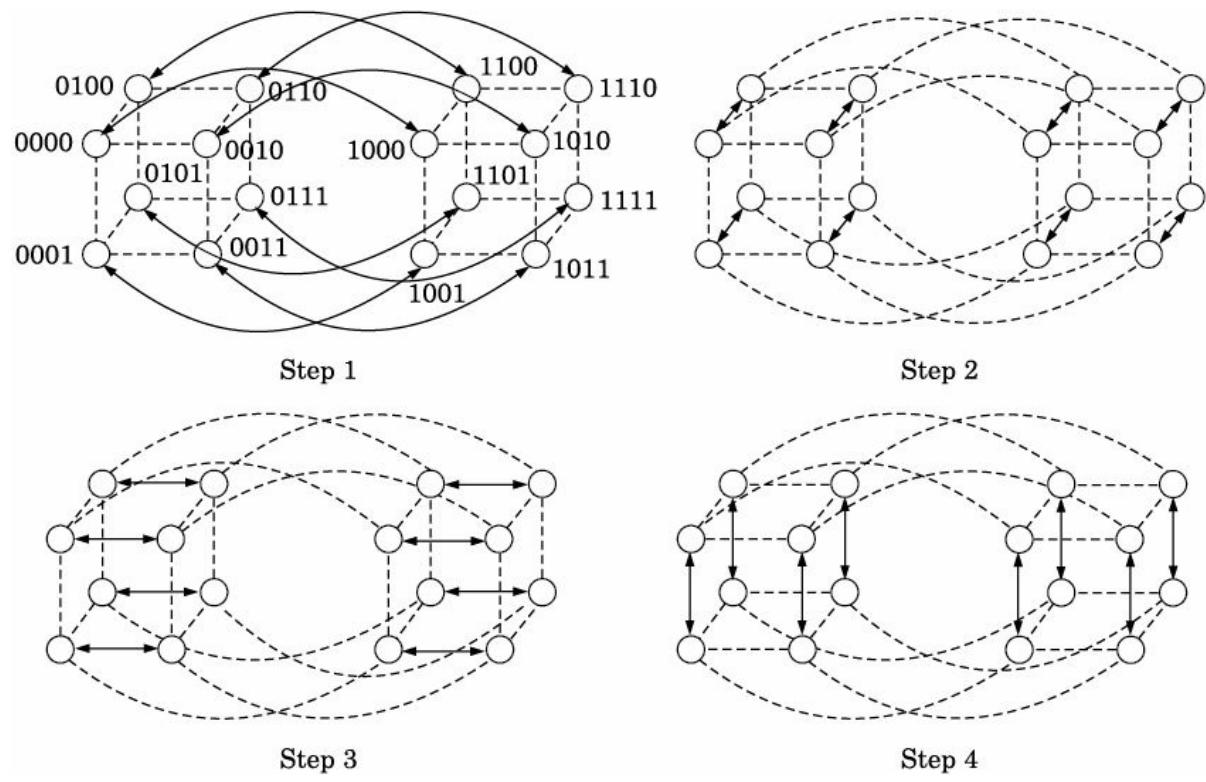


Figure 7.17 Communication during the last stage of bitonic sort.

7.5 SEARCHING

Searching is one of the most fundamental operations encountered in computing. It is used in applications where we need to know whether an element belongs to a list or, more generally, retrieve from a file information associated with that element. The problem of searching, in its most basic form, is stated as follows: Given a sequence $S = \{s_1, s_2, \dots, s_n\}$ of integers and an integer x , it is required to find out whether $x = s_k$ for some s_k in S .

In sequential computing, the problem is solved by scanning the sequence S and comparing x with every element in S , until either an integer equal to x is found or the sequence is exhausted without success. This (solution) algorithm takes $O(n)$ time which is optimal (when S is not ordered) since every element of S must be examined (when x is not in S) before declaring failure. On the other hand, if S is sorted in nondecreasing order, then there exists the *binary search* algorithm which takes $O(\log n)$ time. Binary search is based on the divide-and-conquer principle. During each iteration, the middle element, s_m , of the sorted sequence is probed and tested for equality with the input x . If $s_m > x$, then all the elements larger than s_m are discarded; otherwise all the elements smaller than s_m are discarded. Thus, the next iteration is applied to a sequence half as long as the previous one. The algorithm terminates when the probed element equals x or when all elements have been discarded. Since the number of elements under consideration is reduced by one-half at each iteration, the algorithm requires $O(\log n)$ time in the worst case. Again, this is optimal since this many bits are needed to distinguish among the n elements of S .

In this section we present parallel search algorithms. We first consider the case where S is sorted and then the more general case where the elements of S are in random order.

7.5.1 Searching on PRAM Models

Searching a Sorted Sequence

Here we assume that the sequence $S = \{s_1, s_2, \dots, s_n\}$ is sorted in nondecreasing order. Let x denote the element to be searched in S .

EREW Searching

We assume that we have an N -processor EREW parallel computer. The processors are denoted by P_1, P_2, \dots, P_N . In the first step of the algorithm, the value of the element x to be searched is *broadcast* (i.e., is made known) to all the processors. This can be done in $O(\log N)$ time (see Exercise 7.4). The sequence S is subdivided into N subsequences of length n/N each, and each processor is assigned one of the subsequences. The processor P_i is assigned the subsequence $\{S_{(i-1)(n/N)+1}, S_{(i-1)(n/N)+2}, \dots, S_{i(n/N)}\}$. Every processor now searches for the element x using the sequential binary search algorithm on the subsequence assigned to it. Therefore, this step takes $O(\log (n/N))$. The overall time complexity of the algorithm is therefore, $O(\log N) + O(\log (n/N))$, which is $O(\log n)$. Since this is precisely the time required by the binary search algorithm (running on a single processor!), no speedup is obtained using this EREW algorithm.

CREW Searching

In the case of the CREW parallel computer, we can use the same algorithm, described above, for the EREW model. The broadcast step, in this case, takes only $O(1)$ time as compared to the $O(\log N)$ time in the EREW model, since all processors can read x simultaneously in

constant time. Therefore the CREW algorithm in the worst case requires $O(\log(n/N))$ time thereby running faster than the EREW algorithm.

In fact, it is possible to obtain a better algorithm for the CREW computer than the naive one described above. The basic idea is to use a parallel version of the binary search method. In this version, the binary search is modified into an $(N + 1)$ -ary search. At each stage of the algorithm, the sequence under consideration is split into $N + 1$ subsequences of equal length. The N processors simultaneously probe the elements at the boundary between successive subsequences. Let Y_0, Y_1, \dots, Y_N denote the $N + 1$ subsequences. And let s_i denote the element that occurs at the boundary of Y_{i-1} and Y_i . Then the processor P_i compares s_i with x .

1. If $s_i > x$, then if an element equal to x is in the sequence at all, it must precede s_i . Consequently, s_i and all the elements that follow it (i.e., to its right) are removed from consideration.
2. If $s_i < x$, then all elements to the left of s_i in the sequence are removed from consideration.

Thus each processor splits the sequence into two parts: those elements to be discarded as they definitely do not contain an element equal to x and those that might and are hence retained. This narrows down the search to the intersection of all the parts to be retained. This process of splitting and probing continues until either an element equal to x is found or all the elements of S are discarded. Since at every stage the size of the subsequence that needs to be searched drops by a factor $N + 1$, $O(\log_{N+1}(n+1))$ stages are needed. The algorithm is given below. The sequence $S = \{s_1, s_2, \dots, s_n\}$ and the element x are given as input to the algorithm. The algorithm returns k if $x = s_k$ for some k , else it returns 0. Each processor P_i uses a variable c_i that takes the value *left* or *right* according to whether the part of the sequence P_i decides to retain is to the left or right of the element it compared with x during this stage. The variables q and r keep track of the boundary of the sequence that is under consideration for searching. Precisely one processor updates q and r in the shared memory, and all remaining processors simultaneously read the updated values in constant time.

Procedure CREW Search

```

Step 1: {Initialize indices of sequence to be searched}
    q := 1
    r := n
Step 2: {Initialize results and maximum number of stages}
    k := 0
    g :=  $\lceil \log_{N+1} (n + 1) \rceil$ 
Step 3: while ( $q \leq r$  and  $k = 0$ ) do
     $j_0 := q - 1$ 
    for  $i := 1$  to  $N$  do in parallel
         $j_i := (q - 1) + i(N + 1)^{g-1}$ 
        { $P_i$  compares  $x$  to  $s_j$  and determines the part of the sequence to
         be retained}
        if  $j_i \leq r$ 
        then if  $s_{j_i} = x$  {Element found}
            then  $k := j_i$ 
            else if  $s_{j_i} > x$ 
                then  $c_i := \text{left}$ 
                else  $c_i := \text{right}$ 
                end if
            end if
        else (a)  $j_i := r+1$ 
                (b)  $c_i := \text{left}$ 
        end if
        {The indices of the subsequence to be searched in the next iteration
         are computed}
        if  $c_i \neq c_{i-1}$ 
        then (a)  $q := j_{i-1} + 1$ 
                (b)  $r := j_i - 1$ 
        end if
        if ( $i = N$  and  $c_i \neq c_{i+1}$ ) then  $q := j_i + 1$ 
        end if
    end for
     $g := g-1$ 
end while

```

Analysis

If the element x is found to occur in the input sequence, the value of k is set accordingly and consequently the while loop terminates. After each iteration of the while loop, the size of the sequence under consideration shrinks by a factor $N + 1$. Thus after $\lceil \log_{N+1} (n + 1) \rceil$ steps (iterations), in the worst case, q becomes greater than r and the while loop terminates. Hence the worst case time complexity of the algorithm is $O(\log_{N+1} (n + 1))$. When $N = n$, the algorithm runs in constant time.

Note that the above algorithm assumes that all the elements in the input sequence S are sorted and distinct. If each s_i is not unique, then possibly more than one processor will succeed in finding an element of S equal to x . Consequently, possibly several processors will attempt to return a value in the variable k , thus causing a write conflict which is not allowed in both the EREW and CREW models. However, the problem of *uniqueness* can be solved with an additional overhead of $O(\log N)$ (The reader is urged to verify this). In such a case, when $N = n$ the running time of the algorithm is $O(\log_{N+1} (n + 1)) + O(\log N) = O(\log n)$ which is precisely the time required by the sequential binary search algorithm!

CRCW Searching

The searching algorithm discussed for the CREW model is applicable even to the CRCW

model. The uniqueness criterion which posed a problem for the CREW model is resolved in this case by choosing an appropriate conflict resolution policy. The algorithm, therefore, does not incur the $O(\log N)$ extra cost. Thus the algorithm runs in $O(\log_{N+1} (n + 1))$ time.

Searching a Random Sequence

We now consider the more general case of the search problem. Here the elements of the sequence $S = \{s_1, s_2, \dots, s_n\}$ are not assumed to be in any particular order and are not necessarily distinct. As we will see now, the general algorithm for searching a sequence in random order is straightforward. We use an N -processor computer to search for an element x in S , where $1 < N \leq n$. The sequence S is divided into N subsequences of equal size and each processor is assigned one of the subsequences. The value x is then broadcast to all the N processors. Each processor now searches for x in the subsequence assigned to it. The algorithm is given below.

Procedure Search

```

Step 1: Broadcast x to all the N processors.
Step 2: for  $i := 1$  to  $N$  do in parallel
     $S_i := \{s_{(i-1)(n/N) + 1}, s_{(i-1)(n/N) + 2}, \dots, s_{i(n/N)}\}$ 
    Search for x sequentially in  $S_i$  and store the result in  $k_i$ 
    end for
Step 3: for  $i := 1$  to  $N$  do in parallel
    if  $k_i > 0$  then  $k := k_i$  end if
    end for
```

Analysis

- 1. EREW:** In this model, the broadcast step takes $O(\log N)$ time. The sequential search takes $O(n/N)$ time in the worst case. The final store operation takes $O(\log N)$ time since it involves a concurrent write to a common memory location. Therefore the overall time complexity of the algorithm, $t(n)$, is

$$t(n) = O(\log N) + O(n/N)$$

and its cost, $c(n)$, is

$$c(n) = O(N \log N) + O(n)$$

which is not optimal.

- 2. CREW:** In this model, the broadcast operation can be accomplished in $O(1)$ time since concurrent read is allowed. Steps 2 and 3 take the same time as in the case of EREW. The time complexity of the algorithm remains unchanged.
- 3. CRCW:** In this model, both the broadcast operation and the final store operation take constant time. Therefore the algorithm has a time complexity of $O(n/N)$. The cost of the algorithm is $O(n)$ which is optimal.

7.5.2 Searching on Interconnection Networks

We now present parallel search algorithms for tree and mesh interconnection networks.

Searching on a Tree

We use a complete binary tree network with n leaf nodes (processors). The total number of nodes (and hence the number of processors) is thus $2n - 1$. Let L_1, L_2, \dots, L_n denote the

processors at the leaf nodes. The sequence $S = \{s_1, s_2, \dots, s_n\}$ which has to be searched is stored in the leaf nodes. The processor L_i stores the element s_i . The algorithm has three stages:

1. The root node reads x and passes it to its two child nodes. In turn, these send x to their child nodes. The process continues until a copy of x reaches each leaf node.
2. Simultaneously, every leaf node L_i compares the element x with s_i . If they are equal, the node L_i sends the value 1 to its parent, else it sends a 0.
3. The results (1 or 0) of the search operation at the leaf nodes are combined by going upward in the tree. Each intermediate node performs a logical **OR** of its two inputs and passes the result to its parent. This process continues until the root node receives its two inputs, performs their logical **OR** and produces either a 1 (for yes) or a 0 (for no).

The working of the algorithm is illustrated in Fig. 7.18 where $S = \{26, 13, 36, 28, 15, 17, 29, 17\}$ and $x = 17$. It takes $O(\log n)$ time to go down the tree, constant time to perform the comparison, and again $O(\log n)$ time to go back up the tree. Thus the time taken by the algorithm to search for an element is $O(\log n)$. Surprisingly, its performance is superior to that of the EREW PRAM algorithm.

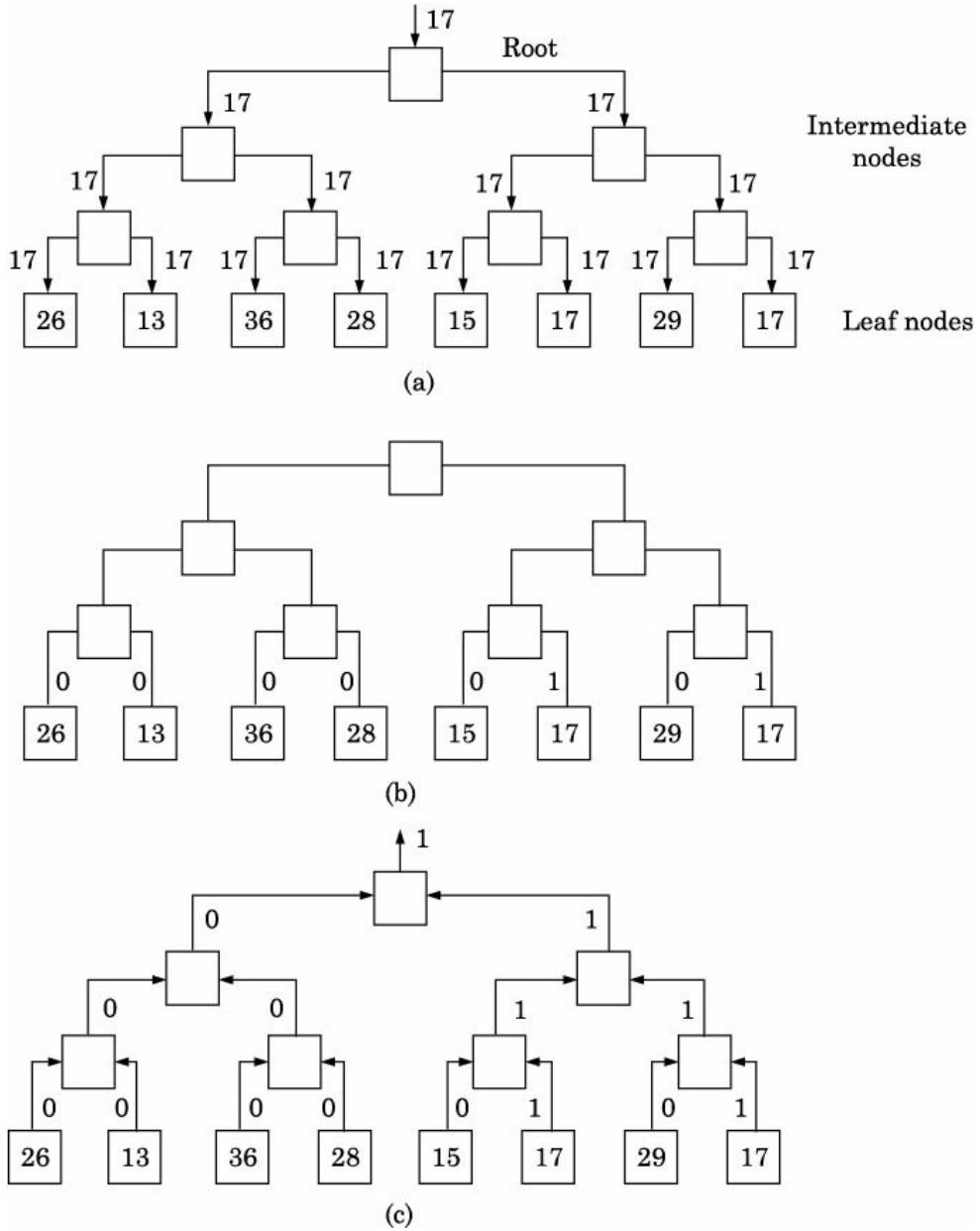


Figure 7.18 Searching on a tree.

Suppose, instead of one value of x (or *query*, which only requires a yes or no answer), we have q values of x , i.e., q queries to be answered. These q queries can be pipelined down the tree since the root node and intermediate nodes are free to handle the next query as soon as they have passed the current one along their child nodes. The same remark applies to the leaf nodes—as soon as the result of one comparison has been performed, each leaf node is ready to receive a new value of x . Thus it takes $O(\log n) + O(q)$ time for the root node to process all the q queries. Note that for large values of q , this behaviour is equivalent to that of the CRCW PRAM algorithm.

Searching on a Mesh

We now present a searching algorithm for the mesh network. This algorithm runs in $O(n^{1/2})$ time. It can also be adapted for pipelined processing of q queries so that q search queries can be processed in $O(n^{1/2}) + O(q)$ time.

The algorithm uses n processors arranged in an $n^{1/2} \times n^{1/2}$ square array. The processor in

the i^{th} row and j^{th} column is denoted by $P(i, j)$. Then elements are distributed among the n processors.

Let $s_{i,j}$ denote the element assigned to the processor $P(i, j)$. The query is submitted to the processor $P(1, 1)$. The algorithm proceeds in two stages namely *unfolding* and *folding*.

Unfolding: Each processor has a bit $b_{i,j}$ associated with it. The unfolding step is effectively a broadcast step. Initially, the processor $P(1, 1)$ reads x . At the end of the unfolding stage, all the processors get a copy of x . The bit $b_{i,j}$ is set to 1 if $x = s_{i,j}$, otherwise it is set to 0. The processor $P(1, 1)$ sends the value x to $P(1, 2)$. In the next step, both $P(1, 1)$ and $P(1, 2)$ send their values to $P(2, 1)$ and $P(2, 2)$, respectively. This unfolding process, which alternates between row and column propagation, continues until x reaches processor $P(n^{1/2}, n^{1/2})$. The exact sequence of steps for a 4×4 mesh is shown in Fig. 7.19 where each square represents a processor $P(i, j)$ and the number inside the square is element $s_{i,j}$, and $x = 15$.

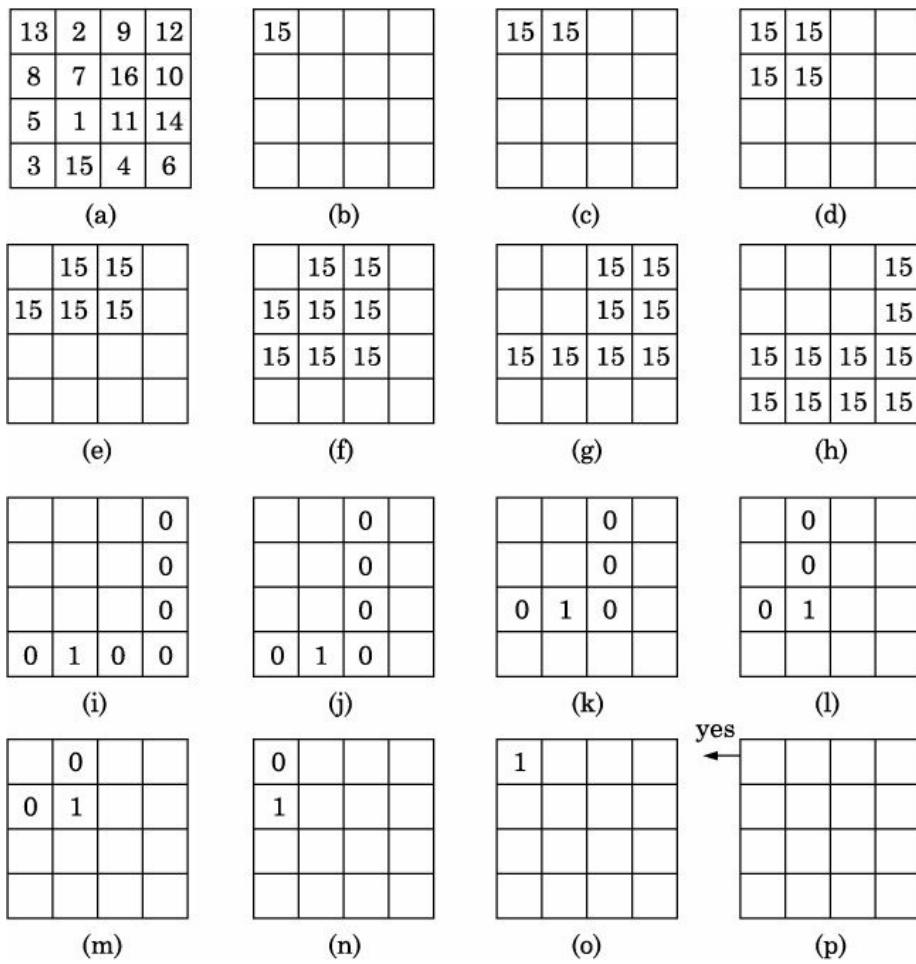


Figure 7.19 Unfolding and folding operations.

Folding: Folding is just the reverse of the unfolding process. The bits $b_{i,j}$ are propagated from row to row and from column to column in an alternating fashion, right to left and bottom to top, until the result of the search query emerges from $P(1, 1)$. Note that the result that is output at $P(1, 1)$ is just a logical *OR* operation of all the bits $b_{i,j}$. The algorithm is given below:

Procedure Mesh Search

```

Step 1: { $P(1, 1)$  reads the input}
  if  $x = s_{1,1}$  then  $b_{1,1} := 1$  else  $b_{1,1} := 0$ 
  end if

Step 2: {Unfolding}
  for  $i := 1$  to  $n^{1/2} - 1$  do
    for  $j := 1$  to  $i$  do in parallel
       $P(j, i)$  transmits  $(b_{j,i}, x)$  to  $P(i, j+1)$ 
      if ( $x = s_{j,i+1}$  or  $b_{j,i} = 1$ ) then  $b_{j,i+1} := 1$  else  $b_{j,i+1} := 0$ 
      end if
    end for
    for  $j := 1$  to  $i+1$  do in parallel
       $P(i, j)$  transmits  $(b_{i,j}, x)$  to  $P(i+1, j)$ 
      if ( $x = s_{i+1,j}$  or  $b_{i,j} = 1$ ) then  $b_{i+1,j} := 1$  else  $b_{i+1,j} := 0$ 
      end if
    end for
  end for

Step 3: {Folding}
  for  $i := n^{1/2}$  downto 2 do
    for  $j := 1$  to  $i$  do in parallel
       $P(j, i)$  transmits  $b_{j,i}$  to  $P(j, i-1)$ 
    end for
    for  $j := 1$  to  $i-1$  do in parallel
       $b_{j,i-1} := b_{j,i}$ 
    end for
    if ( $b_{i,i-1} = 1$  or  $b_{i,i} = 1$ ) then  $b_{i,i-1} := 1$  else  $b_{i,i-1} := 0$ 
    end if
    for  $j := 1$  to  $i-1$  do in parallel
       $P(i, j)$  transmits  $b_{i,j}$  to  $P(i-1, j)$ 
    end for
    for  $j := 1$  to  $i-2$  do in parallel
       $b_{i-1,j} := b_{i,j}$ 
    end for
    if ( $b_{i-1,i-1} = 1$  or  $b_{i,i-1} = 1$ ) then  $b_{i-1,i-1} := 1$  else  $b_{i-1,i-1} := 0$ 
    end if
  end for

Step 4: { $P(1, 1)$  produces the output}
  if  $b_{1,1} = 1$  then answer := yes else answer := no
  end if

```

Analysis

The steps 1 and 4 of the algorithm take constant time and the unfolding (step 2) and the folding (step 3) operations take $O(n^{1/2})$ time. Therefore the time required to process one query is $O(n^{1/2})$.

As in the case of searching on a tree, several queries can be processed in a pipeline fashion since the operations of any two queries do not clash. Inputs are submitted to processor $P(1, 1)$ at a constant rate. The outputs are also produced by $P(1, 1)$ at a constant rate following the answer of the first query. Therefore the time required to process q requests is $O(n^{1/2}) + O(q)$. Finally, note that if only one query is to be answered, the transmission of $b_{i,j}$ along with x during the unfolding stage is not required.

7.6 MATRIX OPERATIONS

Like sorting and searching, matrix multiplication and solution of systems of linear equations are fundamental components of many numerical and non-numerical algorithms. In this section, we examine several parallel algorithms to perform matrix multiplication and to solve systems of linear equations.

7.6.1 Matrix Multiplication

The product of an $m \times n$ matrix A and an $n \times k$ matrix B , is an $m \times k$ matrix C , whose elements are:

$$c_{ij} = \sum_{s=1}^n a_{is} * b_{sj} \quad (1 \leq i \leq m, 1 \leq j \leq k)$$

A simple sequential algorithm for the matrix multiplication is given below. Assuming that $m \leq n$ and $k \leq n$, the algorithm, Matrix Multiplication, runs in $O(n^3)$ time. However, there exist several sequential matrix multiplication algorithms whose running time is $O(n^x)$ where $2 < x < 3$. For example, Strassen has shown that two matrices of size 2×2 can be multiplied using only 7 multiplications. And by recursively using this algorithm he designed an algorithm for multiplying two matrices of size $n \times n$ in $O(n^{2.81})$ time. Interestingly to this date it is not known whether the fastest of such algorithms is optimal. Indeed, the only known lower bound on the number of steps required for matrix multiplication is the trivial $O(n^2)$ which is obtained by observing that n^2 outputs are to be produced. And therefore any algorithm must require at least this many steps. In view of the gap between n^2 and n^x , in what follows, we present algorithms whose cost is matched against the running time of the algorithm, Matrix Multiplication, given below. Further, we make the simplifying assumption that all the three matrices A , B , and C are $n \times n$ matrices. It is quite straightforward to extend the algorithms presented here to the asymmetrical cases.

Procedure Matrix Multiplication

```
for i := 1 to m do
    for j := 1 to k do
        cij := 0
        for s := 1 to n do
            cij := cij + ais * bsj
        end for
    end for
end for
```

CREW Matrix Multiplication

The algorithm uses n^2 processors which are arranged in a two-dimensional array of size $n \times n$.

The processor $P_{i,j}$ computes the element $c_{i,j}$ in the output matrix C . That is, each processor is assigned the task of computing one *inner product* of two n -dimensional vectors as shown in the algorithm given below. Therefore the overall time complexity of the algorithm is $O(n)$.

Procedure CREW Matrix Multiplication

```

for  $i := 1$  to  $n$  do in parallel
    for  $j := 1$  to  $n$  do in parallel
         $c_{i,j} := 0;$ 
        for  $k := 1$  to  $n$  do
             $c_{i,j} := c_{i,j} + a_{i,k} * b_{k,j};$ 
        end for
    end for
end for

```

EREW Matrix Multiplication

The algorithm, to be presented now, for the EREW model is a modification of the algorithm presented for the CREW model. In the CREW model, one had the advantage that a memory location could be accessed concurrently by different processors. In the EREW model one needs to ensure that every processor $P_{i,j}$ reads the value of a memory location which is not being accessed by any other processor at the same time. Assume that in the k^{th} iteration, $P_{i,j}$ selects the pair $(a_{i,l_k}, b_{l_k,j})$ for multiplication where $1 \leq l_k \leq n$. $P_{i,j}$ needs to find l_k in such a way that a_{i,l_k} and $b_{l_k,j}$ are not read by any other processor in the k^{th} iteration. Therefore, the conditions that l_k need to satisfy are:

1. $1 < l_k < n$.
2. l_1, l_2, \dots, l_k are all distinct.
3. a_{i,l_k} and $b_{l_k,j}$ can be read only by processor $P_{i,j}$ in iteration k .

The simplest function for l_k that satisfies all these three conditions is $l_k = ((i + j + k) \bmod n) + 1$. This function ensures that l_k lies between 1 and n , and also guarantees that l_1, l_2, \dots, l_k are all distinct. Suppose two processors $P_{i,j}$ and $P_{a,b}$ read the same element in a particular iteration, then we have either $i = a$ or $j = b$. This forces the other two elements also to be equal since $l_k(i,j) = l_k(a,b) \Rightarrow (i+j) = (a+b) \bmod n$. Therefore, $P_{i,j} = P_{a,b}$. This implies that the function l_k satisfies all the three conditions. Hence $P_{i,j}$ will correctly compute $c_{i,j}$. The algorithm, like the earlier algorithm, has the time complexity of $O(n)$ and requires $O(n^2)$ processors.

Procedure EREW Matrix Multiplication

```

for  $i := 1$  to  $n$  do in parallel
    for  $j := 1$  to  $n$  do in parallel
         $c_{i,j} := 0;$ 
        for  $k := 1$  to  $n$  do
             $l_k := (i + j + k) \bmod n + 1;$ 
             $c_{i,j} := c_{i,j} + a_{i,l_k} * b_{l_k,j};$ 
        end for
    end for
end for

```

CRCW Matrix Multiplication

The algorithm uses n^3 processors and runs in $O(1)$ time. Here the write conflicts are resolved as follows: Whenever more than one processor attempts to write to the same memory location, the *sum* of the values is written onto the memory location. Observe that the algorithm, given below, is a direct parallelization of sequential algorithm Matrix

Multiplication. Conceptually the processors may be thought of as being arranged in an $n \times n$ array pattern, each processor having three indices (i, j, s) .

Procedure CRCW Matrix Multiplication

```

for  $i := 1$  to  $n$  do in parallel
    for  $j := 1$  to  $n$  do in parallel
        for  $s := 1$  to  $n$  do in parallel
             $c_{i,j} := 0$ 
             $c_{i,j} := a_{i,s} * b_{s,j}$ 
        end for
    end for
end for

```

Finally, note that the cost of all the three algorithms given above matches the running time of sequential algorithm Matrix Multiplication.

Matrix Multiplication on a Mesh

The algorithm uses $n \times n$ processors arranged in a mesh configuration. The matrices A and B are fed into the *boundary* processors in column 1 and row 1, respectively as shown in Fig. 7.20. Note that row i of matrix A and column j of matrix B lag one time unit behind row $i-1$ ($2 \leq i \leq n$) and column $j-1$ ($2 \leq j \leq n$), respectively in order to ensure that $a_{i,k}$ meets $b_{k,j}$ ($1 \leq k \leq n$) in processor $P_{i,j}$ at the right time. Initially, $c_{i,j}$ is zero. Subsequently, when $P_{i,j}$ receives two inputs a and b it (i) multiplies them, (ii) adds the result to $c_{i,j}$ (iii) sends a to $P_{i,j+1}$ unless $j = n$, and (iv) sends b to $P_{i+1,j}$ unless $i = n$. At the end of the algorithm, $c_{i,j}$ resides in processor $P_{i,j}$.

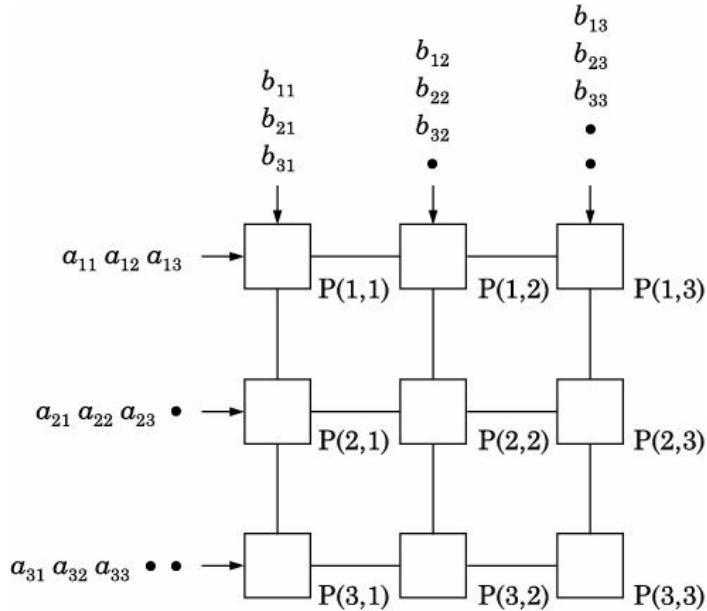


Figure 7.20 Matrix multiplication on a mesh.

Procedure Mesh Matrix Multiplication

```

for  $i := 1$  to  $n$  do in parallel
    for  $j := 1$  to  $n$  do in parallel
         $c_{i,j} := 0$ 
        while  $P_{i,j}$  receives two inputs  $a$  and  $b$  do

```

```

 $c_{i,j} := c_{i,j} + a * b$ 
if  $i < n$  then send  $b$  to  $P_{i+1,j}$ 
end if
if  $j < n$  then send  $a$  to  $P_{i,j+1}$ 
end if
end while
end for
end for

```

Analysis

Note that the processor $P_{i,j}$ receives its input after $(i - 1) + (j - 1)$ steps from the beginning of the computation. Once a processor gets its input it takes exactly n steps to compute the value $c_{i,j}$. Hence the value $c_{i,j}$ is computed after $(i - 1) + (j - 1) + n$ steps. So the time complexity of the algorithm is $O(n - 1 + n - 1 + n)$ which is $O(n)$. The cost of this algorithm is $O(n^3)$ since it requires n^2 processors.

7.6.2 Solving a System of Linear Equations

The problem of solving a system of linear equations ($Ax = b$) is central to many problems in engineering and scientific computing. Thus parallel algorithms and architectures for the high-speed solution of linear equations have been the focus of considerable attention in the last few decades. There are several methods for solving $Ax = b$ such as Gaussian elimination, LU decomposition, and QR decomposition. We will now discuss one of these methods, namely the Gaussian elimination.

Gaussian Elimination

Consider a system of n simultaneous linear equations of the following form

$$\begin{aligned} a_{1,1} x_1 + a_{1,2} x_2 + \cdots + a_{1,n} x_n &= b_1 \\ a_{2,1} x_1 + a_{2,2} x_2 + \cdots + a_{2,n} x_n &= b_2 \\ &\vdots \\ a_{n,1} x_1 + a_{n,2} x_2 + \cdots + a_{n,n} x_n &= b_n \end{aligned}$$

In matrix notation, this system of linear equations is written as $Ax = b$. Here A is a dense $n \times n$ matrix of coefficients such that $A[i, j] = a_{i,j}$, b is an $n \times 1$ column vector $[b_1, b_2, \dots, b_n]^T$, and x is the desired $n \times 1$ solution column vector $[x_1, x_2, \dots, x_n]^T$.

Gaussian elimination method, for solving $Ax = b$, has two phases. In the first phase, $Ax = b$ is reduced to the form $Ux = y$, where U is a unit upper-triangular matrix—one in which all subdiagonal elements are zero and all principal diagonal elements are equal to one, i.e., $U[i, i] = 1$ for $1 \leq i \leq n$, $U[i, j] = 0$, if $i > j$, otherwise $U[i, j] = u_{i,j}$. In the second phase, the upper-triangular system is solved, for the variables in reverse order from x_n to x_1 , using a method called *back-substitution*.

A sequential version of the Gaussian elimination algorithm, which converts $Ax = b$ to $Ux = y$, is given below. Here the matrix U shares storage with A and overwrites the upper-triangular portion of A . For simplicity, we assume that A is non-singular and $A[k, k] \neq 0$ when it is used as a divisor.

Procedure Gaussian Elimination

```
for  $k := 1$  to  $n$  do
```

```

for  $j := k + 1$  to  $n$  do
   $A[k,j] := A[k,j]/A[k,k]$  {Division step}
end for
 $y[k] := b[k]/A[k,k]$ 
 $A[k,k] := 1$ 
for  $i := k + 1$  to  $n$  do
  for  $j := k + 1$  to  $n$  do
     $A[i,j] := A[i,j] - A[i,k] * A[k,j]$ 
  end for
   $b[i] := b[i] - A[i,k] * y[k]$  {Elimination step}
   $A[i,k] := 0$ 
  end for
end for

```

Gaussian elimination involves approximately $n^2/2$ divisions (refer to division step) and approximately $(n^3/3) - (n^2/2)$ subtractions and multiplications (refer to elimination step). Assuming that each arithmetic operation takes unit time, the sequential running time of Gaussian elimination is approximately $2n^3/3$ (for large n), which is $O(n^3)$.

Parallel Implementation of Gaussian Elimination

We now provide a parallel implementation of the above algorithm in which one row of A is assigned to one processor i.e., processor P_i ($1 \leq i \leq n$) initially stores elements $A[i,j]$ for $1 \leq j \leq n$. At the beginning of the k^{th} iteration, the elements $A[k, k + 1], A[k, k + 2], \dots, A[k, n]$ are divided by $A[k, k]$ (refer to division step). This requires $n - k - 1$ divisions at processor P_k . Since all matrix operations participating in this operation lie on the same processor, this step does not require any communication. In the second computation step of the algorithm (refer to elimination step), the modified (after division) elements of the k^{th} row are used by all other rows of the *active part* (lower-right $k \times k$ submatrix) of the matrix A . This requires a one-to-all broadcast of the active part of the k^{th} row to the processors $k + 1$ to n . Finally, the computation $A[i, j] = A[i, j] - A[i, k] * A[k, j]$ takes place in the remaining active portion of the matrix. This involves $n - k$ multiplications and subtractions at all processors P_k ($k < i \leq n$). Thus the total time spent on computation (division and elimination steps) is equal to $\sum_k 3(n - k)$ which is equal to $3n(n - 1)/2$, assuming each arithmetic operation takes unit time.

The cost of one-to-all broadcast communication step varies according to the model of computation employed. In a PRAM model which supports concurrent read, the communication time required during the k^{th} iteration is $(n - k)$. The overall parallel running time of the algorithm on CRCW and CREW PRAM models is equal to $3n(n - 1)/2 + n(n - 1)/2 = 2n(n - 1)$, which is $O(n^2)$. In an EREW PRAM model, the sequence in which the active elements of the k^{th} row are read by the other processors can be modelled in such a way that every processor reads a unique element in every step (using the idea of *round-robin* as in EREW matrix multiplication). In this way we obtain an $O(n^2)$ time algorithm on EREW model. The algorithm is therefore cost optimal on the PRAM models (with respect to the sequential algorithm given above).

On a hypercube network, the communication step in the k^{th} iteration takes $t_s + t_w(n - k - 1) \log n$ time, where t_s and t_w denote *startup time* and *per-word transfer time*, respectively (See Exercise 7.5). Hence, the total communication time over all iterations is $\sum_k t_s + t_w(n - k - 1) \log n$ which is $t_s n \log n + t_w(n(n - 1)/2) \log n$. Thus the overall running time of the algorithm

on a hypercube is

$$3n(n - 1)/2 + t_s n \log n + t_w (n(n - 1)/2) \log n$$

The cost of the algorithm is therefore $O(n^3 \log n)$ (due to the term associated with t_w) and hence the algorithm is not cost-optimal.

Solving a Triangular System

We now briefly discuss the second phase of solving a system of linear equations. After matrix A has been reduced to a unit upper-triangular matrix U , we perform back-substitution to determine the vector x . A sequential back-substitution algorithm for solving an upper triangular system of equations $Ux = y$ is given below. The algorithm performs approximately $n^2/2$ multiplications and subtractions. Note that the number of arithmetic operations in the back-substitution is less than that in Gaussian elimination by a factor of $O(n)$. Hence, the back-substitution algorithm used in conjunction with the Gaussian elimination will have no effect on the overall time complexity of the algorithm.

Procedure Back-Substitution

```

for  $k := n$  downto 1 do
   $x[k] := y[k]$ 
  for  $i := k-1$  downto 1 do
     $y[i] := y[i] - x[k]*U[i,k]$ 
  end for
end for
```

Bidirectional Gaussian Elimination

Recall that in $Ax = b$, the value of x_i ($i = 1, 2, \dots, n$) depends on the value of x_j ($j = 1, 2, \dots, n$ and $i \neq j$), indicating the $(n - 1)^{\text{th}}$ level dependency. Hence, the influence of $(n - 1)$ other unknowns has to be unraveled to find one solution. In the Gaussian Elimination (GE) algorithm, the value x_i ($i = 1, 2, \dots, n$) is found by eliminating its dependency on x_k ($k < i$) in the matrix decomposition (or triangularization) phase and x_k ($k > i$) in the back-substitution phase. The major drawback of GE algorithm is that the triangularization phase and the back-substitution phase have to be carried out in a serial order, i.e., the back-substitution cannot begin until the triangularization phase is complete. Sometimes it may be necessary to design two (*systolic*) array structures, one for triangularization and the other for back-substitution, and use them in conjunction. It is obvious that if separate arrays are used for triangularization and back substitution, the I/O overhead (between the two phases) would increase substantially. We now present a back-substitution-free algorithm called Bidirectional Gaussian Elimination (BGE) for solving a set of linear algebraic equations.

Two linear systems are said to be equivalent provided that their solution sets are the same. Accordingly, the given set of n linear equations can be replaced by two sets of equivalent linear independent equations with half the number of unknowns by eliminating $n/2$ variables each in forward (left to right) and backward (right to left) directions. This process of replacing the given set of linear independent equations by two sets of equivalent equations with half the number of variables is used successively in the BGE algorithm. The algorithm involves the following steps to find the solution vector, x , in the equation $Ax = b$.

1. First, we form an augmented matrix $A|b$ (i.e., the b -vector is appended as the $(n + 1)^{\text{th}}$ column) of order $n \times (n + 1)$. This matrix is duplicated to form $A_0 | b_0$ and $b_1 |$

A_1 (where the b -vector is appended to the left of the coefficient matrix A) with A_0 and A_1 being the same as the coefficient matrix A , and similarly, b_0 and b_1 being the same as the b -vector. Note the b -vector is appended to the left or right of the coefficient matrix only for programming convenience.

2. Using the GE method, we triangularize $A_0 | b_0$ in the forward direction to eliminate

the subdiagonal elements in the columns $1, 2, \dots, \left(\frac{n}{2}\right)$ to reduce its order to $\frac{n}{2} \times \left(\frac{n}{2} + 1\right)$ (ignoring the columns eliminated and the corresponding rows). Concurrently, we triangularize $b_1 | A_1$ in the backward direction to eliminate the

superdiagonal elements in the columns $n, (n - 1), \dots, \left(\frac{n}{2} + 1\right)$ to reduce the order of $A_1 | b_1$ also to $\frac{n}{2} \times \left(\frac{n}{2} + 1\right)$ (again ignoring the columns eliminated and the corresponding rows). With this, $A_0 | b_0$ may be treated as a new augmented matrix

with columns and rows $\left(\frac{n}{2} + 1\right), \left(\frac{n}{2} + 2\right), \dots, n$ and the modified b -vector appended to its right, and similarly $b_1 | A_1$ as a new augmented matrix with columns and rows $1, 2, \dots, (n/2)$ and the modified b -vector appended to its left.

3. We duplicate the reduced augmented matrices $A_0 | b_0$ to form $A_{00} | b_{00}$ and $b_{01} | A_{01}$, and $b_1 | A_1$ to form $A_{10} | b_{10}$ and $b_{11} | A_{11}$ (each of these duplicated matrices will

be of the same order, $\frac{n}{2} \times \left(\frac{n}{2} + 1\right)$. We now triangularize $A_{00} | b_{00}$ and $A_{10} | b_{10}$ in the forward direction and $b_{01} | A_{01}$ and $b_{11} | A_{11}$ in the backward direction through $n/4$ columns using the GE method, thus reducing the order of each of these matrices to

half of their original size, i.e., $\frac{n}{4} \times \left(\frac{n}{4} + 1\right)$. Note that the above four augmented matrices are reduced in parallel.

4. We continue this process of halving the size of the submatrices using GE and doubling the number of submatrices $\log n$ times so that we end up with n submatrices each of order 1×2 . The modified b -vector part, when divided by the modified A matrix part in parallel, gives the complete solution vector, x .

The solution of $Ax = b$ using the BGE algorithm is shown in Fig. 7.21 in the form of a binary tree for $n = 8$. We leave the implementation of this BGE algorithm on mesh-connected processors to the reader as an exercise. Finally, note that in the BGE algorithm, all x_i ($i = 1, 2, \dots, n$) are found simultaneously, unlike in the GE algorithm. Hence the BGE algorithm is expected to have better numerical stability characteristics (i.e., lesser *rounding errors* due to the finite precision of floating-point numbers stored in the memory) than the GE algorithm. The idea of bidirectional elimination is used in efficiently solving sparse systems of linear equations and also dense systems of linear equations using Givens rotations.

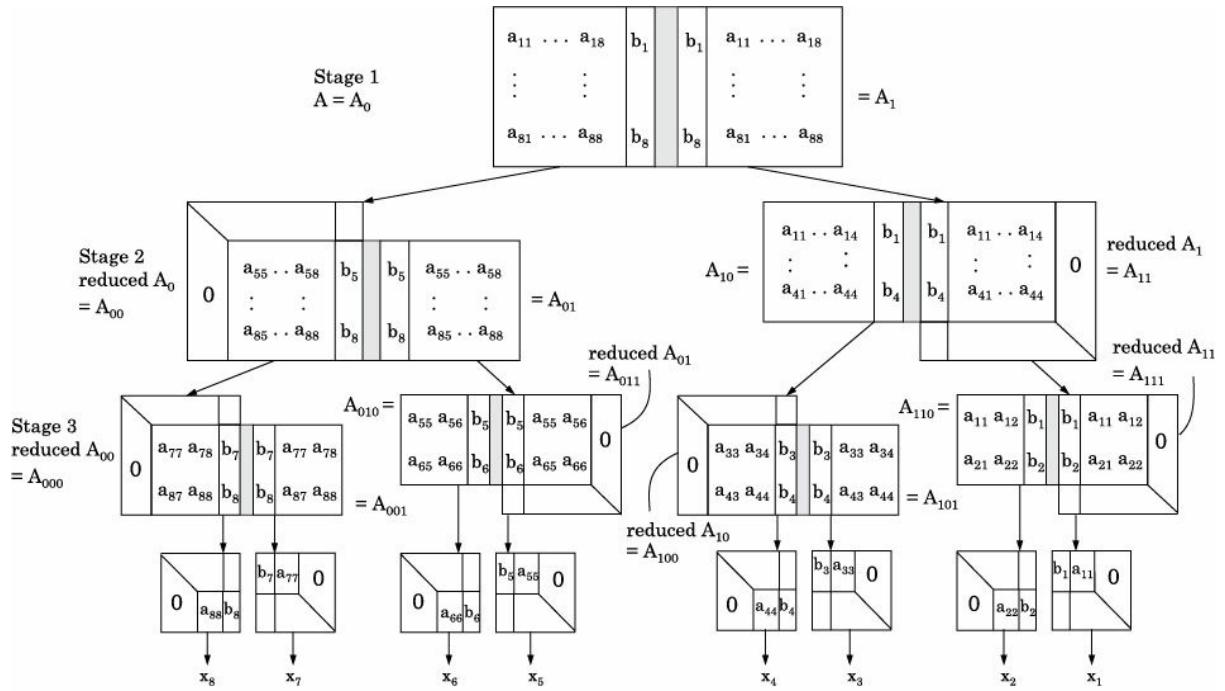


Figure 7.21 Progression of the BGE algorithm.

7.7 PRACTICAL MODELS OF PARALLEL COMPUTATION

So far we looked at a variety of parallel algorithms for various models of a parallel computer. In this section, we first look at some drawbacks of the models we studied and then present some recent models of a parallel computer which make realistic assumptions about a parallel computer.

The PRAM is the most popular model for representing and analyzing the complexity of parallel algorithms due to its simplicity and clean semantics. The majority of theoretical parallel algorithms are specified using the PRAM model or a variant thereof. The main drawback of the PRAM model lies in its unrealistic assumptions of zero communication overhead and instruction level synchronization of the processors. This could lead to the development of parallel algorithms with a degraded performance when implemented on real parallel machines. To reduce the theory-to-practice disparity, several extensions of the original PRAM model have been proposed such as *delay* PRAM (which models delays in memory access times) and *phase* PRAM (which models synchronization delays).

In an interconnection network model, communication is only allowed between directly connected processors; other communication has to be explicitly forwarded through intermediate processors. Many algorithms have been developed which are perfectly matched to the structure of a particular network. However, these elegant algorithms lack robustness, as they usually do not map with equal efficiency onto interconnection networks different from those for which they were designed. Most current networks allow messages to *cut through* (Exercise 7.5) intermediate processors without disturbing the processor; this is much faster than explicit forwarding.

7.7.1 Bulk Synchronous Parallel (BSP) Model

The BSP model, proposed by Valiant, overcomes the shortcomings of the PRAM model while keeping its simplicity. A BSP *computer* consists of a set of n processor/memory pairs (*nodes*) which are interconnected by a communication network as shown in Fig. 7.22.

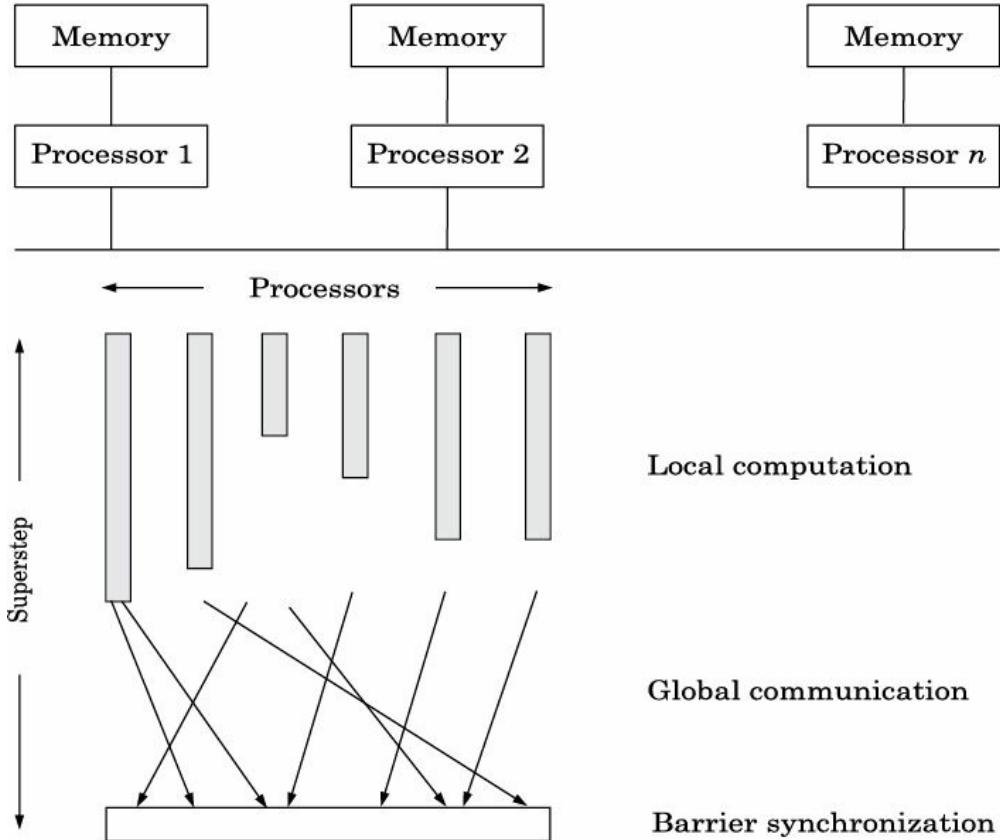


Figure 7.22 BSP model.

A BSP program has n processes with one process per node. The basic time unit is a *cycle* (or time step). The program executes as a strict sequence of *supersteps*. Within a superstep, a process executes the computation operations in at most w cycles, a communication operation that takes c cycles, and a *barrier synchronization* that takes l cycles. The barrier forces the processes to wait so that all the processes have to finish the current superstep before any of them can begin the next superstep. Thus the time for a superstep is $w + c + l$. If there is an overlap of these three (computation, communication, and synchronization) operations, the time for a superstep becomes $\max(w, c, l)$.

The processes in the BSP computer can execute different instructions simultaneously i.e., the BSP computer is an MIMD machine. Within a superstep, different processes execute asynchronously at their own pace. The BSP model abstracts out the communication operations in a BSP superstep by the *h relation* concept. An *h relation* is an abstraction of any communication operation, where each node sends at most h words to various nodes and each node receives at most h words. (A communication is always realized in a point-to-point manner.) On a BSP computer, the time to realize any *h relation* is no more than gh cycles, where g is a constant decided by the machine platform. g has a small value on a computer with more efficient communication support. Thus the number of cycles taken by a communication operation, namely c , equals gh .

Consider the computation of inner product s of two N -dimensional vectors A and B on an n -processor EREW PRAM computer. Each processor performs $2N/n$ additions and multiplications to generate a local result in $2N/n$ cycles. These n local sums are then added to obtain the final sum s in $\log n$ cycles. Thus the total execution time is $2N/n + \log n$ cycles. Note that the sequential algorithm takes $2N$ cycles. We now solve the same problem on a 4-processor BSP computer in three supersteps:

- Superstep 1: Each processor computes its local sum in $w = 2N/4$ cycles. There is a communication of 1 word (and thus $h = 1$). There is one barrier synchronization.
- Superstep 2: Two processors each performs one addition ($w = 1$). Again there is a communication of 1 word. There is one barrier synchronization.
- Superstep 3: One processor performs one addition ($w = 1$) to generate the final sum. No more communication or synchronization is needed.

The total execution time is $2N/4 + 2 + 2g + 2l$ cycles. On an n -processor BSP it is $2N/n + \log n(g + l + 1)$ cycles. This is in contrast to the $2N/n + \log n$ cycles on an EREW PRAM computer. The two extra terms, $g \log n$ and $l \log n$, correspond to *communication* and *synchronization* overheads, respectively.

Multi-BSP Model

The BSP model has been extended to the Multi-BSP model to capture the most basic resource parameters of multicore architectures. The Multi-BSP model extends BSP in two ways: (i) It is a hierarchical model with an arbitrary number d of levels modelling multiple memory and cache levels. (ii) At each level the model incorporates memory size as a further parameter. The model is based on a tree structure of depth (number of levels) d with memory/caches at the internal nodes and processors/cores at the leaves. The computational model has explicit parameters, for various important aspects of parallel computation in multicores, which are number of processors, sizes of memories/caches, communication cost, and synchronization cost.

The computational model is recursive and consists of multiple levels of components as shown in Fig. 7.23. Each level j consists of 4 parameters $(p_j; g_j; L_j; m_j)$, where p_j is the number of level $(j - 1)$ components inside a level j component, g_j is the data rate at which the j^{th} component communicates with the $(j + 1)^{\text{th}}$ component, L_j is the synchronization cost, and m_j is the size of the memory. As mentioned earlier, the components at the lowest level are processors or cores and the components at higher levels are memories or caches. An example to model an instance of p Sun Niagara UltraSparc T_1 multicores which are connected to an external storage that is large enough to store the input to the problem to be solved is as follows:

level 1: $(p_1 = 4; g_1 = 1; L_1 = 3; m_1 = 8 \text{ KB})$

(each core runs 4 threads and has an L_1 cache of size 8 KB);

level 2: $(p_2 = 8; g_2 = 3; L_2 = 23; m_2 = 3 \text{ MB})$

(each chip has 8 cores with a shared L_2 cache of size 3 MB);

level 3: $(p_3 = p; g_3 = \infty; L_1 = 108; m_3 = 128 \text{ GB})$

(p chips with external memory of size m_3 accessible via a network with rate g_2).

In this example, L_j parameters are not exactly synchronizing costs but latency parameters given in the chip specification. In addition, caches on the chip are managed implicitly by hardware and not by the parallel algorithm, and cores share a common arithmetic unit. Thus, while the relative values of the g and L parameters shown here may be meaningful, it is hard to obtain their exact values for this instance.

It may be noted that an instance of Multi-BSP model with $d = 1$ and parameters $(p_1 = 1; g_1 = \infty; L_1 = 0; m_1)$ is the von Neumann model, while an instance with parameters $(p_1 \geq 1; g_1 = \infty; L_1 = 0; m_1)$ is the PRAM model, where m_1 is the size of the memory. In the BSP model,

direct communication is allowed horizontally between units at the same level, while in the Multi-BSP model such communication needs to be simulated via memory at a higher level.

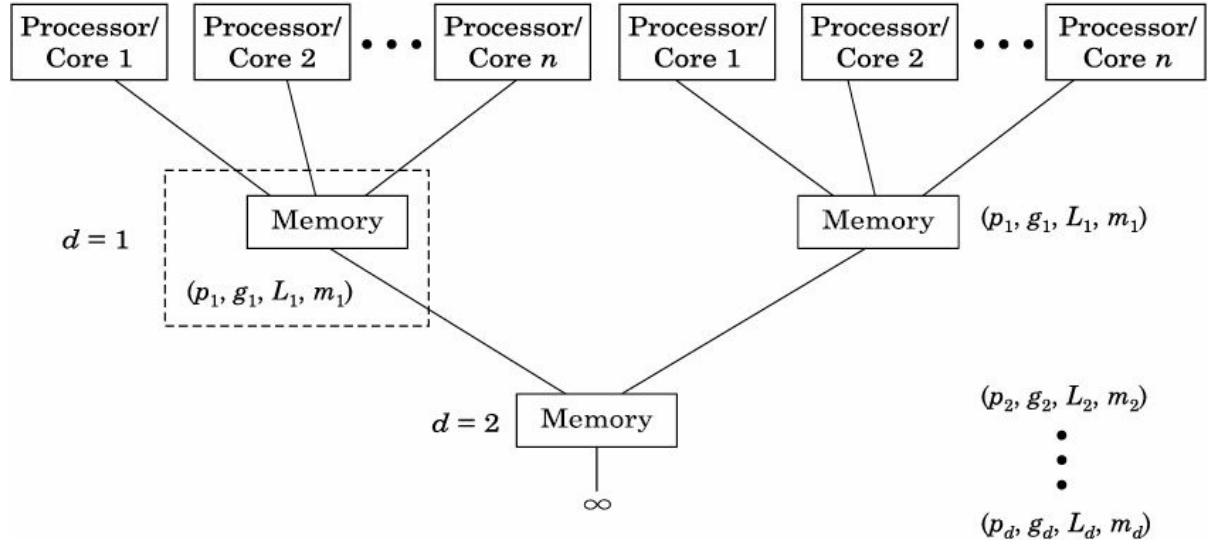


Figure 7.23 Multi-BSP model.

7.7.2 LogP Model

We now look at yet another practical model of parallel computation called the LogP model. The LogP model, proposed by Culler *et al.*, was influenced by the following technological developments:

1. The processor speed is increasing at the rate of 50–100% per year.
2. The memory capacity is increasing at a rate comparable to the increase in capacity of DRAM chips; quadrupling in size every three years. (DRAMs, or dynamic random access memory chips, are the low-cost, low-power memory chips used for main memory in most computers.) The increase in the speed of DRAM, however, is much lower than that of CPU. Thus sophisticated cache structures are required to bridge the difference between processor cycle times and memory access times.
3. The network speed is also increasing, but not at a rate that of the processor and the memory. The communication bandwidth available on a network is considerably lower than that available between a processor and its local memory.
4. There appears to be no consensus emerging on interconnection network topology: the networks of new commercial machines are typically different from their predecessors and different from each other. Note that each interconnection network is optimized for a particular set of applications.

Culler *et al.*, argue that the above technological factors are forcing parallel computer architectures to converge towards a system made up of a collection of essentially complete computers (each consisting of a microprocessor, cache memory, and sizable DRAM memory) connected by a communication network. This convergence is reflected in the LogP model.

The LogP model abstracts out the interconnection network in terms of a few parameters. Hence the algorithms designed using the LogP model do not suffer a performance degradation when ported from one network to another. The LogP model tries to achieve a good compromise between having a few design parameters to make the task of designing algorithms in this model easy, and trying to capture the parallel architecture without making

unrealistic assumptions.

The parameters of the LogP model to characterize a parallel computer are:

1. **L :** This is an upper bound on the *latency* or *delay* incurred in communicating a message containing a word (or small number of words) from one processor to another. (Note that the exact routing algorithm used is not specified as part of the LogP model.)
2. **o :** This is the *overhead* which is defined as the length of time that a processor is engaged in the transmission or reception of each message. The processor is not allowed to perform other operations during this time.
3. **g :** This denotes the *gap* which is defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
4. **P :** This is the number of processors in the network.

The LogP model assumes the following about the processors and the underlying network:

(i) The processors execute asynchronously, (ii) The messages sent to a destination processor may not arrive in the order that they were sent, and (iii) The network has a finite capacity such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time.

Notice that, unlike the PRAM model, the LogP model takes into account the inter-processor communication. It does not assume any particular interconnection topology but captures the constraints of the underlying network through the parameters L , o , g . These three parameters are measured as multiples of the processor cycle.

We now illustrate the LogP model with an example. Consider the problem of broadcasting a single datum from one processor to the remaining $P - 1$ processors. Each processor that has the datum transmits it as quickly as possible while ensuring that no processor receives more than one message. The optimal broadcast tree is shown in Fig. 7.24 for the case of $P = 8$, $L = 6$, $g = 4$, $o = 2$. The number shown here for each processor is the time at which it has received the datum and can begin sending it on. The broadcast operation starts at time $t = 0$ from processor P_1 . The processor P_1 incurs an overhead of o ($= 2$) units in transferring the datum to processor P_2 . The message reaches P_2 after L ($= 6$) units of time i.e., at time $t = 8$ units. The processor P_2 then spends o ($= 2$) units of time in receiving the datum. Thus at time $t = L + 2o = 10$ units the processor P_2 has the datum. The processor P_1 meanwhile initiates transmissions to other processors at time g , $2g$, ..., (assuming $g \geq o$) each of which acts as the root of a smaller broadcast tree. The processor P_2 now initiates transmission to other nodes independently. The last value is received at time 24 units. Note that this solution to the broadcast problem on the LogP model is quite different from that on the PRAM model.

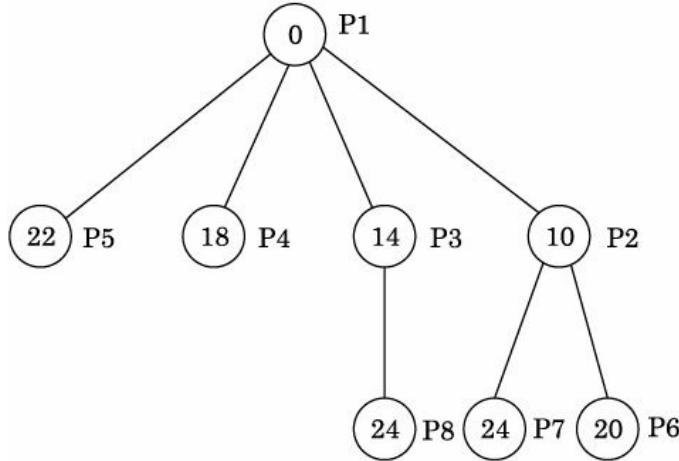


Figure 7.24 Optimal broadcast tree for $P = 8, L = 6, g = 4, o = 2$.

The LogP model only deals with short messages and does not adequately model parallel computers with support for long messages. Many parallel computers have special support for long messages which provide a much higher bandwidth than for short messages. A refinement to the LogP model, called LogGP model, has been proposed which accurately predicts communication performance for both short and long messages.

LogGP Model

Here the added parameter G , called *gap per byte*, is defined as the time per byte for a long message. The reciprocal of G characterizes the available per processor communication bandwidth for long messages.

The time to send a short message can be analyzed, in the LogGP model, as in the LogP model. Sending a short message between two processors takes $o + L + o$ cycles: o cycles in the sending processor, L cycles for the communication latency, and finally, o cycles on the receiving processor. Under the LogP model, sending a k -byte message from one processor to another requires sending $\lceil k/w \rceil$ messages, where w is the underlying message size of the parallel computer. This would take $o + (k/w - 1) * \max\{g, o\} + L + o$ cycles. In contrast, sending everything as a single large message takes $o + (k - 1)G + L + o$ cycles when we use the LogGP model (Note that g does not appear here since only a single message is sent).

