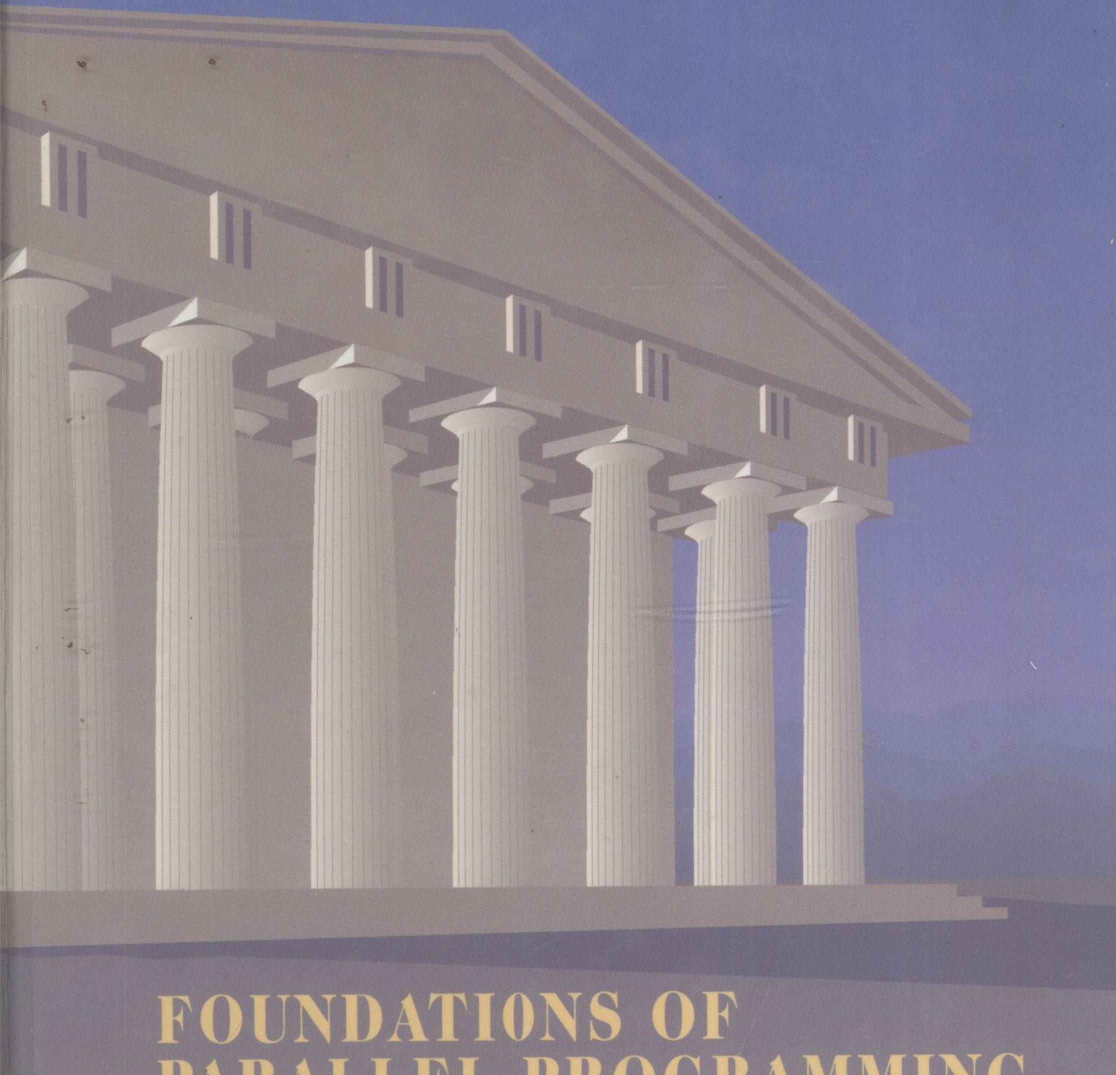


FOUNDATIONS OF PARALLEL PROGRAMMING

A Machine-Independent Approach

Ted G. Lewis

IEEE COMPUTER SOCIETY PRESS

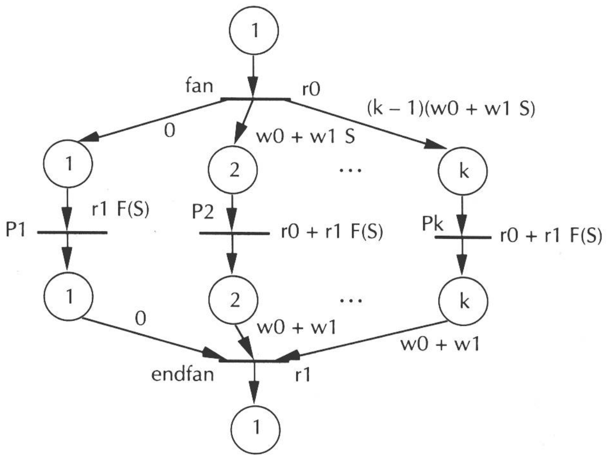
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

