Data-Parallel Programming on MIMD Computers

Data-Parallel Programming on MIMD Computers

Philip J. Hatcher and Michael J. Quinn

The MIT Press
Cambridge, Massachusetts
London, England

Foreword

The world of modern computing potentially offers many helpful methods and tools to scientists and engineers, but the fast pace of change in computer hardware, software, and algorithms often makes practical use of the newest computing technology difficult. The Scientific and Engineering Computation series focuses on rapid advances in computing technologies and attempts to facilitate transferring these technologies to applications in science and engineering. It will include books on theories, methods, and original applications in such areas as parallelism, large-scale simulations, time-critical computing, computer-aided design and engineering, use of computers in manufacturing, visualization of scientific data, and human-machine interface technology.

The series will help scientists and engineers to understand the current world of advanced computation and to anticipate future developments that will impact their computing environments and open up new capabilities and modes of computation.

This book is the first in the series and describes data-parallel programming. In general, parallel computation has not yet been fully assimilated into the world of practical computer applications. Among the main reasons for this are limited portability of parallel software and scarcity of programming tools. Hatcher and Quinn suggest in this volume that the data-parallel model of computation offers programmers an approach that overcomes some of these difficulties. Data-parallel programming is easy to learn and can be used to solve many problems in science and engineering. Resulting codes can be ported to radically different computer architectures in the shared-memory and message-passing machine classes.

The book suggests a very likely future trend for a practical and economically justifiable mode of parallel computation.

Janusz S. Kowalik

Preface

MIMD computers are notoriously difficult to program. Typical MIMD programming languages are too low-level and lack portability. One solution is to introduce a high-level notation that simplifies programming, enhances portability, and provides compilers with enough information to allow them to generate efficient parallel code. This book illustrates how programs written in a high-level SIMD programming language may be compiled and efficiently executed on MIMD computers—both shared-memory multiprocessors and distributed-memory multicomputers. The language presented is Dataparallel C, a variant of the original C*™ language developed by Thinking Machines Corporation for its Connection Machine™ processor array. Separate chapters describe the compilation of Dataparallel C programs for execution on the Intel and nCUBE hypercube multicomputers and the Sequent multiprocessor. Later chapters document the performance of these compilers on a variety of benchmark programs and case studies.

We have designed this book to be suitable for several audiences. Some readers will use this book to learn more about high-level parallel programming languages. People who want to study the problem of compiling languages for distributed- or shared-memory parallel computers should also find this book helpful. Last, but not least, are those who will use this book as a reference manual for the Dataparallel C programming language. Real implementations of high-level portable parallel programming languages are still few and far between. We hope that our Dataparallel C compilers will stimulate research in the areas of parallel algorithms and programming languages.

We want to emphasize that this book is a "snapshot" of the state of our compilers in April 1991. We have seen significant improvements in the performance of many of our compiled programs over the past several months, and we anticipate further gains, as we continue to implement compiler optimizations.

The results we report in this book are largely due to the efforts of many graduate students at the University of New Hampshire and Oregon State University. Charles A. Grasso implemented the first generic host (`ghost`) program for the nCUBE™ 3200. Jeffrey E. F. Friedl modified the University of Virginia's Very Portable C Compiler to generate code for the nCUBE node processors, developed the valuable UNIX™-to-nCUBE and nCUBE-to-UNIX binary file conversion programs, and implemented the second `ghost` program for the nCUBE. Karen C. Jourdenais designed and implemented our first Dataparallel C compiler for the nCUBE. Lutz H. Hamel built the second-generation Dataparallel C compiler for the nCUBE and ported the GNU C compiler to generate code for the nCUBE node processors. Robert R. Jones built the front end

C* is a registered trademark of Thinking Machines Corporation.
Connection Machine is a registered trademark of Thinking Machines Corporation.
nCUBE is a trademark of nCUBE Corporation.
UNIX is a registered trademark of AT&T Bell Laboratories.