LogGPS Model

The LogGP model has been extended to the LogGPS model by including one additional parameter ‘S’ in order to capture synchronization cost that is needed when sending long messages by high-level communication libraries. The parameter ‘S’ is defined as the threshold for message length above which synchronous messages are sent (i.e., synchronization occurs, before sending a message, between sender and receiver).

7.8 CONCLUSIONS

The primary ingredient in solving a computational problem on a parallel computer is the parallel algorithm without which the problem cannot be solved in parallel. In this chapter we first presented three models of parallel computation—(i) PRAM, (ii) interconnection networks, and (iii) combinational circuits, which differ according to whether the processors communicate among themselves through a shared memory or an interconnection network, whether the interconnection network is in the form of an array, a tree, or a hypercube, and so on. We then studied the design and analysis of parallel algorithms for the computational problems: (i) prefix computation, (ii) sorting, (iii) searching, (iv) matrix operations on the above models of parallel computation. Finally, we presented some recent models of parallel computation, namely, BSP, LogP and their extensions which intend to serve as a basis for the design and analysis of fast, portable parallel algorithms on a wide variety of current and future parallel computers.

EXERCISES

- 7.1 CRCW PRAM model is the most powerful of the four PRAM models. It is possible to emulate any algorithm designed for CRCW PRAM model on the other PRAM models. Describe how one can emulate an algorithm of SUM CRCW PRAM model on (a) CREW PRAM model and (b) EREW PRAM model. Also compute the increase in running time of the algorithms on these two weaker models.
- 7.2 Consider an EREW PRAM model with n processors. Let m denote the size of the common memory. Emulate this model on n -processor (a) ring, (b) square mesh and (c) hypercube interconnection network models, where each processor has m/n memory locations. Also compute the increase in running time of an n -processor EREW PRAM algorithm when it is emulated on these three interconnection network models.
- 7.3 Design an efficient algorithm for finding the sum of n numbers on a CREW PRAM model.
- 7.4 Develop an $O(\log n)$ time algorithm for performing a *one-to-all broadcast* communication operation (where one processor sends an identical data element to all the other $(n - 1)$ processors) on a CREW PRAM model, where n is the number of processors.
- 7.5 The time taken to communicate a message between two processors in a network is called communication latency. Communication latency involves the following parameters.
- (a) Startup time (t_s): This is the time required to handle a message at the sending processor. This overhead is due to message preparation (adding header, trailer, and error correction information), execution of the routing algorithm, etc. This delay is incurred only once for a single message transfer.
 - (b) Per-hop time (t_h): This is the time taken by the header of a message to reach the next processor in its path. (This time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.)
 - (c) Per-word transfer time (t_w): This is the time taken to transmit one word of a message. If the bandwidth of the links is r words per second, then it takes $t_w = 1/r$ seconds to traverse a link.

There are two kinds of routing techniques, namely, *store-and-forward* routing, and *cut-through* routing. In store-and-forward routing, when a message is traversing a path with several links, each intermediate processor on the path forwards the message to the next processor after it has received and stored the entire message. Therefore, a message of size m (words) takes

$$T_{\text{comm}} = t_s + (mt_w + t_h) l$$

time to traverse l links. Since, in general, t_h is quite small compared to mt_w , we simplify the above equation as

$$t_{\text{comm}} = t_s + mt_w l$$

In cut-through routing, an intermediate node waits for only parts (of equal size) of a message to arrive before passing them on. It is easy to show that in cut-through routing a message of size m (words) requires

$$t_{\text{comm}} = t_s + lt_h + mt_w$$

time to traverse l links. Note that, disregarding the startup time, the cost to send a message of size m over l links (hops) is $O(ml)$ and $O(m + l)$ in store-and-forward routing and cut-through routing, respectively.

Determine an optimal strategy for performing a one-to-all broadcast communication operation on (a) a ring network with p processors and (b) a mesh network with $p^{1/2} \times p^{1/2}$ processors using (i) store-and-forward routing and (ii) cut-through routing.

7.6 Repeat Exercise 7.5 for performing an *all-to-all broadcast* operation. (This is a generalization of one-to-all broadcast in which all the processors simultaneously initiate a broadcast.)

7.7 Using the idea of parallel prefix computation develop a PRAM algorithm for solving a first-order linear recurrence:

$$x_j = a_j x_{j-1} + d_j,$$

for $j = 1, 2, \dots, n$ where the values $x_0, a_1, a_2, \dots, a_n$ and d_1, d_2, \dots, d_n are given. For simplicity, and without loss of generality, you can assume that $x_0 = a_1 = 0$.

7.8 Design and analyze a combinational circuit for computing the prefix sums over the input $\{x_0, x_1, \dots, x_{n-1}\}$.

7.9 Consider a linked list L . The *rank* of each node in L is the distance of the node from the end of the list. Formally, if $\text{succ}(i) = \text{nil}$, then $\text{rank}(i) = 0$; otherwise $\text{rank}(i) = \text{rank}(\text{succ}(i)) + 1$. Develop a parallel algorithm for any suitable PRAM model to compute the rank of each node in L .

7.10 Prove formally the correctness of the odd-even transposition sort algorithm.

7.11 Design a bitonic sorting algorithm to sort n elements on a hypercube with fewer than n processors.

7.12 Extend the searching algorithm for a tree network, presented in this chapter, when the number of elements, n , is greater than the number of leaf nodes in the tree.

7.13 The transpose of the matrix $A = [a_{ij}]_{m \times n}$, denoted by A^T , is given by the matrix $[a_{ji}]_{n \times m}$.

Develop an algorithm for finding the transpose of a matrix on (a) EREW PRAM machine and (b) hypercube network.

7.14 Design an algorithm for multiplying two $n \times n$ matrices on a mesh with fewer than n^2 processors.

7.15 Describe in detail a parallel implementation of the Bidirectional Gaussian Elimination algorithm on (a) a PRAM machine and (b) a mesh network. Also derive expressions for the running time of the parallel implementations.

7.16 A Mesh Connected Computer with Multiple Broadcasting (MCCMB) is shown in Fig. 7.25. It is a mesh connected computer in which the processors in the same row or column are connected to a bus in addition to the local links. On an MCCMB there are two types of data transfer instructions executed by each processor: routing data to one of its nearest neighbours via a local link and broadcasting data to other processors in the same row (or column) via the bus on this row (or column). At any time only one type of data routing instruction can be executed by processors. Further, only one processor is allowed to broadcast data on each row bus and each column bus at a time. Note that other than the multiple broadcast feature the control organization of an MCCMB is the same as that of the well-known mesh connected computer. Design and analyze

algorithms for (a) finding the maximum of n elements and (b) sorting n elements on an MCCMB. Are your algorithms cost-optimal? Assume that each broadcast along a bus takes $O(1)$ time.

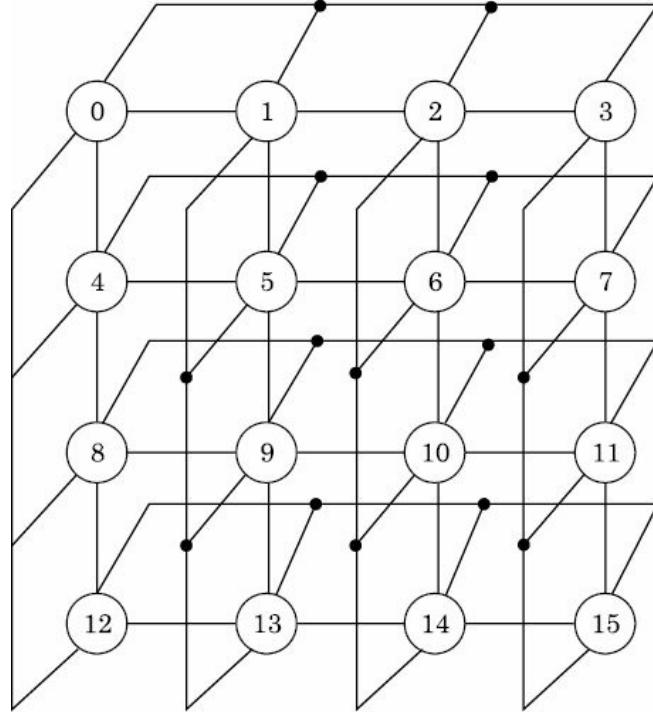


Figure 7.25 Mesh connected computer with multiple broadcasting.

7.17 A reconfigurable mesh consists of an array of processors connected to a grid-shaped reconfigurable broadcast bus. A 4×5 reconfigurable mesh is shown in Fig. 7.26. Here each processor can perform either an arithmetic operation or a logical operation in one time unit.

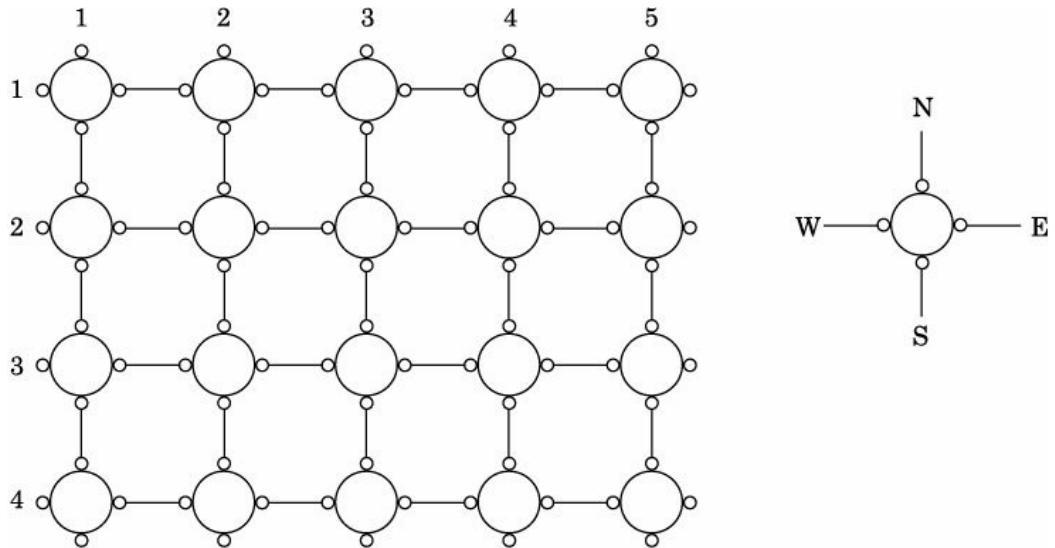


Figure 7.26 Reconfigurable mesh.

Within each processor four ports denoted by N , S , W , E (standing for North, South, West, and East, respectively) are built. The processors are connected to the reconfigurable mesh through these ports. The configuration of the bus is dynamically changeable by adjusting the local connection among the ports within each processor.

For example, by connecting port W to port E within each processor, horizontally straight buses can be established to connect processors in the same row together. Furthermore, each of the horizontally straight buses is split into sub-buses, if some processors disconnect their local connections between port W and port E . Note that when no local connection among ports is set within each processor, the reconfigurable mesh is functionally equivalent to a mesh connected computer. Design and analyze algorithms for (a) finding the maximum of n elements and (b) sorting n elements on a reconfigurable mesh. Are your algorithms cost-optimal? Assume that each broadcast along a bus takes one time unit.

- 7.18 What is the (non-overlapping) cost of a superstep in the BSP model?
- 7.19 Compute the execution time of FFT shown in Fig. 7.4 on a BSP machine with four processors.
- 7.20 Under LogP model, what is the (ping-pong) round trip delay between two processors?
- 7.21 Design and analyze an optimal algorithm for finding the sum of n numbers on LogP model with 4 processors in the network.
- 7.22 Design and analyze an algorithm for multiplying two $n \times n$ matrices on an n -processor BSP and LogP machines.
- 7.23 $\text{BSP} = \text{LogP} + \text{barriers} - \text{overhead}$. Is this true?
- 7.24 Verify if $\text{BSP}(p, g, L)$ with memory m per processor = Multi-BSP with $d = 2$, $(p_1 = 1; g_1 = g; L_1 = 0; m_1 = m)$ and $(p_2 = p; g_2 = \infty; L_2 = L; m_2)$.

BIBLIOGRAPHY

- Akl, S.G., *Parallel Computation: Models and Methods*, Prentice Hall, Upper Saddle River, New Jersey, USA, 1997.
- Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1989.
- Alejandro Salinger, “Models for Parallel Computation in Multicore, Heterogeneous, and Ultra Wide-Word Architectures”, PhD Thesis, University of Waterloo, 2013.
- Alexandrov, A., Ionescu, M.F., Schausler, K.E. and Scheiman, C., “LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation”, *Journal of Parallel and Distributed Computing*, Vol. 44, No. 1, July 1997, pp. 71–79.
- Balasubramanya Murthy, K.N. and Siva Ram Murthy, C., “Gaussian Elimination Based Algorithm for Solving Linear Equations on Mesh-connected Processors”, *IEEE Proc. Part E, Computers and Digital Techniques*, Vol. 143, No. 6, Nov. 1996, pp. 407–412.
- Balasubramanya Murthy, K.N., Bhuvaneswari, K. and Siva Ram Murthy, C., “A New Algorithm Based on Givens Rotations for Solving Linear Equations on Fault-tolerant Mesh-connected Processors”, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 9, No. 8, Aug. 1998, pp. 825–832.
- Chowdhury, R.A., Ramachandran, V., Silvestri, F. and Blakeley, B., “Oblivious Algorithms for Multicores and Networks of Processors”, *Journal of Parallel and Distributed Computing*, Vol. 73, No. 7, July 2013, pp. 911–925.
- Culler, D., Karp, R., Patterson, D., Sahay, A., Schausler, K.E., Santos, E., Subramanian, R. and Eicken, T., “LogP: Towards a Realistic Model of Parallel Computation”, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 235–261.
- Ghosh, R.K., Rajat Moona, and Gupta, P., *Foundations of Parallel Processing*, Narosa Publishing House, New Delhi, 1995.
- Grama, A., Gupta, A., Karypis, G. and Kumar, V., *Introduction to Parallel Computing*, Pearson, London, 2003.
- Hwang, K. and Xu, Z., *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, USA, 1998.
- Ino, F., Fujimoto, N. and Hagiraha, K., “LogGPS: A Parallel Computational Model for Synchronization Analysis”, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001, pp. 133–142.
- Krishnan, C.S.R. and Siva Ram Murthy, C., “A Faster Algorithm for Sorting on Mesh Connected Computers with Multiple Broadcasting Using Fewer Processors”, *International Journal of Computer Mathematics*, Vol. 48, 1993, pp. 15–20.
- Kumar, V., Grama, A., Gupta, A. and Karypis, G., *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, USA, 1994.
- Pradeep, B. and Siva Ram Murthy, C., “Constant Time Algorithm for Theorem Proving in Propositional Logic on Reconfigurable Meshes”, *Information Sciences*, Vol. 85, No. 1–3, 1995, pp. 175–184.
- Rajasekaran, S., Fiondella, L., Ahmed, M. and Ammar, R.A. (Eds.), *Multicore Computing: Algorithms, Architectures, and Applications*, CRC Press, USA, 2013.
- Rauber, T. and Runger, G., *Parallel Programming for Multicore and Cluster Systems*,

- Springer-Verlag, Germany, 2010.
- Siva Ram Murthy, C., Balasubramanya Murthy, K.N. and Srinivas, A., *New Parallel Algorithms for Direct Solution of Linear Equations*, John Wiley & Sons, New York, USA, 2001.
- Strassen, V., “Gaussian Elimination is Not Optimal”, *Numerische Mathematik*, Vol. 13, 1969, pp. 354–356.
- Valiant, L.G., “A Bridging Model for Parallel Computation”, *Communications of the ACM*, Vol. 33, No. 8, Aug. 1990, pp. 103–111.
- Valiant, L.G., “A Bridging Model for Multi-core Computing”, *Journal of Computer and System Sciences*, Vol. 77, No. 1, Jan. 2011, pp. 154–166.
- Yzelman, A.N., “Towards a Highly Scalable Multi-BSP Fast Fourier Transformation”, *SIAM Conference on Parallel Processing for Scientific Computing*, 2014.
- Yzelman, A.N., Bisseling, R.H., Roose, D. and Meerbergen, K., “MulticoreBSP for C: A High-performance Library for Shared-memory Parallel Programming”, *International Journal of Parallel Programming*, Vol. 42, No. 4, Aug. 2014, pp. 619–642.

Parallel Programming

In the previous chapter, we have presented parallel algorithms for many problems. To run these algorithms on a parallel computer, we need to implement them in a programming language. Exploiting the full potential of a parallel computer requires a cooperative effort between the user and the language system. There is clearly a trade-off between the amount of information the user has to provide and the amount of effort the compiler has to expend to generate efficient parallel code. At one end of the spectrum are languages where the user has full control and has to explicitly provide all the details while the compiler effort is minimal. This approach, called *explicit parallel programming*, requires a parallel algorithm to explicitly specify how the processors will cooperate in order to solve a specific problem. At the other end of the spectrum are sequential languages where the compiler has full responsibility for extracting the parallelism in the program. This approach, called *implicit parallel programming*, is easier for the user because it places the burden of parallelization on the compiler. Clearly, there are advantages and disadvantages to both, explicit and implicit parallel programming approaches. Many current parallel programming languages for parallel computers are essentially sequential languages augmented by library functions calls, compiler directives or special language constructs. In this chapter, we study four different (explicit) *parallel programming models* (a programming model is an abstraction of a computer system that a user sees and uses when developing a parallel program), namely, message passing programming (using MPI), shared memory programming (using OpenMP), heterogeneous programming (using CUDA and OpenCL) and MapReduce programming (for large scale data processing). Implicit parallel programming approach, i.e., automatic parallelization of a sequential program, will be discussed in the next chapter.

8.1 MESSAGE PASSING PROGRAMMING

In message passing programming, programmers view their programs (applications) as a collection of cooperating processes with private (local) variables. The only way for an application to share data among processors is for the programmer to explicitly code commands to move data from one processor to another. The message passing programming style is naturally suited to message passing parallel computers in which some memory is local to each processor but none is globally accessible.

Message passing parallel computers need only two primitives (commands) in addition to normal sequential language primitives. These are *Send* and *Receive* primitives. The *Send* primitive is used to send data/instruction from one processor to another. There are two forms of *Send* called *Blocked Send* and *Send*. *Blocked Send* sends a message to another processor and waits until the receiver has received it before continuing the processing. This is also called *synchronous* send. *Send* sends a message and continues processing without waiting. This is known as *asynchronous* send. The arguments of *Blocked Send* and *Send* are:

Blocked Send (destination processor address, variable name, size)

Send (destination processor address, variable name, size)

The argument “destination processor address” specifies the processor to which the message is destined, the “variable name” specifies the starting address in memory of the sender from where data/instruction is to be sent and “size” the number of bytes to be sent. Similarly *Receive* instructions are:

Blocked Receive (source processor address, variable name, size)

Receive (source processor address, variable name, size)

The “source processor address” is the processor from which message is expected, the “variable name” the address in memory of the receiver where the first received data item will be stored and “size” the number of bytes of data to be stored.

If data is to be sent in a particular order or when it is necessary to be sure that the message has reached its destination, *Blocked Send* is used. If a program requires specific piece of data from another processor before it can continue processing, *Blocked Receive* is used. *Send* and *Receive* are used if ordering is not important and the system hardware reliably delivers messages. If a message passing system’s hardware and operating system guarantee that messages sent by *Send* are delivered correctly in the right order to the corresponding *Receive* then there will be no need for a *Blocked Send* and *Blocked Receive* in that system. Such a system is called an *asynchronous message passing system*.

We illustrate programming a message passing parallel computer with a simple example. The problem is to add all the components of an array $A[i]$ for $i = 1, N$, where N is the size of the array.

We begin by assuming that there are only two processors. If N is even then we can assign adding $A[1], \dots, A[N/2]$ to one processor and adding $A[N/2 + 1], \dots, A[N]$ to the other. The procedures for the two processors are shown below.

Processor0

Read $A[i]$ for $i = 1, \dots, N$

Send to Processor1 $A[i]$ for $i = (N/2 + 1), \dots, N$

Find $\text{Sum0} = \text{Sum of } A[i] \text{ for } i = 1, \dots, N/2$

Receive from Processor1, Sum1 computed by it

$\text{Sum} = \text{Sum0} + \text{Sum1}$

Write Sum

Processor1

Receive from Processor0 $A[i]$ for $i = (N/2 + 1), \dots, N$

Find $Sum1 = \text{Sum of } A[i] \text{ for } i = (N/2 + 1), \dots, N$

Send $Sum1$ to Processor0

If we use the notation *Send* (1, $A[N/2 + 1], N/2$), it will be equivalent to: Send to Processor1, $N/2$ elements of the array A starting from $(N/2 + 1)$ th element.

Similarly the notation *Receive* (0, $B[1], N/2$) will be equivalent to saying receive from Processor0, $N/2$ values and store them starting at $B[1]$. Using this notation, we can write programs for Processor0 and Processor1 for our example problem using a Pascal-like language as shown in Program 8.1.

Program 8.1 Two-processor addition of array

Program for Processor0

```
Array A[1:N];
begin
  for i := 1 to N do
    Read A[i]
  end for;
  Send(1, A[N/2 + 1], N/2);
  Sum0 := 0;
  for i := 1 to N/2 do
    Sum0 := sum0 + A[i]
  end for;
  Receive(1, Sum1, 1);
  Sum := Sum0 + Sum1;
  Write Sum
end.
```

Program for Processor1

```
Array B[1:N/2];
begin
  Receive(0, B[1], N/2);
  Sum1 := 0;
  for i := 1 to N/2 do
    Sum1 := Sum1 + B[i]
  end for;
  Send(0, Sum1, 1)
end.
```

We can extend the same Program to p processors as shown in Program 8.2. One point to be observed is that when the array size N is not evenly divisible by N the integer quotient $N \text{ div } p$ is allocated to Processor0, Processor1, ... up to Processor($p - 2$) and $(N \text{ div } p + N \text{ mod } p)$ is allocated to Processor($p - 1$). In the worst case Processor($p - 1$) is allocated $(N \text{ div } p + p - 1)$ array elements. If $N \gg p$ then $N \text{ div } p \gg p$ and the extra load on Processor($p - 1$) compared to the load on the other processors is not significant.

Program 8.2 Adding array using p processors

Program for Processor0

```

Array A[1: N], Sum[0 : p-1];
begin
    for i := 1 to N do
        Read A[i]
    end for;
    increment := N div p;
    last_inc := increment + N mod p;
    for j := 1 to (p-2) do
        Send (j, A[j * increment + 1], increment)
    end for;
    Send (p-1, A[(p-1) * increment + 1], last_inc);
    Sum[0] := 0;
    for i := 1 to increment do
        Sum[0] := Sum[0] + A[i]
    end for;
    for j := 1 to (p-1) do
        Receive (j, Sum[j], 1)
    end for;
    grand_sum := 0;
    for j := 0 to (p-1) do
        grand_sum := grand_sum + Sum[j]
    end for;
    Write grand_sum
end.

```

Program for Processor k , $k = 1, \dots, (p-2)$

```

Array B[1: increment];
begin
    Receive (0, B[1], increment);
    Sum[k] := 0; {Sum[k] is the partial sum of processor k}
    for i := 1 to increment do
        Sum[k] := Sum[k] + B[i]
    end for;
    Send (0, Sum[k], 1)
end.

```

Program for Processor($p-1$)

```

Array B[1: last_inc];
begin
    Receive (0, B[1], last_inc);
    Sum[p-1] := 0;
    for i := 1 to last_inc do
        Sum[p-1] := Sum[p-1] + B[i]
    end for;
    Send (0, Sum[p-1], 1)
end.

```

Single Program Multiple Data Programming—Message Passing Model

Instead of writing multiple programs as in Programs 8.1 and 8.2, programmers implement an application by writing one program, and then replicating it as many times as their parallel computer has processors. This style of writing a program is called Single Program Multiple Data (SPMD) programming in which each instance of the program works on its own data, and may follow different conditional branches or execute loops a different number of times.

An SPMD program is loosely synchronous. Between synchronization points, processors asynchronously execute the same program (instance) but manipulate their own portion of data. An SPMD program corresponding to Program 8.2 is given in Program 8.3. Note that all the processors execute the same program but follow different conditional branches based on their unique identifier, *procid*.

Program 8.3 SPMD program for addition of array

```

begin
    Sum_local, sum_rcvd, increment;
    Array A[1:N];
    if procid = 0 then
        for i := 1 to N do
            Read A[i]
        end for
    end if
    if procid ≠ p-1 then
        increment := N div p
    else
        increment := N div p + N mod p
    end if
    if procid = 0 then
        for j := 1 to (p-2) do
            Send (j, A[j*increment + 1], increment)
        end for
        Send (p-1, A[(p-1) * increment + 1], increment+N mod p)
    else
        Receive (0, A[1], increment)
    end if
    Sum_local := 0;
    for i := 1 to increment do
        Sum_local := Sum_local + A[i]
    end for
    if procid ≠ 0 then
        Send (0, Sum_local, 1)
    else
        for j := 1 to p-1 do
            Receive (j, sum_rcvd, 1);
            Sum_local := Sum_local + sum_rcvd
        end for
        Write Sum_local
    end if
end.

```

We present below an SPMD message passing program for parallel Gaussian elimination that we described in the previous chapter.

Program 8.4 SPMD program for Gaussian elimination

Gaussian Elimination (INOUT: A[1:N,1:N], INPUT: B[1:N], OUTPUT: Y[1:N]);

```

begin
  if procid = 0 then
    for i := 2 to N do
      Send (i-1, A[i, 1:N], N);
      Send (i-1, B[i], 1)
    end for
  else
    Receive (0, A[procid+1, 1:N], N);
    Receive(0, B[procid+1], 1)
  end if;
  for k := 1 to N do
    if k = procid+1 then
      for j := k+1 to N do
        A[k, j] := A[k, j]/A[k, k] {Division step}
      end for;
      Y[k] := B[k]/A[k, k];
      A[k, k] := 1
      for i := k+1 to N do
        Send(i-1, A[k, k+1:N], N-k);
        Send (i-1, Y[k], 1)
      end for
    else if procid  $\geq$  k then
      Receive (k-1, tempA[k+1:N], N-k);
      Receive (k-1, tempY, 1)
      for j := k+1 to N do
        A[procid+1,j] := A[procid+1,j]-A[procid+1,k]*tempA[j]
      end for;
      B[procid+1] := B[procid+1]-A[procid+1,k] *tempY;
                                         {Elimination step}
      A[procid+1,k]:= 0
    end if
  end for
  if procid  $\neq$  0 then
    Send (0, A[procid+1, 1:N], N);
    Send (0, Y[procid+1], 1)
  else
    for i := 1 to N do
      Receive (i-1, A[i, 1:N], N);
      Receive (i-1, Y[i], 1)
    end for
  end if
end.

```

8.2 MESSAGE PASSING PROGRAMMING WITH MPI

In this section, we show how parallel processing can be achieved using a public domain message passing system, namely MPI. It should be noted that the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

8.2.1 Message Passing Interface (MPI)

MPI, developed by the *MPI Forum* consisting of several parallel computer vendors, library writers, and application specialists, is a standard specification for a library of message passing functions. MPI specifies a public-domain, platform-independent standard of message passing library thereby achieving portability. Documentation of MPI standard is available at <http://www.mpi-forum.org>. This specification does not contain any feature that is specific to any particular hardware, operating system, or vendor. Note that MPI is a library and not a programming language. It defines a library of functions that can be called from FORTRAN 77 or FORTRAN 95 programs and C or C++ programs. The programs that users write in FORTRAN 77/95 and C/C++ are compiled with ordinary compilers and linked with the MPI library. Since MPI has been implemented on personal computers on Windows, Unix workstations, and all major parallel computers, a parallel program written in FORTRAN 77/95 or C/C++ using the MPI library could run without any change on a single personal computer, a workstation, a network of workstations, a parallel computer, from any vendor, on any operating system.

The design of MPI is based on four orthogonal concepts. They are message datatypes, communicators, communication operations, and virtual topology. We now discuss each of these concepts in detail.

Running an MPI Program

The MPI standard does not specify every aspect of a parallel program. For example, MPI does not specify how processes come into being at the beginning. This is left to implementation. MPI assumes static processes; all processes are created when a parallel program is loaded. No process can be created or terminated in the middle of program execution. All processes stay alive till the program terminates. There is a default process group consisting of all such processes, identified by MPI_COMM_WORLD.

Typical MPI routines used in programs include MPI_Init which must be the first MPI call to initialize the MPI environment, MPI_Comm_size to find out the number of processes that the user has started for the program (Precisely how the user caused these processes to be started depends on the implementation.), MPI_Comm_rank to find out the rank of each process, MPI_Send and MPI_Recv for sending and receiving a message (to and from a process), and MPI_Finalize to terminate the MPI environment.

Consider the message passing program with MPI routines as shown in Program 8.5. This is an SPMD program to compute $\sum x(i)$, $i = 0, 1, \dots, N$.

Program 8.5 An example MPI program

```

#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[])
{
    int i, x(i), tmp, sum = 0, group_size, my_rank, N;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &group_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0)
    {
        printf ("Enter N: " );
        scanf ( "%d", &N) ;
        for (i = 1; i < group_size; i++)
            MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);
        for (i= my_rank; i < N; i = i + group_size)
            sum = sum + x(i);
        for (i = 1; i < group_size; i++)
        {
            MPI_Recv(&tmp, 1, MPI_INT, i, i, MPI_COMM_WORLD, &status);
            sum = sum + tmp;
        }
        printf ("The result = %d\n", sum);
    }
    else
    {
        MPI_Recv(&N, 1, MPI_INT, 0, i, MPI_COMM_WORLD, &status);
        for (i = my_rank; i < N; i = i + group_size)
            sum = sum + x(i);
        MPI_Send(&sum, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}

```

Note that the above program includes the “mpi.h” header file to get function prototypes and macros defined by MPI. In order to run this program, stored in a “myprog.c” file, the program is first compiled using a parallel C compiler *mpicc*:

```
mpicc myprog.c -o myprog
```

The executable *myprog* is then loaded as *n* processes in nodes (processors) using one of the following commands:

```
mpirun -np n myprog
mpiexec -n n myprog
```

These *n* processes which stay alive till the end of the program form the default process group *MPI_COMM_WORLD*. For process *k* (*k* = 0, 1, ..., *n*-1), *MPI_Comm_rank* returns *k* and *MPI_Comm_size* returns *n*. Since *n* is a load-time parameter, the same program can be run on varying number of nodes.

In the above example program, we assumed that *N* > *n*. The process with rank 0 reads the input number *N* and sends it to all the other processes. Then, a process with rank *k* computes the partial sum $x(k) + x(n + k) + x(2n + k) + \dots$. All the processes other than the one with rank 0 send their resulting partial sums to process 0. Process 0 then adds these partial sums

including its share to get the total sum.

Message Passing in MPI

Processes in MPI are heavy-weighted (like Unix processes) and single threaded with separate address spaces. Since one process cannot directly access variables in another process's address space, message passing is used for interprocess communication. The routines `MPI_Send` and `MPI_Recv` are used in order to send/receive a message to/from a process. As an analogy, a message is like a letter. It has two parts namely, the contents of the message (*message buffer*—the letter itself) and the destination of the message (*message envelope*—what is on the letter/envelope). We now discuss the primitive `MPI_Send`. An example usage of `MPI_Send` is shown below:

```
MPI_Send(&N; 1, MPI_INT, i, i, MPI_COMM_WORLD)
```

The `MPI_Send` routine has six parameters. The first three parameters specify the message address, message count, and message datatype, respectively. These three parameters together constitute the message buffer. MPI introduces datatype parameter to support heterogeneous computing and to allow messages from non-contiguous memory locations. The last three parameters specify the destination process id of the message, tag, and communicator, respectively. These three parameters together constitute the message envelope. A message tag, also known as a message type, is an integer used by the programmer to label types of messages and to restrict message reception. A communicator is a *process group* with a *context*. We will now discuss the significance of each of the parameters of `MPI_Send`.

Message Buffer

The message buffer in MPI is specified by the message address, message count, and the message datatype. The message address could be any address in the sender's address space and refers to the starting address of the message buffer. Message count specifies the number of occurrences of data items of the message datatype starting at the message address. As mentioned earlier, the datatype parameter supports heterogeneous computing (which refers to running programs or applications on a system consisting of different computers, each could come from a different vendor, using different processors and operating systems) and allows messages from noncontiguous memory locations.

For example, a string of 10 characters is to be sent from one computer (machine) to another. In a conventional message passing system, the call to the send routine would appear as `send (destination, string, 10)` or `send (string, 10, destination)` which specifies that a string of 10 characters is to be sent. But, if a character is represented using two bytes (2 bytes/character) at the destination machine as opposed to the 1 byte/character representation in the sending machine, then the string consisting of 10 characters at the source would be interpreted wrongly to be a string of 5 characters at the destination. In MPI, communicating 10 characters can be realized by the calls

```
MPI_Send(string, 10, MPI_CHAR, ...), at the sending machine  
MPI_Recv(string, 10, MPI_CHAR, ...), at the receiving machine.
```

Here, 10 is no longer the message length in bytes but the occurrences of data items of the message type `MPI_CHAR`. Thus the datatype `MPI_CHAR` is implemented as one byte on the sending machine, but two bytes on the destination machine. The implementation is responsible for conversions between the 8-bit and the 16-bit formats. The basic types in MPI include all basic types in C and FORTRAN, plus two additional types, namely `MPI_BYTE` and `MPI_PACKED`. `MPI_BYTE` indicates a byte of 8 bits.

Packing

We now examine the role of the MPI_PACKED datatype in order to send non-contiguous data items. When a sequence of non-contiguous array items is to be sent, the data items can be first packed into a contiguous temporary buffer and then sent. The following program illustrates the use of packing and sending all the even indexed elements of an array A.

```
double A[100];
MPI_Pack_size(50, MPI_DOUBLE, MPI_COMM_WORLD, &BufferSize);
TempBuffer = malloc(BufferSize);
j = sizeof(MPI_DOUBLE);
Position = 0;
for (i = 0; i < 50; i++)
    MPI_Pack(A+i*j, 1, MPI_DOUBLE, TempBuffer, BufferSize, &Position,
              MPI_COMM_WORLD);
MPI_Send(TempBuffer, Position, MPI_PACKED, destination, tag, MPI_COMM_WORLD);
```

In the above program, the routine MPI_Pack_size determines the size of the buffer required to store 50 MPI_DOUBLE data items. The buffer size is next used to dynamically allocate memory for TempBuffer. The MPI_Pack routine is used to pack the even numbered elements into the TempBuffer in a **for** loop. The input to MPI_Pack is, the address of an array element to be packed, the number (1) and datatype (MPI_DOUBLE) of the items, output buffer (TempBuffer) and the size of the output buffer (BufferSize) in bytes, the current Position (to keep track of how many items have been packed) and the communicator. The final value of Position is used as the message count in the following MPI_Send routine. Note that all packed messages in MPI use the datatype MPI_PACKED.

Derived Datatypes

In order to send a message with non-contiguous data items of possibly mixed datatypes wherein a message could consist of multiple data items of different datatypes, MPI introduces the concept of *derived datatypes*. MPI provides a set of routines in order to build complex datatypes. This is illustrated in the example below.

```
double A[100];
MPI_Data_type EvenElements ;
...
MPI_Type_vector(50, 1, 2, MPI_DOUBLE, &EvenElements);
MPI_Type_commit(&EvenElements);
MPI_Send(A, 1, EvenElements, destination, ...);
...
...
MPI_Type_free(&EvenElements);
```

In order to send the even numbered elements of array A, a derived datatype EvenElements is constructed using the routine MPI_Type_vector whose format is as follows:

`MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`

The derived datatype *newtype* consists of *count* copies of blocks, each of which in turn consists of *blocklength* copies of consecutive items of the existing datatype *oldtype*. *stride* specifies the number of *oldtype* elements between two consecutive blocks. Thus *stride blocklength* is the gap between two blocks.

Thus the MPI_Type_vector routine in the above example creates a derived datatype EvenElements which consists of 50 blocks, each of which in turn consists of one double

precision number. These blocks are separated by two double precision numbers. Thus only the even numbered elements are considered, skipping the odd numbered elements using the gap. The MPI_Type_commit routine is used to commit the derived type before it is used by the MPI_Send routine. When the datatype is no longer required, it should be freed with MPI_Type_free routine.

Message Envelope

As mentioned earlier, MPI uses three parameters to specify the message envelope, the destination process id, the tag, and the communicator. The destination process id specifies the process to which the message is to be sent. The tag argument allows the programmer to deal with the arrival of messages in an orderly way, even if the arrival of messages is not in the order desired. We now study the roles of tag and communicator arguments.

Message Tags

A message tag is an integer used by the programmer to label different types of messages and to restrict message reception. Suppose two send routines and two receive routines are executed in sequence in node A and node B, respectively, as shown below without the use of tags.

Process X:

send(M, 64, Y)
send(N, 32, Y)

Process Y:

recv(P, X, 64)
recv(Q, X, 32)

Here, the intent is to first send 64 bytes of data which is to be stored in P in process Y and then send 32 bytes of data which is to be stored in Q. If N, although sent later, reaches earlier than M at process Y, then there would be an error. In order to avoid this error tags are used as shown below:

Process X:

send(M, 64, Y, tag1)
send(N, 32, Y, tag2)

Process Y:

recv(P, X, 64, tag1)
recv(Q, X, 32, tag2)

If N reaches earlier at process Y it would be buffered till the recv() routine with tag2 as tag is executed. Thus, message M with tag 1 is guaranteed to be stored in P in process Y as desired.

Message Communicators

A major problem with tags (integer values) is that they are specified by users who can make mistakes. Further, difficulties may arise in the case of libraries, written far from the user in time and space, whose messages must not be accidentally received by the user program. MPI's solution is to extend the notion of tag with a new concept—the context. Contexts are allocated at run time by the system in response to user (and library) requests and are used for matching messages. They differ from tags in that they are allocated by the system instead of the user.

As mentioned earlier, processes in MPI belong to groups. If a group contains n processes, its processes are identified within a group by ranks, which are integers from 0 to $n-1$.

The notions of context and group are combined in a single object called a communicator, which becomes an argument in most point-to-point and collective communications. Thus the destination process id or source process id in an MPI_Send or MPI_Recv routines always refers to the rank of the process in the group identified within the given communicator. Consider the following example:

Process 0:

```
MPI_Send(msg1, count1, MPI_INT, 1, tag1, comm1);
parallel_matrix_inverse();
```

Process 1:

```
MPI_Recv(msgl, count1, MPI_INT, 0, tagl, comm1);
parallel_matrix_inverse();
```

The intent here is that process0 will send msg1 to process 1, and then both execute a routine `parallel_matrix_inverse()`. Suppose `parallel_matrix_inverse()` code contains an `MPI_Send(msg2, count1, MPI_INIT, 1, tag2, comm2)` routine, If there were no communicators, the `MPI_Recv` in process 1 could mistakenly receive the msg2 sent by the `MPI_Send` in `parallel_matrix_inverse()`, when tag2 happens to be the same as tag1. This problem is solved using communicators as each communicator has a distinct system-assigned context. The communicator concept thus facilitates the development of parallel libraries, among other things.

MPI has a predefined set of communicators. For instance, `MPI_COMM_WORLD` contains the set of all processes which is defined after `MPI_Init` routine is executed. `MPI_COMM_SELF` contains only the process which uses it. MPI also facilitates user defined communicators.

`status`, the last parameter of `MPI_Recv`, returns information (source, tag, error code) on the data that was actually received. If the `MPI_Recv` routine detects an error (for example, if there is no sufficient storage at the receiving process, an overflow error occurs), it returns an error code (integer value) indicating the nature of the error.

Most MPI users only need to use routines for (point-to-point or collective) communication within a group (*called intra-communicators* in MPI). MPI also allows group-to-group communication (inter-group) through *inter-communicators*.

Point-to-Point Communications

MPI provides both blocking and non-blocking send and receive operations, and non-blocking versions whose completion can be tested for and waited for explicitly. Recall that a *blocking* send/receive operation does not return until the message buffer can be safely written/read while a *non-blocking* send/receive can return immediately. MPI also has multiple communication modes. The *standard* mode corresponds to current common practice in message passing systems. (Here the send can be either synchronous or buffered depending on the implementation.) The *synchronous* mode requires a send to block until the corresponding receive has occurred (as opposed to the standard mode blocking send which blocks only until the message is buffered). In *buffered mode* a send assumes the availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call `MPI_Buffer_attach(buffer,size)` that allocates a user buffer. This buffer can be released by `MPI_Buffer_detach(*buffer,*size)`. The *ready* mode (for a send) is a way for the programmer to notify the system that the corresponding receive has already started, so that the underlying system can use a faster protocol if it is available. Note that the send here does not have to wait as in the synchronous mode. These modes together with blocking/non-blocking give rise to eight send routines as shown in Table 8.1. There are only two receive routines. `MPI_Test` may be used to test for the completion of a receive/send stared with `MPI_Isend/MPI_Irecv`, and `MPI_Wait` may be used to wait for the completion of such a receive/send.

At this point, the reader is urged to go through Program 8.5 again, especially the MPI routines and their arguments in this program.

TABLE 8.1 Different Send/Receive Operations in MPI

<i>MPI Primitive</i>	<i>Blocking</i>	<i>Non-blocking</i>
Standard Send	<code>MPI_Send</code>	<code>MPI_Isend</code>
Synchronous Send	<code>MPI_Ssend</code>	<code>MPI_Issend</code>
Buffered Send	<code>MPI_Bsend</code>	<code>MPI_Ibsend</code>
Ready Send	<code>MPI_Rsend</code>	<code>MPI_Irsend</code>
Receive	<code>MPI_Recv</code>	<code>MPI_Irecv</code>
Completion Check	<code>MPI_Wait</code>	<code>MPI_Test</code>

Collective Communications

When all the processes in a group (communicator) participate in a *global communication* operation, the resulting communication is called a collective communication. MPI provides a number of collective communication routines. We describe below some of them assuming a communicator Comm contains n processes:

1. *Broadcast*: Using the routine `MPI_Bcast(Address, Count, Datatype, Root, Comm)`, the process ranked Root sends the same message whose content is identified by the triple (Address, Count, Datatype) to all processes (including itself) in the communicator Comm. This triple specifies both the send and the receive buffers for the Root process whereas only the receive buffer for the other processes.
2. *Gather*: The routine `MPI_Gather (Send Address, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Root, Comm)` facilitates the Root process receiving a personalized message from all the n processes (including itself). These n received messages are concatenated in rank order and stored in the receive buffer of the Root process. Here the send buffer of each process is identified by (SendAddress, SendCount, SendDatatype) while the receive buffer is ignored for all processes except the Root where it is identified by (RecvAddress, RecvCount, RecvDatatype).
3. *Scatter*: This is the opposite of gather operation. The routine `MPI_Scatter()` ensures that the Root process sends out personalized messages, which are stored in rank order in its send buffer, to all the n processes (including itself).
4. *Total Exchange*: This is also known as an all-to-all. In the routine `MPI_Alltoall()` each process sends a personalized message to every other process including itself. Note that this operation is equivalent to n gathers, each by a different process and in all n^2 messages are exchanged.
5. *Aggregation*: MPI provides two forms of aggregation—reduction and scan. The `MPI_Reduce (SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)` routine reduces the partial values stored in SendAddress of each process into a final result and stores it in RecvAddress of the Root process. The reduction operator is specified by the Op field. The `MPI_Scan(SendAddress, RecvAddress, Count, Datatype, Op, Comm)` routine combines the partial values into n final results which it stores in the RecvAddress of the n processes. Note that the root field is absent here. The scan operator is specified by the Op field. Both the scan and the reduce routines allow each process to contribute a vector, not just a scalar

value, whose length is specified in Count. MPI supports user-defined reduction/scan operations.

6. **Barrier:** This routine synchronizes all processes in the communicator Comm i.e., they wait until all n processes execute their respective MPI_Barrier(Comm) routine.

Note that in a collective communication operation, *all* processes of the communicator must call the collective communication routine. All the collective communication routines, with the exception of MPI_Barrier, employ a standard, blocking mode of point-to-point communication. A collective communication routine may be or may not be synchronous depending on the implementation. The Count and the Datatype should match on all the processes involved in the collective communication. There is no tag argument in a collective communication routine. The first five collective communication routines are illustrated in Fig. 8.1.

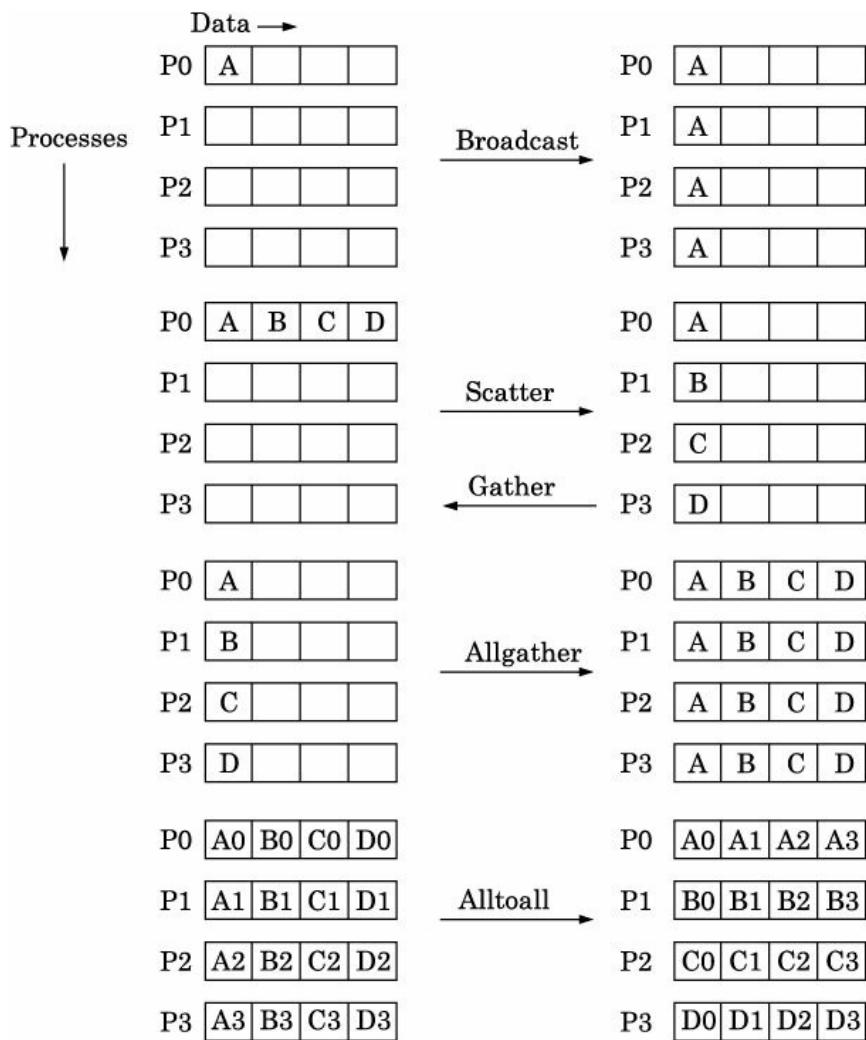


Figure 8.1 Collective communications.

Consider the problem of computing π by numerical integration. An approximate value of π can be obtained through the following integration:

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx \approx \sum_{0 \leq i < n} \frac{4}{1 + \left(\frac{i+0.5}{N}\right)^2} \times \frac{1}{N}$$

An MPI program to compute the value of π is given below (Program 8.6). It illustrates the use of collective communication routines such as broadcast and reduce.

Program 8.6 MPI program to compute π

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[])
{
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x, a;
    MPI_Init (&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if (myid == 0)
    {
        printf ("Enter the number of intervals: (0 terminates) " );
        scanf ( "%d", &n);
    }
    MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0)
        break;
    else
    {
        h = 1.0 / (double) n;
        sum= 0. 0;
        for (i = myid + 1; i<= n; i += numprocs)
        {
            x = h * ((double)i + 0.5 );
            sum += (4.0/(1.0+x*x));
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE , MPI_SUM , 0,
                   MPI_COMM_WORLD);
        if (myid == 0)
            printf ("pi is approximately %f\n", pi);
    }
    MPI_Finalize ();
}
```

In the above program, there are *numprocs* number of processes in the default communicator, MPI_COMM_WORLD. The process with rank (myid) 0 reads the value of the number of intervals (*n*) and uses MPI_Bcast routine to broadcast it to all the other processes. Now each process computes its contribution (partial sum) mypi. Then all these partial sums are added up by using the MPI_Reduce routine with operator MPI_SUM and the final result is thus stored in the process ranked 0. Process 0 prints the computed value of π .

Virtual Topologies

The description of how the processors in a parallel computer are interconnected is called the topology of the computer (or more precisely, of the interconnection network). In most parallel programs, each process communicates with only a few other processes and the pattern of communication in these processes is called an application topology (or process graph or task graph). The performance of a program depends on the way the application topology is fitted onto the physical topology of the parallel computer (see Exercises 8.14 and 8.15). MPI supports the mapping of application topologies onto virtual topologies, virtual topologies onto physical hardware topologies.

MPI allows the user to define a particular application, or virtual topology. Communication among processes then takes place within this topology with the hope that the underlying physical network topology will correspond and expedite the message transfers. An important virtual topology is the Cartesian or (one- or multi-dimensional) mesh topology as the solution of several computation-intensive problems results in Cartesian topology. MPI provides a collection of routines for defining, examining, and manipulating Cartesian topologies.

8.2.2 MPI Extensions

MPI Forum announced a revised standard, called MPI-2, in 1997. The original MPI standard, published in 1994, is renamed MPI-1. MPI-2 has added many features. Some of these are: (i) Dynamic processes (creation and deletion of processes at any time during program execution to allow more efficient use of resources and load balancing, and to support fault tolerance; note that no processes can be added during program execution in MPI-1), (ii) One-sided communication (also called remote memory access to allow a process to send a message to a destination or receive a message from a source without the other party's participation; note that in MPI-1 both sender and receiver processes must participate in a point-to-point communication and hence the communication is two-sided. In one-sided communication, a process can update either local memory with a value from another process or remote memory with a value from the calling process thereby eliminating the overhead associated with synchronizing sender and receiver.), (iii) Scalable (parallel) I/O which is absent in MPI-1, (iv) Language bindings for FORTRAN 90 and C++, and (v) External interfaces (to define routines that allow developers to layer on top of MPI, such as for debuggers and profilers).

MPI-3 standard, adopted in 2012, contains (i) Non-blocking collective communications (to permit processes in a collective to perform operations without blocking, possibly resulting in improved performance), (ii) New one-sided communications (to handle different memory models), (iii) FORTRAN 2008 bindings, and (iv) MPIT tool interface (to allow implementation to expose certain internal variables, counters, and other states to the user). Though originally designed for message passing (distributed memory) parallel computers, today MPI runs on virtually any platform—distributed memory, shared memory and hybrid systems.

8.3 SHARED MEMORY PROGRAMMING

In shared memory programming, programmers view their programs as a collection of cooperating processes accessing a common pool of shared variables. This programming style is naturally suited to shared memory parallel computers. In such a parallel computer, the processors may not have a private program or data memory. A common program and data are stored in the main memory shared by all processors. Each processor can, however, be assigned a different part of the program stored in memory to execute with data also stored in specified locations. The main program creates separate processes for each processor and allocates them along with information on the locations where data are stored for each process. Each processor computes independently. When all processors finish their assigned tasks they have to rejoin the main program. The main program will execute after all processes created by it finish. Statements are added in a programming language to enable creation of processes and for waiting for them to complete. Two statements used for this purpose are:

1. **fork:** to create a process and
2. **join:** when the invoking process needs the results of the invoked process(es) to continue.

For example, consider the following statements:

Process X;	Process Y;
:	:
fork Y;	:
:	:
join Y;	end Y;

Process X when it encounters **fork Y** invokes another Process Y. After invoking Process Y, it continues doing its work. The invoked Process Y starts executing concurrently in another processor. When Process X reaches **join Y** statement, it waits till Process Y terminates. If Y terminates earlier, then X does not have to wait and will continue after executing **join Y**. When multiple processes work concurrently and update data stored in a common memory shared by them, special care should be taken to ensure that a shared variable value is not initialized or updated independently and simultaneously by these processes. The following example illustrates this problem. Assume that $\text{Sum} := \text{Sum} + f(A) + f(B)$ is to be computed and the following program is written.

Process A;	Process B;
:	:
:	$\text{Sum} := \text{Sum} + f(A);$
fork B;	:
:	end B.
Sum := Sum + f(A);	
:	
join B;	
:	
end A.	

Suppose Process A loads Sum in its register to add $f(A)$ to it. Before the result is stored back in main memory by Process A, Process B also loads Sum in its local register to add $f(B)$ to it. Process A will have $\text{Sum} + f(A)$ and Process B will have $\text{Sum} + f(B)$ in their respective local registers. Now both Process A and B will store the result back in Sum. Depending on which process stores Sum last, the value in the main memory will be either $\text{Sum} + f(A)$ or

Sum + $f(B)$ whereas what was intended was to store $(\text{Sum} + f(A)) + f(B)$ as the result in the main memory. If Process A stores Sum + $f(A)$ first in Sum and then Process B takes this result and adds $f(B)$ to it then the answer will be correct. Thus we have to ensure that only one process updates a shared variable at a time. This is done by using a statement called **lock** <variable name>. If a process locks a variable name no other process can access it till it is unlocked by the process which locked it. This method ensures that only one process is able to update a variable at a time. The above program can thus be rewritten as:

Process A;	Process B;
:	:
fork B;	lock Sum;
:	Sum := Sum + $f(B)$;
Lock Sum;	unlock Sum;
Sum := Sum + $f(A)$;	:
Unlock Sum;	end B.
:	
join B;	
:	
end A.	

In the above case whichever process reaches **lock** Sum statement first will lock the variable Sum disallowing any other process from accessing it. It will **unlock** Sum after updating it. Any other process wanting to update Sum will now have access to it. The process will **lock** Sum and then update it. Thus updating a shared variable is serialized. This ensures that the correct value is stored in the shared variable.

We illustrate the programming style appropriate for shared memory parallel computer using as an example adding the components of an array. Program 8.7 has been written assuming that the parallel computer has 2 processors. Observe the use of **fork**, **join**, **lock**, and **unlock**. In this program the statement:

fork(1) Add_array(input: A[N/2 + 1:N], output: Sum)

states that the Procedure Add_array is to be executed on Processor1 with the specified input parameters. Following this statement the procedure Add_array is invoked again and this statement is executed in Processor0 (the main processor in the system).

Program 8.7 Adding an array with 2-processor shared memory computer

{Main program for the 2-processor computer [Processor0 (main), Processor1 (slave)]}

```

begin {main}
  global Sum, Array A[1:N];
  for i := 1 to N do
    Read A[i]
  end for;

```

```

Sum := 0;
{The following process is executed by Processor1}
fork(1)Add_array(input: A[N/2 + 1:N], output: Sum);
{The following process is executed by Processor0}
Add_array(input: A[1:N/2], output: Sum);
join 1; { The processors wait till both complete}
Write Sum
end. {main}

```

Procedure invoked by the main program

```

Add_array(input: A[P:M], output: Sum);
begin {Add_array}
  Sum_local := 0;
  for i := P to M do
    Sum_local := Sum_local + A[i]
  end for;
  lock Sum;
  Sum := Sum + Sum_local;
  Unlock Sum
end; {Add_array}

```

The above program may be extended easily to p processors as shown in Program 8.8.

Program 8.8 Adding an array with p-processor shared memory computer

{Procedure for main Processor, Processor0}

```

begin { main }
  global Sum, Array A[1:N];
  for i := 1 to N do
    Read A[i]
  end for;
  Sum := 0;
  increment := N div p;
  for j := 1 to (p-2) do
    fork(j) Add_array(input: A[j *increment+ 1: (j + 1) *increment],
      output: Sum)
  end for;
  fork(p-1) Add_array(input: A[(p-1) *increment+ 1 : N], output: Sum);
  Add_array(input: A[1:increment], output: Sum);
  for i := 1 to (p-1) do
    join i
  end for;
  Write Sum
end. {main}

```

```

Add_array(input: A[P:M], output: Sum);
begin { Add_array }
  Sum_local := 0;
  for i := P to M do
    Sum_local := Sum_local + A[i];
  end for;
  lock Sum;
  Sum := Sum + Sum_local;
  unlock Sum
end; { Add_array }

```

Single Program Multiple Data Programming—Shared Memory Model

The SPMD programming style is not limited to message passing parallel computers. Program 8.9 shows how SPMD could be used on a shared memory parallel computer. Here synchronization is achieved by using the *barrier* primitive. Each process waits at the barrier for every other process. (After synchronization, all processes proceed with their execution.)

Program 8.9 SPMD program for addition of array

```

begin
  global Sum, increment, Array A[1:N];
  if procid = 0 then
    for i := 1 to N do
      Read A[i]
    end for;
    Sum := 0;
    increment := N div p
  end if
  if procid ≠ p-1 then
    Add_array(input: A[procid*increment+1:(procid+1)*increment],
              output: Sum)
  else
    Add_array(input: A[(p-1) * increment+1: N],output: Sum)
  end if
  barrier;
  if procid = 0 then
    Write sum
  end if
end.
Add_array(input: A[P:M], output: Sum);
begin
  local Sum_local;
  Sum_local := 0;
  for i := P to M do
    Sum_local := Sum_local + A[i]

  end for
  lock Sum;
  Sum:= Sum + Sum_local;
  unlock Sum
end;

```

We present below an SPMD shared memory program (Program 8.10) for the parallel Gaussian elimination.

Program 8.10 SPMD program for Gaussian elimination

Gaussian Elimination (INOUT: A[l:N,l:N], INPUT: B[l:N], OUTPUT:Y[l:N]);

```
begin
    for k := 1 to N do
        if procid >= k then
            A[k, procid+1] := A[k, procid+1]/A[k, k] {Division step}
        end if;
        if procid = 0 then
            Y[k] := B[k]/A[k, k];
            A[k, k] := 1
        end if;
        barrier;
        if procid >= k then
            for j := k+1 to N do
                A[procid+1, j] := A[procid+1, j]-A[procid+1, k]*A[k, j]
            end for;
            B[procid+1] := B[procid+1]-A[procid+1, k]*Y[k];
                            {Elimination step}
            A[procid+1, k] := 0
        end if
    end for
end.
```

8.4 SHARED MEMORY PROGRAMMING WITH OpenMP

In this section, we present a portable standard, namely, OpenMP for programming shared memory parallel computers. Like MPI, OpenMP is also an explicit (not automatic) programming model, providing programmers full control over parallelization. A key difference between MPI and OpenMP is the approach to exploiting parallelism in an application or in an existing sequential program. MPI requires a programmer to parallelize (convert) the entire application immediately. OpenMP, on the other hand, allows the programmer to *incrementally* parallelize the application without major restructuring.

OpenMP (Open specifications for Multi-Processing) is an API (Application Programmer Interface) for programming in FORTRAN and C/C++ on shared memory parallel computers. The API comprises of three distinct components, namely compiler directives, runtime library routines, and environment variables, for programming shared memory parallel computers, and is supported by a variety of shared memory architectures and operating systems. Programmers or application developers decide how to use these components.

The OpenMP standard designed in 1997 continues to evolve and the OpenMP Architecture Review Board (ARB) that owns the OpenMP API continues to add new features and/or constructs to OpenMP in each version. Currently most compliers support OpenMP Version 2.5 released in May 2005. The latest OpenMP Version 4.0 was released in July 2013. The OpenMP documents are available at <http://www.openmp.org>.

8.4.1 OpenMP

OpenMP uses the fork-join, multi-threading, shared address model of parallel execution and the central entity in an OpenMP program is a thread and not a process. An OpenMP program typically creates multiple cooperating threads which run on multiple processors or cores. The execution of the program begins with a single thread called the *master thread* which executes the program sequentially until it encounters the first *parallel* construct. At the parallel construct the master thread creates a *team* of threads consisting of certain number of new threads called *slaves* and the master itself. Note that the fork operation is implicit. The program code inside the parallel construct is called a *parallel region* and is executed in parallel typically using SPMD mode by the team of threads. At the end of the parallel region there is an implicit barrier synchronization (join operation) and only the master thread continues to execute further. Threads communicate by sharing variables. OpenMP offers several synchronization constructs for the coordination of threads within a parallel region. Figure 8.2 shows the OpenMP execution model. Master thread spawns a team of threads as required. Notice that as parallelism is added incrementally until the performance goals are realized, the sequential program evolves into a parallel program.

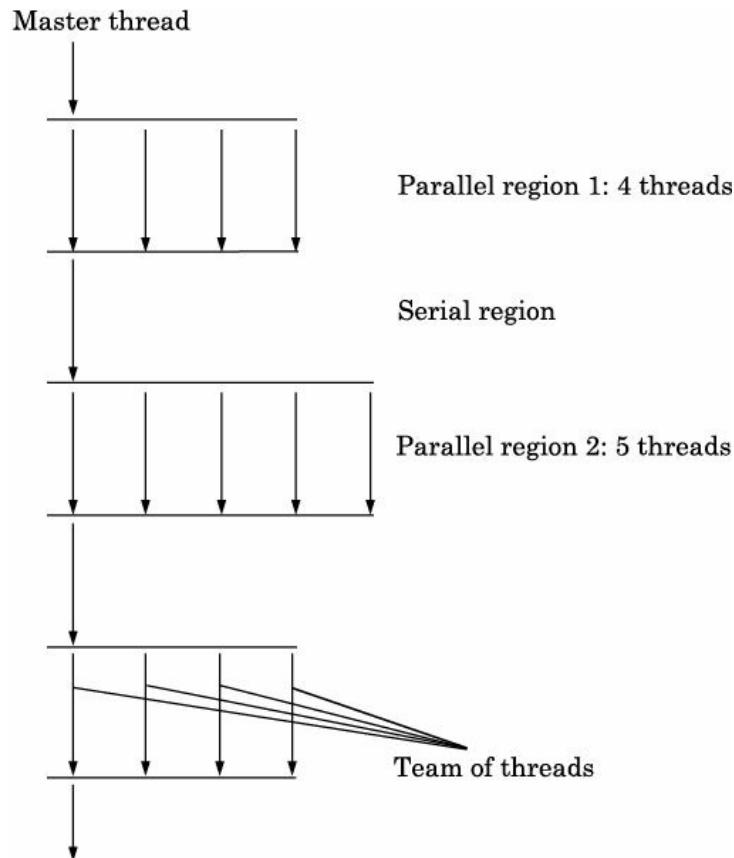


Figure 8.2 OpenMP execution model.

Running an OpenMP Program

Consider the simple program, stored in “myprog.c” file, as shown in Program 8.11. This is a multi-threaded program where each thread identifies itself by its ID (or rank) and the master thread prints the number of threads created or present in the parallel region. In order to compile this program with gcc and then run it, we type

```
gcc -fopenmp myprog.c -o myprog
./myprog
```

Program 8.11 A simple OpenMP program

```
#include <stdio.h>
#include <omp.h>
int main (void)
{
    int noOfThreads, threadId, THREAD_COUNT = 4;
    #pragma omp parallel private(threadId) num_threads(THREAD_COUNT)
    /* fork */
```

```

    { /* threadId contains thread number or ID */
threadId = omp_get_thread_num();
printf("Hi, I am thread %d\n", threadId);
#pragma omp barrier
/* master thread executes the following if statement */
if (threadId == 0) {
noOfThreads = omp_get_num_threads();
printf("Master: My team has %d threads.\n", noOfThreads);
}
} /* join */
return 0;
}

```

The output is as follows:

```

Hi, I am thread 1
Hi, I am thread 2
Hi, I am thread 3
Hi, I am thread 0
Master: My team has 4 threads.

```

Parallelism in OpenMP is controlled by compiler directives. In C/C++, these are specified by `#pragma omp` string (called *sentinel*). The syntax of the most important compiler directive, parallel construct, is as follows:

```

#pragma omp parallel [clause [clause] ...]
{ structured block }

```

It specifies that the structured block of code (a block of one or more statements with one point of entry at the top and one point of exit at the bottom), called parallel region, should be executed in parallel. A team of threads (in this program, four) is created to execute the parallel region in parallel (typically using SPMD mode). The clauses, which are used to influence the behaviour of a directive, are optional. If `num_threads` (`THREAD_COUNT`) clause is omitted, then the number of threads created depends on either a default number or the number of processors/cores available. Each thread carries a unique number or ID, starting from zero for the master thread up to the number of threads minus one. The library functions, `omp_get_thread_num()` and `omp_get_num_threads()`, return ID (rank) for the thread and number of threads in the team respectively. It should be noted that the threads are competing for access to `stdout`, and (the first four lines in) the output shown above might also be as follows (or any other permutation of the thread IDs):

```

Hi, I am thread 3
Hi, I am thread 1
Hi, I am thread 2
Hi, I am thread 0

```

Master: My team has 4 threads.

`#pragma omp barrier` synchronizes the threads. That is every thread waits until all the other threads arrive at the barrier.

Scope of a Variable

The variables that are declared in the main function (in this program, `noOfThreads`, `threadId`, `THREAD_COUNT`) or those declared before the parallel directive are shared by all the threads in the team started by the parallel directive. Thus they have *shared* scope, while the

variables declared inside the structured block (for example, local variables in functions) have *private* scope. In this program, the scope for the variable `threadId` is changed from shared to private using the *private* clause so that each thread, having its own private copy of `threadId`, can be uniquely identified in the team. OpenMP provides `private(list-of-variables)` and `shared(list-of-variables)` clauses to declare variables in their list to be private to each thread and shared among all the threads in the team respectively.

Parallel Loops

The OpenMP “parallel for” directive (“#pragma omp parallel for”) creates a team of threads to execute the following structured block, which must be a *for* loop. The loop is parallelized by dividing the iterations of the loop among the threads, that is, the iterations of the loop are assigned to the threads of the parallel region and are executed in parallel. Consider the program as shown in Program 8.12 and the corresponding output.

Program 8.12 An OpenMP program with parallel for directive

```
#include <stdio.h>
#include <omp.h>
#define N 7
int main(void)
{
    int i, THREAD_COUNT = 3;
    #pragma omp parallel for num_threads(THREAD_COUNT)
    for (i = 0; i < N; i++)
        printf("ThreadID %d is executing loop iteration %d\n", omp_get_
        thread_num(), i);
    return 0;
}
```

Output:

```
ThreadID 0 is executing loop iteration 0
ThreadID 0 is executing loop iteration 1
ThreadID 0 is executing loop iteration 2
ThreadID 2 is executing loop iteration 6
ThreadID 1 is executing loop iteration 3
ThreadID 1 is executing loop iteration 4
ThreadID 1 is executing loop iteration 5
```

Note that the scope of the for loop index variable *i* in the parallel region is private by default. We observe from the output that out of the three threads, thread numbers 0 and 1 are working on three iterations each and thread number 2 is working on only one iteration. We will shortly see how to distribute the work of the parallel region among the threads of the team.

It is possible to incrementally parallelize a sequential program that has many *for* loops by successively using the OpenMP parallel for directive for each of the loops. However, this directive has the following restrictions: (i) the total number of iterations is known in advance and (ii) the iterations of the for loop must be independent of each other. This implies that the following for loop cannot be correctly parallelized by OpenMP parallel for directive.