TP31
L676

9560858

Foundations of Parallel Programming

A Machine-Independent Approach



Ted G. Lewis

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

Library of Congress Cataloging-in-Publication Data
Lewis, T.G. (Theodore Gayle), 1941–
Foundations of parallel programming: a machine-independent approach / Ted G. Lewis
p. cm.
Includes bibliographical references and index.
ISBN 0-8186-5692-1 (case). — ISBN 0-8186-5691-3 (fiche)
1. Parallel programming (Computer science) I. Title.
QA76.642.L48 1994
005.2 — dc20 94-4873
CIP

Published by the IEEE Computer Society Press
10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264

Technical Editor: Jon Butler
Production Editor: Lisa O'Conner
Cover Artist: Alex Torres

IEEE Computer Society Press Order Number 5692-04
IEEE Catalog Number EH0393-9
Library of Congress Number 94-4873
ISBN 0-8186-5691-3 (microfiche)
ISBN 0-8186-5692-1 (case)

This book was acquired, developed, and produced by Manning Publications Co.
3 Lewis Street, Greenwich, CT 06830.

Design: Frank Cunningham
Copyediting: Margaret Marynowski
Typesetting: Aaron Lyon

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Preface

In parallel computing applications, one faces the difficult challenge of building software which can take advantage of highly parallel systems. Parallel software is so poorly understood that a new branch of software engineering will be required to achieve the transition from sequential to parallel applications. At the heart of this transition is parallel programming.

The "genius compiler" advocates believe it is possible to construct compilers that translate sequential programs written in our favorite sequential languages into parallel programs. In this scenario, FORTRAN, Pascal, and C source programs are converted into highly efficient parallel equivalents. Thus, a sequential program can be moved onto a parallel processor, compiled, and run many times faster than on a sequential processor.

The problem with this approach is the underlying assumption that sequential algorithms *can* be converted into parallel algorithms. In many cases this is true, but in other cases it is impossible to convert a good sequential algorithm into a good parallel algorithm because the two approaches are radically different. For example, a binary search is known to be a good sequential search algorithm, but a linear search done on N processors is better, given enough processors. Yet, it is unlikely that a genius compiler will ever be able to convert a program which implements binary search into a parallel program which implements a much faster linear search algorithm.

The second option is to discard all sequential programs and start over again! This extreme view has the advantage that one can redesign programs to use algorithms which are efficient on parallel processors. Clearly, it will be a long time before the world converts all of its existing sequential software into parallel programs which implement efficient parallel algorithms.

An intermediate prescription borrows from both extremes. Some sequential programs can be converted automatically into parallel equivalents, and others will need to be thrown away and rewritten. In this approach, new applications would be coded directly in new explicitly parallel languages so that after a period of perhaps decades, the transition to parallelism would be 100% complete. Sequential programming would become a subset of parallel programming, and there would no longer be a division between the sequential and parallel paradigms.

The problem with this approach is that it is still too early to decide on the features of a good parallel programming language. Each parallel computer seems to offer its own dialect of parallel FORTRAN, parallel C, or some new language altogether. If we are going to begin now, what language should we use? Selecting a new parallel programming language at this early stage might be just as wasteful as continuing to program in sequential languages.

This book is not about parallel algorithms; nor is it about how to write parallel programs for a specific machine. Instead, it is about the *design* of parallel programs using a small number of fundamental constructs which are powerful enough to express any parallel algorithm. Its emphasis is on foundations and concepts rather than syntax and machine dependencies.

This book's approach might also be called *performance-based design*, because it develops analytical measures of performance for each of the fundamental constructs of parallel programming. These analytical results can be used by a human programmer or a machine compiler to optimize the performance of any parallel program. This is especially important in parallel programming, because the object of parallelism is performance.

This approach is also machine independent. The only assumption made is that the parallel processor has N processors, linked by some interconnection network. Different machines may exhibit different performance characteristics, but they all possess multiple processors and some form of interconnection. One can use the performance-based design formulas to optimize a given program for a given architecture.