of the third-generation multicomputer Dataparallel C compiler. Anthony J. Lapadula wrote the back end and optimizer for the third-generation multicomputer Dataparallel C compiler. Bradley K. Seevers designed, implemented, and benchmarked the Dataparallel C compiler for the Sequent multiprocessor family, and he wrote part of Section 4.5. Ray J. Anderson ported Friedl's tool set to the Sun™-hosted nCUBE system, implemented the routing and parallel I/O libraries for the nCUBE and Intel multicomputers, and helped test both the multiprocessor and the multicomputer compilers by programming a variety of case studies. Margaret M. Cawley tested the multicomputer Dataparallel C compiler by implementing several case studies. David Judge implemented the first Dataparallel C version of the shallow-water model. We thank these students for their splendid efforts.

We are pleased to have been able to collaborate with Andrew Bennett, Professor of Oceanography at Oregon State University. Andrew developed the model used as the basis of the ocean case study of Chapter 7, and he wrote parts of Sections 7.2 and 7.3.

We appreciate the careful proofreading and copy editing performed by Darcy J. McCallum and Jenya Weinreb.

We would like to thank Bob Prior of The MIT Press, who encouraged us to write this book, supported our efforts in a variety of important ways, and never gave us a hard time when we missed our original deadline by almost a year. Thanks, Bob. We hope this book is worth the wait.

We thank our families for their unconditional support: Peggy, Christina, and John; Vicki, Shauna, Brandon, and Courtney.

Finally, we are grateful to the organizations that supported this research: the National Science Foundation, the Defense Advanced Research Projects Agency, the Oregon Advanced Computing Institute, Oregon State University, the University of New Hampshire, and Intel Corporation. The Department of Computer Sciences at the University of Wisconsin-Madison and the Advanced Computing Research Facility of the Mathematics and Computer Science Division at Argonne National Laboratory gave us free access to their Sequent Symmetry™ multiprocessors.

Sun is a registered trademark of Sun Microsystems, Inc.
Symmetry is a trademark of Sequent Computer Systems, Inc.

Contents

Chapter 1
Introduction

Since 1985 vendors have announced a large number of multiple-CPU computers. Some of these computers contain thousands of processors, and many systems can perform hundreds of millions of floating-point operations per second. Unfortunately, the accompanying programming languages are not nearly as glamorous. The typical commercial programming language is little more than a bag of parallel constructs hung on the side of an existing sequential language, such as C or FORTRAN. These low-level parallel languages lead to programs that are difficult to design, implement, debug, and maintain.

The research community has recognized the need for better parallel programming languages and has proposed dozens of alternatives. An examination of these languages reveals that they represent virtually every possible answer to the fundamental design questions. Should the parallelism be implicit or explicit? Should the language be imperative, functional, or based on logic programming? Should the processes execute synchronously or asynchronously? Should the level of parallelism be fixed at compile time, be chosen at run time, or be dynamic? Should the programmer view memory as distributed or shared?

We anticipate that there will be a variety of successful higher-level parallel programming languages available to programmers of MIMD computers in the next decade. Our work has focused on *data-parallel programming languages*, languages in which you express parallelism through the simultaneous application of a single operation to a data set. We have three reasons to believe that data-parallel languages will assume an important role in the future of parallel computing: you can solve a significant number of problems using data-parallel algorithms, it is easier to write data-parallel programs for these problems than programs written using lower-level parallel constructs, and compilers can translate data-parallel programs into efficient code.

This book describes the implementation of two compilers for the data-parallel programming language Dataparallel C. The target machines are from two radically different MIMD architecture classes: distributed-memory multicomputers and shared-memory multiprocessors. We have claimed that compilers can translate data-parallel programs into code that executes efficiently on MIMD architectures. We validate our claim by benchmarking the performance of the code produced by our two compilers on

a wide variety of programs, which we execute on the Intel iPSC™/2 and nCUBE 3200 multicomputers and the Sequent Symmetry multiprocessor.