```
for (i = 2; i < 7; i++)
{
    a[i] = a[i] + c; /* statement 1 */
```

```

    b[i] = a[i-1] + b[i]; /* statement 2 */
}

```

Here, when $i = 3$, statement 2 reads the value of $a[2]$ written by statement 1 in the previous iteration. This is called *loop-carried dependence*. Note that there can be dependences among the statements within a single iteration of the loop but not dependences across iterations because OpenMP compilers do not check for dependences among iterations, with *parallel for* directive. It is the responsibility of the programmers to identify loop-carried dependences (by looking for variables that are read/written in one iteration and written in another iteration) in *for* loop and eliminate them by restructuring the loop body. If it is impossible to eliminate them, then the loop is not amenable to parallelization. OpenMP does not parallelize *while* loops and *do-while* loops and it is the responsibility of the programmers to convert them into equivalent *for* loops for possible parallelization.

OpenMP *for* directive, unlike the *parallel for* directive, does not create a team of threads. It partitions the iterations in *for* loop among the threads in an existing team. The following two code fragments are equivalent:

```

#pragma omp parallel
#pragma omp for
  for ( ... )
{
  ...
}

```

```

#pragma omp parallel for
  for ( ... )
{
  ...
}

```

Loop Scheduling

OpenMP supports different scheduling strategies, specified by the *schedule* parameters, for distribution or mapping of loop iterations to threads. It should be noted that a good mapping of iterations to threads can have a very significant effect on the performance. The simplest possible is static scheduling, specified by *schedule(static, chunk_size)*, which assigns blocks of size *chunk_size* in a round-robin fashion to the threads. See program 8.13. The iterations are assigned as: <Thread 0: 0,1,2,9>, <Thread 1: 3,4,5>, <Thread 2: 6,7,8>.

If the amount of work (computation) per loop iteration is not constant (for example, Thread 0 gets vastly different workload), static scheduling leads to load imbalance. A possible solution to this problem is to use dynamic scheduling, specified by *schedule(dynamic, chunk_size)*, which assigns a new block of size *chunk_size* to a thread as soon as the thread has completed the computation of the previously assigned block. However, there is an overhead (bookkeeping in order to know which thread is to compute the next block/chunk) associated with dynamic scheduling. Guided scheduling, specified by *schedule(guided, chunk_size)*, is similar to dynamic scheduling but the *chunk_size* is relative to the number of iterations left. For example, if there are 100 iterations and 2 threads, the chunk sizes could be 50,25,12,6,3,2,1,1. The *chunk_size* is approximately the number of iterations left divided by the number of threads. Another scheduling strategy supported by the OpenMP is *schedule(auto)* where the mapping decision is delegated to the compiler and/or runtime system.

Program 8.13 An OpenMP program with schedule clause

```

#include <stdio.h>
#include <omp.h>
int main(void)

```

```

{
    int i, N = 10, THREAD_COUNT = 3, CHUNK_SIZE = 3;
    #pragma omp parallel for num_threads(THREAD_COUNT) schedule
                                (static, CHUNK_SIZE)
        for (i = 0; i < N; i++)
            printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);
    return 0;
}

```

Output:

```

ThreadId: 0, iteration: 0
ThreadId: 0, iteration: 1
ThreadId: 0, iteration: 2
ThreadId: 0, iteration: 9
ThreadId: 2, iteration: 6
ThreadId: 2, iteration: 7
ThreadId: 2, iteration: 8
ThreadId: 1, iteration: 3
ThreadId: 1, iteration: 4
ThreadId: 1, iteration: 5

```

Non-Iterative Constructs

OpenMP *for* construct shares iterations of a loop across the team representing “data parallelism”. OpenMP *sections* construct, which breaks work into separate independent sections that can be executed in parallel by different threads, can be used to implement “task or functional parallelism”.

```

#pragma omp sections
{
    #pragma omp section
    { structured block }

    #pragma omp section
    { structured block }
    .
    .
    .
}

```

As stated above structured blocks are independent of each other and are executed in parallel by different threads. There is an implicit synchronization at the end of the construct. If there are more sections than threads, it is up to the implementation to decide how the extra sections are executed.

Synchronization

The OpenMP offers several high-level (such as critical, atomic and barrier) and low-level (such as locks and flush) synchronization constructs to protect critical sections or avoid race conditions as multiple threads in a parallel region access the same shared data.

1. #pragma omp critical
{ structured block }

The *critical* construct guarantees that the structured block (critical section) is executed by only one thread at a time.

2. #pragma omp atomic statement_expression

The *atomic* construct ensures that a single line of code (only one statement, for example $x = x + 1$) is executed by only one thread at a time. While the critical construct protects all the elements of an array variable, atomic construct can be used to enforce exclusive access to one element of the array.

3. #pragma omp barrier

The *barrier* construct is used to synchronize the threads at a specific point of execution. That is every thread waits until all the other threads arrive at the barrier.

4. *nowait*, for example in ‘#pragma omp parallel for nowait’, eliminates the implicit synchronization that occurs by default at the end of a parallel section.
5. OpenMP provides several lock library functions for ensuring mutual exclusion between threads in critical sections. For example, the functions `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, initialize a lock variable, set the lock, unset or release the lock respectively.

6. #pragma omp flush [(list)]

Different threads can have different values for the same shared variable (for example, value in register). To produce a consistent view of memory, *flush* construct updates (writes back to memory at this synchronization point) all variables given in the list so that the other threads see the most recent values. If no list is specified, then all thread visible variables are updated. OpenMP uses a relaxed memory consistency model.

7. Reduction (operator: list) reduces list of variables into one, using operator. It provides a neat way to combine private copies of a variable into a single result at the end of a parallel region. Note that (i) without changing any code and by adding one simple line, the sequential code fragment shown below gets parallelized, (ii) with *reduction* clause, a private copy for the variable sum is created for each thread, initialized and at the end all the private copies are reduced into one using the addition operator, and (iii) no synchronization construct such as *critical* construct is required to combine the private copies of the variable sum.

Sequential code fragment:	Parallelized code fragment:
<pre>int A[MAX], i, sum = 0; double average; for (i = 0; i < MAX; i++) { sum = sum + A[i]; } average = sum/MAX;</pre>	<pre>int average, A[MAX], i, sum = 0; double average; #pragma omp parallel for reduction (+:sum) for (i = 0; i < MAX; i++) { sum = sum + A[i] } average = sum/MAX;</pre>

In Section 8.2, we provided an MPI parallel program (Program 8.6) to compute the value of π . Sequential and OpenMP π programs are given below (Program 8.14 and Program 8.15). Observe that without making any changes to the sequential program and by just inserting two lines into it, the sequential program is converted to an OpenMP parallel program.

Program 8.14 A sequential program to compute π

```

#include <stdio.h>
int main (void)
{
    static long n = 100000; double h;
    int i; double x, pi, sum = 0.0;
    h = 1.0/(double) n;
    for (i = 0; i < n; i++) {
        x = h * ((double) i + 0.5));
        sum += (4.0/(1.0+x*x));
    }
    pi = h * sum;
    return 0;
}

```

Program 8.15 An OpenMP program to compute π

```

#include <stdio.h>
#include <omp.h>
int main (void)
{
    static long n = 100000; double h;
    int i; double x, pi, sum = 0.0;
    h = 1.0/(double) n;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i = 0; i < n; i++) {
        x = h * ((double) i + 0.5));
        sum += (4.0/(1.0+x*x));
    }
    pi = h * sum;
    return 0;
}

```

Environment Variables

OpenMP provides several environment variables that can be used to control the parallel execution environment. We list a few of them here.

- (i) OMP_SCHEDULE: It alters the behaviour of the schedule clause (mapping of iterations to threads) when schedule(runtime) clause is specified in *for* or *parallel for* directive.
- (ii) OMP_NUM_THREADS: It sets the maximum number of threads to use in a parallel region during execution.
- (iii) OMP_DYNAMIC: It specifies whether the runtime can adjust the number of threads in a parallel region.
- (iv) OMP_STACKSIZE: It controls the size of the stack for slaves (non-master threads).

In this section, we presented some of the most important directives, clauses and functions of the OpenMP standard for exploiting thread-level parallelism. We also presented how to start multiple threads, how to parallelize *for* loops and schedule them, and also how to coordinate and synchronize threads to protect critical sections. Another important directive, added in the Version 3.0 of OpenMP, is the *task* directive which can be used to parallelize recursive functions and *while* loops. Also the *collapse* clause added in this version enables parallelization of nested loops. The most recent OpenMP Version 4.0 released in July 2013 has SIMD constructs to support SIMD or vector-level parallelism to exploit full potential of

today's multicore architectures.

8.5 HETEROGENEOUS PROGRAMMING WITH CUDA AND OpenCL

8.5.1 CUDA (Compute Unified Device Architecture)

CUDA, a parallel programming model and software environment developed by NVIDIA, enables programmers to write scalable parallel programs using a straightforward extension of C/C++ and FORTRAN languages for NVIDIA's GPUs. It follows the data-parallel model of computation and eliminates the need of using graphics APIs, such as OpenGL and DirectX, for computing applications. Thus CUDA, to harness the processing power of GPUs, enables programmers for general purpose processing (not exclusively graphics) on GPUs (GPGPUs) without mastering graphic terms and without going into the details of transforming mathematical computations into equivalent pixel manipulations.

As mentioned in Chapter 5, a GPU consists of an array of Streaming Multi-processors (SMs). Each SM further consists of a number of Streaming Processors (SPs). The threads of a parallel program run on different SPs with each SP having a local memory associated with it. Also, all the SPs in an SM communicate via a common shared memory provided by the SM. Different SMs access and share data among them by means of GPU DRAM, different from the CPU DRAM, which functions as the GPU main memory, called the *global memory*. Note that CPU (having a smaller number of very powerful cores, aimed at minimizing the latency or the time taken to complete a task/thread) and GPU (having thousands of less powerful cores, aimed at maximizing the throughput, that is, number of parallel tasks/threads performed per unit time) are two separate processors with separate memories.

In CUDA terminology, CPU is called the *host* and GPU, where the parallel computation of tasks is done by a set of threads running in parallel, the *device*. A CUDA program consists of code that is executed on the host as well as code that is executed in the device, thus enabling *heterogeneous computing* (a mixed usage of both CPU and GPU computing). The execution of a CUDA device program (parallel code or “kernel”) involves the following steps: (i) (device) global memory for the data required by the device is allocated, (ii) input data required by the program is copied from the host memory to the device memory, (iii) the program is then loaded and executed on the device, (iv) the results produced by the device are copied from the device memory to the host memory, and (v) finally, the memory on the device is deallocated.

CUDA Threads, Thread Blocks, Grid, Kernel

The parts of the program that can be executed in parallel are defined using special functions called *kernels*, which are invoked by the CPU and run on the GPU. A kernel launches a large number of *threads* that execute the same code on different SPs (Streaming Processors) thus exploiting data parallelism. A thread is the smallest execution unit of a program. From a thread's perspective, a kernel is a sequential code written in C. The set of all threads launched by a kernel is called a *grid*. The kernel grid is divided into *thread blocks* with all thread blocks having an equal number of threads (See Fig. 8.3). Each thread block has a maximum number of threads it can support. Newer GPUs support 1024 threads per block while the older ones can only support 512. A maximum of 8 thread blocks can be executed simultaneously on an SM. Also, the maximum number of threads that can be assigned to an SM is 1536. Note that each SM can execute one or more thread blocks, however, a thread block cannot run on multiple SMs.

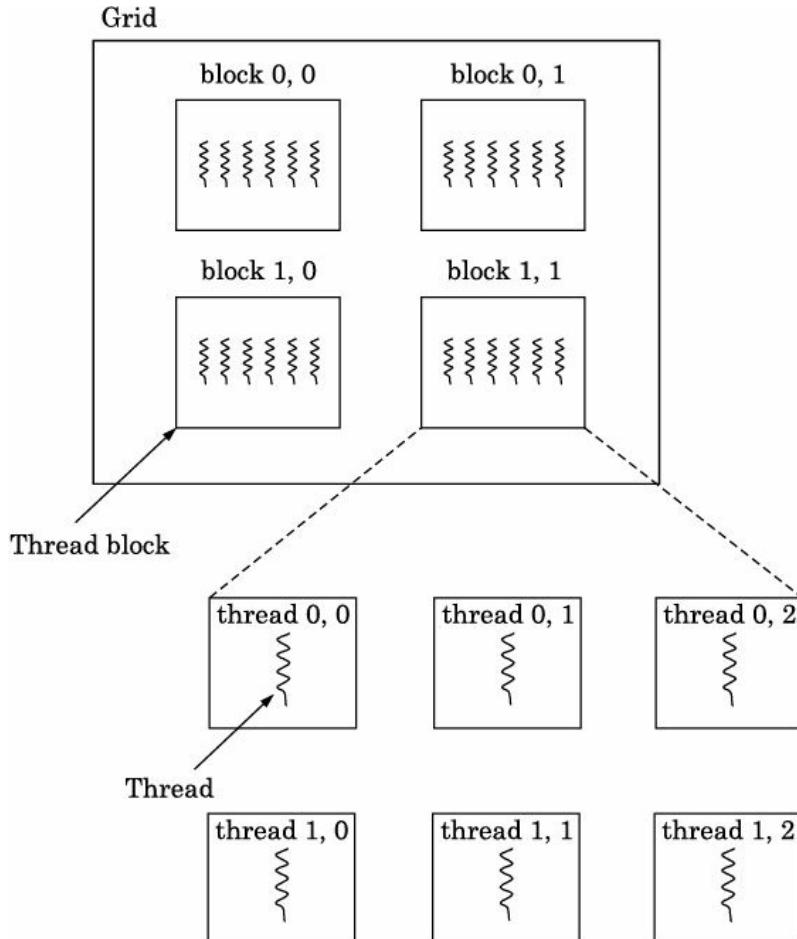


Figure 8.3 2D grid is 2×2 thread blocks; each 2D thread block is 2×3 threads.

The hierarchical grouping of threads into thread blocks that form a grid has many advantages. First, different thread blocks operate independent of each other. Hence, thread blocks can be scheduled in any order relative to each other (in serial or in parallel) on the SM cores. This makes CUDA architecture scalable across multiple GPUs having different number of cores. Second, the threads belonging to the same thread block are scheduled on the same SM concurrently. These threads can communicate with each other by means of the shared memory of the SM. Thus, only the threads belonging to the same thread block need to synchronize at the barrier. Different thread blocks execute independently and hence no synchronization is needed between them. In short, the grid is a set of loosely coupled thread blocks (expressing coarse-grained data parallelism) and a thread block is a set of tightly coupled threads (expressing fine-grained data/thread parallelism).

Once the execution of one kernel gets completed, the CPU invokes another kernel for execution on the GPU. In newer GPUs (for example, Kepler), multiple kernels can be executed simultaneously (expressing task parallelism). CUDA supports three special keywords for function declaration. These keywords are added before the normal function declaration and differentiate normal functions from kernel functions. They include:

global keyword indicates that the function being declared is a kernel function which is to be called only from the host and gets executed on the device.

device keyword indicates that the function being declared is to be executed on the device and can only be called from a device.

host keyword indicates that the function being declared is to be executed on the host.

This is a traditional C function. This keyword can be omitted since the default type for each function is `_host_`.

Let `functionDemo()` be a device kernel. Its definition can be given as:

```
_global_ functionDemo(int param1, int param2) {
    ...
    ...
}
```

The keyword `_global_` indicates that this is a kernel function which will be invoked by the host and executed on the device. It takes two integer arguments `param1` and `param2`.

Whenever a kernel is launched, the number of threads to be generated by the kernel is specified by means of its *execution configuration* parameters. These parameters include *grid dimension* defined as the number of thread blocks in the grid launched by the kernel and *block dimension* defined as the number of threads in each thread block. Their values are stored in the predefined variables `gridDim` and `blockDim` respectively. They are initialized by the execution configuration parameters of the kernel functions. These parameters are specified in the *function call* between `<<<` and `>>>` after the function name and before the function arguments as shown in the following example.

```
functionDemo<<<32, 16>>>(param1, param2);
```

Here the function `functionDemo()` will invoke a total of 512 threads in a group of 32 thread blocks where each thread block consists of 16 threads.

CUDA has certain built-in variables with pre-initialized values. These variables can be accessed directly in the kernel functions. Two such important variables are `threadIdx` and `blockIdx` representing the thread index in a block and the block index in a grid respectively. These keywords are used to identify and distinguish threads from each other. This is necessary since all the threads launched by a kernel need to operate on different data. Thus by associating a unique thread identifier with each thread, the thread specific data is fetched. `gridDim` and `blockDim` are also the pre-initialized CUDA variables. A thread is uniquely identified by its `threadIdx`, `blockIdx` and `blockDim` values as:

```
id = blockIdx.x * blockDim.x + threadIdx.x
```

Here `.x` represents that the threads and blocks are one dimensional only.

Memory Operations

As mentioned earlier, GPU (device) does not have access to the system (host) RAM and has its own device memory. Hence, before executing a kernel on the device, memory has to be allocated on the device global memory and necessary data required by the kernel is to be transferred from the host memory to the device memory. After the kernel execution, the results available in the global memory have to be transferred back to the host memory. All these are achieved by means of the following APIs provided with CUDA and are used in the host code.

`cudaMalloc((void)A, size)`:** This API function is used to allocate a chunk of device global memory for the data objects to be used by the kernel. It is similar to the `malloc()` function of standard C programming language. The parameter `A` is the address of the pointer variable that points to the address of the allocated device memory. The function `cudaMalloc()` expects a generic pointer, hence the address of the pointer variable `A` should be cast to `(void**)`. Second parameter, `size`, is the allocation size in bytes reserved for the data object pointed to by `A`.

cudaFree(A): This API function is used to free the device global memory allocated previously by cudaMalloc().

cudaMemcpy(A, B, size, direction): This API function is used to transfer or copy the data from host to device memory and vice-versa. It is used after the memory is allocated for the object in the device memory using cudaMalloc(). First parameter, A, to the cudaMemcpy() function is a pointer to the destination location where data object has to be copied. Second parameter, B, is again a pointer to the data source location. Third parameter, size, represents the number of bytes to be transferred. The direction of memory operation involved is represented by the fourth parameter, *direction*, that is, host to device, device to host, host to host and device to device. For data transfer from host to device memory, the direction parameter would be cudaMemcpyHostToDevice. Similarly, the parameter value for data transfer from device to host memory is cudaMemcpyDeviceToHost. Here cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are predefined constants of the CUDA environment.

Array Addition Example

We now present a simple CUDA program to add two given arrays or vectors and explain the difference between sequential and parallel programming. Consider a function that takes four parameters as input. First two arguments are the addresses of the integer array elements whose sum is to be computed. Third argument is the address where the resulting sum should be stored. Fourth argument is the number of elements in the arrays to be added (assuming equal size of both the arrays). We will first write the sequential function for the problem and then add the relevant statements that make the function parallel thereby enabling it for GPU execution.

Sequential CPU Execution

Under sequential execution, the function is as given below. The function addArray() takes the integer arrays A and B, computes the sum and stores the result in C.

```
1. void addArray(int *A, int *B, int *C, int count)
2. {
3.     for (int i = 0; i < count; i++)
4.         C[i] = A[i] + B[i];
5. }
```

Parallel GPU Execution

Under parallel execution in the GPU, the function is as given below.

```
1. void addArray(int *A, int *B, int *C, int count)
2. {
3.     int allocationSize = count * sizeof(int);
4.     int *deviceA, *deviceB, *deviceC;
5.     cudaMalloc((void**)&deviceA, allocationSize);
6.     cudaMemcpy(deviceA, A, allocationSize, cudaMemcpyHostToDevice);
7.     cudaMalloc((void**)&deviceB, allocationSize);
8.     cudaMemcpy(deviceB, B, allocationSize, cudaMemcpyHostToDevice);
9.     cudaMalloc((void**)&deviceC, allocationSize);
10.    addArrayKernel<<<ceil(count/128), 128>>>
11.    (deviceA, deviceB, deviceC, count);
12.    cudaMemcpy(C, deviceC, allocationSize, cudaMemcpyDeviceToHost);
```

```

12. cudaFree(deviceA);
13. cudaFree(deviceB);
14. cudaFree(deviceC);
15. }

```

In the function addArray(), statements 5,7 and 9 allocate memory for the array elements using cudaMalloc() API. The pointers deviceA, deviceB and deviceC point to the addresses of the arrays involved in computation in the device global memory. Arrays A and B are transferred to the device memory using the cudaMemcpy() API in the statements 6 and 8. Statement 10 launches the kernel function. Understanding the way to choose the execution configuration parameters is very important. The way GPU groups the threads depends upon the kernel configuration parameters only. As explained earlier, here the first configuration parameter specifies the number of thread blocks to be launched by the kernel while the second parameter specifies the number of threads within each thread block.

In our example, we want array elements of each index to be handled by a separate thread. This requires the total number of threads to be equal to the value of count, the fourth parameter to the function addArray(). Out of the count array elements (or threads), let us consider that each thread block is made up of 128 threads. This is specified by the second configuration parameter. This means that 128 threads of a given thread block will get executed by the SPs of the same SM. The total number of thread blocks needed to hold a total of count threads would be equal to $\text{ceil}(\text{count}/128)$, if count is an integer multiple of 128. However, this might not be the case always, thus we use $\text{ceil}(\text{count}/128)$. Statement 11 copies the result of addition from the device global memory present in the array deviceC to the host memory array C. Statements 12, 13 and 14 free the memory allocated in the device.

The operation of kernel function, addArrayKernel(), is given below:

```

1. __global__ void addArrayKernel(int *A, int *B, int *C, int count)
2. {
3.     int id = blockIdx.x * blockDim.x + threadIdx.x;
4.     if (id < count)
5.         C[id] = A[id] + B[id];
6. }

```

Statement 3 computes the identifier associated with the thread under execution by using the built-in variables blockIdx, threadIdx and blockDim. This is used as an index into the array. Each thread manipulates the array element whose index is equal to the thread identifier as given by the statement 5. Since the number of array elements might not be multiple of 128, the number of threads created would be more than the actual number of array elements. Thus, the check in the conditional statement 4 is necessary.

Grid and Block Dimensions

The kernel grids and the thread blocks are three dimensional variables of the type dim3. A grid can be viewed as a 3D array of thread blocks. Similarly, a thread block is composed of a 3D array of threads. It is upto the programmer to use all the dimensions or keep higher dimensions unused by setting the corresponding dimension parameters to 1.

The data type dim3 is a C struct which is made up of the three unsigned integer fields x, y and z. The number of thread blocks that can be present in each dimension of a grid is 65,536. Thus, the values of each of the grid variables gridDim.x, gridDim.y and gridDim.z range from 1 to 65,536.

In case of a thread block, the value of blockIdx.x ranges from 0 to gridDim.x–1. It is true for blockIdx.y and blockIdx.z variables also. Only 1024 threads can be accommodated within

a thread block (including all the dimensions). They can be grouped as (512,2,1), (128,2,2) or (64,4,2) or any other valid combination as long as they do not exceed the total thread count of 1024.

The dim3 variables are declared and used as kernel execution configuration parameters as shown in the following example:

1. `dim3 numBlocks(8,4,2);`
2. `dim3 numThreads(64,2,1);`
3. `funcDemo<<<numBlocks, numThreads>>>(param1, param2);`

The statement 1 defines the dimensions of the kernel grid in terms of the number of thread blocks. There are 64 thread blocks in this grid which can be visualized as a 3D array with 8 unit blocks in x-direction, 4 unit blocks in y-direction and 2 unit blocks in z-direction. Similarly, the statement 2 gives the structure of a thread block. In this example, each thread block is two dimensional with 64 unit threads in x-direction and 2 unit threads in y-direction, thus having 128 threads. In the statement 3, the variables numBlocks and numThreads are used as configuration parameters. The kernel functionDemo() invokes a total of $64 \times 128 = 8192$ threads.

In case a kernel consists of single dimensional threads and blocks, then the variables numBlocks and numThreads can be replaced by the corresponding integer value arguments, as is done in the previous example. Note that the configuration parameters numBlocks and numThreads can be accessed inside the kernel function funcDemo() by using the predefined variables blockDim and gridDim respectively.

The number of dimensions chosen for grids and blocks depends upon the type of data being operated upon, thus helping in better visualization and manipulation of the data. For example, for an image that is made up of a 2D array of pixels, it is imperative to use a 2D grid and block structure. In this case, each thread can be identified by a unique ID in x- and y-directions, with each thread operating on a specific image pixel. An example of using 3D variables could be for the software involving multiple layers of graphics where each layer adds another dimension to the grids and blocks.

In C language, multidimensional arrays are stored in the main memory in row-major layout. This means that in case of a 2D array, first all the elements in the first row are placed in the consecutive memory locations. Then the elements of the next row are placed afterwards. This is in contrast with the column-major layout followed by FORTRAN, in which all elements of the first column are assigned consecutive memory locations and the next columns are assigned memory afterwards in a similar way. For dynamically allocated multidimensional arrays, there is no way to specify the number of elements in each dimension beforehand. Hence while accessing such arrays using CUDA C, programmers convert the higher dimensional element indexes to the equivalent 1D array indexes. The programmers being able to access higher dimensional arrays by a single dimensional index is possible because of the flat memory system being used, which linearizes all multidimensional arrays in a row-major fashion.

Consider a 2D array of size $m \times n$. Since it is a two dimensional data object, it is intuitive that the kernel accessing it creates a two dimensional grid and block structure. Let us consider that each thread block consists of 16 threads in x- and y-dimensions. Then, the variable specifying the dimensions of the block is declared as:

```
dim3 numThreads(16,16,1);
```

The total number of thread blocks in x dimension (which represents the columns) will be $\text{ceil}(n/16)$ and y dimension (which represents the rows) will be $\text{ceil}(m/16)$. Thus, the

variable specifying the dimensions of the grid is declared as:

```
dim3 numBlocks(ceil(n/16), ceil(m/16),1);
```

If the 2D array has a size of 100×70 and the block dimension is 16×16 , then the number of thread blocks in x and y dimensions is 5 and 7, respectively, creating a total of $(7*16=112) \times (5*16=80) = 8960$ threads. However, the number of data elements in the array = $100 \times 70 = 7000$. For the extra created threads, a check is required on their identifier whether it is exceeding the number of rows or columns of the 2D array.

Matrix Multiplication Example

Consider two matrices A and B of sizes $m \times n$ and $n \times p$, respectively. Let C be the matrix that stores the product of A and B. The size of C would be $m \times p$. We want each cell element of the matrix C to be evaluated by a separate thread. Thus, we follow 2D structure of the kernel grid with each thread pointing to a separate cell element of C. Assuming that the size of each thread block is 16×16 , dimensions of the grid computing matrix C would be $\text{ceil}(m/16) \times \text{ceil}(p/16)$. Thus, each block will consist of 256 threads ranging from $T_{0,0}...T_{0,15}...$ to $T_{15,15}$. The program for matrix multiplication is given below:

```
1 void productMatrix(float **A, float **B, float **C, int m, int n, int p)
2 {
3     int allocationSize_A = m*n*sizeof(float);
4     int allocationSize_B = n*p*sizeof(float);
5     int allocationSize_C = m*p*sizeof(float);
6     float *deviceA, *deviceB, *deviceC;
7     cudaMalloc((void**) &deviceA, allocationSize_A);
8     cudaMalloc((void**) &deviceB, allocationSize_B);
9     cudaMalloc((void**) &deviceC, allocationSize_C);
10    cudaMemcpy(deviceA, A, allocationSize_A, cudaMemcpyHostToDevice);
11    cudaMemcpy(deviceB, B, allocationSize_B, cudaMemcpyHostToDevice);
12    dim3 numBlocks(ceil(p/16), ceil(m/16),1);
13    dim3 numThreads(16,16,1);
14    productMatrixKernel<<<numBlocks, numThreads>>>
15    (deviceA, deviceB, deviceC, m, n, p);
16 }
17 __global__ void productMatrixKernel(float *deviceA, float *deviceB,
18     float *deviceC, int m, int n, int p)
19 {
20     /* y_id corresponds to the row of the matrix */
21     int y_id = blockIdx.y * blockDim.y + threadIdx.y;
22     /* x_id corresponds to the column of the matrix */
23     int x_id = blockIdx.x * blockDim.x + threadIdx.x;
24     if((y_id < m) && (x_id < p)) {
25         float sum = 0;
26         for (int k = 0; k < n; k++)
27             sum += deviceA[y_id*n + k] * deviceB[k*p + x_id];
28     }
29     deviceC[y_id*p + x_id] = sum;
30 }
```

Statements 1–11 perform the basic steps required before the execution of a kernel, that is, memory allocation and data transfer. The configuration parameters numBlocks and numThreads are defined based on the data to be operated (on matrix in our example) in statements 12 and 13.

It is very important to understand how the kernel productMatrixKernel() works, since it involves two dimensions. Statements 20 and 22 compute the y and x thread indexes, respectively. For our 2D data, these indices map to the rows and columns of the matrix. Every thread accesses data elements based on its unique thread ID. If condition in statement 23 is necessary because there will be some threads which will not map to any data element in the matrix and should not process any data.

An important point to note is that accessing the matrices using these IDs directly in the form deviceA[y_id][x_id] and deviceB[y_id][x_id] is not possible because the number of columns in the arrays is not available during the compile time. Hence, the arrays are linearized as explained earlier and programmers have to map the 2D indices into the corresponding linear or 1D indices, as is done in the statement 26. Since, C uses row-major layout, memory is allocated row-wise. Each row of matrix deviceA has n elements, thus we need to move $y_id * n$ memory locations to reach the row having the required elements. Then by traversing k adjacent locations, the data deviceA[y_id][k] is fetched by using the index [y_id*n+k]. Similarly, the data element deviceB[k][x_id] is accessed using the equivalent linearized form deviceB[k*p+x_id], since each row of matrix deviceB has p elements. Each element of matrix deviceC is the sum of products of corresponding row elements of deviceA and column elements of deviceB. Once the final value of sum is calculated after the completion of the loop, it is copied to the corresponding element location in the matrix deviceC using the statement 28.

Figure 8.4 shows in detail various threads and thread blocks for the matrix multiplication example with $m=n=p=4$. Block size is taken as 2×2 .

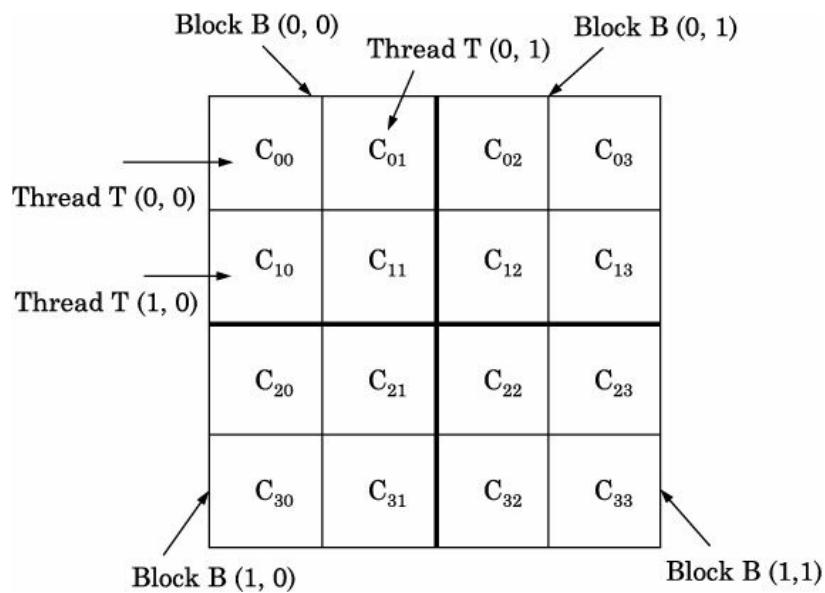


Figure 8.4 Matrix multiplication: Blocks and Threads.

Thread Synchronization and Scheduling

In CUDA, a basic way to coordinate threads is by means of barrier synchronization function, `__syncthreads()`. The threads of a block wait at the point in the program where `__syncthreads()` is used and their execution is stalled till other threads reach the same point. Once all the block threads reach the barrier, they continue their execution. Note that

`__syncthreads()` provides coordination among the threads belonging to the same block. As mentioned earlier, different thread blocks execute independent of each other and in any random order, and hence no synchronization is required between them.

If a kernel has **if-else** construct and a barrier `__syncthreads()` in either **if** or **else** statements, then care must be taken to see that either all threads take the **if** path or all threads take the **else** path. Otherwise, the threads waiting at `__syncthreads()` will keep on waiting forever for the other threads to reach at the barrier. Similarly, two barriers within the same kernel (for example, one within **if** statement and the other within **else** statement) are different. If some threads are waiting at one barrier while others at the other barrier, they will again keep on waiting for each other forever. The programmer must take care of such intricacies while using barrier synchronization.

The threads in a thread block are grouped into *warps*. The number of threads present in a warp depends upon the device and in CUDA devices, it is 32. All the threads in a warp are scheduled to be executed at the same time. These threads follow SIMD execution model. The warps consist of threads with consecutive increasing ID values, for example, threads with `threadIdx.x` value between 0 and 31 will form one warp, threads between 32 and 63 will form second warp and so on. The reason for grouping the threads into warps is to reduce the idle time a device spends waiting for memory access operations to get completed. For instance, if a memory read/write instruction is executed by a warp, the device instead of waiting immediately selects some other warp for execution. The overhead involved in switching from one warp to another is very minimal in GPUs and this is referred to as *zero-overhead thread scheduling*.

Coalescing

CUDA applications access a very large amount of data in a very short span of time because of their data parallel model. Apart from using L_1 (size varying between 16 KB and 64 KB) and L_2 (size varying between 128 KB and 1 MB) caches as we saw in Chapter 5, a technique employed by newer GPUs to increase memory access speeds is *coalescing*. Whenever an access request to the global memory is made, not only the requested DRAM address byte is loaded but multiple memory locations with consecutive addresses are also loaded. Thus, data from global memory is transferred in a chunk of bytes. If the memory access pattern of the threads operating in parallel in a warp is such that they access consecutive memory locations, then the number of global memory accesses will be significantly reduced since a single access operation loads all the nearby memory locations. Such a technique in which threads in a warp are programmed to access consecutive data is known as *coalescing*. It is up to the programmer to write his/her program in a way that has coalesced memory access patterns.

8.5.2 OpenCL (Open Computing Language)

OpenCL is a new, open standard for programming heterogeneous systems consisting of devices such as CPUs, GPUs, DSPs (Digital Signal Processors) and FPGAs (Field-Programmable Gate Arrays), unlike CUDA which is proprietary to NVIDIA GPUs. The standard, maintained by an industry consortium called Khronos Group (<https://www.khronos.org/opencl/>), enables programmers to write their application programs in C/C++ languages which execute on a wide variety of devices produced by different vendors, thereby achieving application code portability. It supports both data-parallel and task-parallel models of parallel computing.

An OpenCL program has two parts: (i) compute *kernels* which are executed by one or more *devices* and (ii) a *host* program which is executed by one CPU to coordinate the

execution of these kernels. The smallest unit of execution on a device is called *work-item*. A set of work-items that are executed simultaneously together constitute a *work-group*. The set of all work-groups that are invoked by the host constitute an index space called *NDRange* (*n*-dimensional range). Work-item, work-group and NDRange in OpenCL correspond to thread, thread block and grid in CUDA respectively. Synchronization among work-items belonging to the same work-group is realized by means of barriers that are equivalent to `__syncthreads()` in CUDA. NDRange and work-group sizes are specified as an array of type `size_t` and they can be specified, for example, as:

```
size_t indexSpaceDimension[3] = {512, 2, 1};  
size_t workgroupDimension[3] = {32, 1, 1};
```

`get_local_id(0)` is used to fetch the local index of a work-item within a work-group in x dimension (similarly 1,2 for y,z dimensions respectively). This is equivalent to `threadIdx.x` in CUDA. `get_global_id(0)` is used to fetch the unique global index of the work-item. This is equivalent to `blockIdx.x*blockDim.x+threadIdx.x` in CUDA. `get_local_size(0)` determines the size of each work-group and corresponds to `blockDim.x` in CUDA. `get_global_size(0)` gives the size of NDRange which is equivalent to `gridDim.x*blockDim.x` in CUDA.

OpenCL device architecture is similar to that of CUDA. The device consists of *compute units* (CUs) (corresponding to SMs in CUDA or execution units in DSPs and FPGAs) and each CU in turn consists of one or more *processing elements* (PEs), corresponding to SPs in CUDA, in which computations are carried out.

Declaration of kernels in OpenCL starts with `__kernel` (`__global` in CUDA). The kernel code for vector addition example is given as follows.

```
1. __kernel void addArrayKernel2  
(__global int *A, __global int *B, __global int *C)  
2. {  
3. int id = get_global_id(0);  
4. C[id] = A[id] + B[id];  
5. }
```

The `__global` keyword in kernel arguments indicates that these arrays reside in the device global memory. The reader can see the similarity between this kernel and the kernel written for vector addition in CUDA.

The following are the typical steps in the host code to execute an OpenCL kernel:

- (i) Discover and select the OpenCL computing/target devices
- (ii) Create “context” (It defines OpenCL environment in which work-items execute, which includes devices, their memories and “command queues”) and create “command queues” (In OpenCL, the host submits work, such as kernel execution and memory copy, to a device through a command queue)
- (iii) Compile the OpenCL kernel program
- (iv) Create/allocate OpenCL device memory buffers
- (v) Copy the host memory buffer to the device buffer
- (vi) Execute the OpenCL kernel
- (vii) Copy the results back to host memory
- (viii) Free memory on devices

OpenCL programs require more “set up” code (corresponding to the steps (i) and (ii) above, to deal with device diversity) than their CUDA counterparts. However, unlike CUDA programs, OpenCL programs are portable across different devices produced by different

vendors, which is the main motivation for the development of OpenCL standard.

8.6 PROGRAMMING IN BIG DATA ERA

Big data is a popular term used to describe massive volume of a variety of data (for example, text, video, sound, images) that exceeds the processing capacity of conventional data processing systems. A few examples of big data are data generated from: (i) scientific measurements (particle accelerators in physics, genome sequencers in biology), (ii) peer-to-peer communication (text messages, voice calls), (iii) smart cities (sensors), (iv) health care (electronic health records, clinical data), (v) business (e-commerce, stock market), (vi) enterprises (product sales, customer interaction) and (vii) social networks (Facebook posts, Twitter tweets). Big data needs to be processed to extract meaningful value (for example, trends, data correlations or patterns, if any) from it.

MapReduce programming model enables easy development of scalable, data parallel applications to process big data on large clusters of off-the-shelf low-cost machines (that are available for purchase, for example PCs, servers). It allows programmers to write a simple program, using map and reduce functions, to process peta-/exa-bytes of data on thousands of machines without the knowledge of (i) how to parallelize the program, (ii) how to distribute/manage data across the machines, (iii) how to schedule parallel tasks in the program onto the machines, (iv) how to achieve synchronization among tasks and (v) how to recover from failures of a task or a machine when the program is being executed. All the above (i)–(v) are accomplished by the MapReduce implementation. A popular implementation of MapReduce is Hadoop, developed initially by Yahoo, where it runs MapReduce applications (also known as jobs) that process hundreds of terabytes of data. A most well-known Hadoop user is Facebook, where it runs MapReduce jobs, for example, to create recommendations for a user based on his/her friends' interests.

8.6.1 MapReduce

This celebrated new, data-parallel programming model popularized by Google, having its roots in functional languages' *map* and *fold*, defines a computation step using two functions: *map()* and *reduce()*.

Map: $(key1,value1) \rightarrow list(key2,value2)$

Reduce: $(key2,list(value2)) \rightarrow list(key3,value3)$

The map function takes an input key/value pair (*key1,value1*) and outputs a list of intermediate key/value pairs (*key2,value2*). The reduce function takes all values associated with the same intermediate key and produces a list of key/value pairs (*key3,value3*). Programmers express their computation using the above two functions and as mentioned earlier the MapReduce implementation or runtime system manages the parallel execution of these two functions.

The following pseudocode illustrates the basic structure of a MapReduce program that counts the number of occurrences of each word in a collection of documents. Map and Reduce are implemented using system provided APIs, *EmitIntermediate()* and *Emit()* respectively.

```

/* input: a document */
/* intermediate output: key = word; value = 1 */
Map(void *doc) {
    for each word w in doc
        EmitIntermediate(w,1); /* count each word once */
}

/* intermediate output: key = word; value = 1 */
/* output: key = word; value = occurrences */
Reduce(void *word, Iterator values) {
    int result = 0;
    for each v in values
        result = result + v;
    Emit(word, result); /* output word and its count */
}

```

Figure 8.5 illustrates an execution overview of MapReduce word count program. The Map function emits each word in a document with a temporary count 1. The Reduce function, for each unique word, emits word count. In this example, we assume that the run-time system decides to use three Map instances (or three Map tasks) and two Reduce instances (or two Reduce tasks). For ease of understanding of our example program execution, we also assume that the three map instances operate on three lines of the input file as shown in the figure. In practice, a file is split into blocks as explained later. Observe that there are two phases, namely shuffle and sort, in between the three mappers and two reducers. The shuffle phase performs an all-map-to-all-reduce personalized communication so that all the $\langle\text{key}, \text{value}\rangle$ tuples for a particular key are sent to one reduce task. As the number of unique keys is usually much more than the reduce tasks, each reduce task may process more than one key. The sort phase sorts the tuples on the key field essentially grouping all the tuples for the same key. As each reduce task receives tuples from all the map tasks, this grouping may not take place naturally and the tuples for the different keys meant for a reduce task may be jumbled.

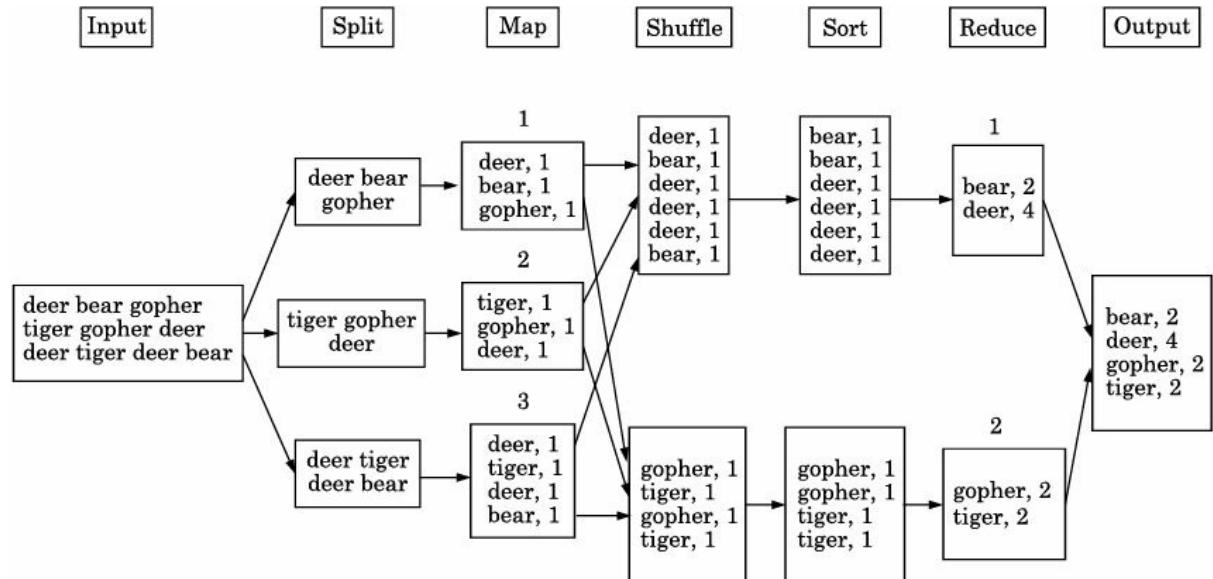


Figure 8.5 Word count execution.

Observe that the choice of the number of map and reduce tasks (3 and 2 respectively in our example) has a bearing on task granularity. Both Map and Reduce tasks can run in parallel,

however there is dependency between the map and reduce phases. Each map task's output data should be communicated to various reduce tasks such that (i) tuples with the same key must be sent to the same reduce task, (ii) the overhead of shuffle phase is minimized and (iii) the load across reduce tasks is balanced.

The intermediate output from the map function typically has to be sent over the network in order to perform the reduce function. If the intermediate data is large, it can cause high network overhead. For associative operations such as count, sum and maximum, a combiner function can be used which will decrease the size of the intermediate data. The combiner function operates on the intermediate output of map. The output of the combiner is then sent to the reduce function. In our example, the map instance M3 with a combiner (performing the same operation as the reduce function) produces $\langle\text{deer}, 2\rangle$ instead of $\langle\text{deer}, 1\rangle$ twice. Thus the combiner function, which runs on the output of map phase, is used as an optimization for MapReduce program.

8.6.2 Hadoop

Hadoop is an open-source Java-based MapReduce implementation (as opposed to Google's proprietary implementation) provided by Apache Software Foundation (<http://hadoop.apache.org>). It is designed for clusters of commodity machines that contain both CPU and local storage. Like MapReduce, Hadoop consists of two layers: (i) Hadoop Distributed File System (HDFS) or data storage layer and (ii) MapReduce implementation framework or data processing layer. These two layers are loosely coupled.

HDFS is primarily designed to store large files and built around the idea of "write-once and read-many-times". It is an append-only file system. Once a file on HDFS is created and written to, the file can either be appended to or deleted. It is not possible to modify any portion of the file. Files are stored as blocks, with block size 64 or 128 MB. Different blocks of the same file are stored on different machines or nodes (striping of data blocks across cluster nodes enables efficient MapReduce processing) and each block is replicated across (usually, 3) nodes for fault-tolerance. The HDFS implementation has a master-slave architecture. The HDFS master process, called NameNode, is responsible for maintaining the file system metadata (directory structure of all files in the file system, location of blocks and their replicas, etc.), thus manages the global file system name space. The slaves, called DataNodes, perform operations on blocks stored locally following instructions from the NameNode. In typical Hadoop deployments, HDFS is used to store both input/output data to/from a MapReduce job. Any intermediate data (for example, output from Map tasks) is not saved in HDFS, instead stored on local disks of nodes (where the map tasks are executed).

The MapReduce implementation framework handles scheduling of map/reduce tasks of a job (processes that invoke user's Map and Reduce functions are called map and reduce tasks respectively), moving the intermediate data from map tasks to reduce tasks and fault-tolerance. A user's (client's) program submitted to Hadoop is sent to the JobTracker process running on the master node (NameNode). Another process called TaskTracker is run on each slave node (DataNode). The JobTracker splits a submitted job into map and reduce tasks and schedules them (considering factors such as load balancing and locality, for example, it assigns map (reduce) tasks close to the nodes where the data (map output) is located) to the available TaskTrackers. The JobTracker then requests the TaskTrackers to run tasks and monitor them. Each TaskTracker has a fixed number of map (reduce) slots for executing map (reduce) tasks. When a TaskTracker becomes idle, the JobTracker picks a new task from its queue to feed it. If the request from the JobTracker is a map task, the TaskTracker will process job split or data chunk specified by the JobTracker. If the request is a reduce task, it

initializes the reduce task and waits for the notification from the JobTracker (that the map tasks are completed) to start processing (reading intermediate results, sorting, running the user-provided reduce function and saving the output in HDFS). Communication between the JobTracker and a TaskTracker processes, for information and requests, happens using a heartbeat protocol. Similar to how data storage follows master-slave architecture, the processing of map/reduce tasks follows the master-slave model.

Figure 8.6 shows the execution of our word count example program on Hadoop cluster. The secondary NameNode shown in this figure is not a standby node for handling the single point of failure of the NameNode. It (a misleading named component of Hadoop) is used for storing the latest checkpoints of HDFS states. Typically, a map task executes sequentially on a block and different map tasks execute on different blocks in parallel. Map function is executed on each record in a block. A record can be defined based on a delimiter such as ‘end of line’. Typically, a line delimiter is used and map function is executed for each line in a block. Sequentially executing map functions record-by-record in a block improves the data read time.

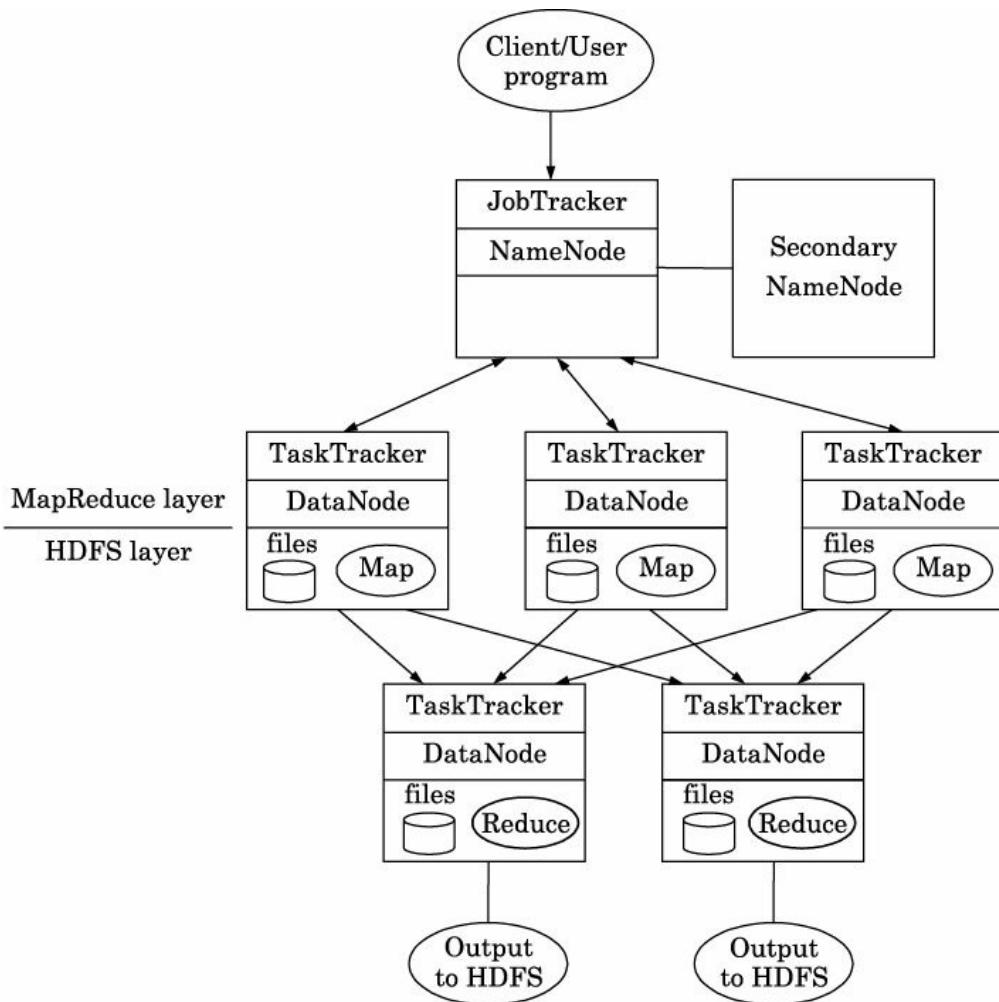


Figure 8.6 Word count execution on Hadoop cluster.

The MapReduce implementation framework also handles failures (note that inexpensive cluster nodes fail, especially when there are many) hiding the complexity of fault-tolerance from the programmer. If a slave node crashes (when the JobTracker does not receive periodic heartbeats from the TaskTracker), the runtime system automatically reruns its (unexecuted) tasks on another healthy node in the cluster. Similarly, if a node is available but is performing

poorly (*straggler* which lengthens completion time), the runtime system launches a *speculative copy*, also called *backup task*, on another slave node and takes the output of whichever copy finishes first, killing the other one.

In addition to the two core components, HDFS and MapReduce implementation, Hadoop includes a wide range (family) of related software systems/tools, for example higher-level languages such as Pig and Hive, and hence it is known as *Hadoop ecosystem*. PigLatin higher-level language constructs (supported by Pig) and HiveQL queries (similar to SQL queries in RDMS, supported by Hive) are converted into Java MapReduce programs and then submitted to the Hadoop cluster. This enhances the productivity of a programmer/application developer as using Java MapReduce model requires a lot of code to be written and debugged as compared to PigLatin and HiveQL programs. The Hadoop ecosystem, a widely used open source big data processing platform, continues to evolve with new/improved technologies/tools.

YARN

The next version of Apache Hadoop is called YARN (Yet Another Resource Negotiator). The fundamental idea of YARN is to decouple the resource management features built into MapReduce from the programming model of MapReduce. Thus the responsibilities of the JobTracker, which are resource management and job scheduling/monitoring (that limit scaling to larger cluster nodes), are split into a global Resource Manager (RM) and per-application (or MapReduce) Application Master (AM), one of which is started for managing each MapReduce job and terminated when the job finishes. The RM coordinates the allocation of resources (CPU, memory, bandwidth, etc.) through underlying per-node agents called the Node Managers (NMs). AM is responsible for negotiating resources from the RM and working with NM(s) to execute and monitor application tasks. Different AMs can be built for different types of applications to be executed on YARN. Thus, a MapReduce job becomes one of the many applications which could be executed on YARN, whereas Hadoop was designed to run MapReduce jobs (applications) only.

The YARN's efficient dynamic allocation of cluster resources by the RM results in better utilization of the resources as compared to static MapReduce rules used in the earlier version (for example, traditional Hadoop implementation does not support changing the map or reduce slot configurations which prevents dynamically adapting to variations during a job's life-cycle, thereby reducing the cluster resource utilization). Other features of YARN include multi-tenancy (allowing multiple, diverse, for example batch, interactive or real-time, user applications developed in languages other than Java running simultaneously on the platform) and compatibility (running the existing MapReduce applications developed for Hadoop).

Spark

Apache Spark, originally developed at the University of California Berkeley, is a recent open source cluster computing system compatible with Apache Hadoop. A limitation of Hadoop MapReduce is that it writes all the output data to the HDFS (disk-based file system) after execution of each MapReduce job. Many common machine learning algorithms require several iterative computations to be performed on the same data. If this is implemented using Hadoop MapReduce, reading and writing files from the disk during each iteration, which is expressed as a MapReduce job, will delay job completion. Spark overcomes this limitation by storing data in main memory (DRAM, instead of disk) of cluster nodes and allowing multiple iterative computations on the same data, that is, subsequent iterations repeatedly access the same data present in DRAM. This in-memory cluster computing enables Spark to significantly outperform Hadoop MapReduce for some machine learning algorithms like

logistic regression. Fault tolerance is ensured by logging only the sequence of operations performed on the data.

Besides the Spark core execution engine, Spark ecosystem includes higher-level frameworks for machine learning (MLlib), stream processing (Spark Streaming), graph processing (GraphX) and high level query processing (Spark SQL).

8.7 CONCLUSIONS

In this chapter, we studied the following: (i) MPI, a de facto standard for parallel programming on message passing (distributed memory) parallel computers. MPI programs are portable due to the existence of MPI standard library implementation on a wide range of parallel computers. Though originally designed for message passing parallel computers, today MPI programs run on virtually any platform—distributed memory, shared memory and hybrid systems. (ii) OpenMP, a portable approach for parallel programming on shared memory systems and multicore processors. Based on compiler directives and a set of supporting library functions, OpenMP, unlike MPI, offers an incremental approach towards parallelization. (iii) CUDA and OpenCL, libraries that let programmers to write parallel programs using a straightforward extension of C/C++ and FORTRAN languages for NVIDIA’s GPUs and vendor neutral heterogeneous devices (including NVIDIA’s GPUs) respectively. (iv) Finally, MapReduce, a programming model that enables easy development of scalable, data parallel applications to process big data on large clusters of off-the-shelf low-cost machines, and its open source implementation Hadoop.

EXERCISES

- 8.1 How do you emulate a message passing parallel computer containing n processors on a shared memory parallel computer with an identical number of processors?
- 8.2 How do you emulate a shared memory parallel computer containing n processors on a message passing parallel computer with an identical number of processors?
- 8.3 What are the advantages and disadvantages of explicit and implicit parallel programming approaches?
- 8.4 Using Send/Receive and Fork/Join primitives write parallel programs to compute π .
- 8.5 Consider the following message passing program for computing a and b as described (by a task graph) in Chapter 2.

Program for Processor0

```
begin
    Read x;
    P1 := f(x) ** 3; { Tasks T1, T2 }
    P2 := cos (P1); { Task T3 }
    Send(1, P2, 1)
end.
```

Program for Processor1

```
begin
    Read y;
    Q1 := cos(g(y)); { Tasks T4, T5 }
    Receive(0, Q2, 1);
    Receive(2, Q3, 1);
    a := Q1 + Q2 + Q3; { Task T6 }
    Write a
end.
```

Program for Processor2

```
begin
    Read z;
    R1 := exp (h(z)); { Tasks T7, T8 }
    Send(1, R1, 1);
    Send(3, R1, 1);
    R2 := sin (R1); { Task T9 }
    Send(3, R2, 1)
end.
```

Program for Processor3

```
begin
    Read u;
    S1 := p(u); { Task T10 }
    Receive(2, S2, 1);
    S3 := S1 * S2; { Task T11 }
    Receive(2, S4, 1);
    b := S4 + S3 ; { Task T12 }
    Write b
end.
```

Convert the above program into an equivalent SPMD message passing parallel program.

8.6 The following is an equivalent program of Exercise 8.5.

Program for Processor0

```
begin
    global x, y, z, u, a, b;
    Read(x, y, z, u);
    fork(1) Proc1 (input: x, output: Q);
    fork(2) Proc2 (input: y, output: R);
    fork(3) Proc3 (input: z, output: S,T);
    W := p(u);
    join;
    b := W * S + T;
    a := Q + R + S;
    Write a, b;
end. {Main}

{Procedure Proc1 executed in Processor1}
Proc1 (input: x, output: Q);
begin
    Q := cos (f(x)** 3);
end {Proc1};

{Procedure Proc2 executed in Processor2}
Proc2 (input: y, output: R);
begin
    R := cos (g(y))
end { Proc2 } ;