How can a programming book be both language and machine independent? This book proposes a simple *pseudo-code* notation for describing parallel programs. This notation is rigorous enough to be incorporated into a language some day, but for the purposes of this book it is strictly a pedagogical device. Even so, one can express any parallel algorithm succinctly and correctly in this notation, and derive performance formulas accordingly.

In a sense, this notation is a specification language for parallel programs. These blueprints are given in a very structured manner, so that students of parallel computing can understand the ideas without getting lost in details. Thus, the concepts of data distribution, synchronization, tasking, allocation of tasks to processors, and the trade-off between communication and computation are all made explicit without obfuscating details.

What are the fundamentals of parallel programming? Each chapter of this book, commencing after the introductory chapters, deals with a major building block used in parallel programming. Each of these building blocks is rooted in a fundamental concept which can be expressed as a programming construct. Thus, Chapter 4 introduces the data-parallel **fan**; Chapter 5 introduces the reduction **tree**, and so forth. These building blocks are sufficient to express any parallel algorithm. Furthermore, each construct corresponds to a fundamental concept of *flow-correct* programs.

The text covers concepts of both *fine-grained* and *large-grained* parallelism, beginning with low-level fine-grained parallelism at the statement level, and working up to procedural parallelism. This also corresponds to increasingly more difficult control problems, e.g., synchronization and race conditions.

This book is designed for an upper division undergraduate course for students in the physical sciences and engineering. Many of the examples are taken from science and engineering. Most students will have had calculus, programming languages, and operating systems prerequisites.

The author would like to thank the students of Oregon State University who have suffered through early drafts of this book. Their comments and questions have vastly improved the material. In addition, the manuscript has benefitted from a number of unknown reviewers.

TED LEWIS
lewis@cs.nps.navy.mil

Contents

CHAPTER 1

Models and Measures of Parallelism

There are many varieties of parallelism; each variety is called a *paradigm*. A paradigm is a way of viewing the world, and in computing, a world view becomes a program design and coding style. Therefore, a *programming paradigm* dictates the abstractions used by programmers. One abstraction might be represented by message-passing, while another abstraction might be represented by synchronization mechanisms called *locks*.

We classify parallelism according to two broad paradigms: control-flow and data-parallel. We claim that the control-flow paradigm is the most general, but does not yield a high degree of parallelism. Data-parallelism is more restricted, but generally yields very high levels of parallelism. We will show that control-flow parallelism can be implemented efficiently on multiple-instructions–multiple-data (MIMD) machines using either message-passing or locking. Message-passing is preferred on distributed-memory machines, while locking is preferred on shared-memory machines. Further, we will show that data-parallel parallelism can be implemented efficiently on message-passing machines in either the single-procedure–multiple-data (SPMD) or single-instruction–multiple-data (SIMD) form. Thus, MIMD, SPMD, and SIMD are architectures for supporting either the control-flow or data-parallel paradigms.

We claim that the *Amdahl Law* of speedup governs control-flow parallelism, while the *Gustafson–Baris Law* governs data-parallel parallelism. We will briefly derive these laws to gain an understanding of their bases; one in the world of control-flow parallelism and the other in the world of data-flow parallelism. These are idealized laws, so one must turn to a more detailed analysis of each parallel algorithm to determine practical bounds on performance. We will define scalability as the ability to obtain N-fold speedup in the face of communications overhead; parallel-computable algorithms as scalable algorithms; and quasi-scalability as speedups in excess of unity, but short of parallel-computable speedups.

We will present Petri nets and Gantt charts as means of visualizing the execution semantics and performance of arbitrary parallel programs. In addition to providing a crisp definition of the semantics of each construct, these diagrams allow performance analysis, and lead to greater insights into parallelism.

The models presented here assume a linear relationship between processor speed and process execution; and message-passing and communication delays. Furthermore, we will assume very simple interconnection networks, ignoring contention, for example, as well as routing overhead. These assumptions may not hold in general.

1.1 PROCESSORS, MEMORIES, AND NETWORKS

A *process* is any single flow of control through a set of instructions stored in a computer, and a *processor* is a hardware device for executing a process. A *parallel computer* is a collection of two or more processors connected to one another through an *interconnection network* or memory. A *parallel program* contains more than one process. The purpose of a parallel computer is to run parallel programs. Note that it is possible for a parallel program to run on a single processor, sequentially, such that each of its processes runs one after the other. Clearly, the advantage of parallel computers is that they deliver greater performance than single-processor computers.