Few high-level parallel programming environments are available to those who want to solve problems on parallel computers or design new parallel algorithms. We hope that these compilers, which can produce code for workstations, multiprocessors, and multicomputers, will stimulate further research in parallel computing.

In this chapter we define a variety of parallel computing terms, present the target architectures, and contrast various approaches to programming MIMD computers. We describe what we mean by "data-parallel algorithm," and we contrast data-parallel and control-parallel approaches to the parallelization of the classic prime-finding algorithm, the Sieve of Eratosthenes. We end with a survey of related work.

1.1 Terminology

The parallel computer terminology we use in this book is fairly standard; you can find more detailed explanations of the terms in Quinn, 1987. A *multiprocessor* is a shared-memory multiple-CPU computer designed for parallel processing. In a *tightly coupled multiprocessor* all the processors work through a central switching mechanism to reach a shared global memory. Some people call this a uniform memory access (UMA) multiprocessor. A *multicomputer* is a multiple-CPU computer designed for parallel processing, but lacking a shared memory. All communication and synchronization between processors must take place through message passing.

Flynn's taxonomy of computer architecture is the basis for a variety of programmer models of parallel computation (Flynn, 1966). A programmer can view a *SIMD* (single instruction stream, multiple data stream) computer as a single CPU directing the activities of a number of arithmetic processing units, each capable of fetching and manipulating its own local data. Another common name for this model is *processor array*. Since the processing units work in lock step under the control of a single CPU, we call this programming model *synchronous*. SIMD models can vary in two respects. First, the number of processing elements may be fixed or unbounded. Second, the way in which processing elements interact can vary. For example, a mesh-connected model organizes the processing elements into a mesh; processors may only fetch data from their immediate neighbors. On the other hand, in a global name space model, the processing elements may directly access the memories of the other processing elements.

A *data-parallel model* of parallel computation is a SIMD model with an unbounded number of processing elements and a global name space.

A *MIMD* (multiple instruction stream, multiple data stream) computer allows the concurrent execution of multiple instruction streams, each manipulating its own data. A

iPSC is a registered trademark of Intel Corporation.

MIMD programming language must include some communication and synchronization primitives in order for the processes corresponding to the various instruction streams to coordinate their efforts. It is possible for every processor in a MIMD computer to execute a unique program. However, it is far more common for every processor to execute the same program. *SPMD* (single program, multiple data stream) programming puts the same program on every processor (Karp, 1987). Although you can expect processors to coordinate with each other at synchronization points, we call the MIMD and SPMD programming models *asynchronous*, because between the synchronization points every processor executes instructions at its own pace.

Speedup is the ratio between the time needed for the most efficient sequential program to perform a computation and the time needed for a parallel program to perform the same computation, where the sequential program executes on a single processor of a parallel computer and the parallel program executes on one or more processors of the same parallel computer. *Scaled speedup* is the ratio between how long a given optimal sequential program would have taken, had it been able to run on a single processor of a parallel computer, and the length of time that the parallel program requires, when executing on a multiple processors of the same parallel computer (Gustafson *et al.*, 1988).

The difference between the two speedup definitions is subtle, yet important. In order to measure speedup, the algorithm must run on a single processor. On a multicomputer, that means the problem data must fit in the memory of that one processor. Far larger problems can be, and usually are, solved by systems with hundreds or thousands of processors, but the restriction that the problem be solvable by a single processor means that these large problems cannot be used when determining the speedup achieved by the parallel machine. The definition of scaled speedup allows the solution of these realistic, large problems on a multicomputer and the estimation of the execution time that would have been required if the same problems had been solved on a single processor with a massive primary memory.

The *Amdahl effect* is the observation that for any fixed number of processors, speedup is usually an increasing function of problem size. Because the definition of scaled speedup allows you to apply the parallel computer to larger problems, the scaled speedup achieved by a particular program is usually larger than the program's speedup. For example, three scientific codes implemented by Gustafson *et al.* achieved speedups of 502 to 637 on a 1024–processor nCUBE 3200, while the scaled speedups achieved by these algorithms ranged from 1009 to 1020 (Gustafson *et al.*, 1988).