{Procedure Proc3 executed in Processor3}
Proc3 (input: z, output: S, T);
begin
    S := exp (h(z));
    T := sin(S);
end {Proc3};
```

Convert the above program into an equivalent SPMD shared memory parallel program.

- 8.7 Write SPMD message passing and shared memory parallel programs for processing student grades as described in Chapter 2.
- 8.8 A popular statement is that shared memory model supports fine-grained parallelism better than the message passing model. Is it true? (Hint: Fine-grained parallelism requires efficient communication/synchronization mechanism.)
- 8.9 Another popular statement is that shared memory programming is easier than message passing programming. Is it true? If so justify your answer with examples.
- 8.10 Which programs, shared memory or message passing, are difficult to debug? (Hint: Consider the address space of the processes.)
- 8.11 A parallel programming language, by extending (sequential) FORTRAN or C, can be realized using three different approaches—Library Functions (e.g. MPI), Compiler Directives (e.g. OpenMP) and New Constructs (e.g. FORTRAN 90). List the relative merits and shortcomings of these three approaches.
- 8.12 Write an MPI program that sends a message (a character string, say, “hello world”) from processes with rank greater than zero to the process with rank equal to zero.

8.13 Write an MPI program that distributes (using `MPI_Scatter`) the random numbers generated by a root node (the node corresponding to the process with rank equal to zero) to all the other nodes. All nodes should compute the partial sums which should be gathered into the root node (using `MPI_Gather`). The root node should print the final result (sum of all random numbers it generated).

8.14 The need for the embedding of one graph into another (*graph embedding*) arises from at least two different directions. First (portability of algorithms across networks), an algorithm designed for a specific interconnection network (graph) may be necessary to adapt it to another network (graph). Second (mapping/assigning of processes to processors), the flow of information in a parallel algorithm defines a program (task/application) graph and embedding this into a network tells us how to organize the computation on the network. Figure 8.7 shows an embedding of linear array (chain) into a mesh and Fig. 8.8 shows an embedding of complete binary tree on a mesh. Embed (a) a ring into a 4×5 mesh and a 3×3 mesh, and (b) a 2×4 mesh into a 3-dimensional hypercube.

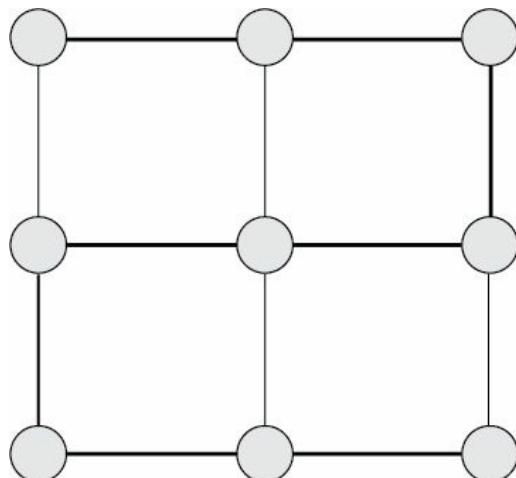


Figure 8.7 Embedding of chain into a mesh.

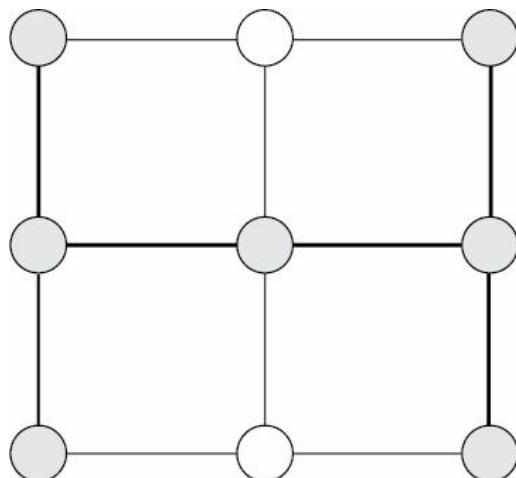


Figure 8.8 Embedding of binary tree into a mesh.

8.15 When embedding a graph $G(V, E)$ into $G'(V', E')$, three parameters are important. First, it is possible that more than one edge in E (set of edges in G) is mapped into a single edge in E' (set of edges in G'). This leads to additional traffic on the corresponding communication link. The maximum number of edges mapped onto any edge in E' is called the *congestion* of the embedding. Second an edge in E , may be

mapped onto multiple contiguous edges in E' . This also leads to additional traffic, on the corresponding link, as it must traverse more than one link. The maximum number of links in E' that any edge in E is mapped onto is called the *dilation* of the embedding. Third, V (set of vertices in G) and V' (set of vertices in G') may contain different number of vertices. If so, a processor in V corresponds to more than one processor in V' . The ratio of number of processors in the set V' to that in V is called the *expansion* of the embedding. Is it possible to embed a complete binary tree with n levels ($2^n - 1$ vertices) into an n -dimensional hypercube with dilation 1 and expansion 1? Prove your answer.

- 8.16 Write MPI and OpenMP programs to multiply two $n \times n$ matrices.
- 8.17 Write an OpenMP program for Gaussian elimination.
- 8.18 Consider the vector addition CUDA kernel with a block size of 256 and each thread processing one output element. If the vector length is 4000, what is the total number of threads in the grid?
- 8.19 Given a matrix of size 400×900 where each element has a dedicated thread for its processing. Consider that the total number of threads in a block cannot be more than 1024 and that the blocks are two dimensional squares. What would be the grid dimension and block dimension of the CUDA kernel such that least number of threads are idle.
- 8.20 Write CUDA and OpenCL programs to transpose a square matrix.
- 8.21 Consider the following four files, each in a different HDFS block. Each file contains two columns (<key,value> in Hadoop terminology) that represent a city and the corresponding temperature (in Celsius) recorded in that city on a particular day. Our goal is to find the maximum temperature for each city across all the four data files. Assuming four instances of map and two instances of reduce functions, show the output at each of the phases (map, shuffle, sort, reduce) in the MapReduce workflow.

File 1	File 2	File 3	File 4
Srinagar 16	Srinagar 15	Srinagar 16	Srinagar 15
New Delhi 24	Bengaluru 30	Bengaluru 32	Kolkata 37
Mumbai 31	Kolkata 38	New Delhi 29	New Delhi 28
Kolkata 36	New Delhi 23	New Delhi 30	Mumbai 33
Bengaluru 33	Mumbai 32	Kolkata 35	Bengaluru 31

- 8.22 Consider the word count program described in this chapter. How do you adopt this program if our goal is to output all words sorted by their frequencies.
(Hint: Make two rounds of MapReduce and in the second round take the output of word count as input and exchange <key,value>)
- 8.23 Hadoop speculative task scheduler uses a simple heuristic method which compares the progress of each task to the average progress. A task with the lowest progress as compared to the average is selected for re-execution on a different slave node. However, this method is not suited in a heterogeneous (slave nodes with different computing power) environment. Why?
- 8.24 How are the following handled by Hadoop runtime system: (i) Map task failure, (ii) Map slave node failure, (iii) Reduce task failure, (iv) Reduce slave node failure, and (v) NameNode failure?

BIBLIOGRAPHY

- Apache Hadoop Project— <http://hadoop.apache.org/>
- Barney, B., “Introduction to Parallel Computing”.
https://computing.llnl.gov/tutorials/parallel_comp/#DesignPerformance.
- Chapman, B., Jost, G. and Pas, R., *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, USA, 2007.
- Dean, J., and Ghemawat, S., “MapReduce: Simplified Data Processing on Large Clusters”, *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–150.
- Diaz, J., Munoz-Caro, C. and Nino, A., “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 8, Aug. 2012, pp. 1369–1386.
- Dobre, C. and Xhafa, F., “Parallel Programming Paradigms and Frameworks in Big Data Era”, *International Journal of Parallel Programming*, Vol. 42, No. 5, Oct. 2014, pp. 710–738.
- Gaster, B.R., Howes, L., Keili, D.R., Mistry, P. and Schaa, D., *Heterogeneous Computing with OpenCL*, Morgan Kaufmann Publishers, USA, 2011.
- Hwang, K. and Xu, Z., *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, USA, 1998.
- Kalavri, V. and Vlassov, “MapReduce: Limitations, Optimizations and Open Issues”, *Proceedings of IEEE International Conference on Trust, Security, and Privacy in Computing and Communications*, 2013, pp. 1031–1038.
- Kirk, D.B. and Hwu, W.W., *Programming Massively Parallel Processors*, Morgan Kaufmann Publishers, USA, 2010.
- Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D. and Moon, B., “Parallel Data Processing with MapReduce: A Survey”, *ACM SIGMOD Record*, Vol. 40, No. 4, Dec. 2011, pp. 11–20.
- Pacheco, P., *An Introduction to Parallel Programming*, Morgan Kaufmann Publishers, USA, 2011.
- Rajaraman, V. and Siva Ram Murthy, C., *Parallel Computers: Architecture and Programming*, PHI Learning, Delhi, 2000.
- Rauber, T. and Runger, G., *Parallel Programming for Multicore and Cluster Systems*, Springer-Verlag, Germany, 2010.
- Sanders, J. and Kandrot, *CUDA by Example: An Introduction to General Purpose GPU Programming*, Addison-Wesley, NJ, USA, 2010.
- Vinod Kumar, V., Arun, C.M., Chris, D., Sharad, A., Mahadev, K., Robert, E., Thomas, G., Jason, L., Hitesh, S., Siddharth, S., Bikas, S., Carlo, C., Owen, O.M., Sanjay, R., Benjamin, R. and Eric, B., “Apache Hadoop YARN: Yet Another Resource Negotiator”, *Proceedings of the ACM Annual Symposium on Cloud Computing*, 2013.
- White, T., *Hadoop: The Definitive Guide*, O'Reilly Media Publisher, Sebastopol, CA, USA, 2012.
- Wilson, G.V., *Practical Parallel Programming*, Prentice-Hall, New Delhi, 1998.
- Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. and Stoica, I., “Spark: Cluster Computing with Working Sets”, *Proceedings of the USENIX Conference on Hot Topics in*

Cloud Computing, 2010.

Compiler Transformations for Parallel Computers

The role of a compiler is to translate code (the input program) from a high level language into an equivalent machine language. In the process, the compiler has a number of opportunities to transform the code into an optimized code and consequently minimize the running time (or size) of the final executable code. This is what an optimizing compiler aims to achieve. In this chapter, we describe several transformations that optimize programs written in (imperative) languages such as Pascal, C and FORTRAN (a de facto standard of the high-performance engineering and scientific computing community) for parallel computers, including vector machines, superscalar and multicore processors. However, the optimizations (transformations) we present here are not specific to Pascal, C and FORTRAN alone.

It is very important to note that a compiler must discover and expose sufficient parallelism in the input program to make efficient use of (parallel) hardware resources. While too little parallelism leaves more room for its efficient execution (due to underutilization of the resources) and too much parallelism can lead to execution inefficiency (due to parallel overhead such as synchronization and load imbalance).

9.1 ISSUES IN COMPILER TRANSFORMATIONS

To apply an optimization, a compiler must do the following:

1. *Decide* upon a portion of the code to optimize and a particular transformation to apply to it.
2. *Verify* if the optimized code is equivalent to the original code in terms of the meaning or the execution of the program.
3. *Transform* the code.

In this chapter, we concentrate on the last step, transformation of the code. However, we also present dependence analysis techniques that are used to select and verify many loop transformations.

9.1.1 Correctness

When a program is transformed by a compiler, the meaning of the program should remain the same. This is the issue of correctness. A transformation is said to be *legal* if the original and the transformed programs produce exactly the same output for all *identical executions* i.e., when supplied with the same input data, every corresponding pair of non-deterministic operations in the two executions produces the same result. The correctness is a complex issue as illustrated below:

Original program:

```
procedure correctness(a, b, n, m, k)
integer n, m, k
real a[m], b[m]
for i := 1 to n do
    a[i] = a[i] + b[k] + 10000.0
end for
end
```

A transformed version of the above program is as follows:

```
procedure correctness(a,b,n,m,k)
integer n, m, k
real a[m], b[m], C
C = b[k] + 10000.0
for i := n downto 1 do
    a[i] = a[i] + C
end for
end
```

Though on the face of it, the above code seems equivalent to the original code, there are a few possible problems.

- *Overflow*. Suppose $b[k]$ is a very large number and $a[1]$ is negative. This changes the order of the additions in the transformed code. This can cause an overflow to occur in the transformed code when one attempts to add $b[k]$ and 10000, while the same would not occur in the original piece of code. This discrepancy complicates debugging since the transformation is not visible to the programmer.

- *Different results.* Even if no overflow occurs, the values of the elements of array a , may be slightly different. The reason is floating-point numbers are approximations of real numbers, and the order in which the approximations are applied (*rounding*) can affect the result.
- *Memory fault.* If $k > m$ and $n < 1$, the reference to $b[k]$ is illegal. While the original code would not access $b[k]$ since it would not run through the loop, the transformed code does access it and throws an exception.
- *Different results.* Partial aliasing of a and b before the procedure ‘correctness’ is called can change the values assigned to a in the transformed code. In the original code, when $i = 1$, $b[k]$ is changed if $k = n$ and $b[n]$ is aliased to $a[1]$. Whereas in the transformed version, the old value of $b[k]$ is used for all the iterations thus giving different results.

Because of the above problems, henceforth we say a transformation is *legal* if, for all semantically correct executions of the original program, the original and the transformed programs perform equivalent operations for identical executions.

9.1.2 Scope

Transformations can be applied to a program at different levels of granularity. It should be noted that as the scope of the transformations is enlarged, the cost of analysis generally increases. Some useful gradations in ascending order of complexity are: (1) Statement, (2) Basic block (a sequence of statements with single entry and single exit), (3) Innermost loop, (4) Perfect loop nest, (5) General loop nest, (6) Procedure, and (7) Inter-procedural.

9.2 TARGET ARCHITECTURES

In this section, we briefly present an overview of different parallel architectures and some optimizations that a compiler performs for each of these architectures.

9.2.1 Pipelines

Pipeline processing is one of the most basic forms of the use of temporal parallelism. A common use of pipeline in computers is in the arithmetic unit, where complicated floating point operations are broken down into several stages whose executions overlap in time. The objective of scheduling a pipeline is to keep it full (*bubble-free* or *stall-free*) as this results in maximum parallelism. An operation can be issued only when there is no pipeline conflict, i.e., the operation has no resource conflicts or dependences on operations already in the pipeline. If there is no hardware interlocking mechanism to detect pipeline conflicts, the compiler must schedule the pipeline completely. If there is full hardware interlocking, the compiler need not worry about the pipeline conflicts. However, it may wish to reorder the operations presented to the pipeline to reduce the conflicts at run (or execution) time in order to enhance the pipelines performance. A common compiler optimization for pipelined machines is *instruction scheduling*, described later.

9.2.2 Multiple Functional Units

A common processor organization consists of multiple units such as a pipelined load/store unit, a pipelined integer and logical operation unit, and a pipelined floating point unit, all of which operate independently. If the operations routed to these units are carried out simultaneously in one clock cycle and are determined at run time by hardware, the architecture is referred to as *superscalar*.

On the other hand, if the operations to be fed to the functional units simultaneously are determined solely by the compiler, the architecture is referred to as a *long instruction word* (LIW machine). An LIW with more than a half-dozen or so functional units is called a VLIW (Very LIW). *Trace scheduling* and *software pipelining* are two techniques for scheduling multiple functional units in VLIW and superscalar machines and are described later.

Another issue in scheduling for VLIW machines is ensuring conflict-free access to memory which is divided into multiple banks to allow multiple independent access. A conflict is said to occur when two or more memory access operations, specified in a VLIW instruction, access the same memory bank. Memory bank conflicts are discussed in the section on memory access transformations.

9.2.3 Vector Architectures

Vector architectures are characterized by separate arithmetic and load/store pipelines for scalars and vectors. The vector arithmetic unit is matched to the memory system in such a way to allow one operand per clock to be delivered from the memory system and fed to the vector arithmetic unit or similarly, one result per clock to be delivered to the memory system to be stored. In order to reduce the speed mismatch between the pipeline logic and memory access times, vector architectures employ many memory banks. Successive memory addresses are contained in successive banks so that the memory can deliver the required one operand per clock if the *stride* (i.e., the distance between addresses of successive memory references) through the addressed memory is such that a bank is not revisited in less than the memory access time. As we will see later, the compiler can change the stride so that the

memory bank will be able to complete its access cycle before the next request to it is issued. Without this, the machine will not be able to sustain one operation per clock.

The main issue involved in compiling for a vector machine is the search for loops that can be *vectorized*, i.e., where the same operation can be applied to each element of a vector of operands. Consider the following piece of code in which all the iterations of the loop can be executed in parallel, in any order or all at once.

```
for i := 1 to 64 do
    a[i] = a [i] + b[i]
end for
```

We use array/vector notation when describing transformations for vector machines. For example, the above loop (in which computations can be vectorized), in array notation is written as

$$a[1:64] = a[1:64] + b[1:64]$$

assuming a vector machine with vector registers of length 64.

In the above example, if we replace the statement $a[i] = a[i] + b[i]$ with $a[i+1] = a[i] + b[i]$ then the loop in array notation cannot be written as

$$a[2:65] = a[1:64] + b[1:64]$$

because of dependence as shown below:

$$a(2) = a(1) + b(1)$$

$$a(3) = a(2) + b(2)$$

$$a(4) = a(3) + b(3)$$

...

Those computations that cannot be vectorized due to dependences or which should not be vectorized (for example, when vectors have very short length) are performed in scalar mode. A number of language constructs are available to users of vector machines. Compiler directives, another form of language extension, generally serve as “hints” to the compiler regarding loop dependences. With these directives, the vectorizing compiler has more information and can therefore produce very efficient code.

9.2.4 Multiprocessor and Multicore Architectures

In SIMD machines, where the processors execute the same instruction at the same time on data from different memories, the principal concern is the data layout across the distributed memories to allow parallel access. We describe compiler techniques to optimize data distribution. In data-parallel languages, for SIMD machines, the user specifies operations that are to be performed on many data objects simultaneously. In some cases, explicit data layout directives and inter-processor communication must be supplied by the user.

In an MIMD machine, rather than receiving a common instruction stream from a single control as in SIMD machines, each processor operates independently on its own thread of control, synchronizing and communicating with the other processors by sending/receiving messages to/from them. In the case of distributed memory MIMD machines, like in SIMD machines, the issues are of organizing the distributed memory and the inter-processor communication. We describe transformations which minimize inter-processor communication overhead. The foremost requirement of a compiler translating a sequential program to parallel form is to find multiple threads of control in the program which can be assigned to different processors. In the case of shared memory MIMD machines, the issues

are memory bank conflicts (here a typical source of conflict is semaphore access), as in vector machines, and data distribution (in NUMA-Non-Uniform Memory Access machines, in which a processor sees varying access times to different portions of memory).

Advancements in processor microarchitectures have led to parallelism and multicore processors becoming mainstream. Many scientific and engineering (compute-intensive) applications often spend significant amount of their execution time in loops. Multicore processors exploit thread-level parallelism by simultaneously executing multiple loop iterations across multiple cores. We describe loop transformations (for example, *loop tiling*), for multicore processors with complex cache hierarchy, that reorder the iterations of a loop to achieve good caching behaviour (hit rate). *Decoupled software pipelining*, which extracts pipeline parallelism in loops, transforms a loop into multiple decoupled loops executing on multiple cores.

9.3 DEPENDENCE ANALYSIS

The accurate analysis of dependences in the data and control flow of both sequential and parallel programs is the key to successful optimization, *vectorization* (that is compiling for vector machines), *parallelization* (compiling for, loosely-coupled, distributed memory multiprocessor systems) and *concurrentization* (compiling for shared memory multiprocessor systems or multicore processors) of the code. We now introduce dependence analysis, its terminology and the underlying theory.

A dependence is a relationship between two computations that places some constraints on their execution order. Dependence analysis identifies these constraints, which help in determining whether a particular transformation can be applied without changing the semantics (meaning) of the computation.

9.3.1 Types of Dependences

Dependences can be broadly classified into two classes: *data dependences* and *control dependences*. These are discussed below:

Data Dependence

Two statements have a data dependence if they cannot be executed simultaneously because of conflicting uses of the same variable. Data dependences are of three types.

1. *Flow dependence*: A statement S_2 has a flow dependence on a statement S_1 (denoted by $S_1 \rightarrow S_2$), when S_1 must be executed first because it writes a value that is read by S_2 .

$$S_1: a = b * 2$$

$$S_2: d = a + c$$

Here, the variable a is written in S_1 and the newly written value is read in S_2 . Hence flow dependence is also called Read After Write (RAW) dependence.

2. *Anti-dependence*: S_4 has an anti-dependence on S_3 (denoted by $S_3 \xrightarrow{a} S_4$) when S_4 writes a variable that is read by S_3 .

$$S_3: e = g * 2$$

$$S_4: g = f + h$$

Here, the variable g is written in S_4 but S_3 is supposed to read the old value in g and not the newly written value. Hence anti-dependence is also called Write After Read (WAR) dependence.

3. *Output dependence*: An output dependence (denoted by $S_5 \xrightarrow{o} S_6$) holds between two statements S_5 and S_6 when both statements write the same variable.

$$S_5: e = g * 2$$

$$S_6: e = f + h$$

Here, both statements write the variable e . Hence output dependence is also called Write After Write (WAW) dependence.

Control Dependence

There is a control dependence between statements S_7 and S_8 . Written as $S_7 \xrightarrow{c} S_8$ when

S_7 determines whether S_8 will be executed or not.

```

 $S_7 : \text{if } (a = 5) \text{ then}$ 
 $S_8 : b = 9$ 
 $\text{end if}$ 
```

9.3.2 Representing Dependences

In order to capture the dependence information for a piece of code, the compiler creates a *dependence graph*. A dependence graph is a directed graph with each node representing a statement and an arc between two nodes indicating that there is a dependence between the computations that they represent. Consider the following program segment:

```

 $S_1 : A = B + D$ 
 $S_2 : C = A * 3$ 
 $S_3 : A = A + C$ 
 $S_4 : E = A / 2$ 
```

The dependence graph for this program segment is given in Fig. 9.1. The types of dependences are also indicated in this figure.

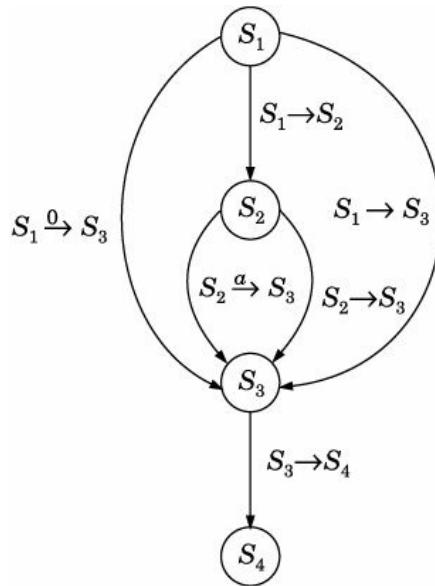


Figure 9.1 Dependence graph.

9.3.3 Loop Dependence Analysis

All the dependences mentioned above pertain to straight-line code where each statement is executed at most once and thus the dependence arcs capture all the possible dependence relationships. In loops, dependences can exist between different iterations and these are called *loop-carried* dependences. For example:

```

for i := 2 to 7 do
    S1 : a[i] = a[i] + c
    S2 : b[i] = a[i-1] + b[i]
end for
```

Here, there is no dependence between S_1 and S_2 within any single iteration of the loop, but there is one between two successive iterations. When $i = k$, S_2 reads the value of $a[k - 1]$

written by S_1 in the iteration $k - 1$. A loop-carried dependence is represented by an *iteration dependence graph*. An iteration dependence graph is a d -dimensional graph where d is the number of nested loops. Each point in the d -dimensional space (considering only integral values on each axis) represents an iteration and the dependence of an iteration T on an iteration S is represented by an arrow from the point representing S to that representing T . The iteration dependence graph for the above example is given in Fig. 9.2.

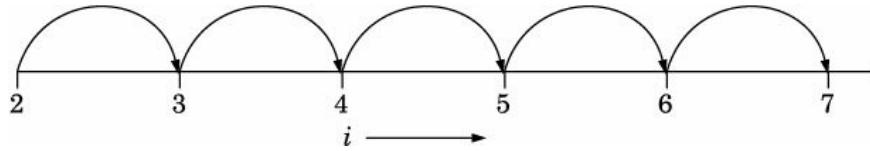


Figure 9.2 Iteration dependence graph.

To know whether there is a dependence in the loop nest, it is sufficient to determine whether any of the iterations can write a value that is read or written by any of the other iterations.

Distance Vectors

Loop dependences are represented using *distance vectors*. A generalized perfect nest of d loops, shown below, will now be used as a basis to describe the concepts related to analysis of loop dependences.

```

for  $i_1 := l_1$  to  $u_1$  do
  for  $i_2 := l_2$  to  $u_2$  do
    ...
    for  $i_d := l_d$  to  $u_d$  do
       $S_1: \quad a[f_1(i_1, \dots, i_d), \dots, f_m(i_1, \dots, i_d)] = \dots$ 
       $S_2: \quad \dots = a[g_1(i_1, \dots, i_d), \dots, g_m(i_1, \dots, i_d)]$ 
    end for
    ...
    end for
end for

```

The *iteration space* of the above nest is a d -dimensional discrete Cartesian space. Each axis of the iteration space corresponds to a particular **for** loop and each point represents the execution of all the statements in one iteration of the loop. An iteration can be uniquely named by a vector of d elements $I = (i_1, \dots, i_d)$, where each index falls within the iteration range of its corresponding loop in the nesting (i.e., $l_p \leq i_p \leq u_p$). Consider a dependence between statements S_1 and S_2 denoted as $S_1 \Rightarrow S_2$. That is, S_2 is dependent on S_1 . The *dependence distance* between the two statements is defined as vector $(S_2) - \text{vector}(S_1) = (y_1 - x_1, \dots, y_d - x_d)$. For example, consider the following loop:

```

for  $i := 2$  to  $n$  do
  for  $j := 1$  to  $n-1$  do
     $a[i, j] = a[i, j] + a[i-1, j+1]$ 
  end for
end for

```

Each iteration of the inner loop writes the element $a[i, j]$. There is a dependence if any iteration reads or writes that same element. Consider the iterations $I = (2, 3)$ and $J = (3, 2)$. Iteration I occurs first and writes the value $a[2, 3]$. This value is read in iteration J , so there is a flow dependence from iteration I to iteration J , i.e., $I \rightarrow J$. The dependence distance is $J - I$

$= (1, -1)$.

When a dependence distance is used to describe the dependences for all iterations, it is called a *distance vector*. $(1, -1)$ is the one and only distance vector for the above loop. Consider another example:

```

for i := 1 to n do
    for j := 2 to n-1 do
        a[j] = a[j] + a[j-1] + a[j+1]
    end for
end for

```

The set of distance vectors for this loop is $\{(0, 1), (1, 0), (1, -1)\}$. Note that distance vectors describe dependences among *iterations*, not among *array elements*. The operations on array elements create the dependences.

Direction Vectors

It may not always be possible to determine the exact dependence distance at compile-time, or the dependence distance may vary between iterations; but there is enough information to partially characterize the dependence. Such dependences are commonly described using *direction vectors*. For a dependence $I \Rightarrow J$, the direction vector is defined as $W = (w_1, \dots, w_d)$, where

$$w_p = \begin{cases} < & \text{if } i_p < j_p \\ = & \text{if } i_p = j_p \\ > & \text{if } i_p > j_p \end{cases}$$

The direction vector for the above first loop (with distance vector set $\{(1, -1)\}$) is the set $\{(<, >)\}$. The direction vectors for the second loop (with distance vector set $\{(0, 1), (1, 0), (1, -1)\}$) are $\{(<=), (<), (<,>)\}$. It can be easily seen that a direction vector entry of $<$ corresponds to a positive distance vector entry, $=$ corresponds to 0, and $>$ corresponds to a negative distance vector entry.

9.3.4 Subscript Analysis

Subscript analysis is the mechanism by which the compiler decides whether two array references might refer to the same element in different iterations. The approach followed by most optimizing compilers while examining a loop nest is as follows: first, the compiler tries to prove that different iterations are independent by applying various tests to the subscript expressions. These tests rely on the fact that the subscript expressions are almost always linear. If dependences are found, the compiler tries to describe them with a direction or distance vector. If the subscript expressions are too complex to analyze, the statements are assumed to be fully dependent on one another and no change in their execution order is permitted. Even for linear subscript expressions, finding dependences is equivalent to the NP-complete problem of finding integer solutions to systems of linear Diophantine equations. A Diophantine equation is a polynomial equation in integer variables. Two general and *approximate* tests are the GCD test and Banerjee's inequalities. Apart from these, there are several *exact* tests which exploit some subscript characteristics to determine whether a particular type of dependence exists. A test (algorithm) to compute dependence is exact if it can be used to find the integer solution to the set of *dependence equations* within the range of the loop(s). An approximate test might indicate that a solution exists, but, in fact, the solution may not lie within the region of the loop range(s) or may not be an integer.

Consider the following single loop:

```

L : for I := 4 to 200 do
S :   A(I) = B(I) + C(I)
T :   B(I + 2) = A(I - 1) + A(I - 3) + C(I - 1)
U :   A(I + 1) = B(2*I + 3) + 1
end for

```

The first four iterations, corresponding to the index values 4, 5, 6, 7 are shown below:

$$\begin{aligned}
S(4) &: A(4) = B(4) + C(4) \\
T(4) &: B(6) = A(3) + A(1) + C(3) \\
U(4) &: A(5) = B(11) + 1 \\
S(5) &: A(5) = B(5) + C(5) \\
T(5) &: B(7) = A(4) + A(2) + C(4) \\
U(5) &: A(6) = B(13) + 1 \\
S(6) &: A(6) = B(6) + C(6) \\
T(6) &: B(8) = A(5) + A(3) + C(5) \\
U(6) &: A(7) = B(15) + 1 \\
S(7) &: A(7) = B(7) + C(7) \\
T(7) &: B(9) = A(6) + A(4) + C(6) \\
U(7) &: A(8) = B(17) + 1
\end{aligned}$$

The dependence graph for this program segment is shown in Fig. 9.3. In this figure, an anti-dependence edge has a cross on it and an output dependence edge has a small circle on it.

Statement T is flow dependent on statement S . The flow dependence of T on S caused by the output variable $A(I)$ of S and the input variable $A(I - 1)$ of T is the set

$$\{(S(4), T(5)), (S(5), T(6)), (S(6), T(7)), \dots, (S(199), T(200))\}$$

The distance vector for this flow dependence is $\{1\}$. The corresponding direction vector is $\{<\}$. The flow dependence of T on S caused by the output variable $A(I)$ of S and the input variable $A(I - 3)$ of T is the set

$$\{(S(4), T(7)), (S(5), T(8)), (S(6), T(9)), \dots, (S(197), T(200))\}$$

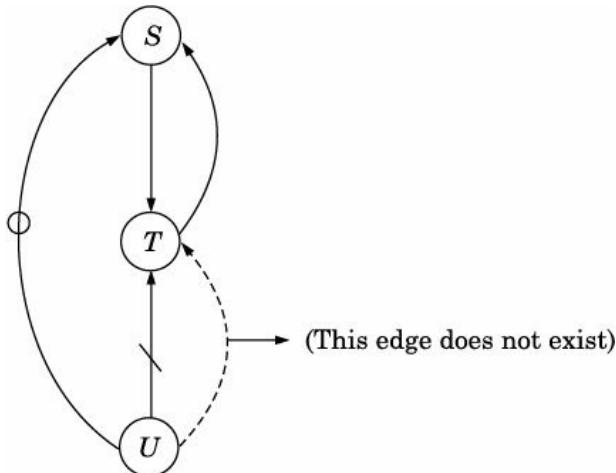


Figure 9.3 Dependence graph.

The distance vector for this flow dependence is $\{3\}$. The corresponding direction vector is $\{<\}$. Statement S is also flow dependent on statement T . This flow dependence is caused by the output variable $B(I + 2)$ of T and the input variable $B(I)$ of S , and is given by

$$\{(T(4), S(6)), (T(5), S(7)), (T(6), S(8)), \dots, (T(198), S(200))\}$$

The distance vector for this flow dependence is $\{2\}$. The corresponding direction vector is $\{<\}$. Statement S is output dependent on statement U . The output dependence of S on U is the set

$$\{(U(4), S(5)), (U(5), S(6)), (U(6), S(7)), \dots, (U(199), S(200))\}.$$

The distance vector for this flow dependence is $\{1\}$. The corresponding direction vector is $\{<\}$. Statement T is anti-dependent on statement U . We find that the anti-dependence of T on U is the set

$$\{(U(4), T(9)), (U(5), T(11)), (U(6), T(13)), \dots, (U(99), T(199))\}.$$

This dependence has the distances: 5, 6, 7, ..., 100.

Note that the statement T is not flow dependent on statement U . Whenever we have two instances $U(i)$ and $T(j)$, such that $U(i)$ is to be executed before $T(j)$ and they refer to the same memory location, an instance of S writes the same location. For example, $U(4)$ and $T(6)$ refer to the location represented by $A(5)$, but $S(5)$ writes this location and it comes between $U(4)$ and $T(6)$ in the execution of the program.

9.3.5 Dependence Equation

This section considers the mathematical problem of computing the dependence of a statement on another statement, caused by a pair of array-element variables. We consider only elements of arrays; scalars can easily be included as a degenerate special case. The dependence equation is first derived for statements having only single dimensional array-element variables and the concept is then extended to statements involving multi-dimensional array-element variables. Another assumption made in this analysis is that all array subscripts are linear functions of one or more index variables (this is the case in many real programs). The analysis is with respect to a perfect nest of d loops. Here the loop bounds l_1 and u_1 are integer constants; l_r and u_r are integer-valued functions of i_1, i_2, \dots, i_{r-1} for $1 < r \leq d$. Since we are assuming that the variables in the program are array-elements with subscripts linear in i_1, i_2, \dots, i_d , a variable that is an element of a one-dimensional array X has the form

$$X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$$

where the coefficients are all integer constants. This can be written as $X(\mathbf{IA} + \mathbf{a}_0)$, where

$$\mathbf{A} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{pmatrix}$$

$$\mathbf{I} = (i_1, i_2, \dots, i_d) \text{ and } \mathbf{a}_0 = (a_0)$$

Let S and T denote two statements in the loop such that $X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$ is a variable in S and $X(b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0)$ is a variable in T . The instance of $X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$ and the instance of $X(b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0)$ will represent the same memory location iff

$$a_1i_1 + a_2i_2 + \dots + a_di_d + a_0 = b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0$$

This equation is called the *dependence equation* for the variables $X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$ of S and $X(b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0)$ of T . It can be seen that the dependence equation

has $2d$ variables.

Consider the example below which finds the dependence equation for the pair of statements S and T in the following double loop:

```

L1: for  $I_1 := 0$  to 100 do
L2:   for  $I_2 := 12$  to 300 do
    S :  $X(2I_1 + 4I_2 - 1) = \dots$ 
    T :  $\dots = \dots X(4I_1 + 2I_2 + 6)$ 
        end for
    end for

```

Since the array X is one-dimensional and the number of loops in the loop nest is 2 (i.e., $d = 2$), the dependence equation is a single equation in four (i.e., $2d$) variables:

$$\begin{aligned} 2i_1 + 4i_2 - 1 &= 4j_1 + 2j_2 + 6 \\ 2i_1 + 4i_2 - 4j_1 - 2j_2 &= 7 \end{aligned} \tag{1}$$

The above idea of a dependence equation can be extended to statements where variables can be array-elements of multiple dimensions. A variable that is an element of an n -dimensional array has the form

$$X(a_{11}i_1 + \dots + a_{d1}i_d + a_{10}, \dots, a_{1n}i_1 + \dots + a_{dn}i_d + a_{n0})$$

where the coefficients are all integer constants. In matrix notation, this variable can be written a $X(\mathbf{IA} + \mathbf{a}_0)$, where

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \dots & \mathbf{a}_{1n} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \dots & \mathbf{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_{d1} & \mathbf{a}_{d2} & \dots & \mathbf{a}_{dn} \end{bmatrix}$$

$$\mathbf{I} = (i_1, i_2, \dots, i_d) \text{ and } \mathbf{a}_0 = (a_{10}, a_{20}, \dots, a_{n0})$$

Let S and T denote two statements in the loop such that $X(\mathbf{IA} + \mathbf{a}_0)$ is a variable in S and $X(\mathbf{IB} + \mathbf{b}_0)$ is a variable in T , where X is an n -dimensional array, \mathbf{A} and \mathbf{B} are $d \times n$ integer matrices, and \mathbf{a}_0 and \mathbf{b}_0 are integer n -vectors. The instance of $X(\mathbf{IA} + \mathbf{a}_0)$ for an index value \mathbf{i} is $X(\mathbf{iA} + \mathbf{a}_0)$ and the instance of $X(\mathbf{IB} + \mathbf{b}_0)$ for an index value \mathbf{j} is $X(\mathbf{jB} + \mathbf{b}_0)$. They will represent the same memory location iff

$$\mathbf{iA} + \mathbf{a}_0 = \mathbf{jB} + \mathbf{b}_0 \tag{2}$$

$$\mathbf{iA} - \mathbf{jB} = \mathbf{b}_0 - \mathbf{a}_0 \tag{3}$$

This equation is the *dependence equation* for the variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$. It consists of n scalar equations in $2d$ variables (the d components of \mathbf{i} and the d components of \mathbf{j}). This equation can also be written in the form

$$(\mathbf{i}, \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b}_0 - \mathbf{a}_0 \tag{4}$$

where (\mathbf{i}, \mathbf{j}) is the $2d$ -vector obtained by adding the components of \mathbf{j} to the components of \mathbf{i} . Equation (4) represents a set of n linear Diophantine equations in $2d$ variables. The two variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$ will cause a dependence between the statements S and T iff there are index points \mathbf{i} and \mathbf{j} satisfying this set of linear Diophantine equations. We now present a basic test for dependence.

9.3.6 GCD Test

The GCD test is a general and approximate test used to determine whether the variable $X(\mathbf{IA} + \mathbf{a}_0)$ of a statement S and the variable $X(\mathbf{IB} + \mathbf{b}_0)$ of a statement T cause a dependence of S on T (or of T on S). First we will describe the test considering only single dimensional array-element variables. The one-dimensional version of the GCD test is given below:

Let $X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$ denote a variable of a statement S and $X(b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0)$ denote a variable of statement T , where X is a one-dimensional array. If $\gcd(a_1, a_2, \dots, a_d, b_1, b_2, \dots, b_d)$ does not divide $(b_0 - a_0)$, then the variables do not cause a dependence between S and T .

The dependence equation for $X(a_1i_1 + a_2i_2 + \dots + a_di_d + a_0)$ and $X(b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0)$ is

$$a_1i_1 + a_2i_2 + \dots + a_di_d + a_0 = b_1j_1 + b_2j_2 + \dots + b_dj_d + b_0$$

This reduces to

$$a_1i_1 + a_2i_2 + \dots + a_di_d - b_1j_1 - b_2j_2 - \dots - b_dj_d = b_0 - a_0 \quad (5)$$

The linear Diophantine equation

$$a_1x_1 + a_2x_2 + \dots + a_mx_m = c$$

in m variables, with integer coefficients: a_1, a_2, \dots, a_m , not all zero, and an integer right-hand side c has a solution iff the gcd of its coefficients divides c i.e., there exists an integer t such that

$$t \times \gcd(a_1, a_2, \dots, a_m) = c \quad (6)$$

The one-dimensional version of the GCD test follows directly from Eqs. (5) and (6).

Let us apply the GCD test to determine if any dependence exists between statements S and T in the previous example. The dependence equation is given by Eq. (1) as

$$2i_1 + 4i_2 - 4j_1 - 2j_2 = 7$$

The GCD of coefficients on the left-hand side is 2. Since 2 does not divide 7, it follows from the GCD test that there is no dependence between S and T .

The GCD test can be extended to statements involving multi-dimensional array-element variables. Before we present this extension, we need to define two types of matrices. A square integer matrix \mathbf{A} is said to be *unimodular* if the determinant of \mathbf{A} has modulus 1. For an $m \times n$ matrix \mathbf{B} , let l_i denote the column number of the leading (first non-zero) element of row i (for a zero row, l_i is undefined). Then, \mathbf{B} is an *echelon* matrix if for some integer p in $0 \leq p \leq m$, the following conditions hold:

- The rows 1 through p are non-zero rows.
- The rows $p + 1$ through m are zero rows.
- For $1 \leq i \leq p$, each element in column l_i below row i is zero.
- $l_1 < l_2 < \dots < l_p$.

The dependence equation in the case of statements involving multi-dimensional array-element variables has been shown in Eq. (4) to be of the form

$$(\mathbf{i}, \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b}_0 - \mathbf{a}_0 \quad (7)$$

This can be reduced to the form

$$\mathbf{U} \cdot \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S} \quad (8)$$

where \mathbf{U} is a $2d \times 2d$ unimodular matrix and \mathbf{S} is $2d \times n$ echelon matrix. The GCD test for statements involving multi-dimensional array variables can now be stated as

if there is no integer vector \mathbf{t} of size $1 \times 2d$ satisfying

$$\mathbf{t}\mathbf{S} = \mathbf{b}_0 - \mathbf{a}_0 \quad (9)$$

then the two variables $X(\mathbf{IA} + \mathbf{a}_0)$ of S and $X(\mathbf{IB} + \mathbf{b}_0)$ of T do not cause a dependence of T on S or of S on T .

Consider the example below to check whether a dependence exists between statements S and T which involve multi-dimensional array variables.

```

L1 : for I1 := 0 to 50 do
L2 :   for I2 := 12 to I1 + 60 do
      S : X(2I1 + 2I2, I1 + 4I2) = ...
      T : ... = ...X(500 - 2I1 - 4I2, 600 - I1 - 5I2)
    end for
  end for

```

Here, we have

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 2 & 4 \end{pmatrix}, \mathbf{a}_0 = (0, 0), \mathbf{B} = \begin{pmatrix} -2 & -1 \\ -4 & -5 \end{pmatrix}, \mathbf{b}_0 = (500, 600) \quad (10)$$

The dependence equation for the two variables of S and T is given by Eq. (4) as

$$(\mathbf{i}_1, \mathbf{i}_2, \mathbf{j}_1, \mathbf{j}_2) \begin{pmatrix} 2 & 1 \\ 2 & 4 \\ 2 & 1 \\ 4 & 5 \end{pmatrix} = (500, 600) \quad (11)$$

This can be reduced to the form (see Exercise 9.7)

$$\mathbf{U} \cdot \begin{pmatrix} 2 & 1 \\ 2 & 4 \\ 2 & 1 \\ 4 & 5 \end{pmatrix} = \mathbf{S} \quad (12)$$

where

$$\mathbf{U} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & -1 \end{pmatrix} \text{ and } \mathbf{S} = \begin{pmatrix} 2 & 1 \\ 0 & 3 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad (13)$$

Here \mathbf{U} is unimodular and \mathbf{S} is echelon. The general solution to the equation

$$(t_1, t_2, t_3, t_4) \mathbf{S} = (500, 600) \quad (14)$$

is $(250, 350/3, t_3, t_4)$ where t_3 and t_4 are undetermined. Since there is no integer solution, the GCD test implies that there is no dependence between statements S and T .

9.4 TRANSFORMATIONS

A major goal of optimizing compilers for parallel computers is to detect or increase parallelism in loops, since generally most of the execution time is spent in loops. We now present several loop transformations and also some standard ones which have been systematized and automated.

9.4.1 Data Flow Based Loop Transformations

Several loop optimizations are based on data-flow analysis, which tracks the flow of data through a program's variables.

Loop Based Strength Reduction

This technique essentially aims at replacing an expensive operation in a loop with an equivalent less expensive one by intelligently making use of a temporary variable. The following examples illustrate this technique:

```
1.  for i := 1 to n do
    a[i] = a[i] + c * i
end for
```

If the above is the code to be optimized, then after strength reduction the code would appear as follows:

```
T = c
for i := 1 to n do
    a[i] = a[i] + T
    T = T + c
end for
```

As seen above, the expensive multiplication operation has been reduced to an addition in each loop iteration. The other examples are similar to this one.

2. Unoptimized code:

```
for i := 1 to n do
    a[i] = a[i] + c ** i
end for
```

Optimized code:

```
T = c
for i := 1 to n do
    a[i] = a[i] + T
    T = T * c
end for
```

3. Unoptimized code:

```
for i := 1 to n do
    a[i] = a[i] + (-1)**i
end for
```

Optimized code:

```
T = -1
for i := 1 to n do
    a[i] = a[i] + T
    T = -T
end for
```

4. Unoptimized code:

```
for i := 1 to n do
    a[i] = a[i] + x[i]/c
end for
```

Optimized code:

```
T = 1/c
for i := 1 to n do
    a[i] = a[i] + T * x[i]
end for
```

A variable whose value depends on the number of iterations is termed as *induction variable*. Strength reduction is basically applied on induction variables and this is its most common use. Strength reduction is also a term applied to optimizations, in non-loop contexts, such as replacing $x * 2$ with $x + x$. This method is effective particularly in machines in which multiplication takes significantly more cycles than addition.

Loop Invariant Code Motion

This optimization attempts to move parts of code, which are within a loop but are invariant with each iteration (i.e., their result does not change between iterations), to the outside of the loop so that their executions take place exactly once for the entire loop. This technique can be applied at the higher level to expressions in the source code or at the assembly level to address computations. The following simple example depicts code motion:

```
for i := 1 to n do
    a[i] = a[i] + sqrt(x)
end for
```

After code motion we have:

```
if (n > 0) then c = sqrt(x)
end if
for i := 1 to n do
    a[i] = a[i] + c
end for
```

Here the costly *sqrt* function is called just once in the optimized code as opposed to n times in the original code. Also, unless the loop has at least one iteration, the *sqrt* function is not invoked.

Loop Unswitching

This optimization is applied when a loop contains a conditional statement with a loop-

invariant test condition. The loop is then replicated inside the *if* and *then* parts of the conditional statement thereby (i) saving the overhead of conditional branching inside the loop, (ii) reducing the code size of the loop body, and (iii) possibly enabling the parallelization of *if* or *then* part of the conditional statement. This technique is illustrated below:

```
for i := 2 to n do
    a[i] = a[i] + e
    if (x < 7) then
        b[i] = a[i] * c[i]
    else
        b[i] = a[i-1] * b[i-1]
    end if
end for
```

The loop unswitching transformation when applied to the above code results in

```
if (n > 1) then
    if (x < 7) then
        for i := 2 to n do in parallel
            a[i] = a[i] + e
            b[i] = a[i] * c[i]
        end for
    else
        for i := 2 to n do
            a[i] = a[i] + e
            b[i] = a[i-1] * b[i-1]
        end for
    end if
end if
```

Conditionals that are candidates for unswitching can be spotted during the analysis for code motion, which identifies loop invariants.

9.4.2 Loop Reordering

We now describe transformations that change the relative order of execution of the iterations of a loop nest or nests. These transformations aid in exposing parallelism and improving the locality of memory.

The compiler determines whether different iterations of a loop can be executed in parallel by examining the loop-carried dependences. As discussed earlier, these dependences can be represented by distance vectors. When all the dependence distances for a loop are 0, there are no dependences carried across iterations of the loop, and thus the loop can be executed in parallel. The following examples reinforce the concept of loop-carried dependences and the parallelizability of a loop.

1. Consider the following piece of code:

```
for i := 1 to n do
    for j := 2 to n do
        a[i,j] = a[i,j-1] + c
    end for
end for
```

Here the distance vector for the loop is $(0, 1)$ and hence the outer loop is parallelizable as shown below:

```
for i := 1 to n do in parallel
    for j := 2 to n do
        a[i, j] = a[i, j-1] + c
    end for
end for
```

2. Now consider another example:

```
for i := 1 to n do
    for j := 1 to n do
        a[i, j] = a[i-1, j] + a[i-1, j+1]
    end for
end for
```

Here the distance vectors are $(1, 0)$, $(1, -1)$, so the inner loop is parallelizable as shown below:

```
for i := 1 to n do
    for j := 1 to n do in parallel
        a[i, j] = a[i-1, j] + a[i-1, j+1]
    end for
end for
```

Loop Interchange

In a perfect loop nest, a loop interchange exchanges the position of two loops. This transformation has several advantages:

- vectorization is facilitated by interchanging an inner, dependent loop with an outer, independent loop,
- vectorization is improved when an independent loop with the largest range is moved into the innermost position,
- parallel performance is enhanced when an independent loop is moved outward in a loop nest to increase the granularity of each iteration as well as to reduce the number of barrier synchronizations,
- stride is reduced, ideally to 1 and
- number of loop invariant expressions in the inner loop is increased.

The above advantages, simple as they may seem, can be extremely tricky to handle since one of the above benefits can cancel another. For example, an interchange that improves register usage may change a stride-1 access pattern to a stride-n access pattern. Consider the following piece of code.

```
for i := 1 to n do
    for j := 1 to n do
        total[i] = total[i] + a[i, j]
    end for
end for
```

Assuming (FORTRAN) column-major storage of the array, the inner loop accesses array a with stride n , i.e., it accesses memory locations which are n apart. This increases the cache

misses which results in lower overall performance. By interchanging the loops, as shown below, we convert the inner loop to stride-1 access.

```
for j := 1 to n do
    for i := 1 to n do
        total[i] = total[i] + a[i,j]
    end for
end for
```

However, the original code, unlike the optimized code, allows `total[i]`, to be placed in a register thereby eliminating the load/store operations in the inner loop. Hence, if the array `a` fits in the cache, the original code is better.

Loop Skewing

This transformation, together with loop interchange, is typically used in loops which have array computations called *wavefront computations*, so called because the updates to the array go on (propagate) in a cascading (wave) fashion. Consider the following *wavefront computation*:

```
for i := 2 to n-1 do
    for j := 2 to m-1 do
        a[i,j] = (a[i-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1])/4
    end for
end for
```

This is an example of *stencil* (nearest-neighbor) *computation*. Stencil computations represent an important class of computations that arise in many scientific and engineering codes. In this example, each array element is computed (or updated according to some fixed pattern called *stencil*) by averaging its four surrounding elements. The two loops in this case, as can be observed easily, cannot be parallelized in their present form. But each array diagonal (“wavefront”) can be computed in parallel. To realize this, we apply the transformation loop skewing. Skewing is performed by adding the outer loop index multiplied by a *skew factor*, f , to the bounds of the inner iteration variable, and then subtracting the same quantity from every use of the inner iteration variable inside the loop. The modified loop is as follows:

```
for i := 2 to n-1 do
    for j := i+2 to i+m-1 do
        a[i,j-i] = (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4
    end for
end for
```

Observe that the transformed code is equivalent to the original, but the effect on the iteration space is to align the diagonal wavefronts of the original loop nest so that for a given value of j , all iterations in i can be executed in parallel. To expose this parallelism, the skewed loop nest must also be interchanged as shown below:

```
for j := 4 to m+n-2 do
    for i := max(2,j-m+1) to min(n-1,j-2) do
        a[i,j-i] = (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4
    end for
end for
```

Loop Reversal

Reversing the direction (indices) of a loop is normally used in conjunction with other iteration space reordering transformations because it changes the distance vector. The change in the distance vector sometimes facilitates loop interchange thus possibly optimizing it. Consider the following example:

```
for i := 1 to n do
    for j := 1 to n do
        a[i,j] = a[i-1,j+1] + 1
    end for
end for
```

The above loop nest has the distance vector $(1, -1)$. The reversed loop nest is as follows:

```
for i := 1 to n do
    for j := n downto 1
        a[i,j] = a[i-1,j+1] + 1
    end for
end for
```

Here the distance vector is $(1, 1)$, which means that the loops may be interchanged now.

Strip Mining

In a vector machine, when a vector has a length greater than that of the vector registers, segmentation of the vector into fixed size *segments* is necessary. This technique is called strip mining. One vector segment (one surface of the mine field) is processed at a time. In the following example, the iterations of a serial loop are converted into a series of vector operations, assuming the length of vector segment to be 64 elements. The strip-mined computation is expressed in array notation, and is equivalent to a **do in parallel** loop.

The original code:

```
for i := 1 to n do
    a[i] = a[i] + C
end for
```

After strip mining:

```
TN = (n/64)*64
for TI := 1 to TN with step size of 64 do
    a[TI:TI+63] = a[TI:TI+63] + C
end for
for i := TN + 1 to n do
    a[i] = a[i] + C
end for
```

Cycle Shrinking

This is just a special case of strip mining. It converts a serial loop into an outer loop and an inner parallel loop. Consider the following example. The result of an iteration of the loop is used only after k iterations and consequently the first k iterations can be performed in parallel after which the same is done for the next k iterations and so on. This results in a speedup of k but since k is quite likely to be small (usually 2 or 3), this optimization is primarily used for exposing fine-grained (instruction level) parallelism.

Before cycle shrinking:

```
for i := 1 to n do
    a[i+k] = b[i]
    b[i+k] = a[i] + c[i]
end for
```

After cycle shrinking:

```
for TI := 1 to n with step size of k do
    for i := TI to TI+k-1 do in parallel
        a[i+k] = b[i]
        b[i+k] = a[i] + c[i]
    end for
end for
```

Loop Tiling

Tiling is the multi-dimensional generalization of strip mining. It is mainly used to improve the cache reuse by dividing an iteration space into tiles and transforming the loop nest to iterate over them. It can also be used to improve processor, register, TLB (translation lookaside buffer or simply translation buffer is a separate and dedicated memory cache that stores most recent translations of virtual memory to physical addresses for faster retrieval of instructions and data) or page locality. The following example shows the division of a matrix into tiles. This sort of division is critical in dense matrix multiplication for achieving good performance.

Original loop:

```
for i := 1 to n do
    for j := 1 to n do
        a[i,j] = b[j,i]
    end for
end for
```

Tiled loop:

```
for TI := 1 to n with step size of 64 do
    for TJ := 1 to n with step size of 64 do
        for i := TI to min(TI + 63, n) do
            for j := TJ to min(TJ + 63, n) do
                a[i,j] = b[j,i]
            end for
        end for
    end for
end for
```

Figure 9.4 shows the iteration dependence graphs (iteration spaces) for some example loop nests that illustrate loop interchange, loop skewing and loop tiling transformations.

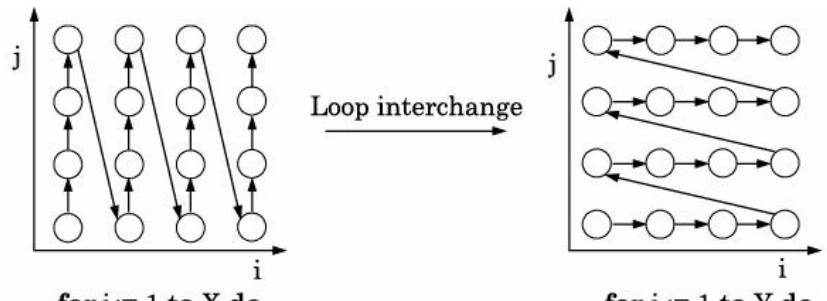
Loop Distribution

Distribution (also called *loop fission* or *loop splitting*) breaks a loop into two or more. Each of the new loops has the same iteration space as the original but with fewer statements. Distribution is used to

- create perfect loop nests,

- create smaller loops and reduce the number of dependences overall,
- improve the cache performance as a consequence of shorter loop bodies,
- reduce memory requirements by iterating over fewer arrays and
- increase register reuse.

The following is an example in which distribution removes dependences and allows a part of a loop to be executed in parallel. Distribution can be applied to any loop, but all statements belonging to a dependence cycle (i.e., all the statements which are dependent on each other) should be executed in the same loop. Also the flow sequence should be maintained.



```

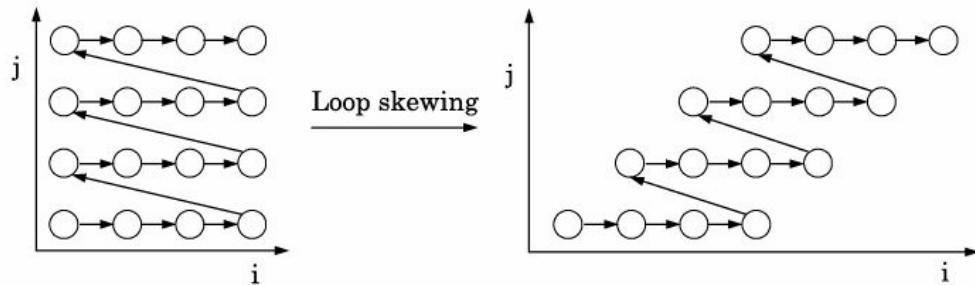
for i := 1 to X do
  for j := 1 to Y do
    a[i, j] = ...
  end for
end for

```

```

for j := 1 to Y do
  for i := 1 to X do
    a[i, j] = ...
  end for
end for

```



```

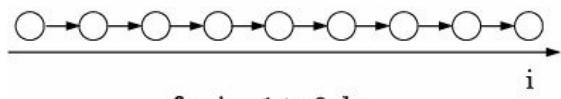
for j := 1 to Y do
  for i := 1 to X do
    a[i, j] = ...
  end for
end for

```

```

for j := 1 to Y do
  for i := 1 + j to X + j do
    a[i-j, j] = ...
  end for
end for

```

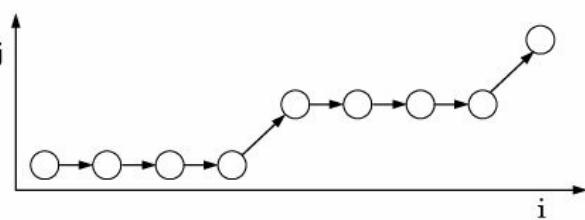


```

for i := 1 to 9 do
  a[i] := ...
end for

```

Loop tiling



```

for j := 1 to 3 do
  for i := 4*(j-1) + 1 to 4*(j-1) + 4 do
    if i = 9 then a[i] = ...
    end if
  end for
end for

```

Figure 9.4 Examples of loop transformations.

Original code:

```
for i := 1 to n do
    a[i] = a[i] + c
    x[i+1] = x[i] * 7 + x[i+1] + a[i]
end for
```

After loop distribution:

```
for i := 1 to n do in parallel
    a[i] = a[i] + c
end for
for i := 1 to n do
    x[i+1] = x[i] * 7 + x[i+1] + a[i]
end for
```

Loop Fusion

This is the inverse of loop distribution and is also called *jamping*. It combines two or more independent loops. In the above example, distribution enables the parallelization of part of the loop while fusion improves register and cache locality as $a[i]$ need be loaded only once. Further, fusion reduces the loop control overhead, which is useful in both vector and parallel machines, and coalesces serial regions, which is useful in parallel execution. With large n (number of iterations), the distributed loop should run faster on a vector machine while the fused loop should be better on a superscalar machine.

9.4.3 Loop Restructuring

We now describe loop transformations that change the structure of the loop, but leave the computations performed by an iteration of the loop body and their relative order unchanged.

Loop Unrolling

Unrolling replicates the body of a loop u (*unrolling factor*) number of times and iterates by step u instead of 1. It is a fundamental technique for generating long instruction sequences required by VLIW machines. Unrolling can improve the performance by

- reducing loop overhead,
- increasing instruction level parallelism and
- improving register, data cache, or TLB locality.

The following example illustrates the usefulness of loop unrolling.

```
for i := 2 to n-1 do
    a[i] = a[i] + a[i-1] * a[i+1]
end for
```

A loop-unrolled version of this loop with the loop unrolled twice is:

```

for i := 2 to n-2 with step size of 2 do
    a[i] = a[i] + a[i-1] * a[i+1]
    a [i+1] = a [i+1] + a[i] * a[i+2]
end for
if (mod(n-2, 2) = 1) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if

```

Here the loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Instruction level parallelism is enhanced because the second assignment can be performed while the results of the first are being stored.

Software Pipelining

This refers to the pipelining of successive iterations of a loop in the source code. Here, the operations of a single iteration are broken into s stages, and a single iteration performs stage 1 from iteration i , stage 2 from iteration $i - 1$, and so on. Software pipelining is illustrated with the execution details of the following loop on a two-issue processor first without software pipelining and then with software pipelining.

```

for i := 1 to n do
    a[i] = a[i] * b + c
end for

```

Here, there is no loop-carried dependence. Let us assume for the analysis that each memory access (*Read* or *Write*) takes one cycle and each arithmetic operation (*Mul* and *Add*) requires two cycles. Without pipelining, one iteration requires six cycles to execute as shown below:

Cycle	Instruction	Comment
1	Read	Fetch a[i]
2	Mul	Multiply by b
3		
4	Add	Add to c
5		
6	Write	Store a[i]

Thus n iterations require $6n$ cycles to complete, ignoring the loop control overhead. Shown below is the execution of four iterations of the software-pipelined code on a two-issue processor. Although each iteration requires 8 cycles to flow through the pipeline, the four overlapped iterations require only 14 clock cycles to execute thereby achieving a speedup of $24/14 = 1.7$.

<i>Cycle</i>	<i>Iteration</i>			
	1	2	3	4
1	Read			
2	Mul			
3		Read		
4		Mul		
5	Add		Read	
6			Mul	
7		Add		Read
8	Write			Mul
9			Add	
10		Write		
11				Add
12			Write	
13				
14				Write

Loop Coalescing

Coalescing combines a loop nest into a single loop, with the original indices computed from the resulting single induction variable. It can improve the scheduling of the loop on a parallel machine and may also reduce loop overhead. An example for loop coalescing is given below. The original loop is:

```
for i := 1 to n do in parallel
    for j := 1 to m do in parallel
        a[i,j] = a[i,j] + c
    end for
end for
```

Here, if n and m are slightly larger than the number of available processors P , say $n = m = P + 1$, then by scheduling any of the loops (inner or outer) first, it will take $2n$ units of time. This is because the P processors can update the first $P = n - 1$ rows in parallel in n units of time, with each processor handling one row. And to update the last row a further n units of time is required. The coalesced form of the loop is shown below:

```
for T := 1 to n * m do in parallel
    i = ((T - 1) / m) * m + 1
    j = (T - 1) mod m + 1
    a[i,j] = a[i,j] + c
end for
```

This loop can be executed by P processors in $n * m/P$ units of time, which is approximately half the time required in the original case.

Loop Collapsing

Collapsing is a simpler, more efficient version of coalescing in which the number of dimensions of the array is actually reduced. The performance benefits of collapsing are the following:

- it eliminates the overhead of nested loops,

- it increases the number of parallelizable loop iterations and
- it increases vector lengths.

The collapsed version of the loop of the previous example is as follows:

```
real TA[n * m]
equivalence (TA,a)
for T := 1 to n * m do in parallel
    TA[T] = TA[T] + c
end for
```

The two-dimensional array $a[n, m]$ is replaced by a uni-dimensional array $TA[n*m]$ (this is indicated by the *equivalence* statement), reducing loop overhead and increasing potential parallelization. Collapsing is best suited to loop nests that iterate over memory with a constant stride. When more complex indexing is involved, coalescing may be a better choice.

9.4.4 Loop Replacement Transformations

Here we describe transformations that operate on whole loops and completely change their structure.

Reduction Recognition

Reduction operation computes a scalar value from an array, such as max, min or the sum of the elements. In the following example, the loop which performs the sum of the elements is a serial loop with a dependence vector of (1). This is parallelized by computing partial sums in parallel. The crucial factor exploited in this case is the associativity of the operator. Commutativity is yet another useful property which can be exploited in reductions. In the case of semi-commutativity and semi-associativity operators (such as float multiplication), the reduction technique varies depending on the programmer's intent and the semantics of the language. Ideally, partial sum calculations should be done in a tree-like fashion which reduces its complexity to $O(\log n)$ from $O(n)$. Similarly, operations such as max and min can be parallelized.

Original code:

```
for i := 1 to n do
    s = s + a[i]
end for
```

Transformed code:

```
real TS[64]
TS[1:64] = 0.0
for TI := 1 to n with step size of 64 do
    TS[1:64] = TS[1:64] + a[TI:TI+63]
end for
for TI := 1 to 64 do
    s = s + TS[TI]
end for
```

Loop Idiom Recognition

This transformation by the compiler is aimed at exploiting special features of the hardware. For example, SIMD machines generally support reduction directly in the processor interconnection network. Some parallel machines apart from reduction also support parallel

prefix operations, in hardware, which allow a loop of the form $a[i] = a[i - 1] + a[i]$ to be parallelized. In general, linear recurrences can be parallelized using parallel prefix operations. The compiler here identifies and converts these special idioms that are specially supported by hardware.

Array Statement Scalarization

Here a loop expressed in array notation is scalarized to give one or more serial loops. This conversion is slightly tricky as can be seen in the following example. Normally a temporary array is used for this purpose. But this can be eliminated if the resulting loops can be fused.

Initial code:

```
a[2:n-1] = a[2:n-1] + a[1:n-2]
```

Scalarized code:

```
for i := 2 to n-1 do
    T[i] = a[i] + a[i-1]
end for
for i := 2 to n-1 do
    a[i] = T[i]
end for
```

After fusion of the two loops:

```
for i := n-1 downto 2
    a[i] = a[i] + a[i-1]
end for
```

9.4.5 Memory Access Transformations

So far we have discussed how a compiler can speed up computations using a host of transformations predominantly on the loops (where generally most of the execution time is spent) thus reducing the processing time. There is yet another aspect that the compiler has to optimize—the use of memory system. The fact that processor speed improvements outpaced memory (DRAM) speed improvements several times can only over-emphasize this. (DRAMs, or dynamic random access memory chips, are the low-cost, low-power memory chips used for main memory in most computers.) Thus memory access optimization is as vital an issue as processing optimization.

Array Padding

In general, in vector machines and shared memory multiprocessors, the memory system is organized into a number of memory banks that can be accessed in parallel (or interleaved). Access to memory usually takes several CPU cycles. Thus attempts to access the same memory bank in the same or consecutive cycle(s) cannot be satisfied. It is therefore advantageous to arrange data so that access to data occurs across memory banks rather than a single bank. Bank conflicts can be caused when the program indexes through an array dimension that is not laid out contiguously in memory.

Consider the following loop which accesses memory with stride 8. If the number of banks is 8, then all memory references are to the first bank resulting in bank conflicts. If an additional unused data location is inserted between the columns of the array, which is called padding, then successive iterations access memory with stride 9 i.e., they go to successive banks.

Original code (Note that a is stored in column major order):

```
real a[8,512]
for i := 1 to 512 do
    a[1,i] = a[1,i] + c
end for
```

After padding:

```
real a[9, 512]
for i := 1 to 512 do
    a[1,i] = a[1,i] + c
end for
```

However, array padding increases memory consumption and makes the subscript calculations for operations over the entire array more complex, as the array has “holes”.

Scalar Expansion

Temporary variables in a loop, as shown in the example below, can create an anti-dependence between the iterations thus preventing the loop from being parallelized. This dependence can be removed by allocating one temporary location for each iteration. This, scalar expansion, is a fundamental technique used in vectorizing compilers.

Original code:

```
for i := 1 to n do
    c = b[i]
    a[i] = a[i] + c
end for
```

After scalar expansion:

```
real T[n]
for i := 1 to n do in parallel
    T[i] = b[i]
    a[i] = a[i] + T[i]
end for
```

9.4.6 Partial Evaluation

Partial evaluation consists of performing part of a computation at compile time.

Constant Propagation

This is a very important optimization technique which involves propagating constants through a program and doing a significant amount of precomputation. Consider the following example:

Original code:

```
n = 64
c = 3
for i := 1 to n do
    a[i] = a[i] + c
end for
```

After constant propagation:

```
for i := 1 to 64 do
    a[i] = a[i] + 3
end for
```

Knowing the range of the loop, the compiler can be more accurate in applying the loop optimizations.

Constant Folding

Constant folding is normally coupled with constant propagation. Here we go one step beyond just propagation and replace a simple expression (containing an operation with constant values as operands) such as $x = 2 * 3$ with $x = 6$.

Copy Propagation

Optimizations such as induction variable elimination and common sub-expression elimination may cause the same value to be copied several times. The compiler can propagate the original name of the value and eliminate redundant copies. This reduces *register pressure* (i.e., the need for more registers than are available) and eliminates redundant register-to-register move instructions. Here is an example to illustrate this:

Original code:

```
t = i * 4
s = t
print*, a[s]
r = t
a[r] = a[r] + c
```

After copy propagation:

```
t = i * 4
print*, a[t]
a[t] = a[t] + c
```

Forward Substitution

This is a generalization of copy propagation, in that it replaces a variable by the expression corresponding to it. Consider the following example. The (original) loop cannot be parallelized because an unknown element of a is being written. On the other hand, the optimized loop can be implemented as a parallel reduction.

Original code:

```
npl = n+1
for i := 1 to n do
    a[npl] = a[npl] + a[i]
end for
```

After forward substitution:

```
for i := 1 to n do in parallel
    a[n+1] = a[n+1] + a[i]
end for
```

Reassociation

This is a technique to increase the number of common sub-expressions in a program. It is generally applied to address calculations within loops when performing strength reduction on induction variable expressions. Address calculations generated by array references involve several multiplications and additions. Reassociation applies the associative, commutative, and distributive laws to rewrite these expressions in a canonical sum-of-products form.

Algebraic Simplification and Strength Reduction

The compiler, by applying algebraic rules, can simplify arithmetic expressions. Here are some of the commonly applied rules.

$$\begin{aligned}x \times 0 &= 0 \\0 / x &= 0 \\x \times 1 &= x \\x + 0 &= x \\x / 1 &= x\end{aligned}$$

As mentioned earlier, the compiler can replace an expensive operator (such as $x \times 2$ and x^2) with an equivalent less expensive operator ($x + x$ and $x \times x$, respectively). This is called strength reduction.

9.4.7 Redundancy Elimination

When a computation is performed repeatedly and needlessly it is termed redundant. Eliminating redundancies is an important part of compiler optimization. Redundancy-eliminating transformations remove two kinds of computations. They are *unreachable* and *useless*. An unreachable computation is never executed and removing it from the program will have no semantic effect on the program's execution. On the other hand, a computation is useless if none of the outputs in the program is dependent on it.

Unreachable-Code Elimination

There are mainly two ways in which code becomes unreachable. If a conditional predicate is known to be true or false, one branch of the conditional is never taken, thus making its code unreachable. The other common source of unreachable code is a loop which goes through no iterations. In a program which uses *goto* statements, to transfer control, unreachable code is not obvious from the program structure but can be detected by traversing the control flow graph of the program. Normally unreachable and useless code arise after constant propagation. Sometimes considered as a separate step, *redundant control elimination* removes control constructs such as loops and conditionals when they become redundant (usually as a result of constant propagation).

Original code:

```
integer c, n, test  
test = 0  
n = 0  
e = b+7  
if (test > 1) then  
    c = e + b + d  
end if  
compute(e)  
for i := 1 to n do  
    a[i] = a[i] + c  
end for
```

After constant propagation and unreachable code elimination:

```
integer c, n, test  
test = 0  
n = 0
```

```

e = b+7
if (0 > 1) then
end if
compute(e)
for i := 1 to 0
end for

```

After redundant control elimination:

```

integer c, n, test
test = 0
n = 0
e = b+7
compute(e)

```

Useless Code Elimination

Useless code is often created by other optimizations such as unreachable-code. When the compiler realizes that a value computed by a piece of code is not necessary, the code can be eliminated. This transformation when applied to the above code results in the following code.

After useless code elimination:

```

integer c, n, test
e = b+7
compute(e)

```

Dead-Variable Elimination

Variables that are never used surface after a series of compiler transformations, particularly loop optimizations. Eliminating these dead-variables reduces the useless code. Applying this technique on the above piece of code, we get

```

e = b+7
compute(e)

```

Common Sub-Expression Elimination and Short Circuiting

These are two important optimizations and are almost universally performed. The first of these exploits the fact that a sub-expression is used frequently in the code. The compiler computes this once, stores it, and reuses the stored result wherever the sub-expression appears in the code.

Short-circuiting refers to an optimization on boolean expressions based on the fact that many boolean expressions, in most cases, can actually be evaluated based on the value of only the first operand. As an example, consider the boolean expression, *a and b*. If it is known that *a* is false, the expression can be evaluated to false right away without even looking at the value of *b*. This is known as short-circuit.

9.4.8 Procedure Call Transformations

Parallelization techniques can be broadly classified based on their granularity as either coarse-grained or fine-grained. In fine-grained parallelism, the parallelism exploited is at the instruction level, whereas in coarse-grained parallelism, the parallelization is at a much higher level. One potential source of coarse-grained parallelism is the procedure call. With intra-procedure dependence tracking alone, a compiler cannot safely restructure code and exploit procedure level parallelism. A simple method here is to *in-line* a call so that the body of the called procedure is substituted for the call. Then normal intra-procedural dependence

analysis can be performed over the entire procedure, including the in-lined procedure. In-lining is attractive because it is easy to perform. But it has the drawbacks of code explosion and the accompanying long compilation time if performed blindly. The choice of when to inline is often guided by the user.

In cases when in-lining is not applied (or even to determine automatically whether in-lining should be applied), it is desirable to perform inter-procedural analysis. Consider the following piece of code in which a procedure *MAIN* invokes another procedure *FUNC*.