The most general form of a parallel computer is shown in Figure 1.1. If the processors operate independently of one another but with occasional pauses to synchronize their processes, we call the parallel computer a multiple-instruction–multiple-data (MIMD) system. Alternately, if the processors operate in lock-step unison, synchronizing with one another after every instruction, we say the parallel computer is a single-instruction-multiple-data (SIMD) system. SIMD processors simultaneously execute exactly the same instructions, but on different data.

Examples of MIMD machines are the Intel iPSC series, nCUBE series, and other commercial products that link a number of commodity microprocessors together to form a single system. MIMD systems contain multiple sequencing units, which means that they can operate asynchronously and independently. Each processor runs under the control of its own sequencing unit, which means that many different instructions can be simultaneously executed, one in each processor.

Examples of SIMD machines are the Thinking Machines, Inc., Connection Machine (CM) series and the Maspar Computer Corp. MP series, products that link together a number of processing elements under the control of a single

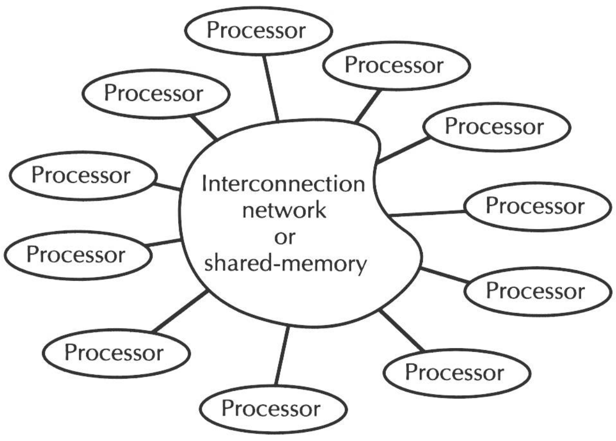


Figure 1.1 General model of a parallel computer

sequencing unit. In a SIMD system, all processing elements do the same thing at the same time, or else they are idle. The processing elements may be incomplete computers that perform simple arithmetic operations on distributed data, or as in Thinking Machines' CM-5 series, they may be entire computers. The most important distinction between MIMD and SIMD architectures is the degree of synchronization among processors: SIMD architectures are much more tightly synchronized.

Processors in SIMD computers are almost always connected by some form of interconnection network that permits them to pass messages among each other. In such systems, memory is associated with individual processors rather than the group of processors; hence, there is no *central memory*. An application's data must be copied and sent to where they will be processed. Thus, SIMD machines are also *distributed-memory machines*.

For example, in the Maspar series, processors and memory are arranged as a mesh-structured array. Each processor/memory subsystem is connected to its nearest neighbors on the north, east, west, and south (NEWS) borders. Other distributed-memory SIMD machines are linked together by *hypercube* interconnection networks, or even more exotic networks. In a hypercube interconnection network, processors are given a binary number designation such as 011 (3 in decimal), and only neighbors that differ in one bit are connected. Thus, processor 011 is connected to processors 111, 001, 010.

The trend has been toward more and more sophisticated interconnections of memories and processors. If the interconnection creates one path through all processors, it is called a *1-D network*; paths that can be drawn on a single sheet of paper without crossing each other are called *2-D networks*; *3-D networks* must be drawn in 3 dimensions, and so forth. Interconnection networks are called *static* if their connections are hardwired into the machine at the factory, and *dynamic* if it is possible for processor–processor connections to be switched while the machine runs.

Some networks collect data into *packets* before transmitting them from one processor to the other, while other networks establish a route between two processors, and transmit data for as long as the route exists. *Packet-switched networks* break messages into packets; *circuit-switched networks* operate like a telephone system and send messages over closed-circuit routes. In a packet-switched network, delays may be introduced by intermediate processors, because the message hops from one processor to the next, along the network. *Worm-hole routing* bypasses intermediate hops, thus achieving greater performance than that of purely packet-switched networks.

MIMD machines are usually either distributed-memory or shared-memory systems. If the architecture is a shared-memory design such as the Sequent Symmetry, Silicon Graphics Onyx, or Sun Microsystems multiple-processing system, processes must synchronize their access to shared data, or else *indeterminate* results may occur. Thus, shared-memory MIMD programmers must be concerned with locking and protection mechanisms.