Our definition of speedup requires that you compare the execution time of the parallel program with the execution time of the best sequential program. Sometimes you can easily determine the best sequential program, but often it is not clear which sequential algorithm is fastest for a particular domain. Another measure of the performance of a parallel program indicates the reduction in execution time achieved as processors are

added. *Parallelizability* is the ratio between the execution time of a parallel program on one processor and its execution time on multiple processors. Many so-called "speedup curves" that appear in the literature are actually illustrations of the parallelizability of the parallel program.

Of course, the purpose of parallel computers is to reduce the time needed to solve particular problems, and given the variety of analyses appearing under the single name "speedup," the least controversial measure of the performance of a parallel program may well be its speed, not its speedup. For this reason we include as part of our performance data later in the book the execution times of the compiled Dataparallel C programs.

The *efficiency* of a parallel program is its speedup divided by the number of processors used. For example, a parallel program that achieves a speedup of 32 on 64 processors exhibits an efficiency of 50%.

Cost is a measure of the total number of operations performed by a sequential or parallel algorithm. We define the cost of an algorithm to be the product of its complexity and the number of processors used. For example, the cost of a sequential binary search algorithm is $\Theta(\log n)$, where n is the length of an ordered list. Imagine a parallelization of binary search for a multiprocessor that gives each of p processors n/p contiguous list elements. Ignoring process creation overhead, the worst-case time complexity of the parallel algorithm is $\Theta(\log(n/p))$; the cost of the parallel algorithm is $\Theta(p \log(n/p))$, which means that the parallel algorithm performs more operations than the sequential algorithm when $p > 1$. If the cost of a parallel algorithm is greater than the cost (i.e., complexity) of the best sequential algorithm, then the parallel algorithm cannot maintain high efficiency as the number of processors increases.

A *barrier synchronization* is a point in a program beyond which no process may proceed until all processes have arrived. The *grain size* of a program is the relative number of instructions performed per barrier synchronization.

1.2 Three Illustrative MIMD Computers

Sequent Symmetry

The Symmetry S81, manufactured by Sequent Computer Systems, Inc., is a tightly coupled multiprocessor that can include up to 30 Intel 80386™ CPUs and between 8 and 240 megabytes of primary memory (Figure 1.1). In systems that use a bus as the central switching mechanism, bus contention has traditionally been the primary factor limiting the number of CPUs that can be utilized. The Symmetry architecture addresses this problem in three ways. First, the bus is 64 bits wide and is able to achieve a sustained transfer rate of 53.3 Mbytes per second. Second, each processor has its

Intel 386 is a registered trademark of Intel Corporation.