```

procedure MAIN
  for i := 1 to n do
    for j := 1 to n do
      a[i,j] = b[i,j+1] + c
      FUNC(b[i,j-1],c)
    end for
  end for
end

procedure FUNC(x, y)
  if (x < 0) then
    y = -x
  else
    y = x
  end

```

An inter-procedural analysis between the two procedures generally collects the following types of information:

- *alias* information related to the procedure call, such as the fact that the actual parameter at the call site ($b[i, j - 1]$ and c) and the formal parameter in the procedure header (x and y , respectively) refer to the same object, if the call by reference mode is used.
- *change* information, such as a variable nonlocal to a procedure being changed as a result of the procedure invocation, as c is changed as a result of the call to *FUNC*.

In the above example, it is necessary to know whether procedure *FUNC* modifies *b* in order to decide whether iterations of the loop on *i* can proceed concurrently.

Inter-procedural analysis is carried out over a program *call graph*. A node in the call graph denotes a procedure and an edge from node P to node Q denotes that procedure P might call procedure Q during execution. Standard data-flow analysis techniques are used to iterate over the call graph, collecting information on, for example, the definition and use of variables. This information, which is usually flow-insensitive, summarizes the effects of all paths through the program.

9.4.9 Data Layout Transformations

The decomposition of data across processors is one of the key issues that a compiler must address while targeting multiprocessors. The main objective here is to minimize communication and maximize parallelism. The data layout is an important factor affecting performance not only in a distributed memory machine, where the communication is explicit, but also in a shared memory machine, where the communication is implicit.

Array Decomposition

The most common method for decomposing an array on a parallel machine is to map each dimension to a set of processors in a regular pattern. The four commonly used patterns are discussed below:

1. Serial Decomposition

In a serial decomposition, an entire dimension is allocated to a single processor. Consider the following piece of code, to be scheduled on four processors:

```
for j := 1 to n do in parallel
    for i := 1 to n do in parallel
        a[i,j] = a[i,j] + c
    end for
end for
```

The decomposition of the array, a, onto the processors is shown in Fig. 9.5. Each column is mapped serially, i.e., all the data in that column is placed on a single processor. As a result of this, the loop that iterates over the columns (the inner loop) is unchanged.

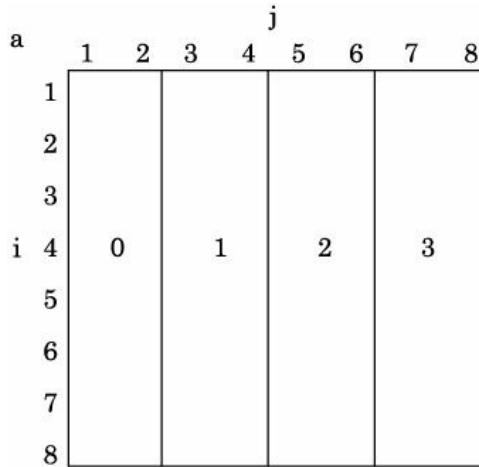


Figure 9.5 Serial decomposition.

2. Block Decomposition

A block decomposition divides the elements into one group of adjacent elements per processor. The block decomposition of the array in the previous example is shown in Fig. 9.6. The advantage of a block decomposition is that adjacent elements are usually on the same processor. Since a loop that computes a value for a given element often uses the values of neighbouring elements, the blocks have good spatial locality and reduce the amount of communication.

a	1	2	3	4	j 5	6	7	8
i 4								
1								
2		0				1		
3								
5								
6		2				3		
7								
8								

Figure 9.6 Block decomposition.

3. Cyclic Decomposition

A cyclic decomposition assigns successive elements to successive processors. A cyclic assignment of columns to processors for the array in the previous example is shown in Fig. 9.7. Cyclic decomposition has the opposite effect of blocking. It has poor locality for neighbour based communication, but spreads load more evenly.

a	1	2	3	4	j 5	6	7	8
i 4	0	1	2	3	0	1	2	3
1								
2								
3								
5								
6								
7								
8								

Figure 9.7 Cyclic decomposition.

4. Block-Cyclic Decomposition

A block-cyclic decomposition combines the two strategies of block decomposition and cyclic decomposition. Here, the elements are divided into many adjacent groups and each of these groups is assigned to a processor cyclically. The block-cyclic decomposition of the array in the previous example on two processors is given in Fig. 9.8. Block-cyclic decomposition is a compromise between the locality of block decomposition and the load balancing of cyclic decomposition.

a	1	2	3	4	5	6	7	8
i	4	0		1		0		1
	1	2	3	4	5	6	7	8

Figure 9.8 Block-cyclic decomposition.

Scalar and Array Privatization

The goal of privatization is to increase the amount of parallelism and to avoid unnecessary communication. When a scalar is used within a loop solely as a scratch variable, each processor can be given a private copy of the variable so that the scalar need not involve any communication. The following piece of code shows a loop that could benefit from privatization. The scalar value c is simply a scratch value and if it is privatized the iterations of the loop become independent.

```

for i := 1 to n do in parallel
    c = b[i]
    a[i] = a[i] + c
end for

```

In array privatization each processor is given its own copy of the entire array. Consider the previous example. Before the execution of the loop, if the arrays are allocated to separate processors (i.e., privatized) then the communication overhead at run time is totally eliminated.

Cache Alignment

To ensure proper synchronization in a shared memory multiprocessor, when multiple processors are updating a single shared variable, any modification of a cache line (or cache block) in one processor results in the flushing of the same cache line in other processors. If several processors are continually updating different variables that reside in the same cache line, the constant flushing of the cache line degrades performance. This problem is called *false sharing*. False sharing of scalars can be addressed by inserting padding between them so that each shared scalar resides in its own cache line. The following example illustrates this.

Consider a shared memory program which involves two arrays a and b each of length 1024 bytes. Suppose the cache line size in the shared memory multiprocessor on which the program is being executed is 2048 bytes. If a processor P_1 modifies only array a and another processor P_2 modifies only array b , and if both these arrays are put in the same cache line, then each modification done by a processor will result in the flushing of the cache line at the other processor resulting in severe performance degradation. This problem of false sharing can be solved by inserting 1024 bytes of padding in between the arrays so that each of them goes to a different cache line.

Computation Partitioning

In addition to decomposing data, a parallel compiler must also allocate the computations of a program to different processors. We now discuss the transformations that seek to enforce correct execution of the program without introducing unnecessary overhead.

Guard Overhead

After a loop is parallelized, all the processors will not compute the same iterations or send the same data. One way a compiler can ensure that correct computations are performed by different processors is by introducing conditional statements known as *guards* into program. Consider the following piece of sequential code.

```
for i := 1 to n do
    a[i] = a[i] + c
    b[i] = b[i] + c
end for
```

A transformed version of this code for parallel execution on P_{num} processors is shown below:

P_{id} denotes the number of each local processor (typically numbered from 0 to $P_{num} - 1$).

```
LBA = (n/Pnum)*Pid + 1
UBA = (n/Pnum)*(Pid + 1)
LBB = (n/Pnum)*Pid + 1
UBB = (n/Pnum)*(Pid + 1)
FORK(Pnum)
for i := 1 to n do
    if (LBA ≤ i and i ≤ UBA) then
        a[i] = a[i] + c
    end if
    if (LBB ≤ i and i ≤ UBB) then
        b[i] = b[i] + c
    end if
end for
JOIN()
```

The above parallelized version of the code computes upper and lower bounds that the guards use to prevent computation on array elements that are not stored locally within the processor. It assumes that n is an even multiple of P_{num} and the arrays are already block distributed across the processors.

Redundant Guard Elimination

A compiler can reduce the number of guards by hoisting them to the earliest point where they can be correctly computed. Hoisting often reveals that identical guards have been introduced and that all but one can be removed. The above parallelized version of the code after redundant guard elimination results in the following:

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
FORK (Pnum)
for i := 1 to n do
    if (LB ≤ i and i ≤ UB) then
        a[i] = a[i] + c
        b[i] = b[i] + c
    end if
end for
JOIN()
```

Bounds Reduction

The guards control which iterations of the loop perform computation. When the desired set of iterations is a contiguous range, the compiler can achieve the same effect by changing the induction expressions to reduce the loop bounds. The following version of the code is obtained after bounds reduction:

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
FORK(Pnum)
for i := LB to UB do
    a[i] = a[i] + c
    b[i] = b[i] + c
end for
JOIN()
```

Communication Optimization

An important part of compilation for distributed memory machines is the analysis of a program's communication requirements and introduction of explicit message passing operations into it. The same fundamental issues arise in communication optimization as in optimizations for memory access: maximizing reuse, minimizing the working set, and making use of available parallelism in the communication system. However, the problems are magnified because the communication costs are at least an order of magnitude higher than those associated with memory access. We now discuss transformations which optimize on communication.

Message Vectorization

This transformation is analogous to the way a vector processor interacts with memory. Instead of sending each element of an array in an individual message, the compiler can group many of them together and send them in a single block to avoid paying the *startup* cost for each message transfer. The idea behind vectorization of messages is to amortize the *startup* overhead over a larger number of messages. Startup overhead includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router. Consider the following loop which adds each element of array *a* with the mirror element of array *b*.

```
for i := 1 to n do
    a[i] = a[i] + b[n+1-i]
end for
```

A parallelized version of the loop to be executed on two processors with the arrays having been block distributed (processor 0 has the first half and processor 1 has the second half) among them is as follows. The format for the SEND and the RECEIVE calls are as follows:

```

SEND(data buffer, number of bytes, destination)
RECEIVE(receive buffer, number of bytes, source)

```

```

LB = Pid * (n/2) + 1
UB = LB + (n/2)
otherPid = 1 - Pid
for i := LB to UB do
    SEND(b[i], 4, otherPid)
    RECEIVE(b[n+1-i], 4, otherPid)
    a[i] = a[i] + b[n+1-i]
end for

```

This version is very inefficient because during each iteration, each processor sends the element of b that the other processor will need and waits to receive the corresponding message. Applying the message vectorization transformation to this loop, we get the following version.

```

LB = Pid * (n/2) + 1
UB = LB + (n/2)
OtherPid = 1 - Pid
OtherLB = otherPid * (n/2) + 1
OtherUB = otherLB + (n/2)
SEND(b[LB], (n/2) * 4, otherPid)
RECEIVE(b[otherLB], (n/2) * 4, otherPid)
for i := LB to UB do
    a[i] = a[i] + b[n+1-i]
end for

```

This is a much more efficient version because it handles all communication in a single message before the loop begins executing. But such an optimization has a drawback if the vectorized messages are larger than the internal buffers available. The compiler may be able to perform additional communication optimization by using *strip mining* to reduce the message length as shown in the following parallelized version of the loop (assuming array size is a multiple of 256).

```

LB = Pid*(n/2) + 1
UB = LB + (n/2)
otherPid = 1 - Pid
otherLB = otherPid *(n/2) + 1
otherUB = otherLB + (n/2)
for j := LB to UB with step size of 256 do
    SEND(b[j], 256*4, otherPid)
    RECEIVE(b[otherLB + (j-LB)], 256*4, otherPid)
    for i := j to j+255 do
        a[i] = a[i] + b[n+1-i]
    end for
end for

```

Collective Communication

Many parallel architectures and message passing libraries offer special purpose communication primitives such as broadcast, hardware reduction, and scatter-gather. These primitives would have been implemented very efficiently on the underlying hardware. So a

compiler can improve performance by recognizing opportunities to exploit these operations. The idea is analogous to idiom and reduction recognition on sequential machines.

Message Pipelining

Another important optimization is to pipeline parallel computations by overlapping communication and computation. Many message passing systems allow the processor to continue executing instructions while a message is being sent. The compiler has the opportunity to arrange for useful computation to be performed while the network is delivering messages.

9.5 FINE-GRAINED PARALLELISM

We now describe some techniques directed towards the exploitation of instruction-level parallelism. They are (i) instruction scheduling, (ii) trace scheduling, and (iii) software pipelining.

9.5.1 Instruction Scheduling

On pipelined and RISC architectures it is often advantageous to reorder instructions to minimize delays caused by having to wait for results from previously issued, partially completed instructions. If an instruction can be issued every cycle, but some instructions take multiple cycles to complete, then it is possible that instruction $n + 1$ will *stall* (wait) until instruction n is completed. This wait cannot be avoided if $n + 1$ needs a result computed by n . It is desirable to place other independent instructions between n and $n + 1$ so that n can complete before $n + 1$ is attempted. The process of arranging sequences of straight line code so that stall will be eliminated or reduced is called instruction scheduling. Instruction scheduling is usually performed on basic blocks, sequences of straight line code of which only the first instruction can be the target of a branch instruction and only the last instruction can be a branch instruction.

9.5.2 Trace Scheduling

The Very Long Instruction Word (VLIW) architecture is another strategy for executing instructions in parallel. A VLIW processor exposes its multiple functional units explicitly. Here each machine instruction is very large and controls many independent functional units. Typically the instruction is divided into pieces, each of which is routed to one of the functional units. (Note that the idea here is similar to *horizontal microcoding*.) With traditional processors, there is a clear distinction between low-level scheduling done via instruction selection and high level scheduling between processors. This distinction is blurred in VLIW architectures. They rely on the compiler to expose the parallel operations explicitly and schedule them at compile time. VLIW machines require more parallelism than is typically available within a basic block. Hence compilers for these machines must look through branches to find additional instructions to execute.

Trace scheduling was developed as an analysis strategy for VLIW architectures and it has proved effective in generating highly parallel code for wide-word machines. Trace scheduling depends on the premise that execution time branch decisions can be predicted with some reliability at compile time. Branch predictions are based on software heuristics or on using profiles of previous program executions. It is claimed that in the domain of scientific codes, in which execution time is spent primarily in loops, compile time branch prediction can be done successfully.

The problem with trying to pack multiple operations per instruction in most programs is that the average length of a basic block is so short that few long-word instructions can be packed fully. The trace scheduling technique overcomes this problem by scheduling across basic blocks. An extended basic block, or *trace*, represents a possible execution time path through the program. A path likely to be taken with a “typical” data set can be inferred in different ways. The compiler can use the looping structure of the program as a guide, since it is likely that a loop will be reexecuted rather than exited. The programmer can insert directives indicating the probabilities attached to a conditional branch and the expected trip count through a loop. Alternatively, the compiler can insert code to monitor the path through

conditional branches and trip counts through loops and empirically derive the values. Using probabilities derived by one (or more) of these methods, the trace scheduler selects the most likely path through the program and schedules operations as if it were a single extended basic block. The most likely trace is then selected and scheduled, and so on until the entire program has been scheduled.

Unlike a basic block, a trace can have multiple entry and exit points. If code is moved in a trace, it may be necessary to insert “compensation code” to preserve correctness. Consider the following piece of code.

```
a = b + 5
if a > 0 then x = x + 10 else x = x + 15
end if
c = d + 20
```

Suppose profile says that $a > 0$, 98% of the time. Then the transformed version of extended basic (super) blocks (primary and secondary traces) for scheduling on a VLIW machine are as follows:

Trace 1 (primary)

```
a = b + 5
x = x + 10
c = d + 20
if a <= 0 then goto fixup_code
end if
```

Trace 2 (secondary)

```
fixup_code: x = x - 10 (compensation code)
x = x + 15
```

Although trace scheduling is very effective, the long compilation time and size of the generated code blocks are negative factors.

9.5.3 Software Pipelining

Recall that software pipelining was already discussed earlier as part of loop restructuring transformations. Software pipelining is complementary to trace scheduling. When successfully applied, it can deliver performance comparable to the trace scheduling of an unrolled loop without the attendant code explosion of trace scheduling. However, trace scheduling is a more general technique, which can be used on code other than loops.

Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP) exploits fine-grained pipeline parallelism in loops. It transforms a loop into multiple decoupled loops which can be run on multiple cores. Consider the execution of the following loop on a dual-core processor.

```
for i := 1 to n do
    A: a[i] = a[i-1]* b + c
    D: d[i] = a[i] + d [i]
end for
```

Assume that statement A takes two time units and statement D takes one time unit to execute. In Fig. 9.9(a), each iteration is assigned alternatively to each core (software pipelining). It takes 12 time units to complete four iterations, assuming that inter-core communication latency is one time unit. Figure 9.9(b) illustrates the DSWP parallelization

technique. It breaks the loop iteration into two parts, and assigns the first part to Core 0 and the second part of the loop body to Core 1. As a consequence of this, it takes 10 time units to complete four iterations. The main idea in DSWP parallelization technique is to avoid communication in the critical path (the longest path that determines the shortest possible execution time) in order to reduce the total execution time.

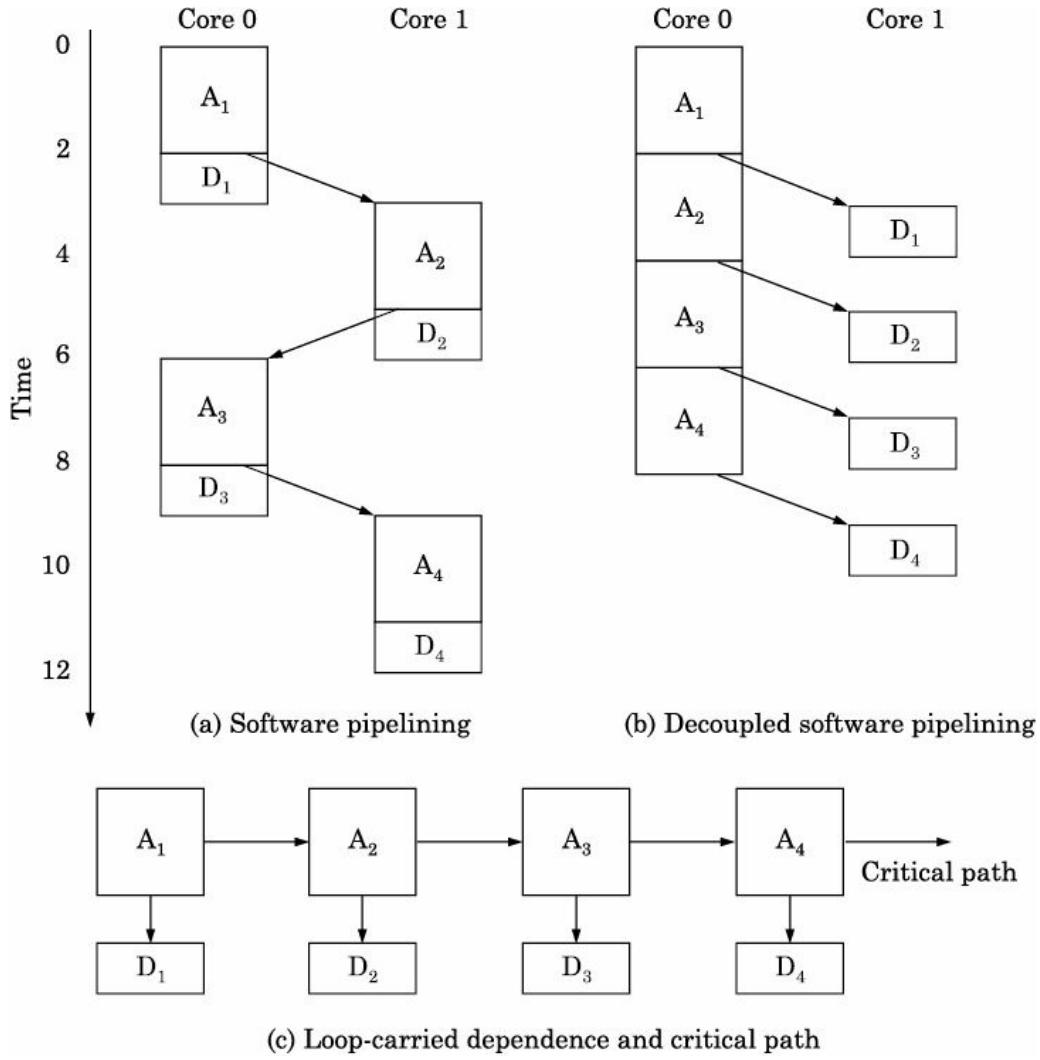


Figure 9.9 An example of DSWP.

9.6 TRANSFORMATION FRAMEWORK

Given the many transformations that we have seen so far; a major task for a compiler writer is to determine which ones to apply and in what order. There is no single best order of application. Current compilers generally use a combination of heuristics and a partially fixed order of applying transformations.

We now study a framework which enables one to encode both the characteristics of the code being transformed and the effect of each transformation. Then the compiler can quickly search the space of possible sets of transformations to find an efficient solution. The framework is based on *unimodular matrix theory*. This theory unifies all combinations of loop interchange, skewing, reversal, and tiling as unimodular transformations. It is applicable to any loop nest whose dependences can be described with a distance vector. The basic principle is to encode each transformation of the loop in a matrix and apply it to the dependence vectors of the loop. The effect on the dependence pattern of applying the transformation can be determined by multiplying the matrix and the vector. The form of the product vector reveals whether the transformation is valid. To model the effect of applying a series of transformations, the corresponding matrices are simply multiplied.

9.6.1 Elementary Transformations

A loop transformation rearranges the execution order of the iterations in a loop nest. Two elementary transformations for nest of n loops are described below:

1. Permutation: A permutation δ on a loop nest transforms iteration (p_1, \dots, p_n) to $(P_{\delta 1}, \dots, P_{\delta n})$. This transformation can be expressed in matrix form as I_δ , the $n \times n$ identity matrix with rows permuted by δ . For $n = 2$, a *loop interchange* transformation maps iteration (i, j) to iteration (j, i) . In matrix notation, we can write this as

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}$$

2. Reversal: Reversal of the i th loop is represented by the identity matrix with the i^{th} element on the diagonal equal to -1 . Consider the following piece of code:

```
for i := 1 to m do
    for j := 1 to m do
        a[i, j] = a[i-1, j] + a[i, j]
    end for
end for
```

A loop reversed version of this code corresponding to reversal of the inner loop is:

```
for i := 1 to m do
    do j := -m to -1 do
        a[i, -j] = a[i-1, -j] + a[i, -j]
    end for
end for
```

In matrix notation, we can write this as

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix}$$

9.6.2 Transformation Matrices

Unimodular transformations are defined by unimodular integer matrices. The product of two unimodular matrices is unimodular, and the inverse of a unimodular matrix is unimodular, so that combinations of unimodular loop transformations and the inverse of unimodular loop transformations are also unimodular loop transformations. A loop transformation is said to be *legal* if the transformed dependence vectors are all *lexicographically positive* (a vector is defined to be lexicographically positive if its first non-zero element is positive).

A compound loop transformation can be synthesized from a sequence of primitive transformations, and the effect of the loop transformation is represented by the products of the various transformation matrices for each primitive transformation. The above concepts are illustrated by the following example:

```
for i := 2 to 10 do
    for j := 1 to 10 do
        a[i, j] = a[i-1, j] + a[i, j]
    end for
end for
```

The distance vector describing the loop nest is $D = \{(1, 0)\}$, representing the dependence of iteration i on $i - 1$ in the outer loop. Suppose we want to check if the outer loop can be reversed, i.e., we want to check if the following transformation of the loop is legal.

```
for i := -10 to -2 do
    for j := 1 to 10 do
        a[-i, j] = a[-i-1, j] + a[-i, j]
    end for
end for
```

The unimodular matrix for reversal of the outer loop is

$$\mathbf{R} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

Applying loop-reverse transformation to the dependence vector describing the above loop,

$$\mathbf{D} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \text{ yields}$$

$$\mathbf{R} \cdot \mathbf{D} = \mathbf{P}_1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

This shows that the loop-reversal transformation is not legal for this loop, because P_1 is not *lexicographically positive*. This can be easily explained from elementary dependence analysis.

Similarly, the matrix for interchange transformation is

$$\mathbf{I} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Applying this to D yields

$$\mathbf{I} \cdot \mathbf{D} = \mathbf{P}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

In this case P_2 is lexicographically positive and hence the interchange of the two loops to get the following transformed version is legal.

```

for j := 1 to 10 do
  for i := 2 to 10 do
    a[i, j] = a[i-1, j] + a[i, j]
  end for
end for

```

We have seen that reversing the outer loop is invalid in this case. Let us see if reversing the outer loop followed by interchange of the loops is valid or not. The transformation matrix for this compound transformation of reversing the outer loop followed by interchange is given by the product of the unimodular transformation matrices for each of these individual transformations taken in the correct order, i.e.,

$$C = I \cdot R = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Applying this transformation we have,

$$C \cdot D = P_3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Since P_3 is not lexicographically positive, the following transformation (reversal followed by interchange) is also invalid.

```

for j := 1 to 10 do
  for i := -10 to -2 do
    a[-i, j] = a[-i-1, j] + a[-i, j]
  end for
end for

```

9.7 PARALLELIZING COMPILERS

A parallelizing compiler analyzes a “dusty deck” (an unaltered and unannotated sequential program) and transforms/compiles it to a parallel program that takes advantage of the presence of multiple processors. This approach, also called as *automatic parallelization*, while analyzing the sequential program essentially identifies opportunities for parallelism through dependence analysis. The first parallelizing compiler, based on dependence analysis, was the Parafrase system developed at the University of Illinois. Parafrase was a source-to-source program restructure (or compiler preprocessor) which transformed sequential FORTRAN 77 programs into forms suitable for vectorization or parallelization. The programs transformed by the Parafrase still needed a conventional optimizing compiler to produce code for a target machine. Parafrase 2 was developed for handling programs written in C. The research and development carried out as part of Parafrase led to two exploratory parallelizing compiler projects—Ptool at Rice University and PTRAN at the IBM T.J. Watson Research Centre.

Automatic parallelization, which essentially relies on dependence analysis, appears to be the only feasible solution for dealing with a large body of legacy sequential programs and also making parallel programming easier as compared to manual parallelization of sequential programs. Automatic parallelization technology is most mature for the FORTRAN 77 language. The simplicity of the language without pointers or user-defined types enabled to analyze and develop advanced compiler optimizations. Researchers have had limited success in developing parallelizing compilers that convert sequential programs in languages such as C, C++ and Java into parallel programs. Autoparallelizers do not guarantee that the generated parallel program results in increased performance. Users need to invest time and effort in tuning their parallel programs when they experience performance degradation.

Autotuning is a general technique useful in both sequential and parallel processing. Recent research work suggests automatic tuning techniques to be integral part of autoparallelizing tools. The idea of autotuning is to automatically adapt the execution of a program to a given parallel computer in order to optimize one or more run-time performance metrics such as execution time and energy consumption. Note that autotuning *per se* does not parallelize a program but closely examines trade-offs among parameters, such as task granularity, inter-processor communication, synchronization, load balancing and locality, in the already parallelized program. For each performance metric, a specific set of tuning parameters is required. For example, for the execution time metric some possible tuning parameters are task granularity, scheduling method (autotuning/programmer can try out different OpenMP loop scheduling directives such as static, dynamic and guided) and data distribution. On the other hand, for the energy consumption metric, the tuning parameters include voltage and frequency settings. Autotuning software helps in identifying efficient parallel program segments and (sometimes) serializing inefficient ones using the performance metrics and the value ranges of the corresponding tuning parameters. As an example, an autotuner may decide to use dynamic thread scheduling for better load balancing if tasks may recursively create other tasks, or may decide to lower voltage during the execution of instructions within an inner-most loop in order to reduce the overall energy consumption. Auto-tuners automatically generate and search a set of alternative implementations of a computation and then select the one that gives the best performance.

Some examples of commercial compilers are Intel’s ICC and PGI. Intel C++ compiler when invoked by the ‘parallel’ option automatically translates sequential portions of the input

program into an equivalent multithreaded program for symmetric multiprocessor systems. The PGI compiler with shared-memory parallelization capabilities incorporates global optimization, vectorization which transforms loops to improve memory access performance and software pipelining. Examples of some recent research parallelizing compilers include OpenUH, Cetus and Rose. OpenUH is an open source, optimizing compiler suite, based on Open64 compiler maintained by HPCTools group of the University of Houston, for C, C++ and FORTRAN. The compiler's major optimization components include global optimizer, loop nest optimizer and inter-procedural analyzer. The Cetus compiler infrastructure, which targets C programs, is a source-to-source translator that supports the detection of loop-level parallelism and generation of C program with OpenMP parallel directives for execution on multicore processors. Rose is an open source, compiler infrastructure to build source-to-source transformation and analysis tools for large-scale FORTRAN, C, C++, Java and OpenMP applications. It supports several loop optimizations.

Automatic parallelization based on polyhedral model, which was born with the seminal work of Richard M. Karp, Raymond E. Miller and Shmuel Winograd on systems of uniform recurrence equations, became increasingly important with the advent of multicore processors. The model with a mathematical framework based on parametric linear algebra and integer linear programming has proven to be a valuable tool to optimize loop programs (for example, communication-optimized coarse-grained parallelization together with locality optimization). However, the loop bounds and memory accesses are limited to affine combinations (linear combination with a constant). Optimizations based on polyhedral model can be applied not only at compile time but also at run time to know more about the structure of the loops thereby enabling more loops to be optimized.

Semi-automatic parallelization involves the knowledge of a programmer. For example, using compiler directives or compiler flags the programmer tells the compiler how to parallelize a piece of code when the autoparallelizer cannot determine what action to take. Although tremendous advances have been made in dependence theory and in the development of a “toolkit” of transformations, parallel computers are used most effectively when the programmer interacts in the optimization process.

9.8 CONCLUSIONS

In this chapter, we described a large number of compiler transformations, with illustrative examples, that can be used to improve program performance on sequential and parallel machines. A prerequisite to the safe use of the transformations discussed here is accurate dependence analysis. The transformation techniques we described here attempt to exploit parallelism at both the fine-grained instruction level and the coarse-grained procedure level. Numerous studies have demonstrated that these transformations, when properly applied, yield high performance.

EXERCISES

9.1 Consider the following piece of code.

```
for i := 2 to n - 1 do
    for j := m - 1 to 2 do
        a[i, j] = (a[i-1, j] + a[i, j-1] + a[i+1, j] + a[i, j+1])/4
    end for
end for
```

The following code is obtained after applying a transformation on the above code. Check whether this transformation is correct.

```
for j := 4 to m + n - 2 do
    for i := max(2, j-m+1) to min(n-1, j-2) do
        a[i, n+1 - (j - i)] = (a[i-1, (n+1) - (j-i)] + a[i, n-j+i+2]) +
        a[i+1, n+1-(j-i)] + a[i, n-j+i])/4
    end for
end for
```

9.2 Vectorize the following loops if possible. Otherwise give reasons why it is not possible.

- (a)

```
for i := 1 to N do
    A(i+1) = A(i) + 3
end for
```
- (b)

```
for i := 1 to N do
    B[i] = C[i] * 4
    D[i] = B[i+1] + A[i]
end for
```
- (c)

```
for i := 1 to N do
    A[i] = B[i] + C[i]
    B[i] = A[i+1] * 5
end for
```

9.3 *Loop Peeling* is a loop restructuring transformation in which two successive loops are merged after “peeling” or separating the first few iterations of the first loop or the last few iterations of the second loop or both. Peeling has two uses: for removing dependences created by these first/last loop iterations thereby enabling parallelization and for matching the iteration control of adjacent loops to enable fusion. Apply this transformation to the following code to obtain a single parallelizable loop.

```
for i := 2 to n do
    b[i] = b[i] + b[2]
end for
for i := 3 to n do in parallel
    a[i] = a[i] + c
end for
```

9.4 *Loop Normalization*, another loop restructuring transformation, converts all loops so that the induction variable is initially 1 or 0 and is incremented by 1 on each iteration. This transformation can expose opportunities for fusion and simplify inter-loop dependence analysis. The most important use of normalization is to permit the compiler to apply subscript analysis tests, many of which require normalized iteration ranges. Now show how this loop transformation should be applied to the following code such

that the different iterations of a loop can be parallelized.

```

for i := 1 to n do
    a[i] = a[i] + c
end for
for i := 2 to n+1 do
    b[i] = a[i-1] + b[i]
end for
```

9.5 Consider the following double loop (L_1, L_2)

```

L1 : for I1 := 0 to 4 do
L2 :   for I2 := 0 to 4 do
        S : A(I1 + 1, I2) = B(I1, I2) + C(I1, I2)
        T : B(I1, I2 + 1) = A(I1, I2 + 1) + 1
        U : D(I1, I2) = B(I1, I2 + 1) - 2
        end for
    end for
```

For this loop

- (a) Find the dependences caused by all possible pairs of variables. Identify the type of each dependence and find its distance and direction vectors.
- (b) Draw the statement and iteration dependence graphs for the loop.

9.6 Find the dependence equation for the variable of the two-dimensional array X in statements S and T in the following loop.

```

L1 : for I1 := 1 to 50 do
L2 :   for I2 := 12 to I1 + 60 do
        S : X(2I1 + 2I2, I1 + 4I2) = ...
        T : ... = ... X(500 - 2I1 - 4I2, 600 - I1 - 5I2) ...
        end for
    end for
```

9.7 Consider the following algorithm which performs *Echelon Reduction*. Given an $m \times n$ integer matrix \mathbf{A} , this algorithm finds an $m \times m$ unimodular matrix \mathbf{U} and $m \times n$ echelon matrix $\mathbf{S} = (s_{ij})$, such that $\mathbf{U}\mathbf{A} = \mathbf{S}$. \mathbf{I}_m is the $m \times m$ identity matrix and $\text{sig}(i)$ denotes the sign of integer i . Use this algorithm to compute \mathbf{U} and \mathbf{S} for the example problem given in the section on GCD test.

```

set  $\mathbf{U} = \mathbf{I}_m$ ,  $\mathbf{S} = \mathbf{A}$ ,  $i_0 = 0$ 
for j := 1 to n do
    if there is at least one non-zero  $s_{ij}$  with  $i_0 < i \leq m$ 
        then
            set  $i_0 = i_0 + 1$ 

            for i := m downto  $i_0 + 1$  do
                while  $s_{ij} \neq 0$  do
                    set  $\sigma = \text{sig}(s_{(i-1)j} * s_{ij})$ 
                     $z = \lfloor |s_{(i-1)j}| / |s_{ij}| \rfloor$ 
                    subtract  $\sigma z$  (row i) from row  $(i-1)$  in  $(\mathbf{U}, \mathbf{S})$ 
                    interchange rows i and  $(i - 1)$  in  $(\mathbf{U}, \mathbf{S})$ 
                end while
            end for
        end if
    end for
```

9.8 Using the GCD test, decide if there could be dependence between the statements S and T in the loop nest

```

L1 : for I1 := p1 to q1 do
    L2 :      for I2 := p2 to q2 do
        S : u = ...
        T : ... = ... v ...
            end for
    end for

```

where

- (a) $u = X(2I_1 + 3I_2 + 1)$, $v = X(3I_1 - I_2 + 4)$
- (b) $u = X(4I_1 - 2I_2 - 1, -I_1 - 2I_2)$, $v = X(I_1 + 2I_2 + 1, -2I_1 + 1)$

9.9 Consider the following loop nest:

```

for i := 1 to n do
    S1 : a[i] = b[i]
    S2 : c[i] = a[i] + b[i]
    S3 : e[i] = c[i+1]
    end for

```

- (a) Determine the dependence relations among the three statements.
- (b) Show how the loop can be vectorized.

9.10 What are the benefits of loop tiling? Consider the following code fragment for multiplying two matrices A and B. Assume that n is very large.

```

for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            C[i,k] = C[i,k] + A[i,j] × B[j,k]
        end for
    end for
end for

```

- (a) In this code, the same rows of C and B are reused in the next iteration of the middle and outer loops, respectively. Then what benefit, if any, does tiling the loop provide?
- (b) Give the tiled version of the loop assuming a tile size s .
- (c) What are the criteria involved in choosing the tile size s ?

9.11 Consider the following code fragment:

```

for i := 1 to n do
    for j := 1 to n do
        A[i,j] = A[i,j] + A[i+1, j-1]
    end for
end for

```

Answer the following questions based on the unimodular transformation framework.

- (a) Is a loop interchange transformation legal for the given loop?
- (b) Is a loop reversal transformation legal?
- (c) Is it legal to do a loop reversal followed by an interchange?

9.12 In the section on transformation frameworks, we discussed loop permutation and reversal as two elementary transformations. Loop skewing is yet another elementary

transformation. Consider the following loop:

```

for i := 2 to n - 1 do
    for j := 2 to m - 1 do
        a[i,j] = (a[i-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1])/4
    end for
end for

```

Skewing the loop by a factor of 1, we get the following version.

```

for 1 := 2 to n - 1 do
    for j := i + 2 to i + m - 1 do
        a[i,j-i] = (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4
    end for
end for

```

Figure 9.10 gives the iteration spaces for the original code as well as for the transformed version.

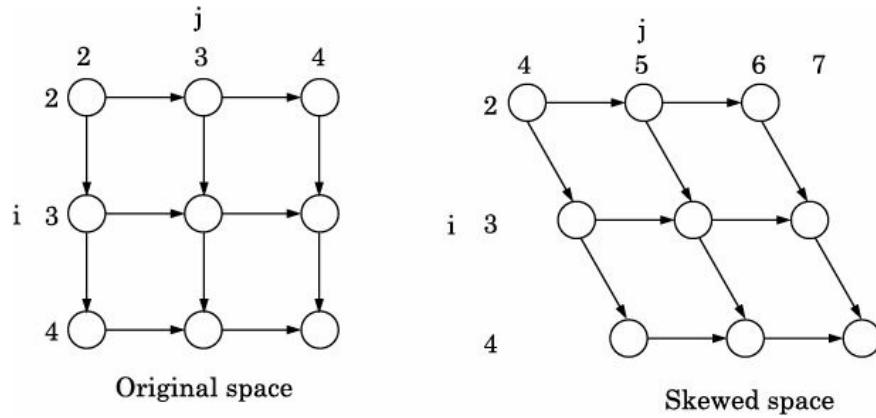


Figure 9.10 Iteration spaces of the two versions.

- (a) What is the transformation matrix for the above skewing transformation?
- (b) What are the dependence vectors for the original loop? Derive the dependence vectors for the skewed version by applying the transformation matrix obtained in part (a) on the original dependence vectors.

9.13 Consider the following piece of code in which a procedure MAIN invokes another procedure FUNC.

```

procedure MAIN
  for i := 1 to n do
    for j := 1 to n do
      a[i,j] = b[i,j+1] + c
      FUNC(b[i,j-1],c)
    end for
  end for
end

procedure FUNC(x,y)
  if (x < 0) then
    y = -x
  else
    y = x
  end if
end

```

Inline the procedure FUNC in the MAIN procedure.

9.14 To allow multiple independent (i.e., faster) access, in systems which employ multiple memory banks, successive elements of an array are stored in successive memory banks. Now consider a program which indexes an array through a dimension that is not laid out contiguously in memory, leading to a non-unit stride s . For example, consider the following code fragment which has a stride $s = 4$ as it accesses the consecutive elements of a column (assuming the elements are stored in column-major order).

```

integer a[4,4]
for j := 1 to 4 do
  a[1,j] = a[1,j] + c
end for

```

The array a has 4 rows and 4 columns as shown below:

$$a = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,4} \\ a_{2,1} & a_{2,2} & \dots & a_{2,4} \\ \vdots & \vdots & \ddots & \vdots \\ a_{4,1} & a_{4,2} & \dots & a_{4,4} \end{bmatrix}$$

- (a) Assume that the array a is stored in the memory in column-major order and the memory has $B = 4$ banks. Then the elements of the array will be stored in the memory as shown in Fig. 9.11. How will the given code perform in terms of time required for consecutive memory accesses?

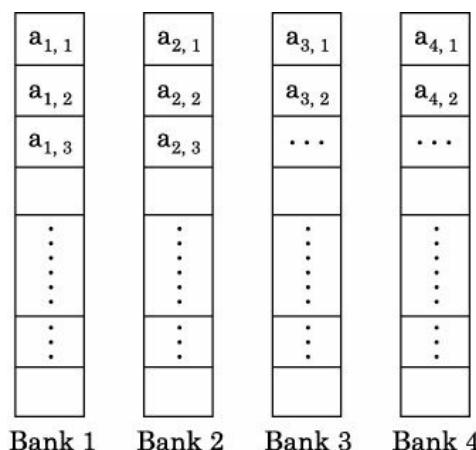


Figure 9.11 Storage of a in the memory.

- (b) From the preceding part, it is clear that if the stride is an exact multiple of the number of banks, then all successive memory accesses will be on the same bank leading to severe performance degradation. One straightforward solution to this problem is to pad the array along the dimension of access with some p (now the stride will be $s + p$) such that the accesses do not fall onto the same bank. Find the condition on the stride s , the number of banks B , and the padding size p such that B successive accesses will all access different banks.

9.15 Consider the following loop

```

for j := 1 to n do
    do i := 1 to j do
        total = total + a[i,j]
    end for
end for

```

The iteration space for this loop is given in Fig. 9.12. Locality for neighbour-based computation and uniform balancing of load across processors are the two important metrics for array decomposition across processors in a parallel machine. Assume that you want to optimize on the uniform balancing of load metric. Then which array decomposition strategy among the following would you choose for decomposing the array a ?

- (a) Serial
- (b) Block
- (c) Cyclic

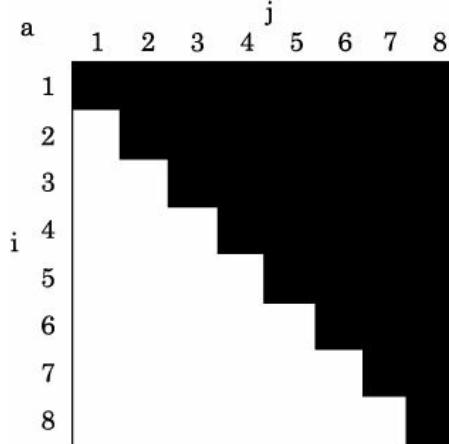


Figure 9.12 Triangular iteration space.

Assume that the array is very large.

- 9.16 VLIW and superscalar architectures are two approaches to instruction level parallelism. Contrast these two techniques with regards to the basic differences as well as their respective merits and demerits.
- 9.17 Both software pipelining and loop unrolling attempt to parallelize the execution of different iterations of a loop. Contrast these two techniques.
- 9.18 Consider the following piece of code.