A distributed-memory MIMD machine like the Intel Paragon uses message-passing to synchronize the parallel parts of an application program. When the message-passing style of parallel programming is adopted, an application's data are distributed among the processors' local memories, where they are processed in parallel. In this paradigm, the MIMD programmer must be concerned with copying and distributing the data.

Thus, even the MIMD paradigm calls for two radically different styles of programming. This necessity is a major hindrance to the progress of parallel computing, because details of the machine architecture creep into the design of software. Portability, and reuseability of software are greatly hampered by such machine dependencies.

For example, shared-memory programming is similar to operating systems programming, where processes are forked and joined to achieve a level of concurrency. No message-passing or copying of data is needed, and data are shared merely by declaring them as shared. But, in distributed-memory systems such as the nCUBE and Intel iPSC series machines, message-passing is used to distribute the data. These machines use Send and Receive primitives as illustrated by the following example. A Send primitive places data in a buffer, which is then emptied by the operating system. A Receive primitive forces a processor to wait for some message, then to copy it into the address space of a waiting destination process. Once distributed, additional effort is needed to update the copies and collect the results.

Example 1.1

Suppose two processors want to access the same value, stored in variable *Y*, as follows. Processor 1 runs a part of a parallel program that sets the value of *Y*, and processor 2 runs a part of the parallel program that increments *Y*. The following code is shown graphically in Figure 1.2.

```
Processor 1                         Processor 2

Y = 100;
Send(Y) to B;                       Receive(X) FROM A;
Receive(Y) FROM B;                  X = X + 1;
Output(Y);                          Send(X) TO A;
```

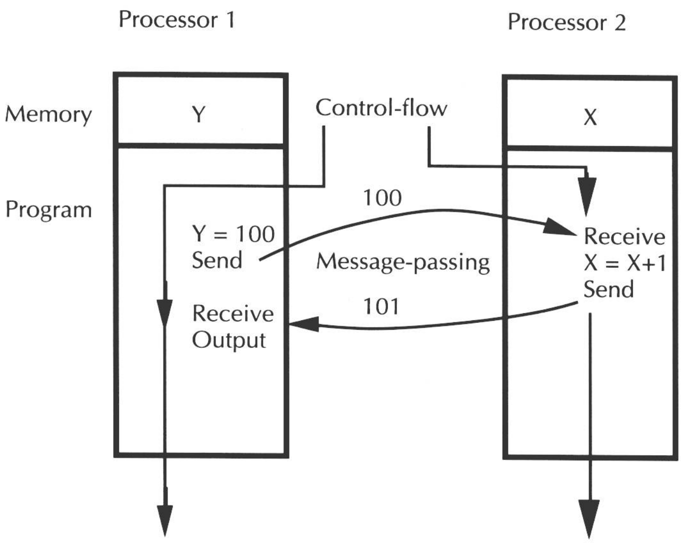


Figure 1.2 Synchronization in distributed-memory machines is achieved by message-passing.

The progress of each parallel part of the program is shown as a time-line in Figure 1.2. Both processors begin at the same time, but 2 immediately blocks on the Receive, waiting for a value to be received from 1. In the meantime, processor 1 sets *Y* to 100, and then places its value in a buffer so the operating system can send it across the interconnection network to processor 2's buffer. Processor 1 continues on to the next statement which is a Receive primitive. The Receive forces processor 1 to block, waiting for a returned value to fill its buffer. In the meantime, processor 2 receives 100, stores this value in its local variable *X*, increments *X*, and then sends 101 to processor 1.

When 101 is received by processor 1 and stored in variable *Y*, the processor unblocks and resumes. The Output function writes 101 from variable *Y*. Thus, the synchronization between these two processors is achieved by message-passing between the two local memories of the parallel computer.

A *shared-memory system* is one in which parallel parts of an application program are synchronized by setting and clearing *locks* on data stored in a centralized, shared memory space, or by synchronizing processes through programmer-created *barriers*. A lock prevents access to data unless certain conditions have been met, such as "only one process has access." A barrier forces any process to wait until all processes have arrived at the same point in the parallel program. Shared-memory MIMD machines share access instead of duplicating data.

A shared-memory MIMD machine like the Sequent Symmetry series, uses locks to coordinate access to shared data as illustrated by the following example.