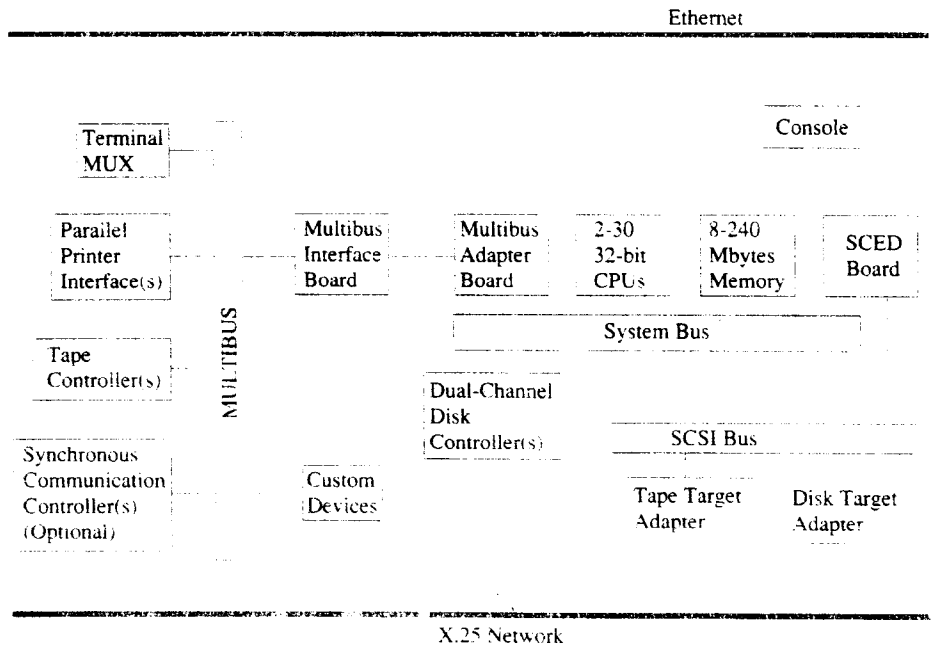

Figure 1.1. Architecture of the Sequent Symmetry S81 multiprocessor.

own 64–Kbyte, two-way set-associative cache memory to reduce traffic on the system bus. Custom VLSI logic ensures cache consistency without requiring a write through operation to main memory every time shared data is modified. Third, every CPU has an associated System Link and Interrupt Controller (SLIC) to "manage system initialization, interprocessor communication, distribution of interrupts among CPUs, and diagnostics and configuration control" (Sequent, 1987). These SLIC chips are connected with a separate bit-serial data path called the SLIC bus. Still, Symmetry computers have at most 30 CPUs, a relatively low ceiling.

nCUBE 3200

The nCUBE 3200 (originally called the nCUBE/ten) was the first commercial MIMD computer offered with more than 1,000 processors. A fully-configured nCUBE 3200 contains 1,024 custom 32–bit processors, each controlling 512 Kbytes of local memory. These node processors are arranged as a ten-dimensional hypercube (see Figure 1.2). Hypercube links represent the paths along which messages between nodes may travel. An Intel 80286™ host processor serves as a front-end computer, managing the hypercube

286 is a registered trademark of Intel Corporation.

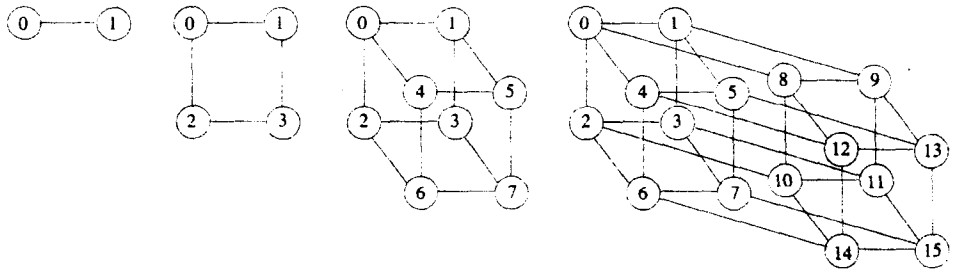

Figure 1.2. One-, two-, three-, and four-dimensional hypercubes. A k-dimensional hypercube has 2^k nodes, labeled $0 \ldots 2^k - 1$; two nodes are adjacent if their labels differ in exactly one bit position. Many multicomputers, including the nCUBE 3200 and the Intel iPSC/2, use the hypercube processor organization.