```

a = log(x)
if (b > 0.01) then c = a + b else c = 0
end if

```

$$y = \sin(c)$$

Suppose profile says that $b > 0.01$ 95% of the time. Show the transformed version of the optimized super blocks (primary and secondary traces) for scheduling on a VLIW machine.

9.19 Construct an example where DSWP is not effective.

BIBLIOGRAPHY

- Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Prentice Hall, USA, 2006.
- Allan, V.H., Jones, R.B., Lee, R.M. and Allan, S.J., “Software Pipelining”, *ACM Computing Surveys*, Vol. 27, No. 3, Sept. 1995, pp. 367–432.
- Allen, R. and Kennedy, K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, San Francisco, USA, 2001.
- Anand, V., Shantharam, M., Hall, M. and Strout, M.M., “Non-affine Extensions to Polyhedral Code Generation”, *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.
- Andion, J.M., Arenaz, M., Rodriguez, G. and Tourino, J., “A Parallelizing Compiler for Multicore Systems”, *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, 2014, pp. 138–141.
- Bacon, D.F., Graham, S.L. and Sharp, O.J., “Compiler Transformations for High Performance Computing”, *ACM Computing Surveys*, Vol. 26, No. 4, Dec. 1994, pp. 345–420.
- Bae, H., Mustafa, D., Lee, J.W., Lin, H., Dave, C., Eigenmann, R. and Midkiff, S.P., “The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation”, *International Journal of Parallel Programming*, Vol. 41, No. 6, Dec. 2013, pp. 753–767.
- Banerjee, U., Elgenmann, R., Nicolau, A. and Padua, D.A., “Automatic Program Parallelization”, *Proceedings of the IEEE*, Vol. 81, No. 2, Feb. 1993, pp. 211–243.
- Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, USA, 1997.
- Bastoul, C., “Code Generation in the Polyhedral Model is Easier Than You Think”, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2004, pp. 7–16.
- Blume, W., et al., “Parallel Programming with Polaris”, *IEEE Computer*, Vol. 29, No. 12, Dec. 1996, pp. 78–82.
- Ellis, J.R., *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, MA, USA, 1986.
- Fisher, J.A. and Rau, B.R., “Instruction-level Parallel Processing”, *Science*, Vol. 253, Sept. 1991, pp. 1233–1241.
- Gohringer, D. and Tepelmann, “An Interactive Tool Based on Polly for Detection and Parallelization of Loops”, *Proceedings of Workshop on Parallel Programming and Runtime Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2014.
- Gokhale, M. and Carlson, W., “An Introduction to Compiler Issues for Parallel Machines”, *Journal of Supercomputing*, Vol. 6, No. 4, Aug. 1992, pp. 283–314.
- Hall, M.W., et al., “Maximizing Multiprocessor Performance with the SUIF Compiler”, *IEEE Computer*, Vol. 29, No. 12, Dec. 1996, pp. 84–89.
- Hall, M.W., Padua, D.A. and Keshav, P., “Compiler Research: The Next 50 Years”, *Communications of the ACM*, Vol. 52, No. 2, Feb. 2009, pp. 60–67.
- Huang, J., Raman, A., Jablin, T., Zhang, Y., Hung, T. and August, D., “Decoupled Software Pipelining Creates Parallelization Opportunities”, *Proceedings of the Annual IEEE/ACM*

- International Symposium on Code Generation and Optimization*, 2010, pp. 121–130.
- Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, USA, 1993.
- Karp, R.M., Miller, R.E. and Winograd, S., “The Organization of Computations for Uniform Recurrence Equations”, *Journal of the Association for Computing Machinery*, Vol. 14. No. 3, July 1967, pp. 563–590.
- Mace, M.E., *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Publishers, Norwell, MA, USA, 1987.
- Mehrara, M., Jablin, T., Upton, D., August, D., Hazelwood, K. and Mahlke, S., “Multicore Compilation Strategies and Challenges”, *IEEE Signal Processing Magazine*, Vol. 26, No. 6, Nov. 2009, pp. 55–63.
- Midkiff, S.P., *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, Morgan & Claypool Publishers, USA, 2012.
- Mustafa, D. and Eigenmann, R., “PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers”, *International Journal of Parallel Programming*, 2014.
- Polychronopoulos, C.D., *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, MA, USA, 1988.
- Pouchet, L., Uday, B., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P. and Vasilache, N., “Loop Transformations: Convexity, Pruning and Optimization”, *Proceedings of the ACM Symposium on Principles of Programming Languages*, 2011, pp. 549–562.
- Rau, B.R. and Fisher, J.A., “Instruction-level Parallel Processing: History, Overview, and Perspective”, *Journal of Supercomputing*, Vol. 7, No. 1–2, May 1993, pp. 9–50.
- Tiwari, A., Chen, C., Chame, J., Hall, M. and Hollingsworth, J.K., “A Scalable Auto-tuning Framework for Compiler Optimization”, *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, 2009.
- Verdoolaege, S., Juega, J.C., Cohen, A., Gomez, J.I., Tenllado, C. and Catthoor, F., “Polyhedral Parallel Code Generation for CUDA”, *ACM Transactions on Architecture and Code Optimization*, Vol. 9, No. 4, Jan. 2013.
- Walter, T., “Auto-tuning Parallel Software: An Interview with Thomas Fahringer: The Multicore Transformation”, *ACM Ubiquity*, June 2014.
- Wolfe, M., *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, USA, 1989.

10

Operating Systems for Parallel Computers

Operating systems for parallel computers are similar to multiprogrammed uniprocessor operating systems in many respects. However, multiprocessor (parallel) operating systems are more complex due to the complexity of parallel hardware (in which there is *physical* as opposed to *virtual* concurrency in multiprogrammed uniprocessors) and more importantly, due to the performance requirements of user programs. Specific problems to be addressed concerning performance include (i) resource (processor) management (i.e., assignment of processes to processors), (ii) process management (representation of asynchronous) active entities like processes or threads), (iii) synchro-nization mechanisms, (iv) inter-process communication, (v) memory management and (vi) input/output. In this chapter we will consider the above six operating systems issues in detail. Throughout this chapter we will be using the terms “process” and “task” interchangeably.

10.1 RESOURCE MANAGEMENT

In Chapter 2, we observed that one of the important issues in parallel computing systems is the development of effective techniques for the assignment of tasks (or processes) of a parallel program to multiple processors so as to minimize the execution time of the program. From a system's point of view, this assignment choice becomes a resource (processors) management problem. The resource management i.e., the task assignment or the scheduling algorithm should not contribute significantly to the overhead of the system. If this algorithm takes longer than the resulting performance improvement of the multiprocessing evaluation, the system will be slower than a uniprocessor system. In general, there are two generic types of scheduling strategies: static and dynamic. Static scheduling is simpler and is performed only once, whereas a dynamic strategy attempts to maximize the utilization of the processors in the system at all times and is more complex. We now study both the static (also called compile-time) and the dynamic (also called dynamic load balancing) scheduling problems in detail.

10.1.1 Task Scheduling in Message Passing Parallel Computers

In static scheduling, the assignment of tasks in a task graph (representing a parallel program) to the processors is done at compile-time with the goal of minimizing the overall execution (completion) time of the task graph (parallel program), while minimizing the inter-task communication delays. This is the classic multiprocessor (parallel) scheduling problem and in the most general case it (i) allows variations in the number of tasks in the task graph, (ii) allows variations in the execution times of the tasks, (iii) imposes no constraints on the precedence relations in the task graph, (iv) does not assume that inter-task communication in the task graph is negligible, (v) allows variations in inter-task communication in the task graph and (vi) allows arbitrary number of processors and arbitrary interconnection topology of the multiprocessor system with non-negligible inter-processor communication cost. This scheduling problem is extremely complex and is known to be NP-hard in the strong sense (i.e., it is computationally intractable or in other words, the only known algorithms that find an optimal solution require exponential time in the worst case). Consequently, research effort has focused on finding polynomial time algorithms which produce optimal schedules for restricted cases of the multiprocessor scheduling problem and on developing heuristic algorithms for obtaining satisfactory suboptimal solutions, to the multiprocessor scheduling problem stated above, in a reasonable amount of computation time.

Polynomial time algorithms exist only for highly restricted cases, such as (i) all tasks take unit time and the task graph is a forest (i.e., no task has more than one predecessor) and (ii) all tasks take unit time and the number of processors is two, which do not hold in practice most of the time. To provide solutions to real-world scheduling problems, we must relax restrictions on the parallel program and the multiprocessor representations. We now precisely define the multiprocessor scheduling problem. As this problem is known to be NP-hard in the strong sense, we present a heuristic algorithm for the problem. It should be noted that the algorithm produces a solution in polynomial time but does not guarantee an optimal solution (i.e., minimum completion time of task graph or parallel program). Since the scheduling problem is computationally intractable, we must be content with polynomial time algorithms that do a good, but not perfect, job.

The Scheduling Problem

A parallel program is characterized by a weighted directed acyclic (task) graph $G = (V, E)$, where $V = \{v_i\}$, $i = 1, 2, \dots, N$ represent the set of tasks of the program with associated weights $\{r_i\}$ where r_i denotes the computation (execution) time of v_i and $E = \{e_{i,j}\}$ is the set of directed edges which define a partial order or precedence constraints on V . An edge $\{e_{i,j}\}$ in G directed from v_i to v_j implies that task v_i must precede task v_j in any schedule. There is a weight $c_{i,j}$, with edge $e_{i,j}$, which denotes the amount of communication from task v_i to task v_j . In task graph G , tasks without predecessors are known as entry tasks and tasks without successors are known as exit tasks. We assume, without loss of generality, that G has exactly one entry task and exactly one exit task. An example of task graph with seven tasks is shown in Fig. 10.1 (Here T_1 is the entry task and T_7 is the exit task; the weight on a vertex and the weight on an edge denote the computation time of the corresponding task and the inter-task communication time between the corresponding tasks, respectively.) The possible architectures for the multiprocessor system onto which task graphs can be scheduled include hypercubes, meshes, rings, etc. The multiprocessor is assumed to be homogeneous and its communication channels (links) are assumed to be halfduplex. Also, each processor contains a separate communication hardware which enables the processor to perform both computation and inter-task communication in parallel. It is also assumed that there is adequate buffer space between the computation and the communication subsystem of a processor, and that the communication subsystem can handle all its channels simultaneously. Given a task graph and a multiprocessor as described above, the multiprocessor scheduling problem is to obtain a non-preemptive schedule of the task graph onto the multiprocessor in a way that minimizes the completion time.

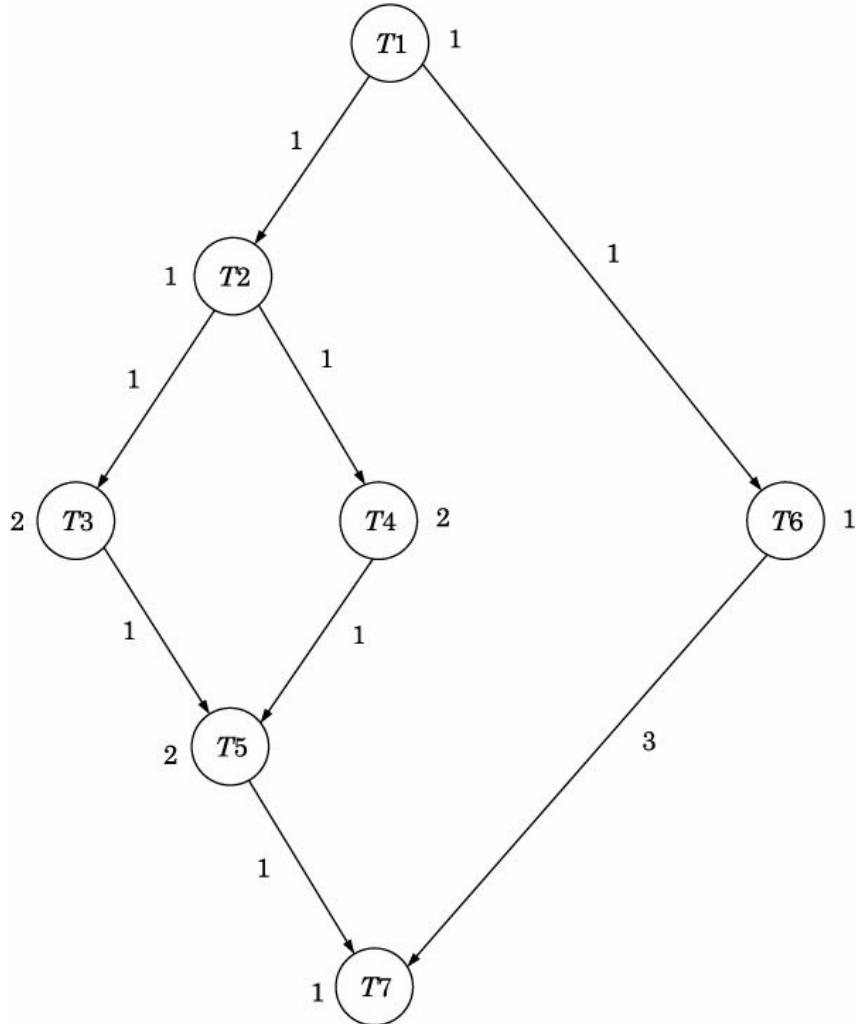


Figure 10.1 Task graph.

Scheduling Algorithm

The algorithm takes exactly N scheduling steps for scheduling a task graph and works as follows: At each scheduling step, it computes a set of *ready* tasks (Unscheduled tasks which have no predecessors or which have all their predecessors scheduled are known as ready tasks.), constructs a priority list of these tasks and then selects the element (task) with the highest priority for scheduling in the current step. The priorities are computed taking into consideration the task graph, the multiprocessor architecture and the partial schedule at the current scheduling step. (A partial schedule at the current scheduling step is the incomplete schedule constructed by the scheduling steps preceding the current scheduling step.) Thus each element (task) in the priority list will have three fields, viz., task, processor and priority of the task with respect to the processor. Now the selected task is scheduled to the selected processor such that it finishes the earliest.

The priority of a ready task, v_i with respect to each processor, p_j ($j = 1, 2, \dots$, number of processors), given the partial schedule (Σ) at the current scheduling step, (which reflects the quality of match between v_i and p_j given Σ) is given by

$$\text{Schedule Priority, } SP(v_i, p_j, \Sigma) = SL(v_i) - EFT(v_i, p_j, \Sigma)$$

where $EFT(v_i, p_j, \Sigma)$ is the earliest time at which task v_i will finish execution if it is scheduled on processor p_j , given the partial schedule Σ at the current scheduling step and $SL(v_i)$, called

the static level of v_i , is the length of the longest path from task v_i to the exit task. Here, the length of the path is defined as the sum of the execution times of all the tasks on the path. The level of a task indicates how far the task is away from the exit task (task completion). The level of the entry task denotes the lower bound on the completion time of the task graph. (The longest path, from the entry task to the exit task is called the *critical* path because it determines the shortest possible completion time.) This means the total execution time for the task graph cannot be shorter than the level of the entry task even if a completely-connected interconnection topology (in which there exists a direct communication channel or link between every pair of processors) with unbounded number of processors is employed. Since the static level of a task depends on the task graph alone, it can be computed for all tasks before the scheduling process begins.

The algorithm considers simultaneous scheduling of the processors and channels. All inter processor communications are scheduled by a router which finds the best path between the communicating processors for each communication and schedules the communication onto the channels on this path. Below we present the algorithm for computing EFT^* .

Algorithm EFT (v_i, p_j, Σ)

1. Tentatively schedule all communications to v_i from its predecessors with processor p_j as the destination using the router so as to find the earliest time at which all data from the predecessors of v_i are available at p_j . This time is called data ready time for v_i on p_j .
2. Find the earliest time at which we can execute v_i on p_j by searching the list of free time slots in processor p_j such that the slot is after the data ready time for v_i on p_j and such that the slot is large enough to execute v_i . Note that the list of free slots contains one slot which starts at the finish time of the latest task to be executed on processor p_j and extends to infinity plus any *schedule holes* (unused processor time intervals) that might have been generated as result of previous schedules. EFT^* is equal to the sum of the time computed above and the execution time of task v_i .
3. Undo all tentatively scheduled channel communications and return the value of EFT^* .

Given the idea of schedule priority and the algorithm for computing EFT^* , we now present the scheduling algorithm below.

Algorithm SCH

while there are tasks to be scheduled **do**

1. Let Σ be the partial schedule at this scheduling step. Compute the set of ready tasks R with respect to Σ .

2. **foreach** task $v_i \in R$ **do**

foreach processor p_j in the multiprocessor **do**

—Compute $EFT(v_i, p_j, \Sigma)$ using algorithm EFT

—Compute the schedule priority $SP(v_i, p_j, \Sigma)$

using the equation

$$SP(v_i, p_j, \Sigma) = SL(v_i) - EFT(v_i, p_j, \Sigma)$$

end foreach

end foreach

3. Let the schedule priority be maximum for task v_m with respect to processor p_n given the partial schedule Σ . Schedule all communications from the immediate predecessors of task V_m to the channels using the router which exploits schedule holes in the channels. Then schedule task v_m onto processor p_n such that it finishes the earliest, exploiting schedule holes in processor p_j whenever possible.

end while

Figure 10.3 gives the output of the algorithm, presented above, at various steps while scheduling the task graph in Fig. 10.1 onto a two-processor system shown in Fig. 10.2.

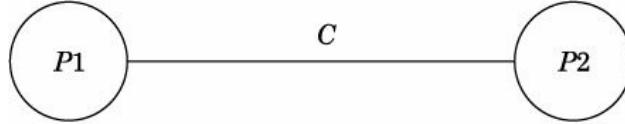


Figure 10.2 Multiprocessor system.

Task Granularity and Clustering (Ideal versus Useful Parallelism)

Figure 10.4 shows a task graph for the problem of finding the maximum of eight numbers, obtained by the application of a divide-and-conquer strategy. A vertex in the task graph represents the comparison of two numbers and an edge represents communication of a maximum of two numbers. Figure 10.5 shows the multiprocessor system onto which this task graph is to be scheduled. The schedule produced by the algorithm, described above, is shown in Fig. 10.6. We observe that the schedule length or the completion time is 8 time units. This is one time unit more than a sequential or a uniprocessor schedule (all the tasks on one processor) having length 7!

Scheduling Step	Ready Tasks	Priority List (Task, Processor, Schedule Priority)	Selected (Task, Processor)	P1	P2	C
(a) 1	T_1	$(T_1, P_1, 6) (T_1, P_2, 6)$	(T_1, P_1)	T_1		
(b) 2	T_2, T_6	$(T_2, P_1, 4) (T_2, P_2, 3)$ $(T_6, P_1, 0) (T_6, P_2, -1)$	(T_2, P_1)	$T_1 T_2$		
(c) 3	T_3, T_4 T_6	$(T_3, P_1, 1) (T_3, P_2, 0)$ $(T_4, P_1, 1) (T_4, P_2, 0)$ $(T_6, P_1, -1) (T_6, P_2, -1)$	(T_3, P_1)	$T_1 T_2 T_3$		
(d) 4	T_4, T_6	$(T_4, P_1, -1) (T_4, P_2, 0)$ $(T_6, P_1, -3) (T_6, P_2, -1)$	(T_4, P_2)	$T_1 T_2 T_3 $	T_4	
(e) 5	T_5, T_6	$(T_5, P_1, -4) (T_5, P_2, -3)$ $(T_6, P_1, -3) (T_6, P_2, -1)$	(T_6, P_2)	$T_1 T_2 T_3 $	$T_6 T_4$	
(f) 6	T_5	$(T_5, P_1, -4) (T_5, P_2, -3)$	(T_5, P_2)	$T_1 T_2 T_3 $	$T_6 T_4 T_5$	
(g) 7	T_7	$(T_7, P_1, -8) (T_7, P_2, -7)$	(T_7, P_2)	$T_1 T_2 T_3 $	$T_6 T_4 T_5 T_7$	
0 1 2 3 4 5 6 7 8 Schedule length = 8 time units						

Figure 10.3 Progression of the scheduling algorithm.

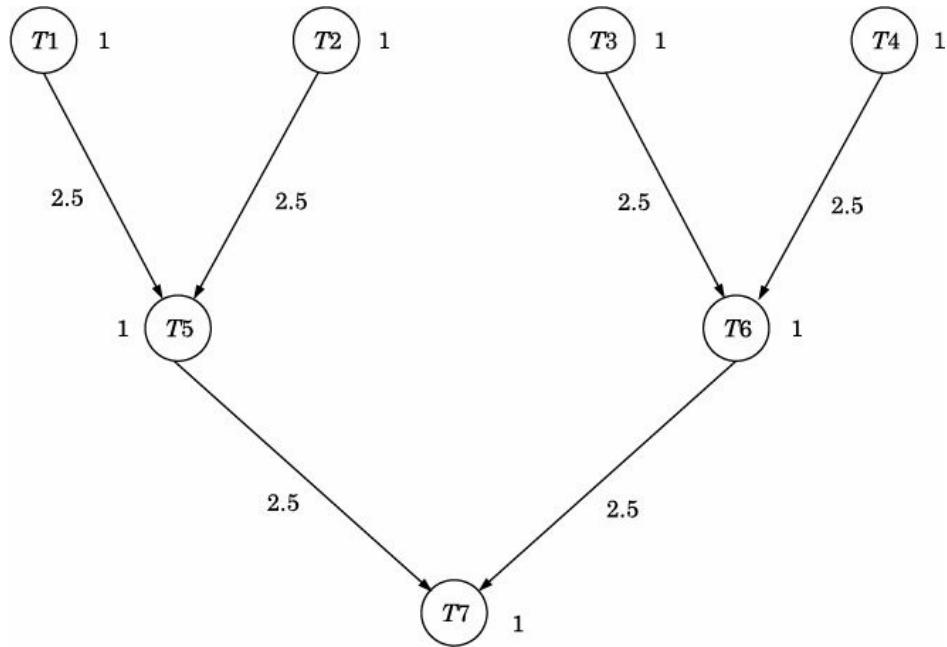


Figure 10.4 Task graph.

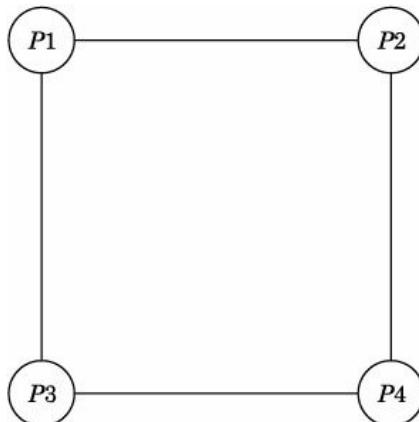


Figure 10.5 Multiprocessor system.

P4	T4					
P3	T3				T6	
P2	T2					
P1	T1		T5			T7

0 1 2 3 4 5 6 7 8

Figure 10.6 Schedule for the task graph.

The granularity of a task is defined as the ratio of the task computation time to the task communication (and overhead). Thus, if E = execution time of a task and C = communication delay generated by the task, E/C is the granularity of the task. A task is said to be fine-grained task if its E/C ratio is relatively low. On the other hand, if E/C is relatively high, then the related task is said to be coarse-grained. In general, fine-grained tasks imply more degree of parallelism among the tasks as well as more amount of overhead (because of more inter-task communication/synchronization points). Conversely, coarse-grained tasks imply less parallelism among the tasks, but less overhead as well. Thus the performance of a multiprocessor system depends on the balance between the amount of parallelism among the

tasks and the associated overhead. To improve system performance fine-grained tasks can be clustered into coarse-grained tasks. These coarse-grained (clustered) tasks are then scheduled on the processors. Therefore, clustering is a preprocessing step to scheduling.

Let us reconsider the task graph shown in Fig. 10.4. The E/C ratio of a task here is 1/2.5 which is low. Now consider the following simple clustering strategy. Tasks T_i , T_j and T_k as shown in Fig. 10.7, are clustered into one task if the shortest parallel processing time $\text{MIN}\{(t_i + t_k + t_{ik}), (t_j + t_k + t_{jk})\}$ is larger than the sequential processing time $(t_i + t_j + t_k)$. This strategy when applied to the task graph of Fig. 10.4 results in a new task graph as shown in Fig. 10.8. This new (clustered) task graph when scheduled onto the multiprocessor system shown in Fig. 10.5 has a completion time of 6.5 time units, as shown in Fig. 10.9. This schedule is better than the earlier multiprocessor schedule (Fig. 10.6) and also the uniprocessor schedule from the completion time point of view. Note that the E/C ratio of a task in this graph is 3/2.5 which is more than that of a task in the original (non-clustered) task graph. Thus clustering eliminates as much as possible the overhead caused by inter-task communication, while preserving as much degree of parallelism as possible. While a programmer expresses the *ideal* parallelism in an application at a fine granularity (Fig. 10.4), the compiler/run-time system extracts coarser-grained *useful* parallelism (Fig. 10.8) for a given multiprocessor system. There is another method called task duplication to eliminate the communication cost among the tasks (see Exercise 10.4).

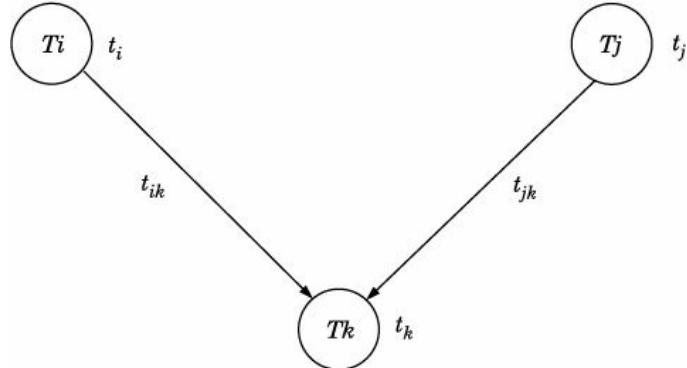


Figure 10.7 Task graph.

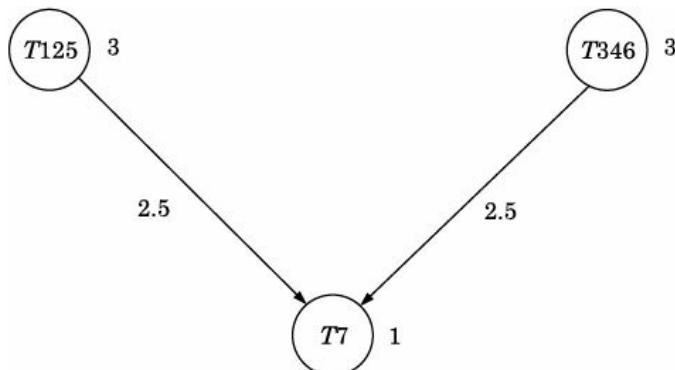


Figure 10.8 Clustered task graph.

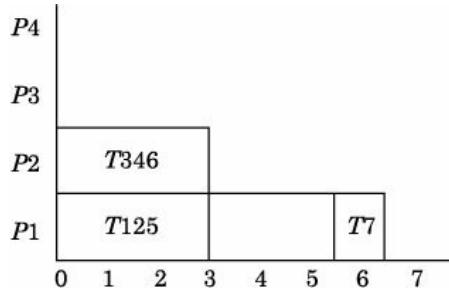


Figure 10.9 Schedule for the clustered task graph.

10.1.2 Dynamic Scheduling

Contrary to static scheduling, dynamic scheduling assumes little or no compile-time knowledge about the run-time parameters of a parallel program, such as task execution times and inter-task communication delays. Dynamic scheduling is based on the reassignment of tasks to the processors during execution time. This reassignment is performed by transferring tasks from the *heavily loaded* processors to the *lightly loaded* processors (called load balancing). A dynamic scheduling algorithm may be centralized or distributed. In a centralized dynamic scheduling algorithm, the responsibility of scheduling physically resides on a single processor. On the other hand, in a distributed dynamic scheduling algorithm, the load balancing operations involved in making task assignment decisions are physically distributed among the various processors in the system.

A dynamic scheduling algorithm is based on four policies:

1. *Load estimation policy*: This determines how to estimate the work load of a particular processor in the system. The work load is typically represented by a *load index* which is a quantitative measure of a processor's load. Some examples of load indices are processor's queue length (number of tasks on the processor) and processor's utilization (number of CPU cycles actually executed per unit of real time). Obviously, the CPU utilization of a heavily loaded processor will be greater than the CPU utilization of a lightly loaded processor.
2. *Task transfer policy*: This determines the suitability of a task for reassignment, i.e., the policy identifies whether or not a task is eligible to be transferred to another processor. In order to facilitate this, we need to devise a policy to decide whether a processor is lightly or heavily loaded. Most of the dynamic scheduling algorithms use the *threshold policy* to make this decision. The threshold value (fixed or varying) of a processor is the limiting value of its work load and is used to decide whether a processor is lightly or heavily loaded. Thus a new task at a processor is accepted locally for processing if the work load of the processor is below its threshold value at that time. Otherwise, an attempt is made to transfer the task to a lightly loaded processor.
3. *State information exchange policy*: This determines how to exchange the system load information among the processors. This policy decides whether the load information is to be distributed periodically (*periodic broadcast*) to all the processors or the load information is to be distributed on-demand (*on-demand exchange*) at load balancing time (a processor, for example, requests the state information of other processors when its state switches from light to heavy.) These two methods generate a number of messages for state information exchange as they involve a broadcast operation. To overcome this problem, a heavily loaded

processor can search for a suitable receiver (partner) by randomly polling the other processors one by one.

4. *Location policy*: This determines the processor to which a task should be transferred. This policy decides whether to select a processor randomly or select a processor with minimum load or select a processor whose load is below some threshold value.

Below we describe two dynamic scheduling algorithms—sender-initiated and receiver-initiated.

Sender-Initiated Algorithm

In a sender-initiated algorithm, a heavily loaded processor (sender) searches for lightly loaded processors (receivers) to which work may be transferred, i.e., when a processor's load becomes more than the threshold value, say L_{high} , it probes its neighbours one by one (assuming immediate or nearest neighbour state information exchange policy) to find a lightly loaded processor that can accept one or more of its tasks. A processor is a suitable candidate for receiving a task from the sender processor only if the transfer of the tasks to that processor would not increase the receiving processor's load above L_{high} . Once both the sender and the receiver(s) are identified, the next step is to determine the amount of load (number of tasks) to be transferred. The average load in the domain (sender and its neighbours which are potential receivers) is

$$L_{\text{avg}} = \frac{1}{K+1} \left(l_p + \sum_{k=1}^K l_k \right)$$

where l_p is the load of the sender, K is the total number of its immediate neighbours and l_k is the load of processor k . Each neighbour is assigned a weight h_k , according to $h_k = L_{\text{avg}} - l_k$ if $l_k < L_{\text{avg}}$; $= 0$ otherwise. These weights are summed to determine the total deficiency H_d :

$$H_d = \sum_{k=1}^K h_k$$

Finally, we define the proportion of processor p 's excess load, which is assigned to neighbour k as δ_k , such that

$$\delta_k = \left\lceil \frac{(l_p - L_{\text{avg}})h_k}{H_d} \right\rceil$$

Once the amount of load to be transferred is determined, then the appropriate number of tasks are dispatched to the receivers. Figure 10.10 shows an example of the sender-initiated algorithm.

Here we assume that $L_{\text{high}} = 10$. Hence, the algorithm identifies processor A as the sender and does its first calculation of the domain's average load:

$$L_{\text{avg}} = \frac{0 + 5 + 20 + 7 + 8}{5} = 8$$

The weight for each neighbourhood processor is then as follows:

Processor	B	C	D	E
Weight, h_k	8	3	1	0

These weights are summed to determine the total deficiency:

$$H_d = 8 + 3 + 1 + 0 = 12$$

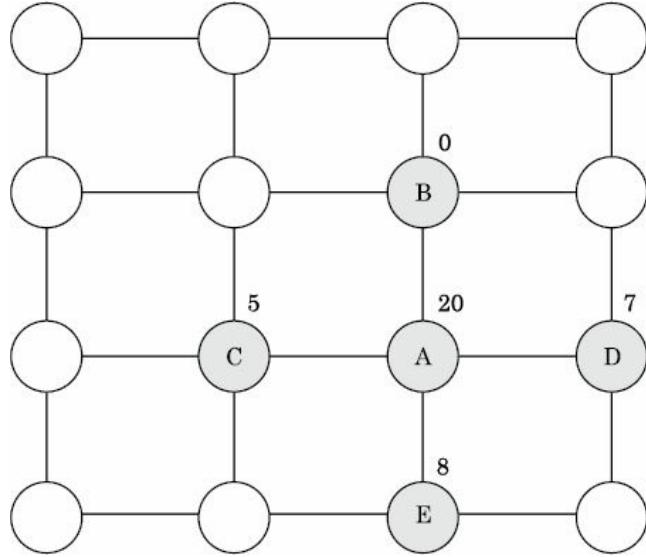


Figure 10.10 Example of sender-initiated algorithm in a 4×4 mesh.

The proportions of processor A's load that are assigned to its neighbours are

Processor	B	C	D	E
Load, δ_k	8	3	1	0

The final load on each processor is therefore 8.

Receiver-Initiated Algorithm

In a receiver-initiated algorithm, a lightly loaded processor (receiver) searches for heavily loaded processors (senders) from which work may be transferred i.e., when a processor's load falls below the threshold value, say L_{low} , it probes its neighbours one by one (assuming immediate neighbour state information exchange policy) to find a heavily loaded processor that can send one or more of its tasks. A processor is a suitable candidate for sending a task(s) only if the transfer of the task(s) from that processor would not reduce its load below L_{low} . Each neighbour k is assigned a weight h_k according to $h_k = l_k - L_{\text{avg}}$ if $l_k > L_{\text{avg}}$; $= 0$ otherwise. These weights are summed to determine the total surplus H_s . Processor p then determines a load portion δ_k to be transferred from its neighbour k :

$$\delta_k = \left\lceil \frac{(L_{\text{avg}} - l_p)h_k}{H_s} \right\rceil$$

Finally, processor p sends respective load requests to its neighbours. Figure 10.11 shows an example of the receiver-initiated algorithm. Here we assume that $L_{\text{low}} = 6$. Hence, the algorithm identifies processor A as the receiver and does its first calculation of the domain's average load:

$$L_{\text{avg}} = \frac{14 + 13 + 2 + 12 + 9}{5} = 10$$

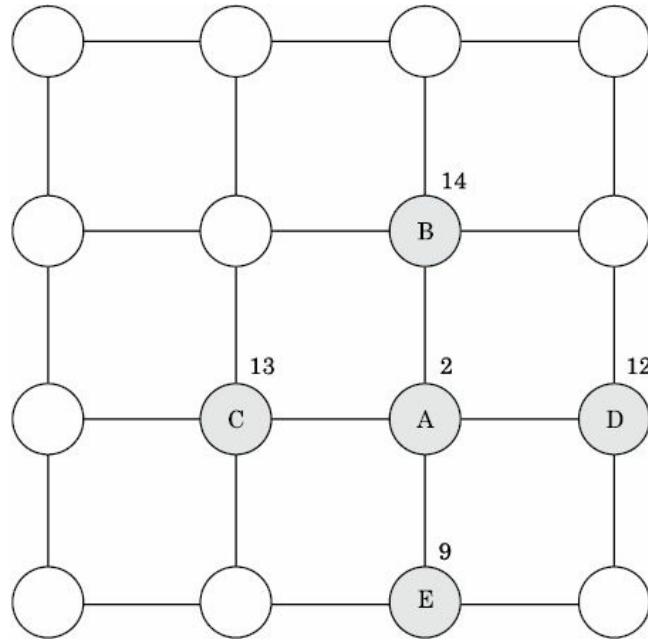


Figure 10.11 Example of receiver-initiated algorithm in a 4×4 mesh.

The weight for each neighbourhood processor is then as follows:

Processor	B	C	D	E
Weight, h_k	4	3	2	0

These weights are summed to determine the total surplus:

$$H_s = 4 + 3 + 2 + 0 = 9$$

The proportion of load that processor A requests from each neighbouring processor is

Processor	B	C	D	E
Load, δ_k	4	3	2	0

Thus the final load on each processor is as follows:

Processor	A	B	C	D	E
Load	11	10	10	10	9

Several researchers believe that dynamic scheduling (load balancing), which attempts to equalize the work load on all processors, is not an appropriate objective as the overhead involved (in gathering state information, selecting a task and processor for task transfer, and then transferring the task) may outweigh the potential performance improvement. Further, load balancing in the strictest sense may not be achievable because the number of tasks in a processor may fluctuate and the temporal unbalance among the processors exist at every moment, even if the average load is perfectly balanced. In fact, for proper utilization of the resources, it is enough to prevent the processors from being idle while some other processors have two or more tasks to execute. Algorithms which try to avoid *unshared states* (states in which some processor lies idle while tasks contend for service at other processors) are called dynamic load sharing algorithms. Note that load balancing (dynamic scheduling) algorithms also try to avoid unshared states but go a step beyond load sharing by attempting to equalize

the loads on all processors.

10.1.3 Task Scheduling in Shared Memory Parallel Computers

A parallel program that runs on a message passing (distributed memory) multi-processor system consists of a collection of tasks (*processes*). The main goal of task assignment is to assign these tasks to the processors in such a way that each processor has about the same amount of computations to perform and the total execution time of the program is minimized. A parallel program that runs on a shared memory multiprocessor system typically starts a single process and forks multiple *threads*. Thus, the program is a collection of threads. A significant feature of threads, which are separate control flows, is that threads belonging to a process share the address space of the process. While assigning these threads to the processors, the number of memory accesses to the shared memory should be taken into consideration. For example, when using a shared address space, it may be useful to assign two threads which share the same data to the same processor as this leads to a good cache usage. We discuss the problem of assigning the threads to the processors in the next section. The threads are scheduled on a pool of “kernel-level” threads (*virtual processors*) and the scheduler of the operating system schedules kernel-level threads to a pool of physical processors (cores). In general, inter-processor communication and migration costs in message passing multiprocessor systems can be significant. These are, in general, less of an issue in shared memory systems.

10.1.4 Task Scheduling for Multicore Processor Systems

We now turn our attention to the task scheduling problem in parallel computers with multicore processors. At a scheduling step, like for a single core processor, a ready task (an unscheduled task with no predecessors or having all its predecessors scheduled) is assigned to a core where it can finish at the earliest. It should be noted that communication between cores on a multicore processor is significantly faster than that between cores on different multicore processors. A ready task is assigned to a core, taking these differential (intra-/inter- multicore processor) communication costs into consideration. Task scheduling in the multicore era mainly focuses on energy-aware (energy-efficient) task scheduling— efficient task scheduling under energy constraints, joint optimization of compute performance (for example, execution time) and energy, etc. For the sake of simplicity and brevity, we consider the assignment of tasks to the cores in a single multicore processor.

There are three major mechanisms which allow the run-time system (or operating system) to manage a processor’s (core’s) energy consumption:

- (i) Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM)
- (ii) Thermal management
- (iii) Asymmetric (heterogeneous versus homogeneous) multicore design

DVFS-aware Task Scheduling

While DPM temporarily suspends the operation of the processor (or a core) and puts it in a low-power state, DVFS throttles the supply voltage and/or clock frequency of the processor (or a core) either to conserve power or to reduce the amount of heat generated by the processor. Dynamically throttling or suspending a processor, though conserves energy, results in reduced speed of the processor thereby increasing the execution time of a task or an application. The mathematical relationship among dynamic power consumption (P), clock

frequency (f) and supply voltage (V) of a core is given by $P \propto V^2f$, and a reduction in V and f reduces dynamic power cubically at the expense of upto a linear performance loss (execution time). While global DVFS allows for scaling of voltages and frequencies of all cores of a processor simultaneously, local DVFS allows for such scaling on a per-core basis thereby providing a flexibility to slow down or stop an overheating core.

A DVFS-aware task scheduling algorithm finds a schedule that saves the most energy while sacrificing the least performance. Figure 10.12 shows an example task graph that needs to be scheduled on a dual-core processor. The 2-tuple on a vertex describes its execution time (in time units)/energy (in energy units) without DVFS and with DVFS applied. Figure 10.13(a) and Fig. 10.13(b) show DVFS oblivious schedule and DVFS-aware schedule respectively. If our problem is to minimize the total execution time of the graph without exceeding the energy budget of 10 energy units, then such a schedule can be found with minimum completion time as 20 time units.

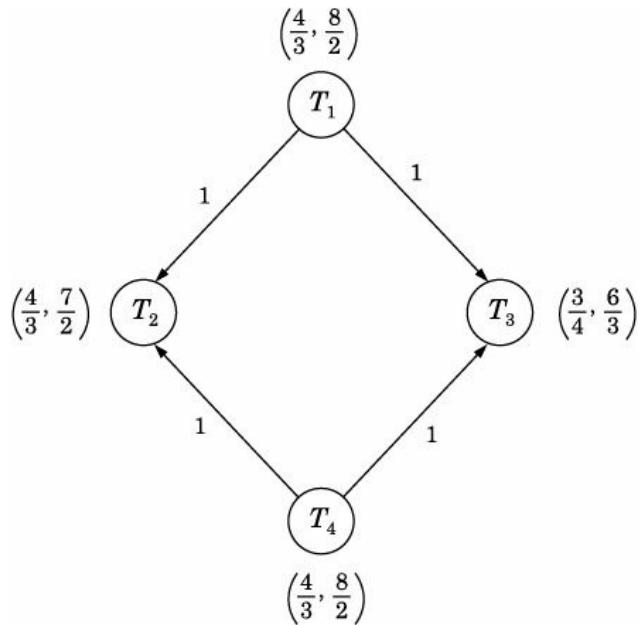
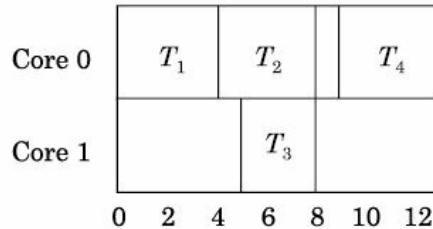


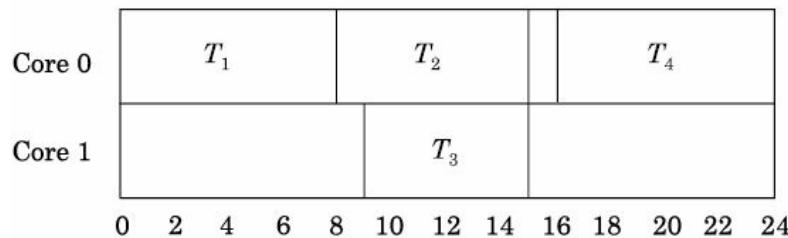
Figure 10.12 Task graph.

Time units = 13; Energy units = 13



(a) DVFS oblivious

Time units = 24; Energy units = 9



(b) DVFS-aware

Figure 10.13 Schedules for the task graph.

Thermal-aware Task Scheduling

Processor temperature has a significant impact on energy consumption. As the temperature of the processor becomes high, its thermal packaging and cooling (electricity) costs go up. Further, the increased heat dissipation may result in processor temperature exceeding a critical/threshold limit leading to processor damage or failure (*thermal emergency*). Thermal management enables the processor to operate at a high performance without exceeding its temperature limit.

Thermal management requires efficient temperature monitoring methods which can be classified into three categories:

- i. Sensor based methods use thermal sensors on the processor to obtain real-time temperature. It may be noted that efficient placement of limited number of thermal sensors on the processor chip (for producing a thermal profile) with their associated overhead (sensor's power and run-time performance overhead) is as important as the sensing (temperature estimation).
- ii. Thermal model based methods use thermal models for estimating the temperature based on chip's hardware characteristics, power consumption and ambient temperature.
- iii. Performance counter based methods estimate the temperature using the values read from specific registers (counters). A processor has many functional units each consuming a different amount of power per access. The access rates of these functional units can be monitored using programmable performance counters. The readings of the counters which reflect the activity information of functional units are used to estimate the power consumption and also temperature using a simple thermal model.

A thermal-aware task scheduling algorithm minimizes the schedule length avoiding thermal hotspots (by task/thread migration from an overheating/hot core to a cool core that is

performing less work) and temperature gradients (to avoid one area on the processor chip becoming much hotter than another. For instance, running two heat-generating tasks/threads on adjacent cores results in higher temperature gradient than running these two tasks/threads on separate parts of the processor chip). Figure 10.14 shows an example task graph that needs to be scheduled on a dual-core processor. The weight on a vertex denotes its execution time (in time units). Figure 10.15(a) and Fig. 10.15(b) show thermal oblivious schedule and thermal-aware schedule respectively. It is assumed that thermal emergency occurs if a core continuously executes more than 3 time units (thermal constraint). Note that in Fig. 10.15(a) thermal constraint is violated.

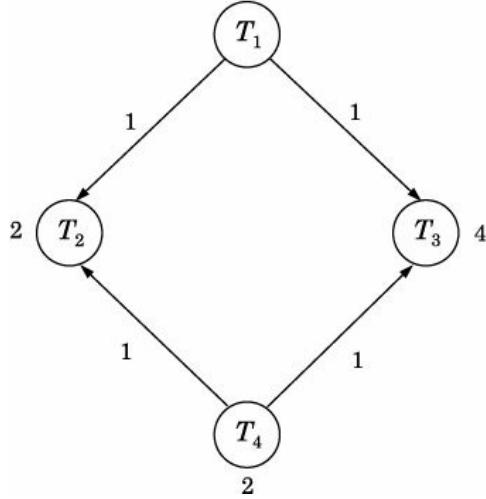


Figure 10.14 Task graph.

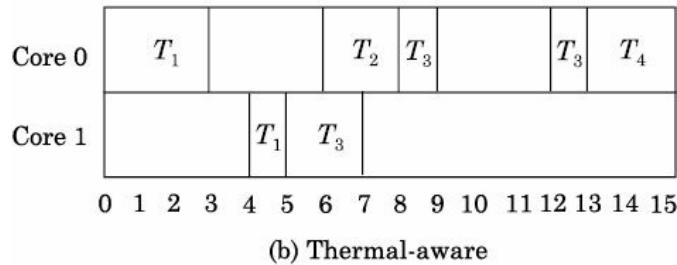
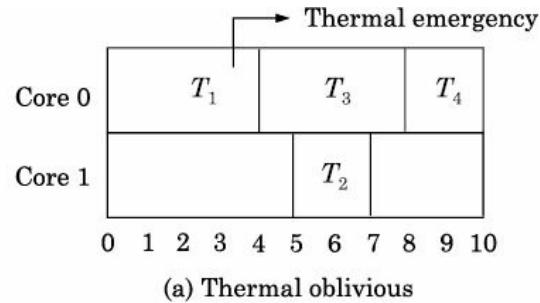


Figure 10.15 Schedules for the task graph.

Asymmetry-aware Task Scheduling

An asymmetric multicore processor typically consists of a small number of fast and powerful cores (with high clock speed, superscalar out of order execution pipeline, aggressive prefetching, sophisticated branch predictor, etc.) and a large number of slower and low power cores (with slower clock speed, simple pipeline, lower power consumption, etc.). This asymmetry by design is different from the (asymmetric) processor in which there is non-uniform DVFS on different cores. An asymmetric-aware task scheduling algorithm assigns

tasks/threads to cores in such a way that tasks/threads with high ILP (and also sequential phases in the application) are assigned to fast cores and tasks/threads with low ILP are assigned to slow cores to achieve better performance per watt. Figure 10.16 shows an example task graph that needs to be scheduled on a dual-core processor. The 2-tuple on a vertex describes its execution time (in time units)/power (in power units) on Cores 0 and 1. Figure 10.17(a) and Fig. 10.17(b) show schedules on symmetric (both cores are Core 0 type) and asymmetric dual-core processors, assuming that the power consumed by the switch/interconnect for effecting inter-core communication is negligible. Taking performance = 1/execution time (if a processor takes longer time to execute the same number of instructions, its performance is considered to be degraded), the performance per power metric for asymmetric (heterogeneous) processor ($1/(9 \times 12)$) is higher than that of symmetric (homogeneous) processor ($1/(12 \times 10)$).

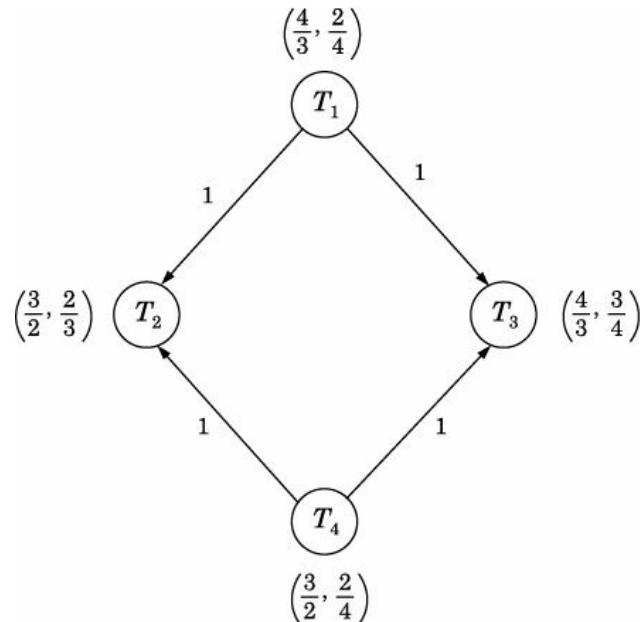
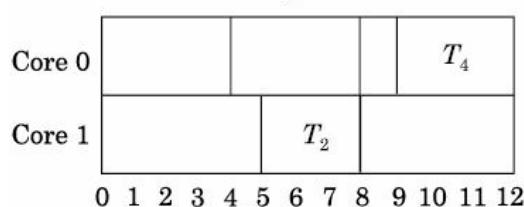


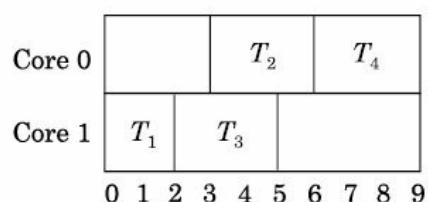
Figure 10.16 Task graph.

Time units = 12, Power units = 10



(a) Symmetric processor

Time units = 9, Power units = 12



(b) Asymmetric processor

Figure 10.17 Schedules for the task graph.

Contention-aware Task Scheduling

As we saw in Chapter 5, the cores of a multicore processor are not fully independent processors but share resources such as caches, prefetching hardware, memory bus and DRAM memory controllers. Contention for these shared resources degrades the performance of an application. Cache contention can occur when two or more threads, belonging to either different applications where threads do not share any data or to the same application where each thread works on its own data, are scheduled to execute on the cores that share the same cache. Threads that share data may actually benefit if they are assigned to run on the cores that share the same cache. An application suffers extra cache misses if its co-runners (the other threads running on cores that share the same cache) bring their own data into the cache evicting the data of the application. Cache miss rate of a thread (the fraction of memory accesses in which a thread fails to find data in the cache and therefore fetches it from the memory) is found to be an excellent predictor for the contention for all the types of shared resources mentioned above. Thus a contention-aware scheduler, to mitigate the performance loss due to shared-resource contention, assigns threads to the cores in such a way that high-miss rate threads are kept apart, that is, they do not run on the cores that share the same cache. Such contention-aware techniques can be combined with energy-aware task/thread scheduling leading to better solutions.

The technological advancements will enable integration of hundreds and thousands of cores onto a single die. The many-core processors impose a big challenge to manage the cores in a scalable manner. In order to achieve a high degree of scalability in such processors, a distributed resource management is necessary as the centralized management is not scalable with the number of cores, especially in handling dynamic workloads (when the number of tasks, their average execution times, dependencies among them, etc. are known only at runtime, not at compile-time). As we have seen earlier, dynamic scheduling involves reassignment of tasks to the cores during execution time. Task migration involves overhead such as the costs associated with interrupting a task that is identified for migration, saving its context, transmission of all the required data to a lightly loaded core and then restarting the task on this core. Development of scalable, distributed resource management schemes which consider thermal issues in many-core, both symmetric (homogeneous) and asymmetric (heterogeneous), processors is an active area of research.

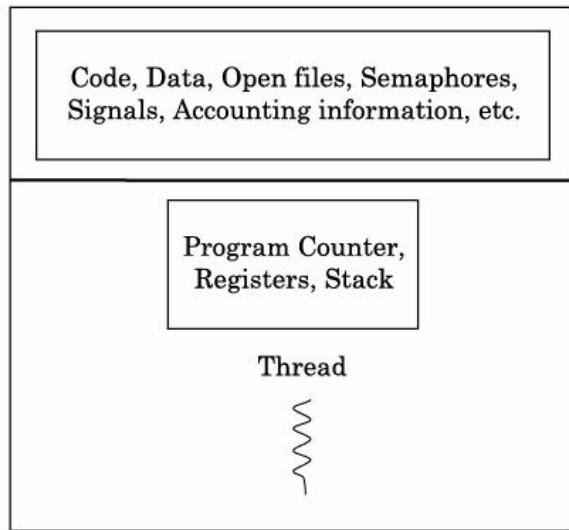
10.2 PROCESS MANAGEMENT

The classic functions of an operating system include creation and management of processes. The effectiveness of parallel computing depends on the performance of such primitives offered by the system to express parallelism within an application. One way to express parallelism is by using *heavyweight* processes (for example, UNIX processes) sharing portions of their address space. Such a process consists of its own program counter, its own register states, its own stack and its own address space. The parallelism expressed using heavyweight processes is coarse-grained and can be inefficient for parallel programs. First, the creation and deletion of heavyweight processes is expensive because the operating system treats a process and its address space as a single entity, so that processes and address space are created, scheduled and destroyed together. Second, context switches for such processes are expensive because there are additional costs due to address space management. This is because the amount of state (page tables, file descriptors, outstanding I/O requests, register values, etc.) associated with a process is large. Finally, there is a long term cost associated with cache and TLB performance due to the address space change. The above problems led to the idea of *threads* or *lightweight* processes.

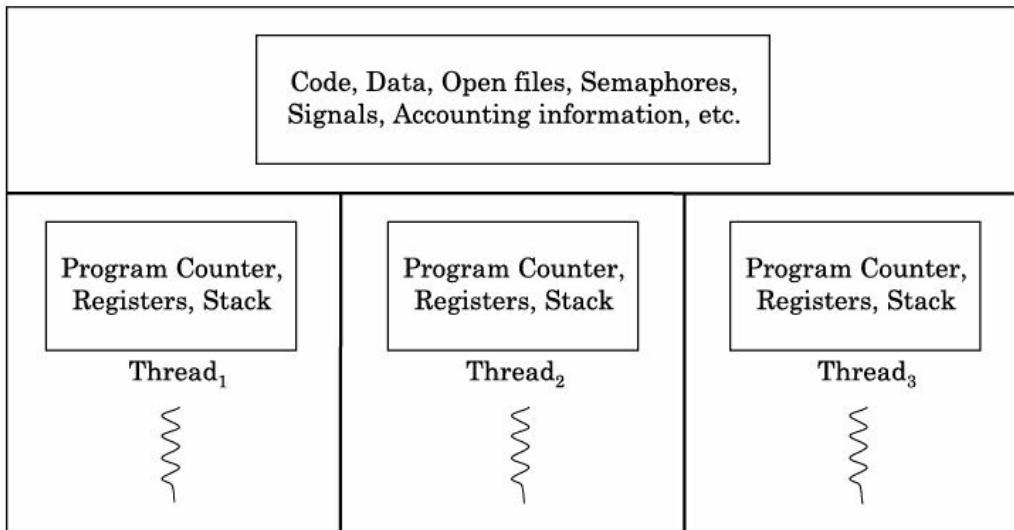
10.2.1 Threads

It should be noted that *hardware multithreading* means the use of multiple-context (for example, multithreaded) processors to hide memory latency as discussed in Chapter 3. What we consider here is *software multithreading* which is a different concept. Hardware threads means physical resources, whereas software threads, as we discuss in this section, are those created by application (user) programs. The Intel Core i7 processor has four cores and each core can execute two software threads at the same time using hyper-threading technology. Thus, the processor has eight hardware threads which can execute upto eight software threads simultaneously. The application program can create any number of software threads. It is the responsibility of the operating system to assign software threads to hardware threads for the execution of the application program.

In operating systems with threads facility, a process consists of an address space and one or more threads, which are executable (i.e., schedulable) entities, as shown in Fig. 10.18. Each thread of a process has its own program counter, its own register states and its own stack. But all the threads of a process share the same address space. In addition, they also share the same set of operating system resources, such as open files, semaphores and signals. Thus, switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address spaces. Because of this, threads are called *lightweight processes*. (In a similar way, a hardware thread can be called as a *lightweight processor*.) Threads can come and go quickly, without a great deal of system overhead. Further, when a newly created thread accesses code and data that have recently been accessed by other threads within its process, it automatically takes advantage of main memory caching that has already taken place. Thus, threads are the right constructs, from an operating system point of view, to efficiently support fine-grained parallelism on shared memory parallel computers or on multicore processors.



(a) Single-threaded process



(b) Multithreaded process

Figure 10.18 Single-threaded and multithreaded processes.

An operating system that supports threads facility must provide a set of primitives, called *threads library* or *threads package*, to its users for creating, terminating, synchronizing and scheduling threads. Threads can be created statically or dynamically. In the static approach, the number of threads of a process is decided at the time of writing the corresponding program or when the program is compiled, and a fixed size of stack is allocated to each thread. In the dynamic approach, a process is started with a single thread, new threads are created (dynamically) during the execution of the process, and the stack size of a thread is specified as a parameter to the system call for thread creation. Termination and synchronization of threads is performed in a manner similar to those with conventional (heavyweight) processes.

Implementing a Threads Library

A threads library can be implemented either in the user space or in the kernel. (The *kernel* that is always in memory is the essential, indispensable program in an operating system which directly manages system resources, handles exceptions and controls processes). In the first approach, referred to as user-level approach, the user space consists of a run-time system

which is a collection of thread management routines. These routines are linked at run-time to applications. Kernel intervention is not required for the management of threads. In the second approach, referred to as kernel-level approach, there is no run-time system and the kernel provides the operations for thread management. As we will see now both approaches have their own advantages and limitations.

In the user-level approach, a threads library can be implemented on top of an existing operating system that does not support threads. This means no changes are required in the existing operating system kernel to support user-level threads. The threads library is a set of application-level utilities shared by all applications. But this is not possible in kernel-level approach as all of the work of thread management is done by the kernel. In the user-level approach, scheduling of threads can be application specific. Users have the flexibility to use their own customized algorithm to schedule the threads of a process. This is not possible in kernel level approach as a scheduler is already built into the kernel. However, users may have the flexibility to select an algorithm, through system call parameters, from a set of already implemented scheduling algorithms.

Thread switching in user-level approach is faster as it does not require kernel mode privileges as all of the thread management data structures are within the user space. This saves the overhead of two mode switches (user to kernel and kernel back to user). A serious drawback associated with the user-level approach is that a multithreaded application cannot take advantage of multiprocessing. The kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time and further, there is no way to interrupt the thread. All this is due to lack of clock interrupts within a single process. Note that this problem is not there in kernel-level approach. A way to solve this problem is to have the run-time system requests a clock interrupt (after every fixed unit of time) to give it control so that the scheduler can decide whether to continue to run the same thread or switch to another thread.

Another drawback of the user-level approach is as follows. In a typical operating system, most system calls are blocking. Thus, when a thread executes a system call, not only is that thread blocked but all threads of its process are blocked and the kernel will schedule another process to run. This defeats the basic purpose of using threads. Again, note that this problem is not there in kernel-level approach. A way to solve this problem of blocking threads is to use a technique called *jacketing*. The purpose of jacketing is to convert a blocking system call into a non-blocking system call. For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine. This jacket routine contains a code which checks whether the I/O device is busy. If it is, the thread enters the *ready* state and passes control (through the threads library) to another thread. When this thread later is given control again, it checks the I/O device again.

Threads Scheduling

Threads library often provides calls to give users or application programmers the flexibility to schedule the threads. We now describe a few threads scheduling algorithms that may be supported by a threads library.

1. *FCFS or Round-robin*: Here the threads are scheduled on a first come, first served basis or using the round-robin method where the CPU cycles are equally timeshared among the threads on a quantum-by-quantum basis. A fixed-length time quantum may not be appropriate on a parallel computer as there may be fewer runnable threads than the available processors. Therefore, instead of using a fixed-

- length time quantum, a scheduling algorithm may vary the size of the time quantum inversely with the total number of threads in the system.
2. *Hand-off Scheduling*: Here a thread can choose its successor or the next thread for running, i.e., the current thread hands-off the processor to another thread thereby eliminating the scheduler interference (or bypassing the queue of runnable threads). Hand-off scheduling has been shown to perform better when program synchronization is exploited (for example, the requester thread hands-off the processor to the holder of a lock) and when inter-process communication takes place (for example, the sender hands the processor off to a receiver).
 3. *Affinity Scheduling*: Here a thread is scheduled on a processor on which it last executed hoping that part of its address space (*working set*) is still present in the processor's cache.
 4. *Coscheduling (Gang Scheduling)*: The goal of coscheduling is to achieve a high degree of simultaneous execution of threads belonging to a single program or application. A coscheduling algorithm schedules the runnable threads of an application to run simultaneously on different processors. Application preemption implies the simultaneous preemption of all its threads. Effectively, the system context switches between applications. Since the overhead involved here can be significant, processors are often dedicated for periods longer than the typical time-sharing quantum to amortize this overhead.

In order to provide portability of threaded programs, IEEE has defined a POSIX (Portable Operating System Interface for Computer Environments) threads standard called as *Pthreads*. The threads library Pthreads defines function calls to create, schedule, synchronize and destroy threads. Other threads libraries include Win32 threads and Java threads. Most operating systems support kernel-level threads.

Threads Execution Models

A parallel program has one or more processes, each of which contains one or more threads and each of these threads is mapped to a processor by the scheduler in the operating system. There are three mapping models used between threads and processors: (i) many-to-one (M:1 or User-level threading), (ii) one-to-one (1:1 or Kernel-level threading) and (iii) many-to-many (M:N or Hybrid threading), as shown in Fig. 10.19.

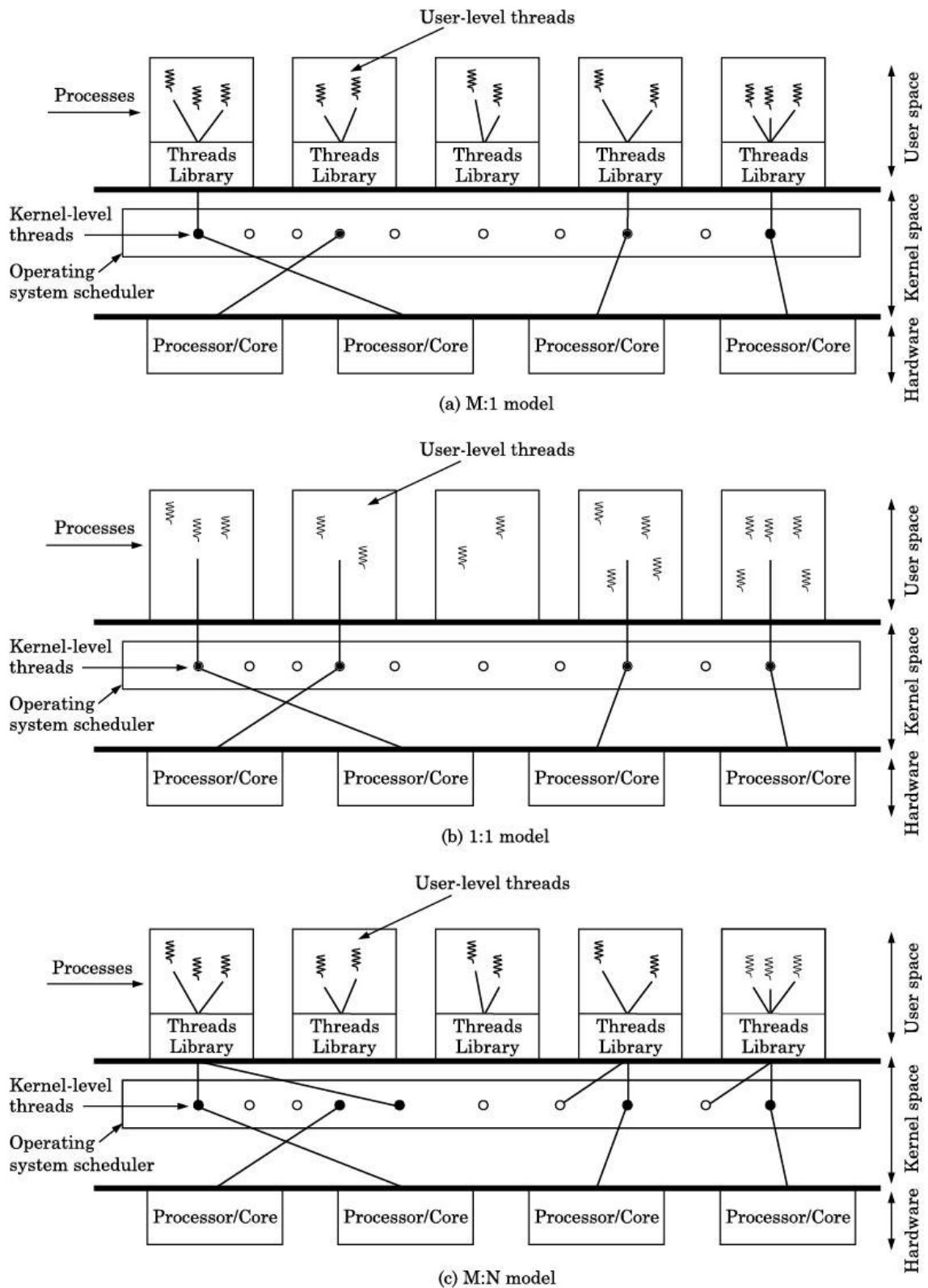


Figure 10.19 Mapping of threads to processors/cores.

In the M:1 model, many user-level threads are mapped to a single kernel-level thread. Since a single kernel-level thread (schedulable entity) can operate only on a single processor, this model does not allow the execution of different threads of one process on different processors. This means only process-level parallelism is possible here, but not thread-level. The library scheduler determines which thread gets the priority. This is also called as

cooperative multithreading. This model is used in systems that do not support kernel-level threads. Some examples of this model are Solaris Green threads and GNU Portable threads.

In the 1:1 model, each user-level thread maps to a kernel-level thread. This facilitates the parallel execution within a single process. This model does not require the threads library scheduler and the operating system does the thread scheduling. This is also called as *preemptive multithreading*. A serious drawback of this model is that generation of every user-level thread of a process must include the creation of a corresponding kernel-level thread and further the mode switches overhead due to transfer of control across multiple user-level threads of the same process cause a significant overhead which has an impact on the performance of the parent process. Some operating systems address this problem by limiting the growth of the thread count. Some examples of this model are Linux, Windows NT/XP/2000, and Solaris 9 and later.

In the (M:N) mapping model, a number of user-level threads are associated to an equal or smaller number of kernel-level threads. At different points in time, a user-level thread may be mapped to a different kernel-level thread and correspondingly a kernel-level thread may execute different user-level threads at different points in time. Some examples of this model include Windows NT/2000 with ThreadFiber package and Solaris prior to version 9.

A variation of M:N mapping model, an example of which is HP-UX, is the two-tier model which allows either M:N or 1:1 operation. It is important to note that in the M:1 model, if a user-level thread blocks (due to a blocking system call for I/O or a page fault) then the entire process blocks, even if the other user-level threads of the same process would otherwise be able to continue. Both 1:1 and M:N models overcome this problem but they require the implementation of threads library at the kernel-level. The M:N mapping model is flexible as the programmer can adapt the number of user-level threads depending on the parallel algorithm (application) used. The operating system can select the number of kernel-level threads in a way that results in efficient utilization of the processors. A programmer, for example by selecting a scheduling algorithm using Pthreads library, can influence only the library scheduler but not the scheduler of the operating system which is fair, starvation-free and tuned for an efficient utilization of the processors.

It is clear by now that the creation of user-level threads is necessary to exploit fine-grained parallelism in a program (application). Since threads of one process share the address space of the process, if one thread modifies a value in the shared address space, the effect is visible to all threads of the same process. A natural programming model for shared memory parallel computers or multicore processors/systems is a thread model in which all threads have access to shared variables. As mentioned earlier, to coordinate access to shared variables, synchronization of threads is performed in a manner similar to those with conventional (heavyweight) processes described in the next section.

10.3 PROCESS SYNCHRONIZATION

The execution of a parallel program on a parallel computer (multiprocessor) may require processors to access shared data structures and thus may cause the processors to concurrently access a location in the shared memory. Hence a mechanism is needed to serialize this access to shared data structures to guarantee its correctness. This is the classic mutual exclusion problem.

In order to realize mutual exclusion, multiprocessor operating systems provide instructions to atomically read and write a single memory location (in the main memory). If the operation on the shared data is very elementary (for example, an integer increment), it can be embedded in a single atomic machine instruction. In such a case, mutual exclusion can be realized completely in hardware. However, if an access to a shared data structure constitutes several instructions (*critical section*), then primitives such as lock and unlock are needed to ensure mutual exclusion. In such a case, the acquisition of a lock itself entails performing an elementary operation on a shared variable (which indicates the status of the lock). Atomic machine instructions can be used to implement lock/unlock operations. We now first present one such machine (hardware) instruction called test-and-set and then describe how it can be used to implement lock/unlock operations.

The test-and-set instruction atomically reads and modifies the contents of a memory location in one memory cycle. It is defined as follows (variable m is a memory location):

```
function test-and-set (var m: boolean) : boolean;
begin
    test-and-set := m;
    m := true
end;
```

The test-and-set instruction returns the current value of variable m and sets it to **true**. This instruction can be used to implement lock and unlock operations in the following way (S, a *binary semaphore*—a non-negative integer variable, is also called a lock):

```
lock(S): while test-and-set (S) do nothing;
unlock(S): S := false;
```

Initially, S is set to **false**. When a **lock(S)** operation is executed for the first time, test-and-set(S) return a **false** value (and sets S to **true**) and the “while” loop of the **lock(S)** operation terminates. All subsequent executions of **lock(S)** keep looping because S is **true** until an **unlock(S)** operation is executed.

In the above implementation of a lock operation, several processors may *wait* for the lock, S, to open by executing the respective atomic machine language instructions concurrently. This wait can be implemented in two ways:

1. *Busy-waiting*: In busy-waiting, processors continuously execute the atomic instructions to check for the status of the shared variable. Busy-waiting (also called *spinlock*) wastes processor cycles and consumes the bandwidth of the network connecting memory modules to the processors. Increased traffic on the network due to busy-waiting can interfere with normal memory accesses and degrade the system performance due to the increased memory access time.
2. *Queueing*: In queueing (also called *blockinglock*), a waiting process is blocked, i.e., it is placed in a queue. It is dequeued and activated by an unlock operation,

i.e., it is awakened by the process releasing the lock. Although queueing eliminates network traffic and the wastage of processor cycles due to busy-waiting, it introduces other processing overhead because the enqueue and the dequeue operations require the execution of several instructions. Further, the queue forms a shared data structure and must be protected against concurrent access.

A *barrier lock* implements a barrier in a parallel program. Once a process reaches a barrier, it is allowed to proceed if and only if all other cooperating processes reach the barrier. A waiting process may either spin or block depending on the implementation of the lock.

In message passing parallel computers, synchronization between processes comes for free; data exchanges synchronize processes.

Many commercially available machines offer a rich set of global operations (for example, collective communications), which are called *idioms* by compiler writers, for use of application programs for efficient inter processor communication/synchronization.

10.3.1 Transactional Memory

Transactional memory is an alternative to using locks to enforce synchronization. It brings the central idea of transactions from database systems into programming languages and has attracted a lot of research focus during the last decade. The basic idea behind transactional memory is to treat critical section codes in shared memory parallel programs as transactions.

In transactional memory system, a *transaction* is a finite code sequence that satisfies two properties:

- i. *Atomicity (all or nothing)*: Either the whole transaction is executed (*committed*) or none of it is done (*aborted*). If a transaction commits, then all its operations (memory updates) appear to take effect as one unit, as if all the operations happened instantaneously. If a transaction aborts, then none of its operations appear to take effect, as if the transaction never happened, which means the execution is rolled back to the start of the transaction.
- ii. *Isolation*: A transaction executes in isolation, meaning individual memory updates within an ongoing transaction are not visible outside the transaction. Only when the transaction commits, all the memory updates are made visible to the rest of the system (other processes).

As discussed in Chapter 3, a modern processor with a branch predictor speculatively executing instructions after a branch, either commits the results or aborts and discards the changes based on whether the branch is really taken. Transactional memory systems can be viewed as an extension of this idea, with cache and registers states treated as processor's internal state, rather than the global state.

We now illustrate the differences between the use of lock mechanism and transactional memory using an example. A company has three accounts, namely A, B and C, with a bank. The accounts are accessed by the company employees concurrently using three functions, withdraw(), deposit() and transfer(). A pseudocode, which is self-explanatory, of these functions is given below using both locks and transactions. The critical section is protected by lock/unlock and transaction tags (Transaction Begin/Transaction End). For simplicity, we ignore error handling codes, that is, code to check whether account A has sufficient funds to effect withdrawal/transfer is not shown.

```

/* Using Locks */
withdraw(Account A, WithdrawAmt)
begin
    lock(S[A])
    A.balance = A.balance - WithdrawAmt
    unlock(S[A])
end

deposit(Account A, DepositAmt)
begin
    lock(S[A])
    A.balance = A.balance + DepositAmt
    unlock(S[A])
end

transfer(Account A, Account B, Amt)
begin
    lock(S[A])
    lock(S[B])))

    A.balance = A.balance - Amt
    B.balance = B.balance + Amt
    unlock(S[A])
    unlock(S[B])
end

/* Using Transactional memory */
withdraw(Account A, WithdrawAmt)
begin
    Transaction Begin
        A.balance = A.balance - WithdrawAmt
    Transaction End
end

deposit(Account A, DepositAmt)
begin
    Transaction Begin
        A.balance = A.balance + DepositAmt
    Transaction End
end