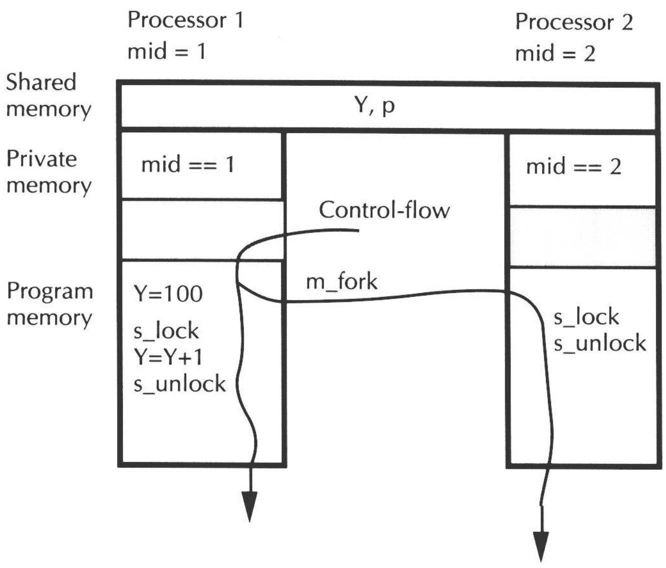


Figure 1.3 **Shared-memory locks synchronize access to shared data.**

Example 1.2

Suppose the previous distributed-memory example is repeated for a shared-memory machine. Once again, the problem is to simply increment the value stored in *Y*. But, this time, *Y* is a shared variable stored in shared memory. We use a simplified version of the Sequent DYNIX model to illustrate how this is done on most shared-memory MIMD machines.

```
shared int Y;                    /* Declare Y to be shared
shared lock p;                   /* Create a lock variable
•
•
•
Y = 100;                         /* Set Y
m_fork(2);                       /* Fork program into two programs
s_lock(p);                       /* Set lock so only one process can access
                                        critical section of code
if (mid == 1) {Y = Y + 1};       /* Processor 1 increments shared Y
s_unlock(p);                     /* Clear lock so all can access
output( Y );                     /* Output the result
•
•
•
```

In this version, the two processors (identified as mid = 1 and mid = 2) are activated by two copies of the same program. In addition, the data that are shared must be placed in a special shared partition of memory, see Figure 1.3. This is done by declaring *Y* to be a shared integer.

When the original program executes the m_fork(2) primitive, the operating system makes a copy of the program and starts running identical code on two processors. That is, the program is halted, and two copies instantiated. Both copies run in parallel, from that point on, as illustrated by the lines in Figure 1.3 that show the flow of control.

When each copy executes the s_lock primitive, the operating system forces one copy to wait while the other copy continues through the critical section of code. It is nondeterministic which processor reaches the s_lock primitive first, but assuming 1 is faster than 2, the lock is set by processor 1. Later, when processor 2 executes its copy of s_lock, it is blocked until shared lock *p* is cleared by processor 1.

Continuing, processor 1 is allowed to increment *Y*, because mid is equal to 1. The s_unlock primitive is executed next, causing processor 2 to become reactivated. But, mid is equal to 2 in processor 2, so the increment is skipped. The s_unlock clears *p*, leaving *p* unlocked.

The observant reader will also notice that both processes output *Y*, but the value of *Y* is different in each one!

Example 1.2 shows how a single copy of an application program can become a model for a parallel program consisting of many parallel parts. These parts must be coordinated by careful placement of locks. This is in stark contrast to message-passing. However, the reader should note that this is a software paradigm; it is possible to implement message-passing on a shared-memory computer. If we were to do so, the paradigm would shift from locks and barriers to sends and receives, regardless of the underlying hardware.

1.2 PARALLEL PROGRAMMING PARADIGMS

Regardless of the architecture of the target parallel computer, parallel programs must harmoniously coordinate two or more program segments to assure correctness as well as high speed. This is the challenge of parallel programming. Exactly how parallelism is controlled is largely determined by the particular *paradigm* used by the programmer and programming language designer. Thus, parallel programming reduces to the study of programming paradigms.

A *parallel program* is a collection of *processes* connected to one another through either message-passing or access to shared data. If the processes operate independently of one another, we call the parallel program *trivially parallel*. If they operate independently but with occasional pauses to coordinate among themselves, then the program must adopt one of two general styles to properly synchronize its parallel parts: *control-flow* programming or *data-parallel* programming (see Figure 1.4).

A control-flow program is one in which more than one thread of control is supported by the underlying hardware, and thus by the parallel program. This means that a single program can perform different operations in the same time interval. Control-flow parallelism is also used to indicate that the order in which (parallel) parts of a program execute is governed by program control rather than by the availability of data.