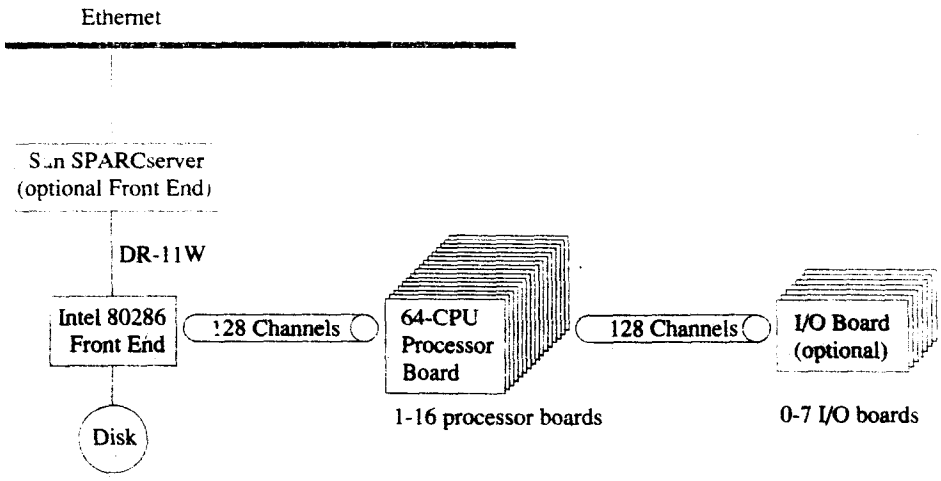

Figure 1.3. Architecture of the nCUBE 3200 multicomputer.

of processors as well as the I/O devices. An option later offered by nCUBE relegates the host processor to the role of intermediary between the nodes and a Sun SPARCserver, which takes over the role as front-end processor (Figure 1.3).

The existence of a front end relegates the node processors to the status of being a computational back end. This distinguishes the nCUBE 3200 from the Sequent Symmetry, in which every processor has direct access to the I/O devices. Multicomputers do not necessarily have a front end, although virtually all first-generation commercial multicomputers, including the Intel iPSC and Ametek S/14, make use of a host processor. One explanation for this may be that these machines trace their ancestry to a common source, Caltech's Cosmic Cube (Seitz, 1985).

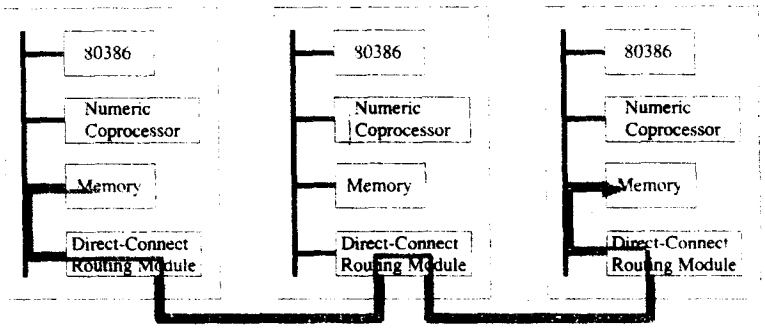

Figure 1.4. When a message is passed between nonadjacent nodes on the Intel iPSC/2, the Direct-Connect Modules along the path between the nodes establish a circuit, which allows the message to be sent from the source to the destination without being stored and forwarded at the intermediate nodes. The CPUs of the intermediate nodes are not interrupted.

The nCUBE 3200 uses store-and-forward message routing. If a node processor sends a message to a nonadjacent node processor, each intermediate processor stores the entire message before forwarding it to the next processor along the message's path.

Intel iPSC/2

The Intel iPSC/2 is a second-generation multicomputer, but in many ways the architecture resembles that of the nCUBE 3200. The node processors are organized in a hypercube topology. This back end may contain up to 128 processors. A System Resource Manager serves as the front end, connecting the back end with the outside world via Ethernet. The System Resource Manager is responsible for allocating and deallocating back-end processors and loading the programs that execute on the back end.

The most important characteristic that distinguishes second-generation multicomputers, such as the Intel iPSC/2, from first-generation multicomputers, such as the nCUBE 3200, is the elimination of store-and-forward message routing. In addition to an Intel 80386 CPU, every iPSC/2 node contains a routing logic daughter card called the Direct-Connect Module™. The Direct-Connect Modules set up a circuit from the source node to the destination node. Once the circuit is set up, the message flows in a pipelined fashion from the source node to the destination node—none of the intermediate nodes store the message. A message being passed from one node to a nonadjacent node does not interrupt the CPUs of the intermediate nodes; only the Direct-Connect modules are involved (Figure 1.4).

Direct-Connect Module is a trademark of Intel Corporation.