transfer(Account A, Account B, Amt)
begin
    Transaction Begin
        withdraw (A, Amt)
        deposit (B, Amt)
    Transaction End
end

```

As you notice a programmer, using locks, specifies both “What” (operations to be carried out) and “How” (to/where to acquire and release locks). On the other hand, the programmer using transactional memory specifies only “What”, but not “How”. The run-time system implements (“How”) synchronization. This means the responsibility for guaranteeing

transaction atomicity and isolation is transferred to the run-time system.

Fine-grained (data element level, for example different array elements) locking is required to promote concurrency for providing good scalability. This can lead to very complicated (messy) code with a number of locks/unlocks spread all over the place that has a potential to create deadlocks due to differences in locking order. Transactions, on the other hand, can be implemented in such a way that they allow both concurrent read accesses and concurrent accesses to disjoint fine-grained data elements. Using transactions, the programmer gets these forms of concurrency without explicitly coding them in the program. Another big advantage of transactional memory is that the program using simple and elegant transaction primitives (tags) is very readable and understandable.

Another serious issue with locks is that they are not composable. This means we cannot take two correct lock-based pieces of code, combine (compose) them and say that the result is still correct, without examining their implementations. This means larger programs cannot be built using smaller programs without modifying the smaller programs. The transfer() function, in the above example, cannot be built by using withdraw() and deposit() functions by replacing the two statements $A.balance = A.balance - Amt$ and $B.balance = B.balance - Amt$ with withdraw(A, Amt) and deposit(B, Amt) respectively as this causes a deadlock. Transactions, on the other hand, allow the programmer to compose libraries into larger programs.

The run-time system, in order to ensure transaction atomicity and isolation, must provide the following two key mechanisms: (i) *version control* and (ii) *conflict detection and resolution*.

When a transaction executes, the run-time system must simultaneously manage two versions of data. A new version (new values of data items) produced by this transaction becomes visible only if the transaction commits and the old version (a copy of old values of data items) is needed when the transaction aborts to revert (roll back) to the beginning of the transaction. With *eager versioning*, also called *direct update*, a write within a transaction is performed directly in memory (new version in memory) and old version is buffered in an *undo log*, also called as *transaction log*. When the transaction commits, the memory already contains the new version and the undo log is flushed. If the transaction aborts, the old version is restored by fetching from the undo log and rewriting to memory. With *lazy versioning*, also called as *deferred update*, a write within a transaction is buffered (new version in a write buffer) till the transaction commits. When the transaction commits, the new version is visible by copying from the write buffer to the actual memory addresses. If the transaction aborts, the write buffer is flushed.

A read-write or write-read or write-write conflict occurs when two or more transactions operate concurrently on the same data. In order to detect conflicts, each transaction needs to keep track of its read-set (the set of data items or memory locations that are read by a transaction) and write-set (the set of data items or memory locations that are written by a transaction). The union of the read-set and write-set, which is the set of data items that have been accessed, is called as *footprint* of the transaction. Similar to version control, there are two basic approaches to conflict detection and resolution to guarantee atomic execution of a transaction. *Eager* (also referred to as *early* or *pessimistic*) conflict detection and resolution checks for conflicts during each read and write thereby detecting a conflict directly when it occurs and handles them either by stalling one of the transactions in place or by aborting one transaction and retrying it later. *Lazy* (also referred to as *late* or *optimistic*) conflict detection and resolution, assuming conflicts are rare, checks for conflicts when a transaction tries to commit (postpones all checks till the end of transaction). The write-set of the committing

transaction is compared with that of the read- and write-sets of other transactions. In case of a conflict, the committing transaction succeeds and the other transactions abort. As conflicts are detected late, after the point a transaction reads/writes the data, stalling in place is not a viable option for conflict resolution. The *granularity* of conflict detection (for example, tracking the read- and write-sets at the word granularity or at the cache line granularity) is an important design decision in a transactional memory system.

Nested transactions (transactions within transactions) may occur frequently with transactional memory systems as we have mentioned earlier that transactions can be composed easily and safely. Such transactions extend atomicity and isolation of inner transactions until the outer-most transaction commits. A simple solution to handle nested transactions is to flatten the nest into a single transaction, by subsuming inner transactions within the outermost. In this approach an abort of an inner transaction may cause a complete abort to the beginning of the outermost transaction, which may result in low performance. Another approach is to allow a committing inner transaction to release isolation immediately. Though this potentially results in higher parallelism, there is a higher cost in terms of more complex software and hardware.

A question arises whether non-transactional code can, from outside the transaction, access non-committed updated values within an ongoing transaction. With *strong atomicity*, also called as *strong isolation*, non-transactional code cannot read non-committed updates. With *weak atomicity*, also called as *weak isolation*, non-committed updates can be read from the outside of a transaction. It is important to note that programs that execute correctly under the weak atomicity model may not execute correctly under the strong atomicity model, as transactions may not appear atomic when interleaved with non-transactional code. The atomicity model for a transactional system is analogous to a memory consistency model for a shared memory multiprocessor system.

A transactional memory system has a major limitation of performance (speed). When a transaction rolls back, all the work that it has done is essentially wasted. A transactional memory system can be implemented in hardware (HTM—Hardware Transactional Memory), in software (STM—Software Transactional Memory) or in both. The most important argument in favour of HTM system is performance. HTM systems have, in general, better performance than both traditional lock-based mechanisms and STM systems. Early designs of HTM systems used cache coherence protocols and kept a transaction's modified state in a cache for conflict detection. Recent proposals of HTM systems have explored using a processor's write buffer which holds transactional updates. A serious drawback of HTM systems is the limitation in hardware resources. Caches used for tracking transaction's footprint and write buffer have finite capacity and may overflow on long and nested transactions. It, therefore, follows that HTM systems are best suited for short and small transactions. Handling transaction state in caches is challenging when interrupts or page faults occur. A solution to the problems mentioned above is to *virtualize* the finite resources. When hardware resources are exceeded, the transactions are rolled back and restarted in STM mode. A challenge with such an implementation is conflict detection between hardware and software transactions. STM systems do not require any changes to the hardware and are not bound by resource limitations. The most serious problem with STM systems is performance. In spite of improved designs of STM systems over the years, they are often significantly slower than traditional lock-based mechanisms and HTM systems. HTMs mitigate most STM overheads by augmenting hardware with state to track read/write sets, leveraging cache coherence to detect conflicts and using write buffers or caches to hold tentative writes. Some recent commercial implementations of HTMs include IBM's BlueGene/Q processor and

Intel's Haswell processor.

Parallel programming using transactions is an active area of research. Transactional memory systems provide a promising approach to write scalable parallel programs for multiprocessor systems with a shared address space like multicore processors.

10.4 INTER-PROCESS COMMUNICATION

Cooperating processes in a multiprocessor environment often communicate and synchronize. In shared memory multiprocessor systems, the processes communicate via shared variables, whereas in message passing systems basic communication primitives such as *send* and *receive* are used for inter-process communication. As we have seen earlier in Chapter 8, these primitives may be *blocking* or *non-blocking*. Thus communication between processes using such primitives may be either *synchronous* or *asynchronous*. The issues that must be considered for an inter-process communication mechanism include (i) whether the underlying hardware supports reliable (packets, constituting a message, are delivered correctly in the right order) or unreliable communication, (ii) whether messages are typed or untyped and of fixed or variable length, (iii) how to support message priority (it may be necessary that some messages are handled at higher priorities than others), and (iv) how the kernel and user programs must interact to result in efficient message transfers.

10.5 MEMORY MANAGEMENT

A typical memory system manages a memory hierarchy consisting of a cache, uniformly accessible main memory and significantly slower secondary memory. The goal in the design of memory hierarchy is to maximize the number of instruction/data accesses from the fast/cache memory thereby leading to a small average memory access time. Shared memory multiprocessor systems commonly use *caching* to reduce memory access time. Under caching, every processor has a private memory, called a *cache*, in addition to the shared global memory. When a processor needs to fetch a word from a data block in the global memory, it fetches the entire block and saves it in its cache for future use. Note that a global memory accesses through the interconnection network is much slower as compared to the cache access time. Further, the global memory access time may not be constant because of contention at the network. If the locality of data reference is high, caching will substantially reduce the effective memory access time. The cost of fetching the entire block from the global memory is amortized over several accesses to that block when it is in the cache. Thus, caching has two advantages: (i) it reduces the traffic over the interconnection network, and (ii) it reduces contention at memory modules as different blocks of a memory module can be fetched by several processors and can be accessed concurrently from their respective caches.

Although exploiting locality by the introduction of a cache improves memory access time, there is still the problem of latency on the first (cold or compulsory) miss. The latency problem can be solved by *prefetching*. The idea of cache prefetching is to predict data (or instruction) access needs in advance so that a specific piece of data (or a block of instructions) is loaded from the main memory before it is actually needed by the application program. For example, when a cold miss fetches memory block i into the cache, block $i + 1$ is prefetched into the cache thereby supplying the cache with data (instructions) ahead of the actual requirements of the application. Cache prefetching can be implemented in hardware (by using a *prefetch engine* which monitors the bus between two memory levels, say L1 and L2, and issues miss requests to L2 cache automatically) or in software (by inserting explicit prefetch instructions in the program by the programmer and/or compiler). Instead of prefetching directly in the cache, prefetching in a small prefetch buffer avoids polluting the cache with prefetch blocks that are not accessed.

Recall that caching poses a problem when a processor modifies a cached block that is also currently cached by some other processors. This is called the *cache coherence* problem. Recall that we studied two approaches, *write update* and *write-invalidate*, in Chapter 4 that address the cache coherence problem. In the write update method, a process that is modifying a block also modifies the copies of this block in the cache of other processors. In the write-invalidate method, a process that is modifying a block invalidates the copies of this block in the cache of other processors. There are several variations of both these approaches and their implementation requires hardware support and also depends on the type of interconnection network employed. (In distributed shared memory systems, the cache coherence problem arises at the software level.) It is important to note that, as mentioned in Chapters 4 and 5 (as the gap between processor cycle time and main memory access time has been widening), today two or three levels of cache are used for each processor, with a small and fast L1 cache, and larger and slower L2 and L3 caches. While L1/L2/L3 caches use SRAM (static random access memory), the main memory is built from DRAM (dynamic random access memory). Typical size and access time are 64 KB and 1–2 ns for the L1 cache, 256 KB and 3–12 ns for the L2 cache, 1–4 MB and 12–20 ns for the L3 cache, 2–10 GB and 50–100 ns for the main

memory, and 2–16 TB and 10–100 ms for the hard disk.

Cache coherence ensures that each processor of a multiprocessor system has the same consistent view of the memory through its local cache. At each point in time, each of the processors gets the same value for each variable if it performs a read access. Cache coherence, however, does not specify in which order, write accesses become visible to the other processors. Recall that we addressed this issue by memory consistency models in Chapter 4. These models provide a formal specification of how the memory system will appear to a programmer by placing restrictions on the values that can be returned by a memory read.

If several processors continuously update different variables that reside in the same cache line (or cache block), the cache coherency mechanism invalidates the whole line across the bus or interconnect with every data write. This is called *false sharing* because the system is behaving as if the variables which are totally unrelated are being shared by the processors. It may be noted that false sharing does not cause incorrect results but significantly degrades the performance of a program due to more accesses to memory than necessary. Recall that we provided some solutions to this problem in the previous chapter.

10.6 INPUT/OUTPUT (DISK ARRAYS)

Advances in semiconductor technology have resulted in faster processors and larger main memories which in turn require larger, faster secondary storage systems (for example, the disk drive hardware). Recall that, with a disk, the actual time to read (write) a sector/block is determined by (i) the seek time (the time required for the movement of the read/write head), (ii) the latency time or rotational delay (the time during which the transfer cannot proceed until the right sector/block rotates under the read/write head) and (iii) the actual data transfer time needed for the data to copy from disk into main memory). Unfortunately, improvements in the processor and main memory speed of parallel computers have outpaced improvements in their input/output (I/O) performance. Thus, for all but the most computation intensive applications, overall system performance (throughput) is typically limited by the speed of the underlying architecture's I/O system. Disk arrays or Redundant Arrays of Inexpensive Disks (RAID), which organize multiple, independent disks into a large, high-performance logical disk, are a natural solution to increase I/O speed (bandwidth). (Since the disks used in such arrays are often state-of-the-art and seldom inexpensive, sometimes RAID is said to stand for "Redundant Arrays of Independent Disks").

Disk arrays stripe (distribute) data across multiple disks and access them in parallel to achieve both higher data transfer rates on large data accesses (*data transfer parallelism*) and higher I/O rates on small data accesses (*access concurrency*). Data striping also provides improved levels of load balancing across all the disks, eliminating hot spots that otherwise saturate a small number of disks while the majority of disks remain idle.

Large disk arrays, unfortunately, are highly vulnerable to disk failures. A disk array with 50 disks is 50 times more likely to fail than a single-disk array as the mean-time-to-failure (MTTF) of a group of disks is inversely proportional to the number of disks, assuming that the MTTF of each disk is independent and exponentially distributed. This means an MTTF of 1,00,000 hours, or approximately 11 years, for a single disk implies an MTTF of only 2,000 hours, or approximately 83 days, for a disk array with 50 disks. The obvious solution is to employ redundancy to tolerate disk failures. But redundancy has negative consequences. Since all write operations must update the redundant information, the performance of writes in redundant disk arrays can be significantly worse than the performance of writes in non-redundant disk arrays.

Thus redundant disk arrays employ two orthogonal concepts: data striping for improved performance and redundancy for improved reliability. A number of different data-striping and redundancy schemes have been developed. The combinations and arrangements of these lead to many set of options, for users and designers of disk arrays, each having tradeoffs among reliability, performance, and cost. In this section, we first describe various data-striping and redundancy schemes employed in disk arrays, and then present different organizations of disk arrays or RAID.

10.6.1 Data Striping

Disk (or Data) striping (also called disk interleaving) folds multiple disk address spaces into a single, unified space seen by the host. This is achieved by distributing consecutive logical data units (called *stripe units*) among the disks in a round-robin fashion (much like interleaving in a multi-bank memory systems). There are two types of disk striping: fine-grained striping and coarse-grained striping.

Fine-grained Striping

Fine-grained striping (Fig. 10.20 shows independent disks and Fig. 10.21 shows fine-grained striping) distributes the data so that all of the array's N disks, which contain a fraction of each accessible block, cooperate in servicing every request. The number of disks and the stripe unit are generally chosen such that their product evenly divides the smallest accessible data unit (from the host's point of view). Stripe unit sizes for fine-grained striping include one bit, one byte and one disk sector. The load is perfectly balanced since all disks receive equal work load. The effective transfer rate is nearly N times that of an individual disk as each disk transfers $(1/N)^{\text{th}}$ of the requested data. However, only one request can be serviced at a time because all N disks work on each request.

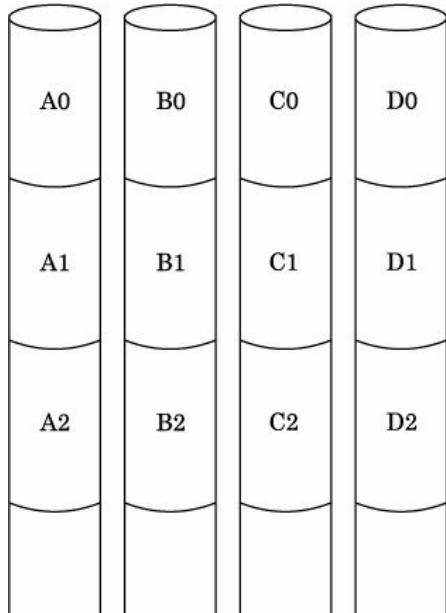


Figure 10.20 Independent disks. Each column represents physically sequential blocks on a single disk.
A0 and B0 represent block of data in separate disk address spaces.

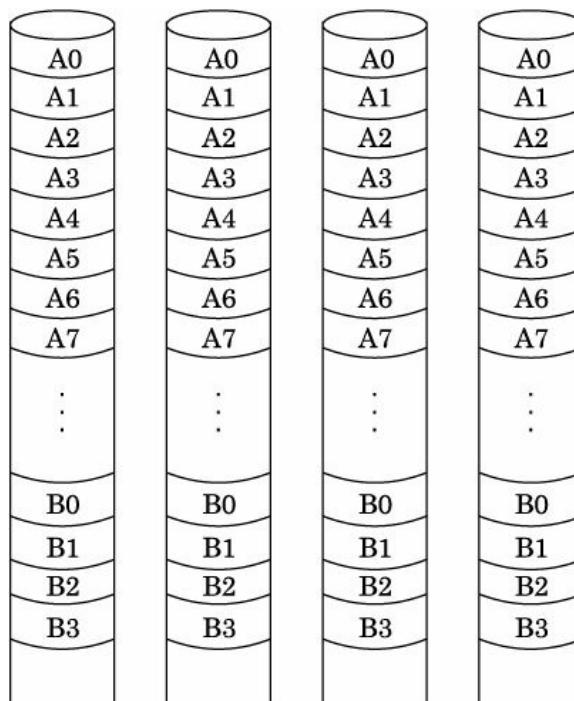


Figure 10.21 Fine-grained striping.

Coarse-grained Striping

Coarse-grained striping (Fig. 10.22) allows the disks to cooperate on large requests and service small requests concurrently. Though coarse-grained striping does not provide perfect load balancing like fine-grained striping, the conflicting effects of biased reference patterns and rapidly changing access distributions lead to statistical (automatic) load balancing. In order to achieve the performance potential of coarse-grained striping the stripe unit size should be correctly chosen. This choice largely decides the number of disks over which each request's data is spread. Large stripe units allow separate disks to handle multiple requests concurrently, reducing overall seek and rotational delays. This often results in reduced queueing delays and higher throughput. On the other hand, small stripe units cause multiple disks to access data in parallel, reducing data transfer times for individual requests. This tradeoff between access concurrency and transfer parallelism is governed by the stripe unit size.

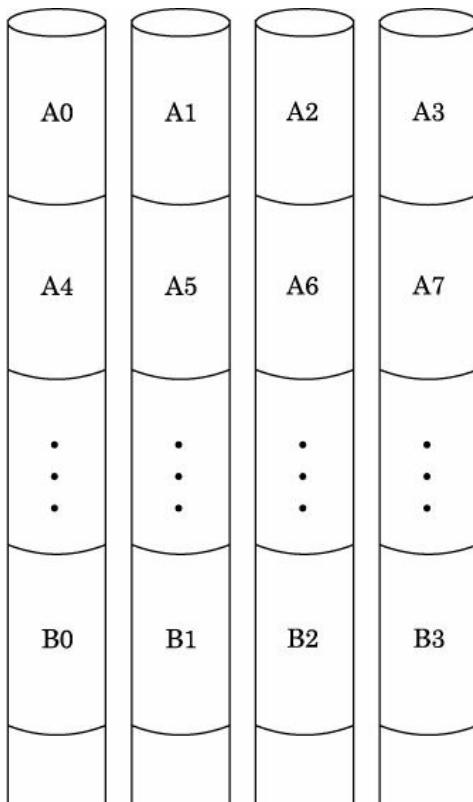


Figure 10.22 Coarse-grained striping.

10.6.2 Redundancy Mechanisms

We have earlier seen that as the number of disks continues to grow, the MTTF of a typical non redundant system shrinks to months or weeks. Further, the number of files affected per failure increases dramatically when disk striping is used. In an array of N disks, a striped system could lose $1/N$ of every file. A system incorporating redundancy in data can survive the failure of one or more components. However, incorporation of redundancy in disk arrays brings up two orthogonal problems. The first problem is to select the method of computing the redundant information. The second problem is that of selecting a method of distributing the redundant information across the disk array.

Data Replication

The simplest way to protect data is to keep separate copies of each data block on D (two or

more) separate disks. Data becomes unavailable only if all disks containing a copy fail. This form of redundancy is called disk mirroring or disk shadowing or disk duplexing (Fig. 10.23). The cost of data replication is proportional to $D-1$. Read performance benefits from disk mirroring. D requests can be serviced simultaneously thereby increasing throughput. In addition, a given read request can be scheduled to the disk on which it will experience the smallest access delay. Write performance, on the other hand, is severely affected as writes must be performed on all copies.

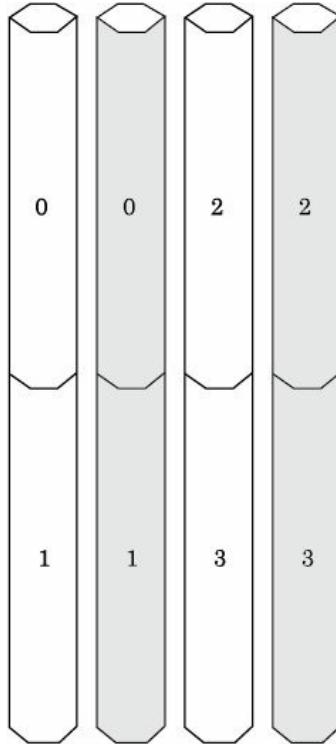


Figure 10.23 Mirroring (Shared regions represent secondary copies.)

In disk mirroring, every set of D identical disks can survive $D-1$ failures without data loss. However, each disk failure reduces the performance of a set relative to other sets. This imbalance can reduce the array's overall performance if a damaged set becomes a bottleneck. Alternately, sets may overlap such that this performance degradation is shared equally by all surviving disks. This method of combining multiple replicated sets is known as declustering which is of two types: chained declustering and interleaved declustering.

Chained declustering (Fig. 10.24) divides each disk into D sections. The first section contains the primary copy of some data, with each disk having different primary data. With clever scheduling, it is possible for chained clustering to maintain a balanced work load after a disk failure, though the probability of surviving multiple failures is reduced.

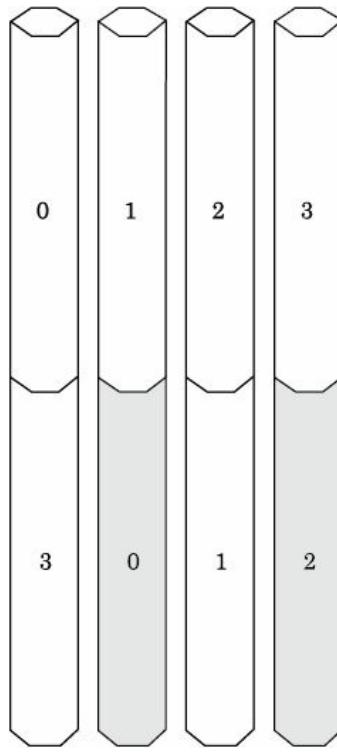


Figure 10.24 Chained declustering.

Interleaved declustering (Fig. 10.25) also partitions each disk but stripes the non-primary copies across all the disks instead of keeping them in contiguous chunks. This balances the additional load caused by a disk failure but further reduces reliability. Note that losing any D disks results in critical failure.

Parity-based Protection

Parity-based protection schemes achieve the required redundancy by using error detecting and error correcting codes. The scope of these schemes varies with the protection mechanism. For example, the erasure of a single bit among a sequence of bits can be recovered by maintaining a single odd or even parity bit. More complicated codes such as Hamming codes and Reed Solomon codes can detect and recover loss of multiple bits.

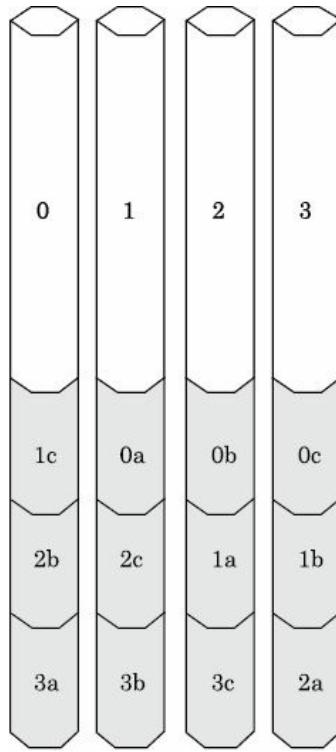


Figure 10.25 Interleaved declustering, 0a, 0b and 0c represent the three component pieces of the secondary copy of 0.

Readers unfamiliar with parity can think of a parity (redundant) disk as having the sum of all the data in the other disks. When a disk fails, we can subtract all the data on the good disks from the parity disk; the remaining information must be the missing information. Parity is simply this sum modulo two.

Although the cost or amount of redundancy information via parity is reduced (only one parity disk), compared to that of data replication, parity maintenance can reduce array performance considerably. Read requests are performed with the same efficiency as in non redundant arrays. Write requests require additional disk accesses to update the parity, often leading to very poor performance. If all data bits associated with a parity bit are updated by a write request, the new parity bit is computed and included as part of the total request. However, if only a subset of the data bits is being written, updating the parity bit requires additional work of reading bits from the disks to construct the new parity bit. There are two approaches to handle this situation. They are Read-Modify-Write (RMW) and Regenerate-Write (RW). With RMW, the new parity bit is constructed from the old value of the parity bit and the old and new values of the data bits being written. Thus RMW requires initial read accesses to all disks being updated (including the disk containing the parity). The other approach, RW, generates new parity from the values of all data bits (that is, the new values of the data bits being updated and the current values of the unchanged data bits). Thus RW requires initial read accesses to all disks not being updated. The performance penalty for maintaining parity can be reduced by dynamically selecting between these two approaches.

A solution to reduce the performance degradation due to parity maintenance is to spread the parity information among the disks in the system, instead of placing all parity on a dedicated parity disk (Fig. 10.26). There are two methods to distribute parity across disks. They are striped parity (Fig. 10.27) and declustered parity (Fig. 10.28). By using striped parity, where the parity is striped among disks, the array can perform multiple parity updates in parallel. In declustered parity, the parity is distributed among disks in such a way that the

performance of losing a disk can be shared equally by all disks.

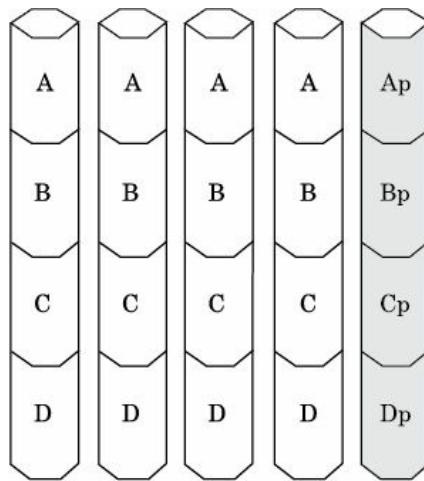


Figure 10.26 Parity-based protection.

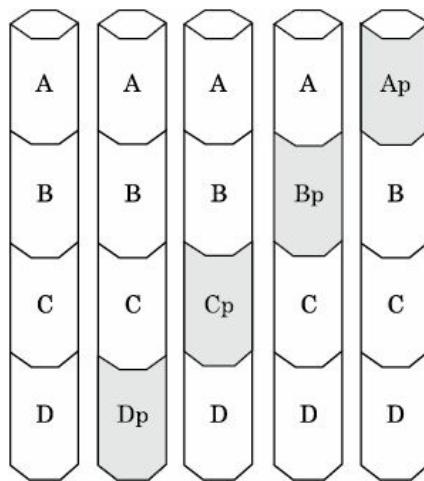


Figure 10.27 Striped parity.

10.6.3 RAID Organizations

We now briefly present the basic RAID organizations. A RAID “level” simply corresponds to a specific data-striping method and redundancy mechanism. It should be noted that the ordering of the levels carries no consistent meaning and does not represent a measure of goodness.

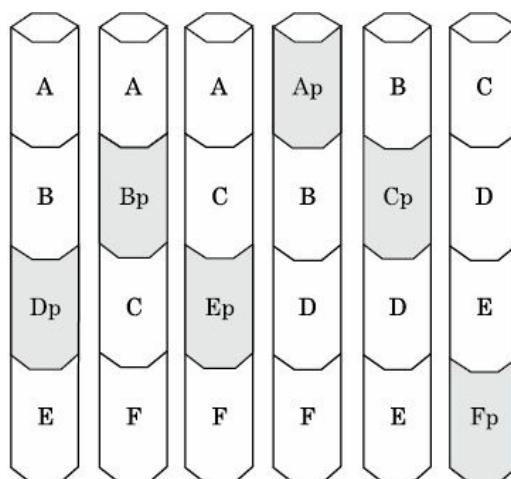


Figure 10.28 Declustered parity.

- (i) *Non-redundant (RAID Level 0)*: RAID level 0, also called *non-redundant disk array*, in which sequential blocks of data are written across multiple disks in stripes, does not employ redundancy at all and hence has the least cost of any RAID organization. Further, it offers the best write performance since it does not have to update any redundant information. Without redundancy, any single disk failure will result in data loss. Non-redundant arrays are used in environments where performance and capacity, rather than reliability, are the primary concerns.
- (ii) *Mirrored (RAID Level 1)*: In this organization, known as *mirroring*, data is written to two disks concurrently thereby using twice as many disks as a non-redundant disk array. It can perform better on reads by selectively scheduling requests on the disk with the shortest expected seek and rotational delays. If a disk fails, the other copy is used to service requests. Mirroring is frequently used in database applications where availability and transaction rate are more important than storage efficiency.
- (iii) *Memory-style (RAID Level 2)*: This organization uses bit-level striping and Hamming code for error correction. The bits of the code computed across corresponding bits on each data disk are stored in the corresponding bit positions on multiple parity disks. Here the number of redundant disks is proportional to the logarithm of the total number of disks in the system. This organization requires spindles of all the drives to be synchronized so that each disk head is in the same position on each disk at any given time. Hence, it cannot service multiple requests simultaneously. Compared to mirroring, the storage efficiency of this organization increases with increasing number of disks.
- (iv) *Bit-interleaved Parity (RAID Level 3)*: In this organization, instead of an error-correcting code, a simple parity is computed for the set of individual bits in the same position on all the data disks. Here data is interleaved bit-wise over the data disks, and a single parity disk is added to tolerate any single disk failure. Thus each read request accesses all data disks, and each write request accesses all data disks and the parity disk. Hence, only one request can be serviced at a time. Bit-interleaved parity disk arrays are used in applications that require high bandwidth (data transfer parallelism) but not high I/O rates (access concurrency).
- (v) *Block-interleaved Parity (RAID Level 4)*: This organization is similar to the bit-interleaved parity disk array except that data is interleaved across disks in blocks (whose size is called the striping unit) rather than in bits. It requires the use of either RMW or RW method to update the parity bit information. Since it has only one parity disk, which must be updated on all writes, the parity disk can easily become a bottleneck.
- (vi) *Block-interleaved Distributed Parity (RAID Level 5)*: This organization eliminates the above parity disk bottleneck by distributing the parity uniformly over all the disks. Block-interleaved distributed-parity disk arrays have the best small read, large read and large write performance of any redundant disk array. Small writes are somewhat inefficient, compared with redundant schemes such as mirroring, due to parity maintenance or updates. RAID levels 4 and 5 do not require synchronized drives.
- (vii) *P + Q Redundancy (RAID Level 6)*: Parity is a redundant code capable of correcting only one failure. We need stronger codes when multiple failures are possible (due to larger disk arrays), especially when applications with more stringent reliability requirements are considered. A scheme called P + Q redundancy uses Reed-Solomon codes which is capable of protecting against upto two disk failures using only two

redundant disks. The P + Q redundant disk arrays are structurally very similar to the block interleaved distributed-parity disk arrays and operate in much the same manner. In particular, they also perform small writes using RMW method.

(viii) *Striped Mirrors (RAID Level 10)*: This is a combination of RAID level 1 (mirroring) for redundancy and RAID level 0 (striping) for maximum performance. It employs mirroring without parity and block-level striping.

Note that comparing different levels of RAID, using performance/cost metrics, can be confusing because a RAID level does not specify a particular *implementation* but just mentions its *configuration* and *use*. For example, a RAID level 5 disk array (block-interleaved distributed parity) with a parity group size of two is comparable to RAID level 1 (mirroring) with the exception that in a mirrored disk array, certain disk scheduling and data optimizations can be performed that, generally, are not implemented for RAID level 5 disk arrays. Similarly, a RAID level 5 disk array can be configured to operate equivalently to a RAID level 3 disk array by choosing a unit of data striping such that the smallest unit of array access always accesses a full parity stripe of data. Thus it is not appropriate to compare the various RAID levels as their performance depends on parameters such as stripe unit size and parity group size. In reality, a specific implementation of a RAID level 3 system can have better performance-cost ratio than a specific implementation of a RAID level 5 system.

10.7 CONCLUSIONS

The complexity of parallel hardware and the performance requirements of the user programs introduce new challenges to the design and implementation of parallel operating systems. In this chapter we discussed in detail the following key issues in parallel operating systems: (i) resource (processor management, (ii) process/thread management, (iii) synchronization mechanisms, (iv) inter-process communication, (v) memory management and (vi) input/output. Another important issue, relating to parallel operating systems, which we did not consider is reliability and fault tolerance. A parallel operating system must promptly detect a component (processor or network) failure and take measures to isolate and contain it. After detecting the failure of a system component, the operating system must be able to recover the processes affected by the failure (i.e., restore the states of these processes to consistent states) so that these processes can resume processing. It is important that both fault detection and fault recovery mechanisms should have low overhead. For this both parallel hardware and operating system should work together.

EXERCISES

10.1 Using the compile-time scheduling algorithm described in this chapter (Section 10.1), obtain the schedule for the task graph of Fig. 10.29 on a 2×2 mesh-connected processors. Also find the complexity of this algorithm.

10.2 Obtain an optimal schedule for the task graph of Fig. 10.29 on a two-processor system and on a 2×2 mesh-connected processors.

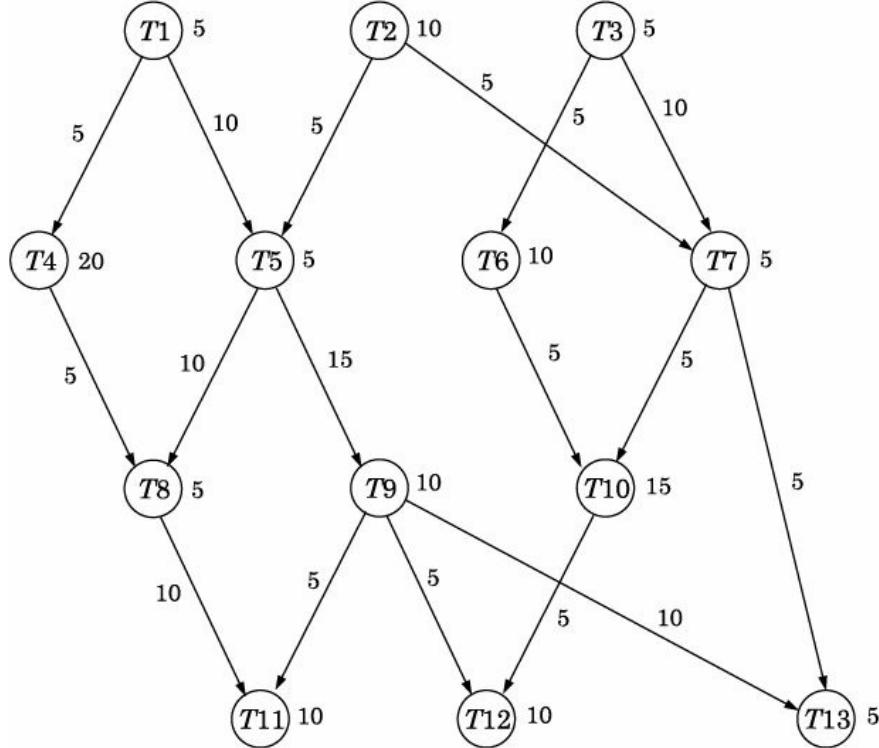


Figure 10.29 Task graph for Exercises 10.1 and 10.2.

10.3 For the task graph shown in Fig. 10.30 obtain the schedule on a two-processor system without clustering and with clustering (try to fuse tasks based on their E/C ratio). Also compute an optimal schedule.

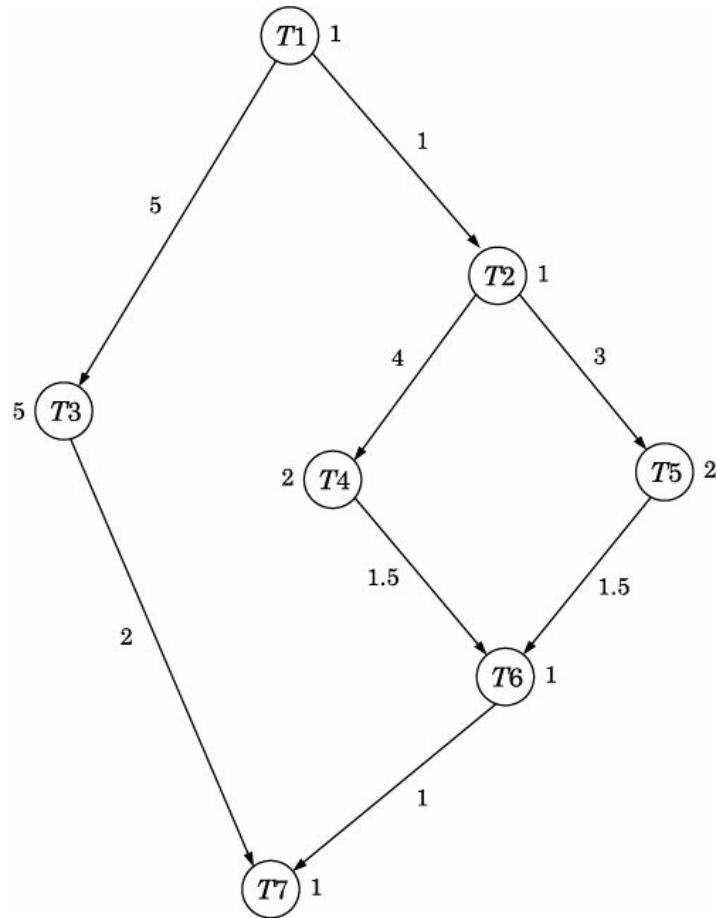


Figure 10.30 Task graph for Exercise 10.3.

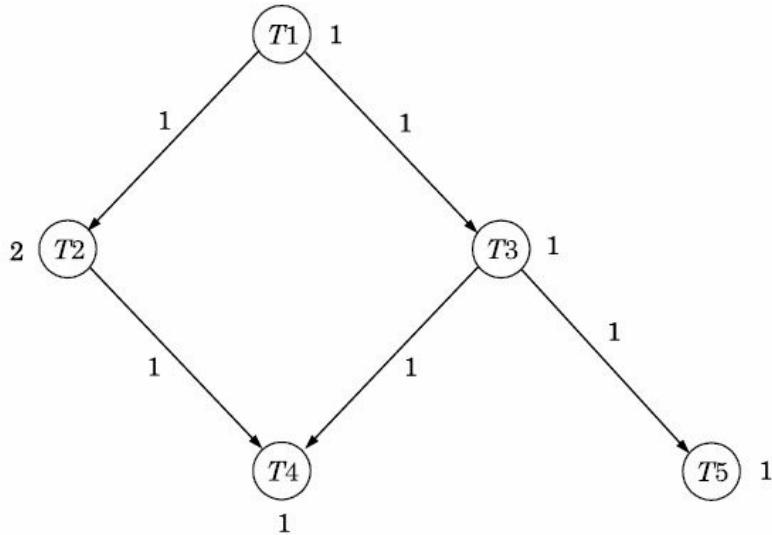
10.4 Task duplication is another method to eliminate the communication cost among the tasks by duplicating the task among the processors. Figure 10.31 shows an example of the task duplication method. In this example, task T1 is duplicated in both the processors. This effectively eliminates the communication cost between T1 and T3. Does task duplication preserve the original program parallelism? Mention two disadvantages of the task duplication method.

10.5 What is the main drawback of a centralized dynamic scheduling algorithm?

10.6 What is the difference between load sharing and load balancing?

10.7 What are the characteristics that a good load index should have?

10.8 A sender-initiated algorithm is preferred at light to moderate system load and a receiver initiated algorithm is preferred at high system load. Why? (Hint: Number of probe messages generated (low/high).)



(a) Task graph

P2		T3	T5			
P1	T1	T2		T4		
	0	1	2	3	4	5

(b) Schedule without task duplication

P2	T1	T3	T5		
P1	T1		T2	T4	
	0	1	2	3	4

(c) Schedule with task duplication

Figure 10.31 Example of task duplication.

10.9 A dynamic scheduling algorithm is said to be *unstable* if it can enter a state in which all the processors in the system are spending all their time transferring tasks without performing any useful work in an attempt to properly schedule the tasks for better performance. This form of fruitless transfer of tasks is called *processor thrashing*. In what situations a processor thrashing can occur? Suggest a solution to overcome this problem.

10.10 The task graph shown in Fig. 10.32 is to be scheduled on a dual-core processor. The 3 tuples on a vertex describes its execution time (in seconds), voltage assignment (in volts) and energy consumption (in joules). Assume inter-core communication is negligible.

- (a) Obtain a schedule with minimum completion time.
- (b) Obtain a schedule with minimum energy consumption.
- (c) Adjust the schedule produced in Exercise (a) such that minimum performance (completion time) is experienced with an energy budget of 50 J.
- (d) Given an initial schedule of a parallel program represented by a task graph, an energy constraint and a multicore processor. Devise an algorithm that utilizes DVS, which effectively determines tasks which are to be penalized (slowed down) such that minimum performance degradation is experienced while satisfying the energy constraint.

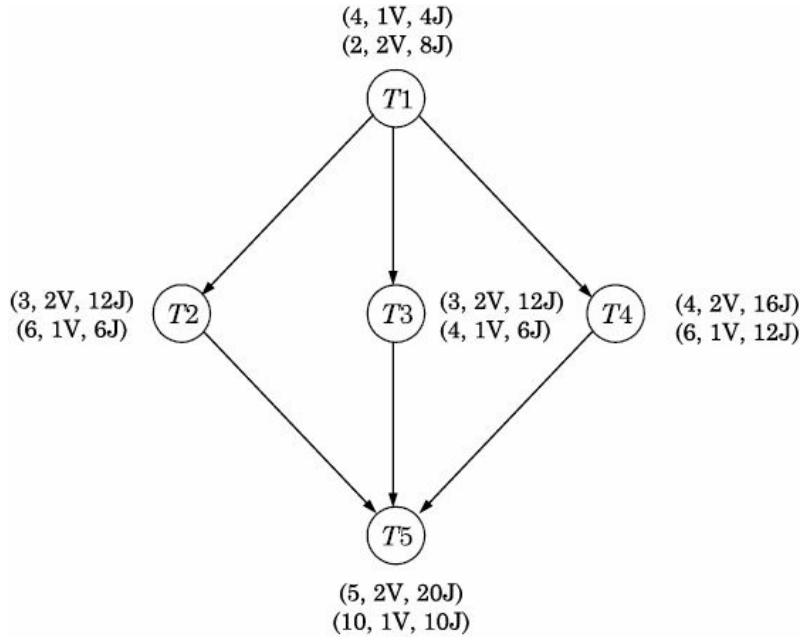


Figure 10.32 Task graph for Exercise 10.10.

- 10.11 Consider a core that runs at a frequency of one GHz and completes one thread in five clock cycles. The critical/threshold limit for the core is 75°C. It takes six threads to raise the temperature 1°C. Find the number of threads it needs to execute concurrently and the time it takes to reach thermal emergency. Assume that the initial temperature of the core is 0°C.
- 10.12 Is multithreading always a useful concept? Give a concrete example of an application that would benefit from the use of threads and another application that would not benefit from the use of threads.
- 10.13 Can you think of a drawback of affinity scheduling? (Hint: Load imbalance!)
- 10.14 Consider the two processes executing as shown below. Each process attempts to perform a lock operation on two variables.

Process 1	Process 2
lock(P)	lock(Q)
:	:
lock(Q)	lock(P)

Will these processes terminate? Why?

- 10.15 Consider the lock operation, using test-and-set, instruction. Instead of busy-waiting, a process inserts a delay after an unsuccessful attempt to acquire the lock. The delay between test-and-set attempts should be neither too long nor too short. Why? (Experimental results have shown that good performance is obtained by having the delay vary “exponentially”, i.e., the delay after the first attempt is a small constant k that increases geometrically, so that after i th attempt, it is $k \times c^i$, where c is another constant. Such a lock is called test-and-set lock with exponential backoff.)
- 10.16 Consider the swap instruction, defined below, which atomically exchanges the contents of two (x and y) variables.

```

procedure swap (var x, y: boolean);
var temp: boolean;
begin
    temp : = x;
    x := y;
    y := temp;
end;

```

Implement lock/unlock operations using the swap instruction.

- 10.17 The fetch-and-add instruction of the NYU (New York University) Ultra-computer is a multiple operation memory access instruction that atomically adds a constant to a memory location and returns the previous contents of the memory location. This instruction is defined as follows (m is a memory location and c is the constant to be added).

```

function fetch-and-add( $m$ : integer;  $c$ : integer);
var temp: integer;
begin
    temp :=  $m$ ;
     $m$  :=  $m + c$ ;
    return temp
end;

```

An interesting property of this instruction is that it is executed by the hardware placed in the interconnection network (not by the hardware present in the memory modules). When several processors concurrently execute a fetch-and-add instruction on the same memory location, these instructions are combined in the network and are executed by the network in the following way. A single increment, which is the sum of the increments of all these instructions, is added to the memory location. A single value is returned by the network to each of the processors, which is an arbitrary serialization of the execution of the individual instructions. If a number of processors simultaneously perform fetch-and-add instructions on the same memory location, the net result is as if these instructions were executed serially in some unpredictable order. Implement lock/unlock operations using the fetch-and-add instruction.

- 10.18 Develop a barrier algorithm using a shared counter. The shared counter maintains the number of processes that have arrived at the barrier and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, a process checks to see if the counter equals p i.e., if it is the last process to have arrived. If not, it busy waits on a flag associated with the barrier; if so, it sets the flag to release the $(p - 1)$ waiting processes.

- 10.19 In the standard barrier, all threads must synchronize at the barrier point i.e., if one thread has not arrived at the barrier point, all other threads must wait and cannot execute any useful operations. A fuzzy barrier (see Fig. 10.33) can overcome this problem of barrier waiting overhead; when a thread reaches the fuzzy barrier, it proceeds to carry out some useful computation without having to just wait for other threads. However, no thread can exit a barrier zone until all threads have entered the zone. Give a concrete example for the use of fuzzy barriers. (Hint: You need to insert some useful computation code in the barrier zone.)

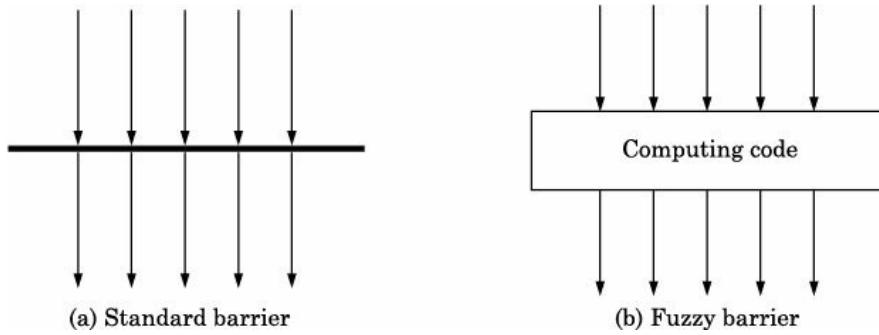


Figure 10.33 Standard and Fuzzy barriers.

10.20 Consider the following variations of the transfer() function considered earlier in this chapter.

- (a) There is a single lock S for all the accounts. Can any other employee access any account while transfer() is in progress?

```
transfer(Account A, Account B, Amt)
begin
    lock(S)
    A.balance = A.balance - Amt
    B.balance = B.balance + Amt
    unlock(S)
end
```

- (b) There is a separate lock for each account. Assume that there is no sufficient fund in Account A which means the function sufficient_funds_in_Account(A) returns false. Will account A gets locked forever?

```
transfer(Account A, Account B, Amt)
begin
    lock(S[A])
    if sufficient_funds_in_Account(A)
        begin
            A.balance = A.balance - Amt
            unlock(S[A])
            lock(S[B])
            B.balance = B.balance + Amt
            unlock(S[B])
        end
    end
end
```

- (c) Does the code limit concurrency? If so, rewrite the code in a way that it promotes concurrency.