1.3 Parallel Programming Languages

Programming parallel computers is widely held to be more difficult than programming sequential computers, but much of the blame can be traced to the programming languages used. Chen has pinpointed a central difficulty in programming parallel computers: How can one reason about a parallel program that embodies concurrent and distributed state changes among a large number of processes (Chen, 1987)? If programmers cannot reason about the behavior of their programs, how can they be expected to produce correct, maintainable code?

In this section we examine a variety of ways proposed to program multiprocessors and multicomputers using imperative programming languages. First we consider the alternative of programming a parallel computer in a conventional sequential language. Next we focus our attention on the parallel C languages provided by Sequent Computer Systems and nCUBE for their respective machines. Finally, we consider the advantages of higher-level parallel programming languages. In each case we consider the model of computation presented to the programmer and two other important attributes: how efficiently the translated program can be made to run on the underlying machine, and the portability of the parallel program to different architectures. If a parallel programming language is to be widely adopted, compiled programs must take good advantage of the resources provided by the target machine. Portability is a particularly important attribute of a parallel programming language, because programs are a valuable commodity that cannot be discarded casually.

Conventional Programming Languages

One solution to the problem of finding a suitable language for programming parallel computers is to stick with an existing imperative programming language, such as FORTRAN or C, and let a parallelizing compiler detect and exploit the parallelism in the program. Conventional programming languages present the programmer with a straightforward, understandable model of computation based on the single instruction stream, single data stream von Neumann computer. No retraining of programmers is needed, and the huge amount of existing software, those legendary "dusty decks" of FORTRAN cards, can be kept.

However, the use of a sequential language pits the programmer against the compiler in a game of hide and seek. The algorithm may have a certain amount of inherent parallelism. The programmer hides the parallelism in a sea of DO loops and other control structures, and then the compiler must seek it out. Because the programmer may have to specify unneeded serializations when writing programs in a conventional imperative language, some parallelism may be irretrievably lost. Explicit parallelism is invaluable when trying to execute programs efficiently on parallel hardware. The introduction of

parallel and/or vector operations into the proposed new FORTRAN standards signifies an acknowledgment by the user community of this principle.

Languages with Low-Level Parallel Constructs

It is not surprising that parallel programming languages with low-level parallel constructs are widespread, given the history of parallel computers. From the days of the first Cray-1, which was delivered without a compiler, to modern parallel computers, the development of innovative hardware has kept ahead of the development of equally sophisticated software. A conventional programming language enhanced with a few constructs allowing the user to create and synchronize parallel processes is the simplest avenue to take, since it puts the least burden on the compiler writer.

Multiprocessor Programming Languages A multiprocessor programming language must have constructs to spawn and terminate parallel processes, manage synchronization between processes, and distinguish between private and shared data.

In Sequent Parallel C, for example, the `m_fork` function forks off a set of parallel processes to execute a function. The processes suspend execution when they reach the end of the called function. Other functions allow for mutual exclusion of processes in critical sections, and the keyword `shared` allows the user to designate global data accessible by all processes.

These low-level constructs can make programs very difficult to debug. Even programs a few dozen lines long can yield numerous, troublesome bugs (Allan and Oldehoeft, 1985; McGraw and Axelrod, 1988; Storc, 1988). It is hard to eliminate timing errors (McGraw and Axelrod, 1988). A general lack of debugging tools often forces programmers to return to paper, pencil, program listings, and hand tracing (McGraw and Axelrod, 1988).

These programming languages are often less elegant than those languages developed in the 1970s for the purpose of implementing multiprogrammed operating systems, including Concurrent Pascal, Modula, and Pascal Plus. Pascal Plus, for example, has the following facilities for parallel programming:

1. the process, which identifies the parts of a program that may execute in parallel;
2. the monitor structure, which guarantees mutual exclusion of processes accessing shared data; and
3. the condition, which allows synchronization of processes.

However, after implementing the system software for C.mmp, Wulf *et al.* concluded that even these constructs such as monitors may not help (Wulf *et al.*, 1981). Managing parallelism and synchronization explicitly is a time-consuming and error-prone activity.