```

transfer(Account A, Account B, Amt)
begin
    lock(S[A])
    if sufficient_funds_in_Account(A)
        begin
            A.balance = A.balance - Amt
            lock(S[B])
            B.balance = B.balance + Amt
            unlock(S[B])
        end
    unlock(S[A])
end

```

- 10.21 Consider the following three transactions running concurrently:

int a = 0, b = 0, c = 0;		
Transaction Begin	Transaction Begin	Transaction Begin
a = a+1;	b = b+1;	c = c+1;
Transaction End	Transaction End	Transaction End
a=a+1;	b=b+1;	a=a+1;
		Transaction End

What are the final values of a, b and c?

- 10.22 Give an example of transactional program that deadlocks. (Hint: consider footprints of the following transactions shown below.)

Boolean flag A = false;	Boolean flag B = false;
Thread 1:	Thread 2:
Transaction Begin	Transaction Begin
while (not(flag A));	flag A = true ;
flag B = true ;	while (not(flag B));
Transaction End	Transaction End

- 10.23 Give an example to show that blindly replacing locks/unlocks with Transaction Begin/Transaction End may result in unexpected behavior.

- 10.24 Consider the following two concurrent threads. What are the final values of x and y?

x = 5; x_shared = true	
Thread 1:	Thread 2:
Transaction Begin	if (x_shared) then x = x+1;
x_shared = false;	
Transaction End	
x = 10	

- 10.25 It appears that input/output operations cannot be supported with transactions. (For example, there is no way to revert displaying some text on the screen for the user to see.) Suggest a way to do this. (Hint: buffering)

- 10.26 When transactions conflict, contention resolution chooses which will continue and which will wait or abort. Can you think of some policies for contention resolution?

- 10.27 Prepare a table which shows the advantages and disadvantages of HTM and STM systems.

- 10.28 A technique related to transactional memory system is *lock elision*. The idea here is to design hardware to handle locks more efficiently, instead of enabling transactions. If two threads are predicted to only read locked data, the lock is removed (elided) and the two threads can be executed in parallel. Similar to transactional memory system, if a thread modifies data, the processor must rollback and re-execute the thread using locks for correctness. What are the advantages and disadvantages of lock elision?
- 10.29 What governs the tradeoff between I/O request rate (i.e., access concurrency) and data transfer rate (i.e., transfer parallelism) in a RAID?
- 10.30 Which RAID (level 3 or level 5) organization is more appropriate for applications that require high I/O request rate?
- 10.31 Which RAID (level 3 or level 5) organization is more appropriate for applications such as CAD and imaging that require large I/O request size?
- 10.32 What is the minimum number of disks required for RAID levels 5 and 10?
- 10.33 What are some inherent problems with all RAIDs? (Hints: (i) correlated failures and (ii) data integrity)
- 10.34 Another approach, apart from RAID, for improving I/O performance is to use the main memory of idle machines on a network as a backing store for the machines that are active thereby reducing disk I/O. Under what conditions this approach is effective? (Hint: Network latency.)
- 10.35 The most common technique used to reduce disk accesses is the disk cache. The cache, which is a buffer in the main memory, contains a copy of some the sectors (blocks) on the disk. What are the issues that need to be addressed in the design of disk caches?

BIBLIOGRAPHY

- Adl-Tabatabai, A., Kozyrakis, C. and Saha, B., “Unlocking Concurrency”, *ACM QUEUE*, Vol. 4, No. 10, Dec.–Jan. 2006–2007, pp. 24–33.
- Akhter, S. and Roberts, J., *Multi-Core Programming*, Intel Press, Hillsboro, 2006.
- Antony Louis Piriyakumar, D. and Siva Ram Murthy, C., “Optimal Compile-time Multi-processor Scheduling Based on the 0–1 Linear Programming Algorithm with the Branch and Bound Technique”, *Journal of Parallel and Distributed Computing*, Vol. 35, No. 2, June 1996, pp. 199–204.
- Casavant, T.L., Tvrlik, P. and Plasil, F. (Eds.), *Parallel Computers: Theory and Practice*, IEEE Computer Society Press, 1996.
- Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S. and Chatterjee, S., “Software Transactional Memory: Why Is It Only a Research Toy?” *ACM QUEUE*, Vol. 6, No. 5, Sept. 2008, pp. 46–58.
- Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H. and Patterson, D.A., “RAID: High Performance, Reliable Secondary Storage”, *ACM Computing Surveys*, Vol. 26, No. 2, June 1994, pp. 145–185.
- Chisnall, D., “Understanding Hardware Transactional Memory in Intel’s Haswell Architecture”, 2013. <http://www.quepublishing.com/articles/article.aspx?p=2142912>
- El-Rewini, H., Lewis, T.G. and Ali, H.H., *Task Scheduling in Parallel and Distributed Systems*, Prentice-Hall, NJ, USA, 1994.
- Ganger, G.R., Worthington, B.L., Hou, R.Y. and Patt, Y.N., “Disk Arrays: High-Performance, High-Reliability Storage Subsystems”, *IEEE Computer*, Vol. 27, No. 3, Mar. 1994, pp. 30–36.
- Grahn, H., “Transactional Memory”, *Journal of Parallel and Distributed Computing*, Vol. 70, No. 10, Oct. 2010, pp. 993–1008.
- Haris, T., Larus, J. and Rajwar, R., *Transactional Memory*, Morgan & Claypool Publishers, CA, USA, 2010.
- Hwang, K. and Xu, Z., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, WCB/McGraw-Hill, USA, 1998.
- Kong, J., Chung, S.W. and Skadron, K., “Recent Thermal Management Techniques for Microprocessors”, *ACM Computing Surveys*, Vol. 44, No. 3, June 2012.
- Krishnan, C.S.R., Antony Louis Piriyakumar, D. and Siva Ram Murthy, C., “A Note on Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing”, *IEEE Trans. on Signal Processing*, Vol. 43, No. 3, Mar. 1995, pp. 802–805.
- Loh, P.K.K., Hsu, W.J., Wentong, C. and Sriskanthan, N., “How Network Topology Affects Dynamic Load Balancing”, *IEEE Parallel & Distributed Technology*, Vol. 4, No. 3, Fall 1996, pp. 25–35.
- Manimaran, G. and Siva Ram Murthy, C., “An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems”, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 9, No. 3, Mar. 1998, pp. 312–319.
- Rauber, T. and Runger, G., *Parallel Programming for Multicore and Cluster Systems*, Springer-Verlag, Germany, 2010.
- Rosario, J.M. and Choudhary, A.N., “High-Performance I/O for Massively Parallel Computers—Problems and Prospects”, *IEEE Computer*, Vol. 27, No. 3, Mar. 1994, pp.

59–68.

- Selvakumar, S. and Siva Ram Murthy, C., “Scheduling of Precedence Constrained Task Graphs with Non-negligible Inter-task Communication onto Multiprocessors”, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 3, Mar. 1994, pp. 328–336.
- Shiraji, B.A., Hurson, A.R. and Kavi, K.M. (Eds.), *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, 1995.
- Singh, A.K., Shafique, M., Kumar, A. and Hankel, J., “Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends”, *Proceedings of the ACM Design Automation Conference*, 2013.
- Sinha, P.K., *Distributed Operating Systems*, IEEE Computer Society Press, USA, 1997.
- Siva Ram Murthy, C. and Rajaraman, V., “Task Assignment in a Multiprocessor System”, *Microprocessing and Microprogramming, the Euromicro Journal*, Vol. 26, No. 1, 1989, pp. 63–71.
- Sreenivas, A., Balasubramanya Murthy, K.N. and Siva Ram Murthy, C., “Reverse Scheduling—An Effective Method for Scheduling Tasks of Parallel Programs Employing Divide-and-Conquer Strategy onto Multiprocessors”, *Microprocessors and Microsystems Journal*, Vol. 18, No. 4, May 1994, pp. 187–192.
- Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R. and Michael, M., “Evaluation of Blue Gene/Q Hardware Support for Transactional Memory”, *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 127–136.
- Yoo, R.M., Hughes, C.J., Rajwar, R. and Lai, K., “Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing”, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.
- Zhuravlev, S., Saez, J.C., Blagodurov, S., Fedorova, A. and Prieto, M., “Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors”, *ACM Computing Surveys*, Vol. 45, No. 1, Nov. 2012.
- Zhuravlev, S., Saez, J.C., Blagodurov, S., Fedorova, A. and Prieto, M., “Survey of Energy-Cognizant Scheduling Techniques”, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 24, No. 7, July 2013, pp. 1447–1464.

Performance Evaluation of Parallel Computers

A sequential algorithm is usually evaluated in terms of its execution time which is expressed as a function of its input size. On the other hand, the execution time of a parallel algorithm depends not only on the input size but also on the parallel architecture and the number of processors employed. Hence, a parallel algorithm cannot be evaluated isolated from a parallel computer. A parallel computing system should be viewed as a combination of a parallel algorithm and the parallel computer on which it is implemented. In this chapter, we first introduce various metrics, standard measures, and benchmarks for evaluating the performance of a parallel computing system. Then we study the sources of parallel overhead followed by speedup performance laws and scalability principles. Finally, we briefly mention the need for performance analysis tools which are essential to optimize an application's performance.

11.1 BASICS OF PERFORMANCE EVALUATION

In this section, we introduce the metrics and measures for analyzing the performance of parallel computing systems.

11.1.1 Performance Metrics

1. Parallel Run Time

The parallel run time, denoted by $T(n)$, of a program (application) is the time required to run the program on an n -processor parallel computer. When $n = 1$, $T(1)$ denotes the (sequential) run time of the program on a single processor.

2. Speedup

Speedup, denoted by $S(n)$, is defined as the ratio of the time taken to run a program on a single processor to the time taken to run it on a parallel computer with identical processors.

$$S(n) = \frac{T(1)}{T(n)}$$

Note that speedup metric is a quantitative measure of performance gain that is achieved by multiple processor implementation of a given program over a single processor implementation. In other words, it captures the relative benefit of running a program (or solving a problem) in parallel.

For a given problem, there may be more than one sequential implementation (algorithm). When a single processor system is used, it is natural to employ a (sequential) algorithm that solves the problem in minimal time. Therefore, given a parallel algorithm it is appropriate to assess its performance with respect to the best sequential algorithm. Hence, in the above definition of speedup, $T(1)$ denotes the time taken to run a program using the fastest known sequential algorithm.

Now consider the problem of adding numbers using an n -processor hypercube. Initially, each processor is assigned one of the numbers to be added. At the end of the computation, one of the processors accumulates the sum of all the numbers. Assuming that n is a power of 2, we can solve this problem on an n -processor hypercube in $\log(n)$ steps. Figure 11.1 illustrates a solution, for $n = 8$, which has three ($= \log(8)$) steps. Each step, as shown in Fig. 11.1, consists of one addition and the communication of a single message between two processors which are directly connected.

Thus,

$$T(n) = O(\log n)$$

Since the problem can be solved in $O(n)$ time on a single processor, its speedup is

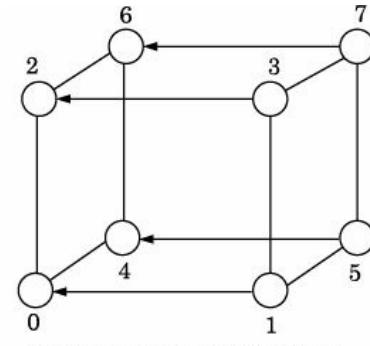
$$S(n) = O\left(\frac{n}{\log n}\right)$$

Superlinear Speedup

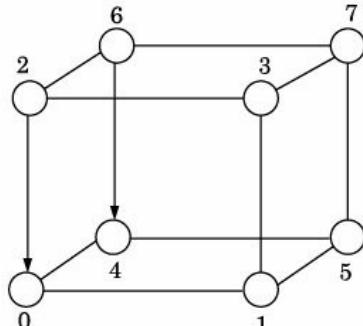
Theoretically, speedup can never exceed the number of processors, n , employed in solving a problem. A speedup greater than n is possible only when each processor spends less than $T(1)/n$ units of time for solving the problem. In such a case, a single processor can emulate the n -processor system and solve the problem in less than $T(1)$ units of time, which contradicts the hypothesis that the best sequential algorithm was used in computing the speedup. Occasionally, claims are made of speedup greater than n , *superlinear speedup*. This

is usually due to (i) non optimal sequential algorithm, (ii) hardware characteristics, and (iii) speculative computation which put a sequential algorithm at a disadvantage.

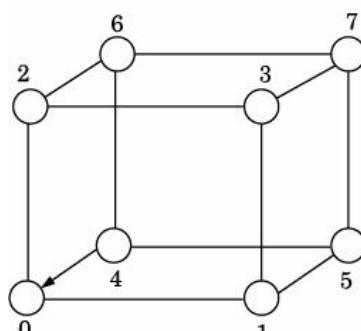
(a) *Superlinear Speedup Due to Non-optimal Sequential Algorithm*: Consider the problem of setting the n elements of an array to zero by both a uniprocessor and an n -processor parallel computer. The parallel algorithm simply gives one element to each processor and sets each element to zero. But the sequential algorithm incurs the overhead of loop construct (i.e., incrementing the loop index variable and checking its bounds) which is absent in the parallel algorithm. Thus the time taken by the parallel algorithm is less than $(1/n)$ th of the time taken by the sequential algorithm.



(a) First communication step



(b) Second communication step



(c) Third communication step

Figure 11.1 Addition of 8 numbers using 8 processors.

(b) *Superlinear Speedup Due to Hardware Characteristics*: Consider a program running on one processor which has a c -byte cache. When this program is decomposed and run on n -processors, each of which has its own c -byte cache, then each processor in the parallel computer may have a higher cache hit rate, and hence the decomposed program may run faster compared to its sequential counterpart.

(c) *Superlinear Speedup Due to Speculative Computation*: Consider the case of sequential and parallel DFS (Depth First Search) performed on a tree (search space) illustrated in

Fig. 11.2. The order of node expansions is indicated by node labels. The sequential DFS generates 13 nodes before reaching the indicated goal node (solution). On the other hand, the parallel DFS, with two processors, generates only 9 nodes. The nodes expanded by the processors are labeled *A* and *B*. In this case, the search overhead factor is 9/13 (less than 1), and if the inter-processor communication overhead is not too large, the speedup will be superlinear.

In what follows, we assume speedups to be sublinear.

3. Efficiency

While speedup measures how much faster a program runs on a parallel computer rather than on a single processor, it does not measure whether the processors in that parallel computer are being used effectively. Since it would not be worthwhile, for example, to use 100 processors to achieve a speedup of 2, efficiency is also commonly used to assess performance.

The efficiency of a program on n processors, $E(n)$, is defined as the ratio of the speedup achieved and the number of processors used to achieve it.

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

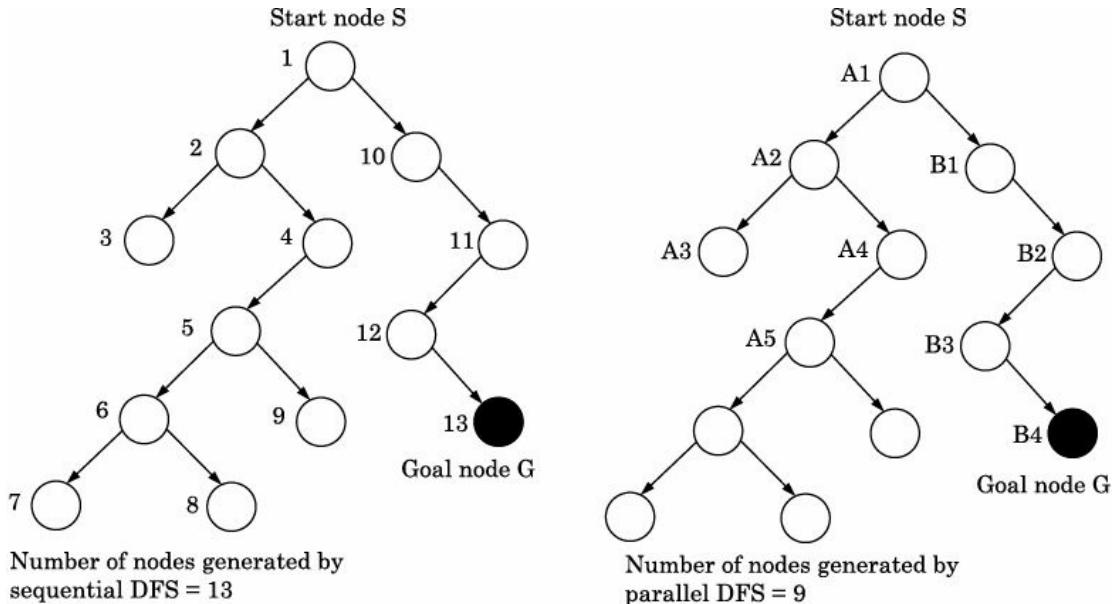


Figure 11.2 Parallel DFS generating fewer nodes than sequential DFS.

This gives the measure of the fraction of the time for which a processor is usefully employed. Since $1 \leq S(n) < n$, we have $1/n < E(n) \leq 1$. An efficiency of one is often not achieved due to parallel overhead. For the problem of adding numbers on an n -processor hypercube,

$$S(n) = O\left(\frac{n}{\log(n)}\right)$$

$$E(n) = O\left(\frac{1}{\log(n)}\right)$$

Now consider the same problem of adding n numbers on a hypercube where the number of processors, $p < n$. In this case, each processor first locally adds its n/p numbers in $O(n/p)$ time. Then the problem is reduced to adding the p partial sums on p -processors. Using the earlier method, this can be done in $O(\log(p))$ time. Thus the parallel run time of this algorithm is $O(n/p + \log(p))$. Hence the efficiency in this case is

$$E(p) = O\left(\frac{\frac{n}{p}}{\frac{n}{p} + \log(p)}\right) = O\left(\frac{1}{1 + \frac{p}{n} \log(p)}\right)$$

Note that, for a fixed p , $E(p)$ increases with increasing n as the computation at each processor increases. Similarly, when n is fixed, $E(p)$ decreases with increasing p as the computation at each processor decreases.

Figures 11.3 to 11.5 show the relation between execution time, speedup, efficiency and the number of processors used. In the ideal case, the speedup is expected to be *linear*, i.e., it grows linearly with the number of processors, but in most cases, it falls due to the parallel overhead.

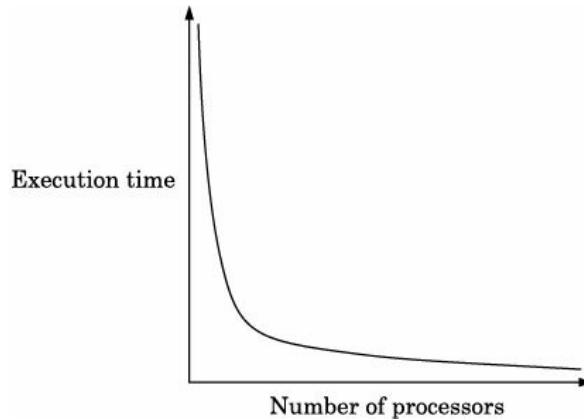


Figure 11.3 Execution time versus number of processors employed.

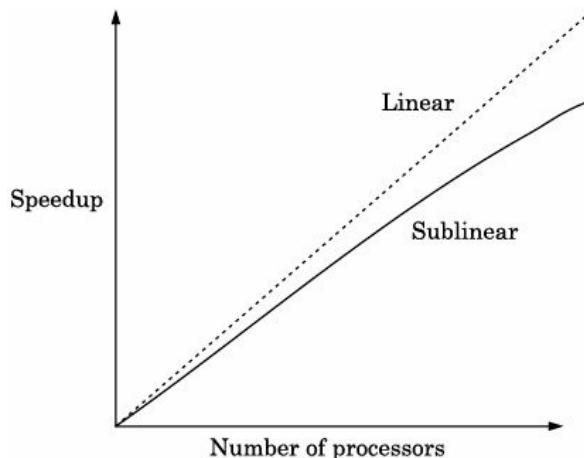


Figure 11.4 Speedup versus number of processors employed.

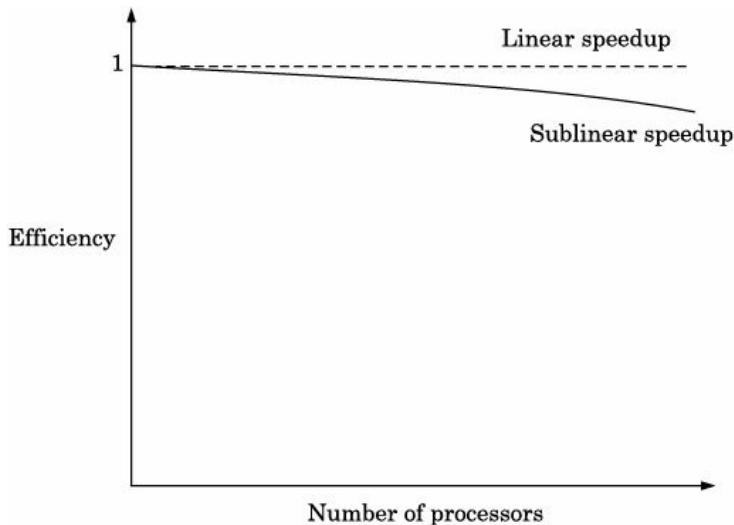


Figure 11.5 Efficiency versus number of processors employed.

11.1.2 Performance Measures and Benchmarks

The above metrics are for evaluating the performance of a parallel algorithm on a parallel computer. There are other standard measures adopted by the industry to compare the performance of various parallel computers.

Most computer manufacturers specify *peak* performance (theoretical maximum based on best possible utilization of all the resources) or *sustained* performance (based on running application-oriented benchmark) in terms of MIPS (Millions of Instructions Per Second) or Mflops (Millions of floating-point operations per second). MIPS and Mflops reflect the *instruction execution rate* and the *floating-point capability* of a computer, respectively.

It should be noted that the MIPS measurement is highly dependent on the instruction mix used to produce the execution times. A program with more simple integer operations runs faster than a program with more complex floating-point operations, even though both the programs have the same instruction count. Hence two different programs with the same number of instructions can produce different MIPS measurement on the same computer.

Like MIPS, Mflops measurement is also program or application dependent as it is highly unlikely that all the instructions in the program are floating-point operations. Further, not all floating-point operations take the same amount of time. For example, a floating-point division may be 4 to 20 times slower than a floating-point addition. The Mflops is a popular measure of machine performance on scientific programs, but it is not a reasonable measure for programs which have a few or no floating-point operations.

There are some benchmarks (a set of programs or program fragments) which are used to compare the performance of various machines. When determining which computer to use or purchase for a given application, it is important to know the performance of that computer for the given application. When it is not possible to test the applications on different machines, then one can use the results of benchmark programs that most resemble the applications run on those machines.

Small programs which are especially constructed for benchmarking purposes and do not represent any real computation (workload) are called *synthetic benchmarks*. Some well-known examples for synthetic benchmarks are Whetstone, which was originally formulated in FORTRAN to measure floating-point performance, and Dhrystone to measure integer performance in C. The major drawback of synthetic benchmarks is that they do not reflect the actual behaviour of real programs with their complex interactions between computations of

the processor and accesses to the memory system. Program fragments which are extracted from real programs are called *kernel benchmarks* as they are the heavily used core (or responsible for most of the execution time) of the programs. Examples for kernel benchmarks are Livermore Loops (consisting of FORTRAN kernels derived from a set of scientific simulations by Lawrence Livermore National Laboratory, USA) and LINPACK (consisting of a set of FORTRAN subroutines to solve a set of simultaneous linear equations, developed by Jack Dongarra of Argonne National Laboratory, USA). The major drawback of kernels is that the performance results they produce, given typically in Mflops, are generally very large for applications that come from non-scientific computing areas. *Application benchmarks* comprise several complete applications (programs) that reflect the workload of a standard user, which are often called *benchmark suites*. The implementation of such benchmark suites which capture all the aspects of the selected programs is very time and resource intensive. But the performance results produced are meaningful for users as the benchmark suite represents their typical workload. SPEC (System Performance Evaluation Cooperation group, founded in 1988, maintains sets of benchmark programs to rate the performance of new computers) is the most popular benchmark suite. The latest benchmark suite for desktop computers is SPEC2006.

The standardized methodology used is as follows:

- (i) Measure execution time of programs
- (ii) Normalize to a fixed reference computer
- (iii) Report geometric mean of ratios as SPECmark.

There are other benchmark suites such as SPECviewperf for graphics, SPECSFC for file servers, and SPEC MPI2007 and SPEC OMP2012 for parallel computing. Table 11.1 provides sources for some parallel benchmark (scientific/engineering) suites.

TABLE 11.1 Parallel Benchmark Websites

<i>Benchmark suite</i>	<i>Website for more information</i>
HPC Challenge	http://icl.cs.utk.edu/hpcc/
NAS	https://www.nas.nasa.gov/publications/npb.html
PARKBENCH	http://www.netlib.org/parkbench/
PARSEC	http://parsec.cs.princeton.edu/
Rodinia	https://www.cs.virginia.edu/~skadron/wiki/rodinia/
SPEC ACCEL, SPEC MPI2007, SPEC OMP2012	http://www.spec.org
SPLASH-2	http://www-flash.stanford.edu/apps/SPLASH http://www.capsl.udel.edu/splash/

The performance results of benchmarks also depend on the compilers used. It is quite possible for separate benchmarking runs on the same machine to produce disparate results, because different (optimizing) compilers are used to produce the benchmarking object code. Therefore, as part of recording benchmark results, it is important to identify the compiler and the level of optimization used in the compilation.

11.2 SOURCES OF PARALLEL OVERHEAD

Parallel computers in practice do not achieve linear speedup or an efficiency of one, as mentioned earlier, because of parallel overhead (the amount of time required to coordinate parallel tasks as opposed to doing useful work). The major sources of overhead in a parallel computer are inter-processor communication, load imbalance, inter-task synchronization, and extra computation. All these are problem or application dependent.

11.2.1 Inter-processor Communication

Any non-trivial parallel algorithm requires inter-processor communication. The time to transfer data between processors is usually the most significant source of parallel computing overhead. If each of the p processors spends t_{comm} units of time for data transfer, then the interprocessor communication contributes $t_{\text{comm}} \times p$ units of time (or $t_{\text{comm}} \times p \times \alpha_0$ units of time, where α_0 ($0 < \alpha_0 < 1$) is the computation and communication overlap factor) to the parallel overhead. Excessive inter-processor communication traffic can saturate the available network bandwidth, further contributing to the parallel overhead.

11.2.2 Load Imbalance

For many problems, (for example, search and optimization) which are to be solved on a parallel computer, it is impossible (or at least difficult) to determine the size of the subproblems to be assigned to various processors. Hence the problem cannot be subdivided among the processors while maintaining a uniform work load i.e., roughly equal amount of work per processor. If different processors have different work loads, some processors may be idle while other processors are working on the problem.

11.2.3 Inter-task Synchronization

During the execution of a parallel program some or all processors must synchronize at certain points due to dependences among its subtasks which are running at various processors. For example, if subtask B running on processor PB depends on subtask A running on processor PA, then the execution of subtask B is delayed (i.e., processor PB should wait) till subtask A completes its execution. If all processors are not ready for synchronization at the same time, then some processors will be idle which contributes to the parallel overhead.

11.2.4 Extra Computation

The best sequential algorithm for a problem may not have a high degree of parallelism thus forcing us to use a parallel algorithm for the same problem which has more *operation count* (This parallel algorithm when run on one processor takes more time than the best sequential algorithm as it does more computations). An example of this is the bidirectional Gaussian elimination described in Chapter 7. The extra computation should be regarded as part of parallel overhead because it expresses the amount of extra work needed to solve the problem in parallel.

11.2.5 Other Overheads

Parallel overhead also includes factors such as task creation, task scheduling, task termination, processor (shared) memory interconnect latency, cache coherence enforcement and those imposed by parallel languages, libraries, operating system, etc. It may be noted that

a coarse-grained task usually has less parallel overhead but potentially less degree of parallelism, while a fine-grained task has a larger overhead cost with a potentially larger degree of parallelism.

11.2.6 Parallel Balance Point

As the number of processors in a parallel computer is increased, the execution time to solve a problem decreases. When the size of the problem is fixed, as the number of processors employed in solving it increases, the computation time (work load) per processor decreases. And at some point, a processor's work load becomes comparable or even more than its parallel overhead. From this point onwards, the execution time of the problem starts increasing. In general, there is an optimal number of processors to be used for any given problem. This is called the *parallel balance point* for a problem, which is the point at which employing more processors without increasing the size of the problem actually increases the overall execution time. This is illustrated in Fig. 11.6.

This is one of the reasons for many parallel computers to allow *space sharing*, i.e., each program, in these machines, is allocated a subset of the available processors. If the characteristics (for example, *parallelism profile*—number of processors used at different time periods during execution) of a program are known a priori, it may be possible to allocate each program the right number of processors.

11.3 SPEEDUP PERFORMANCE LAWS

We now describe three speedup performance laws. Amdahl's law is based on a fixed problem size or a fixed work load. Gustafson's law is for scaled problems, where the problem size increases with the increase in machine size (i.e., number of processors). Sun and Ni's law is applied to scaled problems bounded by memory capacity.

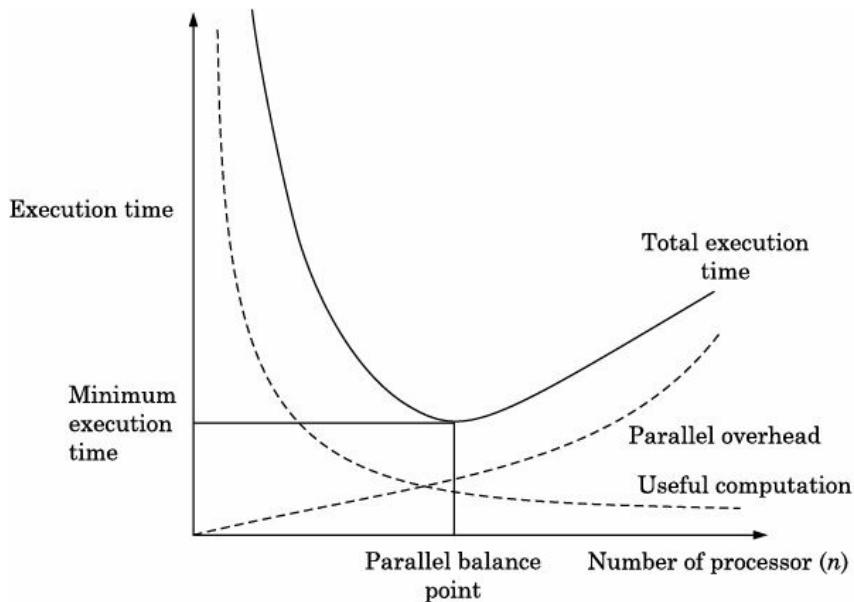


Figure 11.6 Parallel balance point.

11.3.1 Amdahl's Law

We have seen that, for a given problem size, the speedup does not increase linearly as the number of processors increases. In fact, the speedup tends to become saturated. This is a consequence of Amdahl's law formulated in 1967.

According to Amdahl's law, a program contains two types of operations—completely sequential (such as the initialization phase, reading in the data or disk accesses and collecting the results) which must be done sequentially and completely parallel which can be (parallelized to) run on multiple processors. Let the time taken to perform the sequential operations (T_s) be a fraction, α ($0 < \alpha \leq 1$), of the total execution time of the program, $T(1)$. Then the parallel operations take $T_p = (1 - \alpha)$. $T(1)$ time. Assuming that the parallel operations in the program achieve a linear speedup ignoring parallel overhead (i.e., these operations using n -processors take $(1/n)$ th of the time taken to perform them on one processor), then the speedup with n processors is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{\alpha T(1) + (1 - \alpha) \frac{T(1)}{n}} = \frac{1}{\alpha + \frac{1 - \alpha}{n}}$$

Figures 11.7 and 11.8 show the speedup as a function of number of processors and α . Note that the sequential operations will tend to dominate the speedup as n becomes very large. This means, no matter how many processors are employed, the speedup in this problem is limited to $1/\alpha$. This is called the *sequential bottleneck* of the problem. Again note that this sequential bottleneck cannot be removed just by increasing the number of processors. As a result of this, two major impacts on the parallel computing industry were observed. First, manufacturers

were discouraged from building large-scale parallel computers. Second, more research attention was focused on developing optimizing (parallelizing) compilers which reduce the value of α .

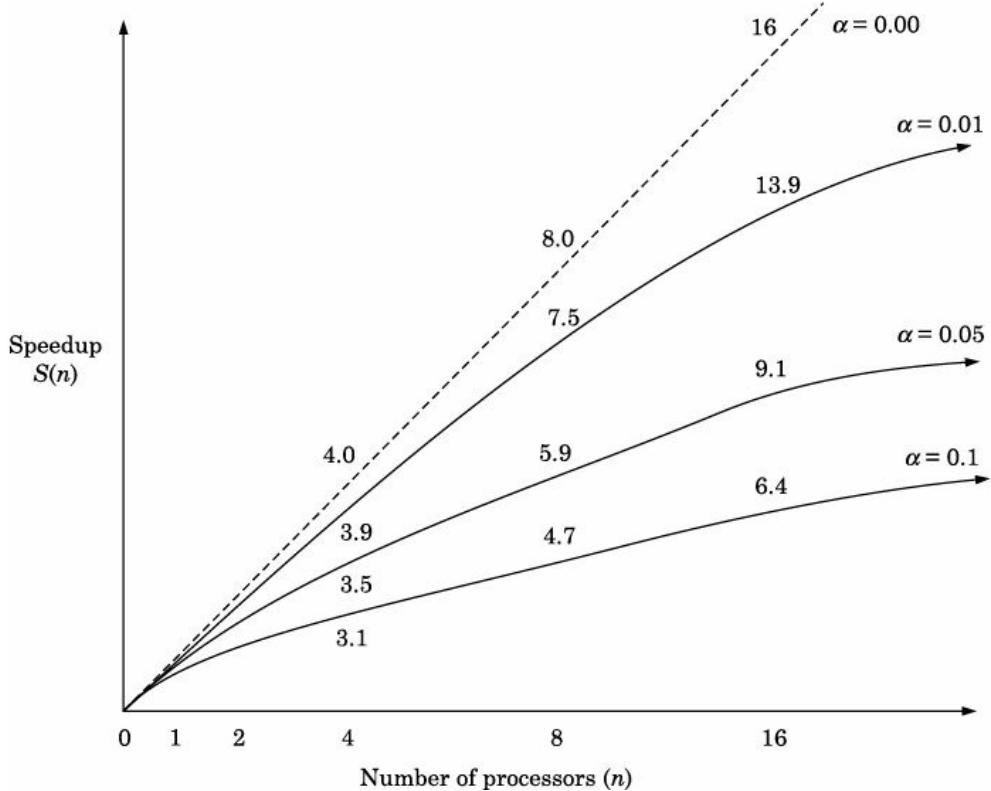


Figure 11.7 Speedup versus number of processors (figure not to scale).

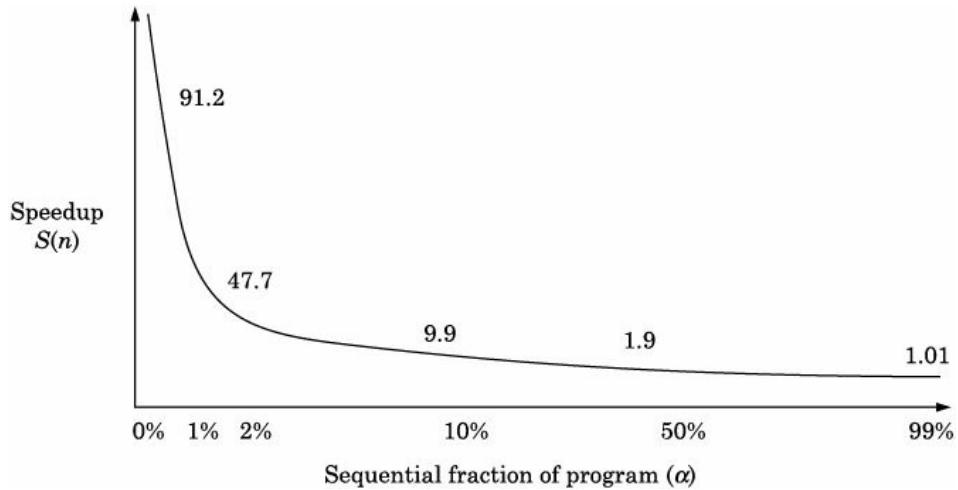


Figure 11.8 Effect of Amdahl's law (for $n = 1024$; figure not to scale).

The major shortcoming in applying Amdahl's law is that the total work load or the problem size is fixed as shown in Fig. 11.9. Here W_s and W_p denote the sequential and parallel work load, respectively. (Note that in Fig. 11.10, the execution time decreases with increasing number of processors because of the parallel operations.) The amount of sequential operations in a problem is often independent of, or increases slowly with, the problem size, but the amount of parallel operations usually increases in direct proportion to the size of the problem. Thus a successful method of overcoming the above shortcoming of Amdahl's law is to increase the size of the problem being solved.

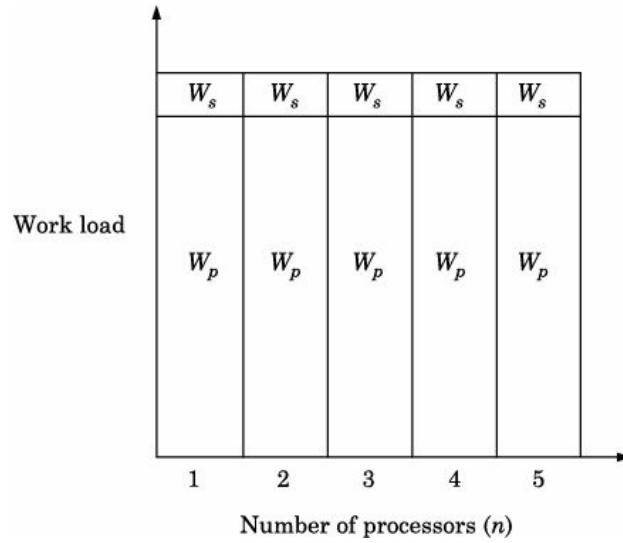


Figure 11.9 Constant work load for Amdahl's law.

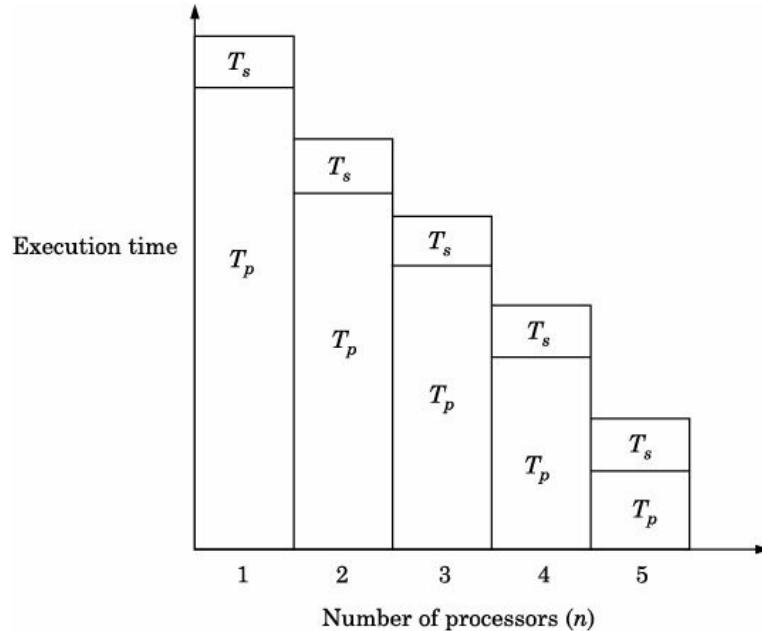


Figure 11.10 Decreasing execution time for Amdahl's law.

Strong Scaling

A program is said to be strongly scalable if we increase the number of processors, we can keep the efficiency fixed without increasing the problem size. The maximum achievable speedup under strong scaling is $1/\alpha$.

Amdahl's Law for Multicore Processors

A modern version of the Amdahl's law states that if f is the portion of the work load that can be parallelized and n is the number of processors (cores), then the parallel processing speedup is given by

$$S(f, n) = \frac{1}{(1-f) + \frac{f}{n}}$$

A simple hardware model considered by Hill and Marty assumes that a multicore processor contains a fixed number, say n , Base Core Equivalents (BCEs) in which either (i)

all BCEs implement the baseline core (homogeneous or symmetric multicore processor) or (ii) one or more cores are more powerful than the others (heterogeneous or asymmetric multicore processor). As an example, with a resource budget of $n = 16$ per chip, a homogeneous multicore processor can have either 16 one-BCE cores or 4 cores of 4-BCE cores whereas a heterogeneous multicore processor can have either 1 four-BCE core (one powerful core is realized by fusing 4 one-BCE cores) and 12 one-BCE cores or 1 nine-BCE core and 7 one-BCE cores. In general, a homogeneous multicore processor has n/r cores of r -BCE cores and a heterogeneous chip has $1 + n - r$ cores because the single larger core uses r resources leaving $n - r$ resources for the one-BCE cores. The performance of a core is assumed to be a function of number of BCEs it uses. The performance of a one-BCE is assumed to be 1; $\text{perf}(1) = 1 < \text{perf}(r) < r$. The value of $\text{perf}(r)$ is hardware/implementation dependent and is assumed to be \sqrt{r} . This means r BCE resources give \sqrt{r} sequential performance.

Homogeneous Multicore Processor

Sequential fraction $(1 - f)$ uses 1 core at performance $\text{perf}(r)$ and thus, sequential time = $(1 - f)/\text{perf}(r)$. Parallel fraction uses n/r cores at rate $\text{perf}(r)$ each and thus, parallel time = $f/(\text{perf}(r) * (n/r)) = f * r/(\text{perf}(r) * n)$. The modified Amdahl's law is given by

$$S(f, n, r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f.r}{\text{perf}(r).n}}$$

It was found that for small r , the processor performs poorly on sequential code and for large r , it performs poorly on parallel code.

Heterogeneous Multicore Processor

Sequential time = $(1 - f)/\text{perf}(r)$, as before. In parallel, 1 core at rate $\text{perf}(r)$, and $(n - r)$ cores at rate 1 give parallel time = $f/(\text{perf}(r) + n - r)$. The modified Amdahl's law in this case is given by

$$S(f, n, r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f}{\text{perf}(r) + n - r}}$$

It was observed that speedups obtained are better than for homogeneous cores. In both the cases, the fraction f should be as high as possible (just as followed from traditional Amdahl's law) for better speedups. This analysis ignores important effects of static and dynamic power, as well as on-/off-chip memory system and interconnect design.

Amdahl's Law for Energy-Efficient Computing on Many-core Processor

A many-core processor is designed in a way that its power consumption does not exceed its power budget. For example, a 32-core processor with each core consuming an average of 20 watts leads to a total power consumption of 640 watts assuming all cores are active. As noted in Chapter 5, multicore scaling faces a power wall. The number of cores that fit on a chip is much more than the number that can operate simultaneously under the power budget. Thus, power becomes more critical than performance in scaling up many-core processors.

Woo and Lee model power consumption for a many-core processor, assuming that one core in active state consumes a power of 1. Sequential fraction $(1 - f)$ consumes $1 + (n - 1)k$ power, where n is the number of cores and k is the fraction of power, the processor consumes in idle state ($0 \leq k \leq 1$). Parallel fraction f uses n cores consuming n amount of power. As it takes $(1 - f)$ and f/n to execute the sequential and parallel code, respectively, the formula for

average power consumption (denoted by W) for a homogeneous many-core processor is as follows:

$$W = \frac{(1-f) \cdot \{1 + (n-1)k\} + \frac{f}{n} \cdot n}{(1-f) + \frac{f}{n}}$$

$$= \frac{1 + (n-1)k(1-f)}{(1-f) + \frac{f}{n}}$$

Performance (reciprocal of execution time) per watt ($Perf/W$), which denotes the performance achievable at the same cooling capacity, based on the average power (W), is as follows:

$$\frac{Perf}{W} = \frac{1}{(1-f) + \frac{f}{n}} \cdot \frac{(1-f) + \frac{f}{n}}{1 + (n-1)k(1-f)}$$

$$= \frac{1}{1 + (n-1)k(1-f)}$$

Here, unfortunately, parallel execution consumes much more energy than sequential execution to run the program. Only when $f = 1$ (ideal case), $Perf/W$ attains the maximum value of 1. A sequential execution consumes the same amount of energy as that of its parallel execution version only when the performance improvement through parallelization scales linearly. When parallel performance does not scale linearly, the many-core processor consumes much more energy to run the program. A heterogeneous many-core processor alternative is shown to achieve better energy efficiency.

11.3.2 Gustafson's Law

Amdahl's law was motivated by time-critical applications where the response time (or faster execution) is more important. There are many other applications where the accuracy of the solution is more important than the response time. As the machine size is increased to obtain more computing power, the problem size can be increased to create a greater work load, producing a more accurate solution and keeping the execution time unchanged (see Figs. 11.11 and 11.12).

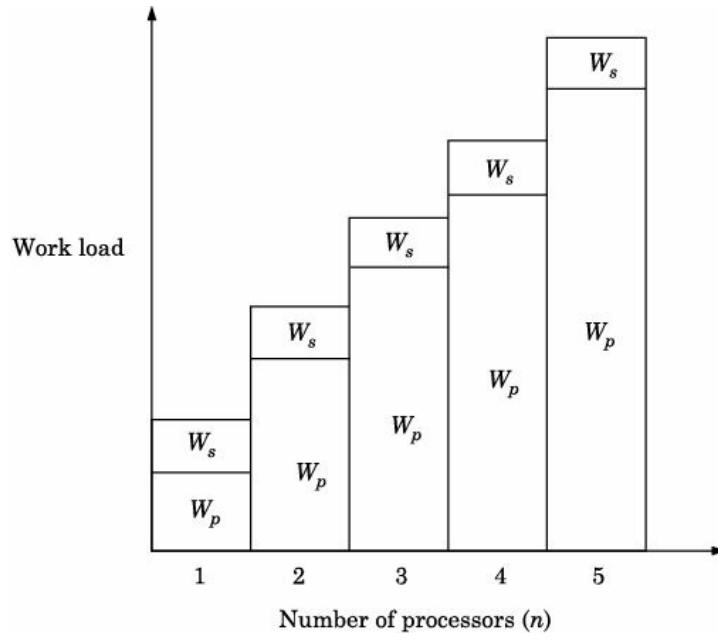


Figure 11.11 Scaled work load for Gustafson's law.

The use of finite-element method to carry out structural analysis and the use of finite difference method to solve computational fluid dynamics problems in weather forecasting are some examples. Coarser grids require fewer computations whereas finer grids require many more computations, yielding greater accuracy. The main motivation lies in producing much more accurate solution and not in saving time.

John Gustafson, in 1988, relaxed the restriction of fixed size of the problem and used the notion of fixed execution time for getting over the sequential bottleneck. According to Gustafson's law, if the number of parallel operations in the problem is increased (or scaled up) sufficiently, then the sequential operations will no longer be a bottleneck. In accuracy-critical applications, it is desirable to solve the largest problem size possible on a larger machine rather than solving a smaller problem on a smaller machine, with almost the same execution time.

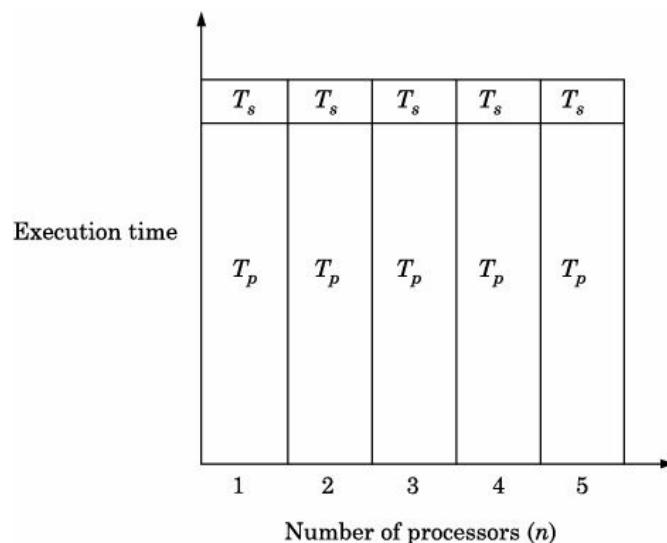


Figure 11.12 Constant execution time for Gustafson's Law.

As the machine size increases, the work load is increased so as to keep a fixed execution time for the problem. Let T_s be the constant time taken to perform the sequential operations

of the program and $T_p(n, W)$ be the time taken to perform the parallel operations of the problem with size or work load W using n processors. Then the speedup with n processors is:

$$S'(n) = \frac{T_s + T_p(1, W)}{T_s + T_p(n, W)}$$

Like in Amdahl's law, if we assume that the parallel operations in the program achieve a linear speedup (i.e., these operations using n processors take $(1/n)$ th of the time taken to perform them on one processor), then $T_p(1, W) = n \cdot T_p(n, W)$. Let α be the fraction of the sequential work load in the problem i.e.,

$$\alpha = \frac{T_s}{T_s + T_p(n, W)}$$

Then the expression for the speedup can be rewritten as

$$S'(n) = \frac{T_s + n \cdot T_p(n, W)}{T_s + T_p(n, W)} = \alpha + n(1 - \alpha) = n - \alpha(n - 1)$$

For large n , $S'(n) \approx n(1 - \alpha)$. A plot of speedup as a function of sequential work load in the problem is shown in Fig. 11.13. Note that the slope of the $S'(n)$ curve is much flatter than that of $S(n)$ in Fig. 11.8. This is achieved by keeping all the processors busy by increasing the problem size. Thus according to Gustafson's law, when the problem can scale to match the available computing power, the sequential bottleneck will never arise.

Weak Scaling

A program is said to be weakly scalable if we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processors. While Amdahl's law predicts the performance of a parallel program under strong scaling, Gustafson's law predicts a parallel program's performance under weak scaling.

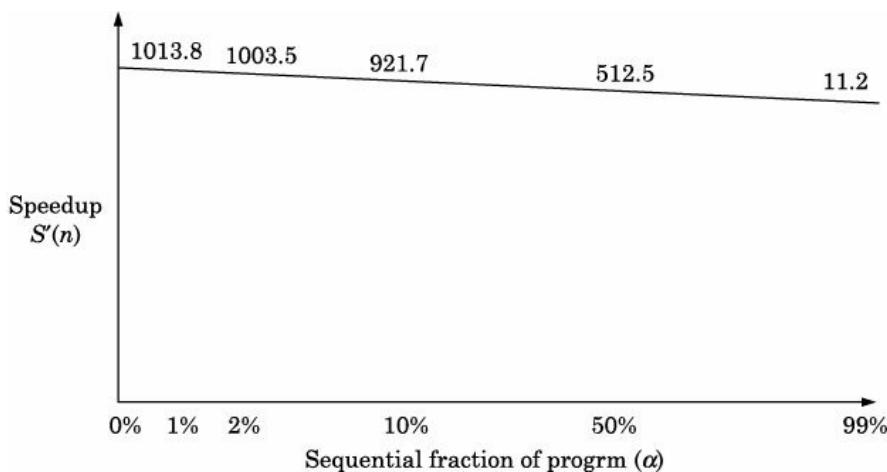


Figure 11.13 Effect of Gustafson's law (for $n = 1024$; figure not to scale).

11.3.3 Sun and Ni's Law

In 1993, Sun and Ni developed a *memory-bounded speedup* model which generalizes both Amdahl's law and Gustafson's law to maximize the use of both processor and memory capacities. The idea is to solve the maximum possible size of the problem, limited by the memory capacity. This inherently demands an increased or scaled work load, providing higher speedup, higher efficiency, and better resource (processor and memory) utilization. In

fact, many large-scale scientific and engineering applications of parallel computers are memory-bound rather than CPUbound and I/O bound.

Consider a distributed memory multiprocessor system where each processor has a small local memory. Therefore, each processor can handle only a small subproblem. When a large number of such processors are used collectively to solve a single large problem, the total memory capacity increases proportionally. This enables the system to solve a scaled problem using program or data partitioning. Instead of keeping the execution time fixed, we can use up all the increased memory by scaling the problem size further. That is, if we have adequate memory space and the scaled problem satisfies the time limit as per Gustafson's law, then we can further increase the problem size to produce a more accurate solution. This idea of solving the largest possible problem, limited only by the available memory capacity gave rise to the memory-bound model. This model assumes a scaled work load and may result in a slight increase in the execution time to achieve scalable (speedup) performance (see Fig. 11.14 and Fig. 11.15).

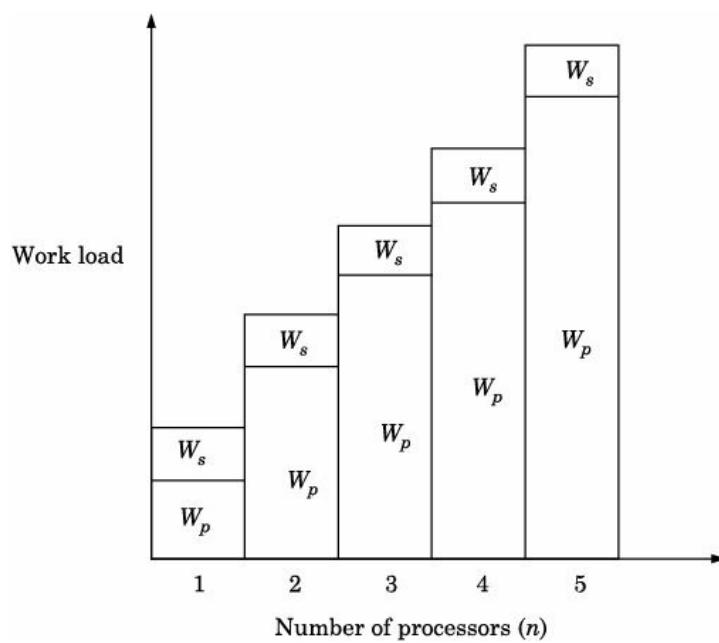


Figure 11.14 Scaled work load for Sun and Ni's law.

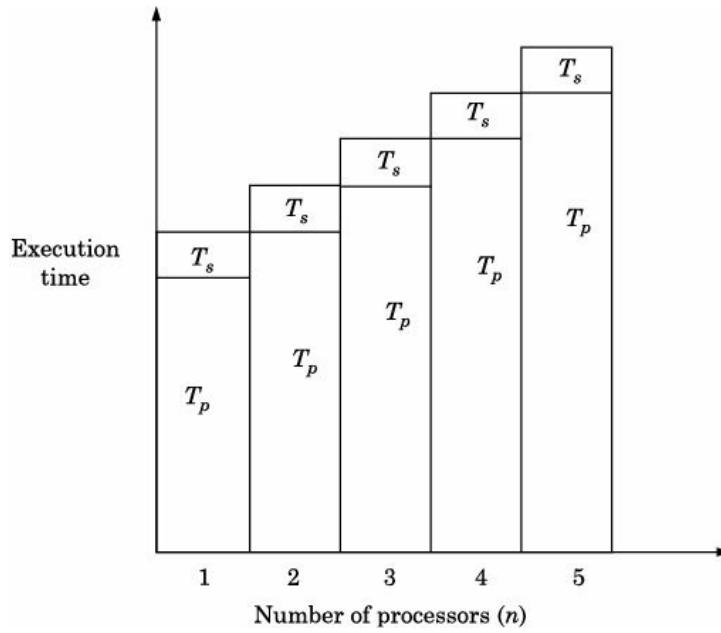


Figure 11.15 Slightly increasing execution time for Sun and Ni's law.

Let M be the memory requirement of a given problem and W be its computational work load. M and W are related to each other through the address space and the architectural constraints. The execution time T , which is a function of number of processors (n) and the work load (W), is related to the memory requirement M . Hence

$$T = g(M) \text{ or } M = g^{-1}(T)$$

The total memory capacity increases linearly with the number of processors available. The enhanced memory can be related to the execution time of the scaled work load by $T^* = g^*(nM)$ where, nM is the increased memory capacity of an n -processor machine.

We can write $g^*(nM) = G(n)g(M) = G(n) T(n, W)$, where $T(n, W) = g(M)$ and g^* is a homogeneous function. The factor $G(n)$ reflects the increase in the execution time as memory increases n times. The speedup in this case is:

$$S^*(n) = \frac{T_s + G(n)T_p(n, W)}{T_s + \frac{G(n)}{n}T_p(n, W)} = \frac{\alpha + G(n)(1-\alpha)}{\alpha + \frac{G(n)}{n}(1-\alpha)}$$

Two assumptions are made while deriving the above speedup expression:

- (i) A global address space is formed from all the individual memory spaces, i.e., there is a distributed shared memory space.
- (ii) All available memory capacity is used up for solving the scaled problem.

In scientific computations, a matrix often represents some discretized data continuum. Increasing the matrix size generally leads to a more accurate solution for the continuum. For matrices with dimension n , the number of computations involved in matrix multiplication is $2n^3$ and the memory requirement is roughly $M = 3n^2$. As the memory increases n times in an n -processor multiprocessor system, $nM = n \times 3n^2 = 3n^3$. If the increased matrix has a dimension of N , then $3n^3 = 3N^2$. Therefore, $N = n^{1.5}$. Thus $G(n) = n^{1.5}$ and

$$S^*(n) = \frac{T_s + n^{1.5}T_p(n, W)}{T_s + \frac{n^{1.5}}{n}T_p(n, W)} = \frac{T_s + n^{1.5}T_p(n, W)}{T_s + n^{0.5}T_p(n, W)}$$

Now consider the following special cases.

Case 1: $G(n) = 1$: This corresponds to the case where we have a fixed problem size i.e., Amdahl's law.

Case 2: $G(n) = n$: This corresponds to the case where the work load increases n times when the memory is increased n times i.e., Gustafson's law. For the scaled matrix multiplication problem, mentioned above,

$$S'(n) = \frac{T_s + nT_p(n, W)}{T_s + T_p(n, W)}$$

Case 3: $G(n) \geq n$: This corresponds to the case where the computational work load (time) increases faster than the memory requirement. Comparing the speedup factors $S^*(n)$ and $S'(n)$ for the scaled matrix multiplication problem, we realize that the memory-bound (or fixed memory) model may give a higher speedup than the fixed execution time (Gustafson's) model.

The above analysis leads to the following conclusions: Amdahl's law and Gustafson's law are special cases of Sun and Ni's law. When computation grows faster than the memory requirement, the memory-bound model may yield a higher speedup (i.e., $S^*(n) \geq S'(n) \geq S(n)$) and better resource utilization.

11.4 SCALABILITY METRIC

Given that increasing the number of processors decreases efficiency and increasing the amount of computation per processor increases efficiency, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processors simultaneously.

Consider the example of adding n numbers on a p -processor hypercube. Assume that it takes one unit of time both to add two numbers and to communicate a number between two processors which are directly connected. As seen earlier, the local addition takes $n/p - 1(O(n/p))$ time. After this, the p partial sums are added in $\log(p)$ steps, each consisting of one addition and one communication. Thus the parallel run time of this problem is $n/p - 1 + 2 \log(p)$, and the expressions for speedup and efficiency can be approximated by

$$S(p) = O\left(\frac{n}{\frac{n}{2} + 2 \log(p)}\right) = O\left(\frac{np}{n + 2p \log(p)}\right)$$
$$E(p) = O\left(\frac{n}{n + 2p \log(p)}\right)$$

From the above expression $E(p)$, the efficiency of adding 64 numbers on a hypercube with four processors is 0.8. If the number of processors is increased to 8, the size of the problem should be scaled up to add 192 numbers to maintain the same efficiency of 0.8. In the same way, if p (machine size) is increased to 16, n (problem size) should be scaled up to 512 to result in the same efficiency. A *parallel computing system* (the combination of a parallel algorithm and parallel machine on which it is implemented) is said to be *scalable* if its efficiency can be fixed by simultaneously increasing the machine size and the problem size. The scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the machine size.

11.4.1 Isoefficiency Function

It is useful to know the rate at which the problem size must be increased with respect to the machine size to keep the efficiency fixed. This rate determines the degree of scalability of the parallel computing system. We now introduce the notion of isoefficiency function, developed by Vipin Kumar and others, to measure scalability of parallel computing systems. Before we do this, we precisely define the term *problem size*. Suppose we express problem size as a parameter of the input size. For example, the problem size as n in case of a matrix operation involving $n \times n$ matrices. Now doubling the input size results in a four-fold increase in the execution time for matrix addition algorithm (as matrix addition takes $O(n^2)$ time). Hence a drawback of this definition is that the interpretation of problem size changes from one problem to another. So we define problem size as the number of basic operations required to solve the problem. This definition is consistent as it ensures that doubling the problem size always means performing twice the amount of computation. By this definition, the problem size is $O(n^2)$ for $n \times n$ matrix addition. Similarly, for matrix multiplication the problem size is $O(n^3)$.

An isoefficiency function shows how the size of a problem must grow as a function of the number of processors used in order to maintain some constant efficiency. We can derive a general form of an isoefficiency function by using an equivalent definition of efficiency:

$$E(p) = \frac{U}{U + O} = \frac{1}{1 + \frac{O}{U}}$$

where U is the time taken to do the useful (essential work) computation and O is the parallel overhead. Note that O is zero for sequential execution. If we fix the efficiency at some constant value, K , then

$$U = \frac{K}{1 - K} \quad O = K' O$$

where K' is a constant for a fixed efficiency K . This function is called the isoefficiency function of the parallel computing system. If we wish to maintain a certain level of efficiency, we must keep the parallel overhead, O , no worse than proportional to the total useful computation time, U . A small isoefficiency function means that small increments in the problem size (or U) are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. On the other hand, a large isoefficiency function indicates a poorly scalable system. To keep a constant efficiency, in such a system, the problem size must be explosive (see Fig. 11.16).

Consider the example of adding n numbers on a p -processor hypercube. Assume that it takes one unit of time both to add two numbers and to communicate a number between two processors which are directly connected. The parallel run time of this problem, as seen earlier, is approximately $n/p + 2 \log(p)$. Thus, $U = n$ and $O = 2p \log(p)$, as each processor, in the parallel execution, spends approximately n/p time for useful work and incurs an overhead of $2 \log(p)$ time. Substituting O with $2p \log(p)$ in the above equation, we get

$$U = K' 2p \log(p)$$

Thus the isoefficiency for this parallel computing system is $O(p \log(p))$. That means, if the machine size is increased from p to p' , the problem size (n) must be increased by a factor of $(p' \log(p'))/(p \log(p))$ to obtain the same efficiency as with p processors. Finally, note that the isoefficiency function does not exist for unscalable parallel computing systems. This is because the efficiency in such systems cannot be kept at any constant value as machine size increases, no matter how fast the problem size is increased.

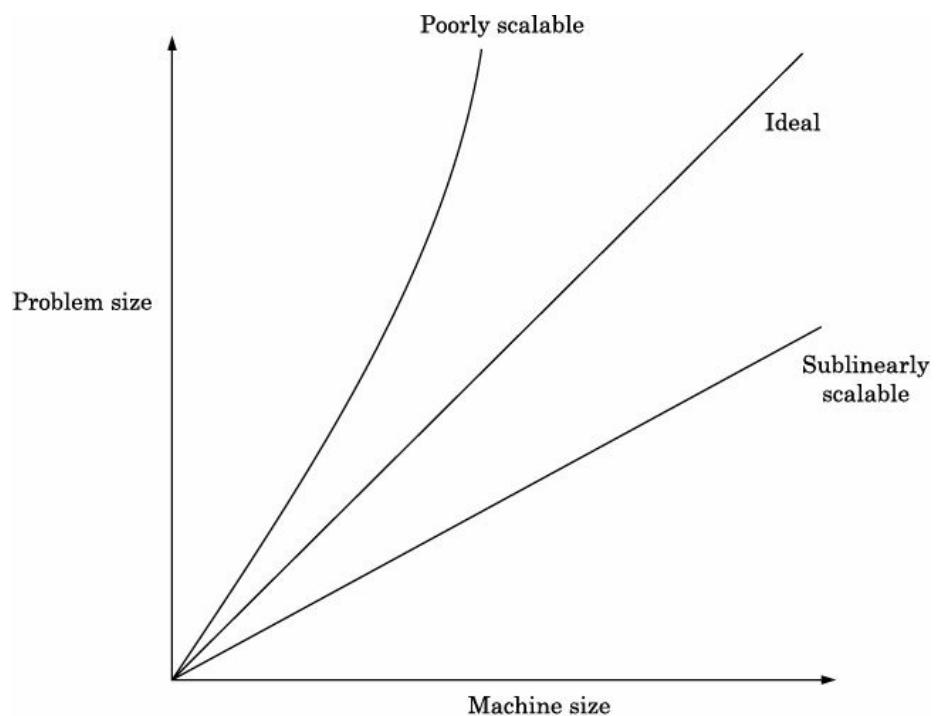


Figure 11.16 Scalable system characteristics.

11.5 PERFORMANCE ANALYSIS

The main reason for writing parallel programs is speed. The factors which determine the speed of a parallel program (application) are complex and inter-related. Some important ones are given in Table 11.2. Once a parallel program has been written and debugged, programmers then generally turn their attention to *performance tuning* of their programs. Performance tuning is an iterative process used to optimize the efficiency of a program (or to close the “gap” between hardware and software). Performance tuning usually involves (i) *profiling* using software tools to record the amount of time spent in different parts of a program and resource utilization, and (ii) finding program’s *hotspots* (areas of code within the program that use disproportionately high amount of processor time) and eliminating *bottlenecks* (areas of code within the program that use processor resources inefficiently thereby causing unnecessary delays) in them.

The software tools, which are either platform specific (refers to a specific combination of hardware and compiler and/or operating system) or cross-platform, provide insight to programmers to help them understand why their programs do not run fast enough by collecting and analyzing statistics about processor utilization rate, load index, memory access patterns, cache-hit ratio, synchronization frequency, inter-processor communication frequency and volume, I/O (disk) operations, compiler/OS overhead, etc. The statistics can be used to decide if a program needs to be improved and in such cases, where in the code the main effort should be put in. The design and development of performance tools, which can pinpoint a variety of performance bottlenecks in complex applications that run on parallel computers which consist of tens of thousands of nodes each of which is equipped with one or more multicore microprocessors, is a challenging problem. For the performance tools to be useful on such parallel systems the techniques, which are used to measure and analyze large volume of performance data for providing fine-grained details about application performance bottlenecks to the programmers, must scale to thousands of threads.

TABLE 11.2 Important Factors which Affect a Parallel Program’s Performance

<i>Application related</i>	<i>Hardware related</i>	<i>Software related</i>
Parallel algorithm Sequential bottleneck (Amdahl’s law) Problem size Memory usage patterns Communication patterns Task granularity Use of I/O	Processor architecture Memory hierarchy Interconnection network I/O configuration	Compiler and run-time system Libraries Communication protocols Operating system

11.6 CONCLUSIONS

In this chapter, we first presented two important performance metrics, speedup and efficiency, for parallel computers. Speedup indicates the amount of speed gain achieved by employing more than one processor in solving a problem while the efficiency measures the useful portion of the total work performed by the processors. Both these metrics depend on the parallel algorithm used in solving the problem and also the parallel architecture on which it is implemented. The parallel overhead prevents a parallel computer from achieving linear speedup and an efficiency of one. For evaluating parallel computers, we also introduced standard performance measures and several benchmarks which must be used with care. Then we presented three speedup performance laws: Amdahl's law for a fixed work load, Gustafson's law for scaled problems, and Sun and Ni's law for memory-bound problems. We also studied the scalability metric for analyzing the scalability of a parallel algorithm, using the concept of isoefficiency, with respect to a parallel architecture.

Finally, we looked at the need and also some issues involved in developing parallel performance measurement tools.

EXERCISES

11.1 Consider the problem of adding n numbers on a 3-dimensional hypercube. t_{comp} ($= 1$) and t_{comm} denote the (computation) time to add two numbers and the (communication) time to communicate between two processors which are directly connected, respectively. For each of the following cases calculate runtime, speedup and efficiency.

- (a) $t_{\text{comm}} = 1$,
- (b) $t_{\text{comm}} = 2$ and
- (c) $t_{\text{comm}} = 4$

11.2 Let $P(n)$ and $T(n)$ denote the total number of unit operations performed and the total execution time in steps on an n -processor system, respectively.

- (a) When $n = 1$, we can write $T(1) = P(1)$ otherwise, $T(n) < P(n)$. Why?
- (b) Consider the following performance metrics. Give physical meaning of each of these.

$$(i) \quad S(n) = \frac{T(1)}{T(n)} \text{ (Speedup)}$$

$$(ii) \quad E(n) = \frac{T(1)}{n \cdot T(n)} \text{ (Efficiency)}$$

$$(iii) \quad R(n) = \frac{P(n)}{P(1)} \text{ (Redundancy)}$$

$$(iv) \quad U(n) = \frac{P(n)}{n \cdot T(n)} \text{ (Utilization)}$$

$$(v) \quad Q(n) = \frac{S(n)E(n)}{R(n)} \text{ (Quality)}$$

11.3 Based on the definitions given in the above problem, prove the following:

- (a) $1 \leq S(n) \leq n$
- (b) $U(n) = R(n) \cdot E(n)$
- (c) $1/n \leq E(n) \leq U(n) \leq 1$
- (d) $1/n \leq R(n) \leq 1/E(n) \leq n$
- (e) $Q(n) \leq S(n) \leq n$

11.4 Visit SPEC website to know more about the benchmark suites SPEC ACCEL, SPEC MPI2007 and SPEC OMP2012 and prepare a short summary, for each of these, about programming language used, computations/application performed/included and information obtained by running the benchmarks.

11.5 What are the different type of programs contained in SPLASH-2 and PARSEC benchmark suites?

11.6 Consider the problem of sorting n numbers using bubble sort algorithm. The worst case time complexity of solving it is $O(n^2)$ on a single processor system. The same problem is to be solved, using the same algorithm, on a two-processor system. Calculate the speedup factor for $n = 100$, assuming unit constant of proportionality for both sorting and merging. Comment on your result.

11.7 The execution times (in seconds) of four programs on three computers are given below:

--	--	--	--

Program	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

Assume that each of the programs has 10^9 floating point operations to be carried out. Calculate the Mflops rating of each program on each of the three machines. Based on these ratings, can you draw a clear conclusion about the relative performance of each of the three computers?

- 11.8 Let α be the percentage of a program code which must be executed sequentially on a single processor. Assume that the remaining part of the code can be executed in parallel by all the n homogeneous processors. Each processor has an execution rate of x MIPS. Derive an expression for the effective MIPS rate in terms of α , n , x assuming only this program code is being executed.
- 11.9 A program contains 20% of non-parallelizable code and the remaining being fully parallelizable. How many processors are required to achieve a speedup of 3? Is it possible to achieve a speedup of 5? What do you infer from this?
- 11.10 What limits the speedup in the case of (a) Amdahl's Law and (b) Sun and Ni's Law?
- 11.11 Extend Amdahl's law to reflect the reality of multicore processors taking parallel overhead $H(n)$ into consideration. Assume that the best sequential algorithm runs in one time unit and $H(n)$ = overhead from operating system and inter-thread activities such as synchronization and communication between threads.
- 11.12 Analyze multicore scalability under (i) fixed-time, and (ii) memory-bounded speedup models.
- 11.13 Derive a formula for average power consumption for a heterogeneous many-core processor.
- 11.14 Consider the problem of adding n numbers on a p -processor hypercube. Assume that the base problem for $p = 1$ is that of adding 256 numbers. By what fraction should the parallel portion of the work load be increased in order to maintain constant execution time if the number of processors used is increased to (a) $p = 4$ and (b) $p = 8$.
- 11.15 Is it possible to solve an arbitrary large problem in a fixed amount of time, provided that unlimited processors are available? Why?
- 11.16 Consider the problem of adding n numbers on an n -dimensional hypercube. What should be the size of the machine in order to achieve a good trade-off between efficiency and execution time? (Hint: Consider the ratio $E(n)/T(n)$.)
- 11.17 Consider the problem of computing the s -point FFT (Fast Fourier Transform), whose work load is $O(s \log(s))$, on two different parallel computers, hypercube and mesh. The overheads for the problem on the hypercube and mesh with n -processors are
- $$O_1(s, n) = O(n \log(n) + s \log(n)) \text{ and } O_2(s, n) = O(n \log(n) + s\sqrt{n})$$
- respectively. For which of these architectures, is the problem more scalable?
- 11.18 What makes the performance tuning of parallel programs substantially different from the analogous process on sequential machines?

11.19 What are the important factors that need to be considered while developing a parallel performance measurement tool?

BIBLIOGRAPHY

- Amdahl, G., "Validity of Single Processor Approach to Achieving Large-Scale Computer Capabilities", *Proceedings of AFIPS Computer Conference*, 1967.
- Barney, B., "Introduction to Parallel Computing", 2012.
https://computing.llnl.gov/tutorials/parallel_comp/#DesignPerformance.
- Barney, B., "Performance Analysis Tools", 2002.
https://computing.llnl.gov/tutorials/performance_tools/
- Casavant, T.L., Tvardik, P. and Plasil, F. (Eds.), *Parallel Computers: Theory and Practice*, IEEE Computer Society Press, USA, 1996.
- Fosdick, L.D., Jessup, E.R., Schable, C.J.C. and Domik, G., *An Introduction to High-Performance Scientific Computing*, Prentice-Hall, Delhi, 1998.
- Grama, A., Gupta, A. and Kumar, V., "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures", *IEEE Parallel and Distributed Technology*, Vol. 1, No. 3, Aug. 1993.
- Gustafson, J.L., "Reevaluating Amdahl's Law", *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 532–533.
- Helmbold, D.P. and McDowell, C.E., "Modeling Speedup (n) Greater Than n ", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 2, Apr. 1990, pp. 250–256.
- Hill, M.D. and Marty, M.R., "Amdahl's Law in the Multicore Era", *IEEE Computer*, Vol. 41, No. 7, July 2008, pp. 33–38.
- Hwang, K. and Xu, Z., *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, USA, 1998.
- Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Singapore, 1993.
- Intel Trace Analyzer and Collector <https://software.intel.com/en-us/intel-trace-analyzer/>
- Intel VTune Performance Analyzer <https://software.intel.com/en-us/intel-vtune>
- Kumar, V., Grama, A., Gupta, A. and Karypis, G., *Introduction to Parallel Computing Design and Analysis of Algorithms*, Benjamin-Cummings, USA, 1994.
- Rauber, T. and Rungger, G., *Parallel Programming for Multicore and Cluster Systems*, Springer-Verlag, Germany, 2010.
- Rice University, HPCToolkit <http://hpctoolkit.org/>
- Sun, X.H. and Chen, Y., "Reevaluating Amdahl's Law in the Multicore Era", *Journal of Parallel and Distributed Computing*, Vol. 70, No. 2, Feb. 2010, pp. 183–188.
- Sun, X.H. and Ni, L.M., "Scalable Problems and Memory-Bound Speedup", *Journal of Parallel and Distributed Computing*, Vol. 19, 1993, pp. 27–37.
- Williams, S., Waterman, A. and Patterson, D., "Roofline: An Insightful Visual Performance Model for Multicore Architectures", *Communications of the ACM*, Vol. 52, No. 4, Apr. 2009, pp. 65–76.
- Wilson, G.V., *Practical Parallel Programming*, Prentice-Hall, Delhi, 1998.
- Woo, D.H. and Lee, H.S., "Extending Amdahl's Law for Energy-Efficient Computing in the Many-core Era", *IEEE Computer*, Vol. 41, No. 12, Dec. 2008, pp. 24–31.

Appendix

List of Acronyms Used in the Book	
ADB	Assignable Data Buffer
ALU	Arithmetic Logic Unit
API	Application Programmer Interface
BCE	Base Core Equivalent
BGE	Bidirectional Gaussian Elimination
BPB	Branch Prediction Buffer
BSP	Bulk Synchronous Parallel
BTB	Branch Target Buffer
CCNUMA	Cache Coherent Non Uniform Memory Access (Parallel Computer)
CE	Computing Element
CI	Communication Interface
CISC	Complex Instruction Set Computer
CMP	Chip Multiprocessor
CPU	Central Processing Unit
CRCW	Concurrent Read Concurrent Write
CREW	Concurrent Read Exclusive Write
CUDA	Compute Unified Device Architecture
DE	Decode instruction
DMAR	Data Memory Address Register
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
DSP	Digital Signal Processor
DSWP	Decoupled Software Pipelining
DVFS	Dynamic Voltage and Frequency Scaling
EFT	Earliest Finish Time
EPIC	Explicitly Parallel Instruction Computer

ERCW	Exclusive Read Concurrent Write
EREW	Exclusive Read Exclusive Write
EX	Execute instruction
FFT	Fast Fourier Transform
FI	Fetch Instruction
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GB	Gigabyte
GE	Gaussian Elimination
GHz	Gigahertz
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HTM	Hardware Transactional Memory
I/O	Input/Output
IaaS	Infrastructure as a Service
IEEE	Institute of Electrical and Electronic Engineers
ILP	Instruction Level Parallelism
IMAR	Instruction Memory Address Register
IR	Instruction Register
KB	Kilobyte
kHz	Kilohertz
LAN	Local Area Network
MAU	Memory Access Unit
MB	Megabyte
MC	Memory Controller
MCCMB	Mesh Connected Computer with Multiple Broadcasting
MCP	Many Core Processors
MDR	Memory Data Register
MEM	Memory Operation
MESI	Modified Exclusive Shared Invalid (Cache coherence protocol)

Mflops	Millions of floating-point operations per second
MHz	Megahertz
MIMD	Multiple Instruction Multiple Data stream
MIPS	Millions of Instructions Per Second
MISD	Multiple Instruction Single Data Stream
MMU	Main Memory Unit
MOESI	Modified Owned Exclusive Shared Invalid (Cache coherence protocol)
MPI	Message Passing Interface
MTTF	Mean-Time-To-Failure
MUX	Multiplexer
NAR	Next Address Register
NIU	Network Interface Unit
NUMA	Non Uniform Memory Access
OpenCL	Open Computing Language
OpenMP	Open specifications for Multi-Processing
OS	Operating System
PaaS	Platform as a Service
PC	Program Counter, Personal Computer
PCIe	Peripheral Component Interconnect express
PE	Processing Element
PRAM	Parallel Random Access Machine
QoS	Quality of Service
QPI	Quick Path Interconnect (Interface)
RAID	Redundant Array of Independent Disks
RAM	Random Access Machine
RAW	Read After Write (Hazard)
RIB	Router Interface Block
RISC	Reduced Instruction Set Computer
RMW	Read-Modify-Write
RW	Regenerate-Write
SaaS	Software as a Service

SAN	System Area Network
SCI	Scalable Coherent Interface (Standard)
SIMD	Single Instruction Multiple Data stream
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data stream
SL	Static Level
SLA	Service Level Agreement
SM	Shared Memory
SM	Streaming Multiprocessor
SMAC2	SMAll Computer 2
SMAC2P	SMAll Computer 2 Parallel
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
SP	Streaming Processor
SP	Schedule Priority
SPMD	Single Program Multiple Data
SR	Store Register, Store Result
SRAM	Static Random Access Memory
STM	Software Transactional Memory
TB	TeraByte
TCP/IP	Transmission Control Protocol/Internet Protocol
TD	Tag Directory
UMA	Uniform Memory Access
URL	Universal Resource Locator
VLIW	Very Long Instruction Word
VM	Virtual Machine
VPU	Vector Processing Unit
WAR	Write After Read (Hazard)
WAW	Write After Write (Hazard)
WSC	Warehouse Sale Computer
YARN	Yet Another Resource Negotiator

Index

- Agenda parallelism, 23
- Animation, 6
- Anti dependency, 68
- ARM Cortex A9 architecture, 75
- ARM cortex A9 multicore processor, 182
- Array processors, 99, 119
- Asynchronous message passing protocol, 159
- Asynchronous send, 157
- Atomic read modify write instruction, 117
- Automatic parallelization, 379
- Autotuning, 379

- Bandwidth of networks, 143
 - bisection, 143
 - link, 143
 - total, 143
- Barrier, 117
- Big data, 318
- Binary search, 248
- Binary tree network, 252
- Branch prediction buffer, 59
- Branch target buffer, 60
- Bus connected multicore processor, 179
- Butterfly circuit, 227
- Butterfly switch, 142

- Cache coherence
 - directory scheme, 132
 - in DSM, 158
 - MESI protocol, 122
 - MOESI protocol, 127
 - in shared bus multiprocessors, 119, 121
- Cache coherent non-uniform memory access, 148
- Cache misses, 120
 - capacity, 120
 - cold, 120
 - conflict, 120
- Chip multiprocessors (CMP), 174, 178
 - cache coherence, 181
 - definition of, 176
 - generalized structure, 176
 - using interconnection networks, 184
- Cloud computing, 210
 - acceptance of, 217
 - advantages, 215
 - applications of, 217
 - characteristics of, 211
 - comparison with grid computing, 219

- definition, 210
- quality of service, 211
- risks in using, 216, 211
- services in, 214
- types of, 213

Cloud services, 214

Coarse grain job, 20

Combinational circuit, 226

- depth, 227
- fan-in, 226
- fan-out, 226
- size, 227
- width, 227

Communication optimization transformation, 370

- collective communication, 372
- message pipelining, 372
- message vectorization, 371

Community cloud, 214

Comparison of grid and cloud computing, 219

Compiler transformation, 331

- correctness, 332
- scope, 333

Computation partitioning transformation, 369

- bounds reduction, 370
- guard overhead, 369
- redundant guard elimination, 370

Computer cluster, 160

- applications of, 161
- using system area network, 161

Concurrentization, 336

Control dependence, 337

Core level parallelism, 173, 175

Cray supercomputer, 108

Critical path, 393

Critical section, 414

Cross bar connection, 138

CUDA, 307

- block, 308
- coalescing, 316
- device (GPU), 307
- grid, 308
- heterogeneous computing with, 307
- host (CPU), 307
- kernel, 308
- thread, 308
- thread scheduling, 316
- thread synchronization, 316
- warp, 316

CUDA Core, 197, 198

Data dependence, 336

- anti (WAR), 336
- flow (RAW), 336
- output (WAW), 336, 337

Data flow based loop transformation, 346

- loop based strength reduction, 346
- loop invariant code motion, 347
- loop unswitching, 348
- Data layout transformation, 366
 - block decomposition (of array), 367
 - block-cyclic decomposition (of array), 368
 - cache alignment, 368
 - cyclic decomposition (of array), 367
 - scalar and array privatization, 368
 - serial decomposition (of array), 366
- Data mining, 6
- Data parallelism, 7, 15
- Data race, 129
- Data striping, 422
 - coarse-grained, 424
 - fine-grained, 423
- Decision tables, 125
- Decoupled software pipelining, 374
- Delay slot, 62
- Delay in pipeline
 - due to branch instructions, 56
 - due to data dependency, 54
 - due to resource constraints, 51
 - reduction by hardware modification, 58
 - reduction by software, 62
- Delays in pipeline execution, 51
 - control hazard, 51, 56
 - data hazard, 51, 54
 - locking, 52
 - pipeline hazards, 51
 - pipeline stall, 51
 - structural hazard, 51, 52
- Dependence, 336
 - equation, 342
 - GCD test, 344
 - graph, 337
- Diophantine equation, 340
- Directory based cache coherence protocol
 - decision tables for, 134, 136
- Dirty bit, 119
- Disk array, 421
 - access concurrency, 422
 - data transfer parallelism, 422
 - RAID, 422
- Distributed shared memory parallel computer, 101, 147
- Dynamic assignment, 18
- Dynamic power dissipation, 173
- Dynamic scheduling, 398
 - receiver-initiated, 400
 - sender-initiated, 399
- Echelon reduction, 382
- Embarrassingly parallel, 29
- Enterprise grid alliance, 209
- EPIC processor, 79

Exascale computing, 5

Exception conditions, 64, 65

False sharing, 369, 421

Features of parallel computers, 8

Fermi GPGPU, 198

Flits, 185

Flow dependency, 69

Flynn's classification

- MIMD computer, 100

- MISD computer, 100

- SIMD computer, 98, 112

- SISD computer, 98

- SPMD computer, 99, 113

Fork, 115

Frequency wall, 173

General Purpose Graphics Processing Unit (GPGPU), 195

Globus project, 206

GPU, 307

- SM, 307

- SP, 307

Grain size in parallel computer, 102

Graph embedding, 327

- congestion, 328

- dilation, 328

- expansion, 328

Grid computing, 206

- comparison with cloud computing, 219

- definition of, 206

- layered architecture, 208

Hadoop, 320

- ecosystem, 322

- HDFS, 321

- MapReduce implementation framework, 321

Hardware register forwarding, 69

Heat dissipation in processors, 173

- components of, 173

Heterogeneous element processor, 88

Hybrid cloud, 214

IA 64 processor, 79

Ideal pipelining, 39

ILP wall, 174

Infrastructure as a Service (IaaS), 214

Instruction level parallelism, 72

Instruction scheduling, 372

Instruction window, 71

- committing instruction, 72

Intel Core i7 processor, 76

Intel i7 multicore processor, 182

Intel Teraflop chip, 195

Intel Xeon-Phi coprocessor, 191

Interconnection networks, 137
 2D, 144, 145
 2D grid, 143
 hypercube, 144, 145
 multistage, 139
 N cube, 145
 parameters of, 146
 ring, 143
 router, 146
Inter-process communication, 420
Isoefficiency, 460

Join, 115

Leakage current in transistors, 173
Load-store architecture, 41
Lock, 116
Loop dependence, 338
 direction vector, 339
 distance vector, 338
Loop reordering transformation, 348
 cycle shrinking, 352
 loop distribution, 353
 loop fusion, 355
 loop interchange, 349
 loop reversal, 351
 loop skewing, 350
 loop tiling, 352
 strip mining, 351
Loop replacement transformation, 358
 array statement scalarization, 359
 loop idiom recognition, 359
 reduction recognition, 358
Loop restructuring transformation, 355
 loop coalescing, 357
 loop collapsing, 358
 loop peeling, 381
 loop normalization, 382
 loop unrolling, 355
 software pipelining, 356
Loosely coupled parallel computer, 101
Lower and upper bounds, 229

Machine parallelism, 72
 loop unrolling, 73
MapReduce, 318
Matrix multiplication, 256
 on interconnection network, 258
 on PRAM, 257
Memory access transformation, 359
 array padding, 360
 scalar expansion, 360
Memory consistency model, 128
 relaxed consistency, 130
Memory management, 420

DRAM, 421
SRAM, 421
Mesh connected many core processors, 193
MESI protocol, 122
Message passing parallel computer, 156
Middleware, 214
Models of computation, 222
combinational circuit, 226
interconnection network, 225
PRAM, 223
RAM, 223
MOESI protocol, 127
Moore's law, 173
Motivation for multicore parallelism, 175
MPI, 283
collective communication, 290
derived datatype, 286
extension, 293
message buffer, 285
message communicator, 288
message envelope, 287
message passing, 285
message tag, 287
packing, 286
point-to-point communication, 289
virtual topology, 293
Multicore processor, 1
Multistage interconnection network
butterfly, 142
three stage cube, 140
three stage omega, 141
Multithreaded processors, 82
coarse grained, 82
comparison of, 89
fine grained, 85
simultaneous multithreading, 88

NEC SX-9 supercomputer, 108, 110
Non Uniform Memory Access, 102, 148
Numerical simulation, 2

Odd-even transposition sort, 245
Omega network, 141
Open grid forum, 208
OpenCL, 317
device (CPU, GPU, DSP, FPGA), 317
kernel, 317
OpenMP, 298
API, 298
compiler directive, 301
environment variable, 306
execution model, 300
loop scheduling, 303
non-iterative construct, 304
parallel loop, 301

- scope of a variable, 301
- synchronization, 304
- Order notation, 228
- Out of order completion, 54
- Output dependency, 69

- Packet switching, 147
- Parallel algorithm, 222
 - cost, 229
 - cost-optimal, 230
 - efficiency, 230
 - number of processors, 229
 - running time, 228
 - speedup, 229
- Parallel balance point, 449
- Parallel Computer
 - CCNUMA, 148
 - classification of, 98
 - comparison of architectures, 165
 - definition, 96
 - distributed shared memory, 101, 147
 - generalized structure, 96
 - grain size in, 102
 - message passing, 156
 - NUMA, 148
 - shared memory, 101, 114, 131
 - shared bus, 118
 - vector, 104, 108
 - warehouse scale, 162
- Parallel overhead, 448
- Parallel programming, 277
 - explicit, 277
 - fork and join, 294
 - implicit, 277
 - message passing, 277
 - model, 277
 - send and receive, 278
 - shared memory, 294
- Parallelism, 394
 - ideal, 397
 - useful, 397
- Parallelization, 336
- Parallelizing compilers, 378
- Parity-based protection, 426
 - RMW, 427
 - RW, 427
- Partial evaluation, 361
 - algebraic simplification, 362
 - constant propagation, 361
 - constant folding, 361
 - copy propagation, 361
 - forward substitution, 362
 - reassociation, 362
 - strength reduction, 362
- Performance benchmark, 446

- HPC challenge, 447
 - kernel, 447
 - LINPACK, 447
 - Livermore loops, 447
 - NAS, 447
 - PARKBENCH, 447
 - PARSEC, 447
 - Rodinia, 447
 - SPECmark, 447
 - SPLASH, 447
 - synthetic, 447
- Performance measure, 446
 - MIPS, 446
 - Mflop, 446
- Performance metrics, 441
 - efficiency, 444
 - parallel run time, 441
 - speedup, 441
- Performance tuning, 462
 - bottleneck, 462
 - hotspot, 462
 - profiling, 462
- Petascale computing, 5
- Pipeline bubble, 14
- Pipelined adder, 105
- Pipelined arithmetic unit, 106
- Pipelining processing elements, 39
- Platform as a Service (PaaS), 214
- Pointer jumping, 234
- Polyhedral model, 380
- Power wall, 175
- Practical models of parallel computation, 265
 - BSP, 265
 - delay PRAM, 265
 - LogGP, 270
 - LogGPS, 270
 - LogP, 268
 - Multi-BSP, 267
 - phase PRAM, 265
- Precise exception, 64
- Predicate registers in IA64, 80
- Prefix computation, 230
 - cost-optimal, 232
 - on linked list, 234
 - on PRAM, 231
- Private cloud, 214
- Problems in data parallelism, 16
- Problems in pipelining, 14
- Procedure call transformation, 364
 - in-line, 365
 - inter-procedural analysis, 365
- Process, 82, 408
 - heavyweight, 408
 - lightweight, 408
- Process synchronization, 414

Public cloud, 213

Quality of Service, 207, 211

Quasi dynamic schedule, 20

RAID organization, 428

Read after Write (RAW) hazard, 69

Reduced Instruction Set Computer (RISC), 40

Reducing delay in pipeline, 58, 62

Redundancy elimination, 363

- common sub-expression elimination, 364

- dead-variable elimination, 364

- short circuiting, 364

- unreachable code elimination, 363

- useless code elimination, 364

Register forwarding, 55

Register renaming, 70

Register score boarding, 70

Ring bus connected multiprocessors, 189

Ring bus, 186, 189

- slotted ring, 188

- token ring, 186

Ring interconnection network, 143

Ring interconnection, 185

Router, 193

Sandy bridge architecture, 190

Scalability metric, 460

Scalable Coherent Interface (SCI) standard, 154

Scaling in parallel processing, 14, 19

Searching, 248

- on interconnection network, 252

- on PRAM, 249

Semi-automatic parallelization, 380

Sequential algorithm, 222

Sequential consistency, 117, 128

Service level agreement, 211

Shared bus architecture, 118

Shared memory computer

- using bus interconnection, 118

- using interconnection network, 131

Shared memory parallel computer, 101

Short circuit current in transistors, 173

Single Instruction Multiple Thread (SIMT), 196, 197

SMAC2P

- architecture of, 40

- data flows in, 47, 48

- instruction format, 41

- instruction set, 41

- pipelined execution, 49

Software as a Service (SaaS), 214

Software pipelining, 356, 374

Sorting, 236

- bitonic network for, 237

- combinational circuit for, 237

- on interconnection network, 244
- on PRAM, 243
- Spark, 323
- Speculative loading, 82
- Speedup in pipeline processing, 14
- Speedup performance law, 449
 - Amdahl's law, 450
 - Amdahl's law for multicore and manycore processor, 453, 454
 - Gustafson's law, 455
 - Sun and Ni's law, 457
- State diagram for MESI protocol, 123
- Static assignment of jobs, 16
- Stencil computation, 350
- Streaming multiprocessor, 199
- Stride, 334
- Strong scaling, 452
- Supercomputer, 108
 - Cray, 108
 - NEC SX 9, 110
- Superlinear speedup, 442
- Superpipelining, 66
- Superscalar processors, 66
- Symmetric multiprocessor, 114
- Synchronization of processes, 114
- Synchronous message passing protocol, 157
- Systems of linear equations, 260
 - BGE, 262
 - GE, 260
- Systolic processor, 100, 113
- Task duplication, 432
- Task granularity, 394
 - course-grained, 396
 - fine-grained, 396
- Task graph, 25, 390
- Task scheduling, 390
 - in message passing parallel computer, 390
 - in shared memory parallel computer, 402
- Task scheduling for multicore processor, 402
 - asymmetry-aware, 406
 - contention-aware, 407
 - DVFS-aware, 403
 - thermal-aware, 404
- Temporal parallelism, 7, 13
- Tera processor, 87
- Test and set instruction, 117
- Thermal emergency, 404
- Thread, 82, 408
 - execution model, 412
 - kernel-level, 410
 - scheduling, 411
 - user-level, 410
- Thread level parallelism, 82
- Threads library, 410
- Three phase protocol, 157

Tightly coupled parallel computer, 101
Trace scheduling, 74, 373
Transactional memory, 415
 HTM, 419
 STM, 419
Transformation framework, 376
Transformation matrix, 377

Uniform memory access, 101
Unimodular matrix, 344
Unlock, 116

Vector computers, 104
Vectorization (vectorize), 336
Very Long Instruction Word Processor (VLIW), 73, 334
Virtual cut through routing, 147
Virtualization, 212
Von Neumann architecture, 1

Warehouse scale computing, 162
 components of, 173
Weak scaling, 457
Wormhole routing, 147, 185
Write after Read (WAR) hazard, 68
Write after Write (WAW) hazard, 69, 70
Write back protocol, 119
Write through protocol, 119

YARN, 323