NEAR EAST UNIVERSITY

Faculty of Engineering
Department of Computer Engineering

PARALLEL PROGRAMMING

GRADUATION PROJECT
COM 400

GÜNEŞ ÖZKAYA
940846
COM. ENG.

SUPERVISOR:
MISS BESİME ERİN

ACKNOWLEDGEMENT

I would like to thank Miss. Besime Erin for accepting to be my supervisor and her support for this project.

I am so grateful to my parents who had always shown patience and understanding to me.

Also, I would like to tkank all the lecturers for helping me see this graduation term.

And finally, I would like to thank all my friends for their support in school and in social life.

Güneş Özkaya

ABSTRACT

Ever since conventional serial computers were invented, their speed has steadily increased to match the needs of emerging applications. However, the fundamental physical limitation imposed by the speed of light makes it impossible to achieve further improvements in the speed of such computers indefinitely. Recent trends show that the performance of these computers is beginning to saturate. A natural way to circumvent this saturation is to use an ensemble of processors to solve problems.

The transition point has become sharper with the passage of time, primarily as a result of advances in very large scale integration (VLSI) technology. It is now possible to construct very fast, low-cost processors. This increases the demand for and production of these processors, resulting in lower prices.

Currently, the speed of off-the-shelf microprocessors is within one order of magnitude of the speed of the fastest serial computers. However, microprocessors cost many orders of magnitude less. This implies that, by connecting only a few microprocessors together to form a parallel computer, it is possible to obtain raw computing power comparable to that of the fastest serial computers. Typically, the cost of such a parallel computer is considerably less.

Furthermore, connecting a large number of processors into a parallel computer overcomes the saturation point of the computation rates achievable by serial computers. Thus, parallel computers can provide much higher raw computation rates than the fastest serial computers as long as this power can be translated into high computation rates for actual applications.

TABLE OF CONTENTS

INTRODUCTION

The technological driving force behind parallel computing is VLSI, or very large scale integration-the same technology that created the personal computer and workstation market over the last decade. In 1980, the Intel 8086 used 50,000 transistors; in 1992, the latest Digital alpha RISC chip contains 1,680,000 transistors-a factor of 30 increase. The dramatic improvement in chip density comes together with an increase in clock speed and improved design so that the alpha performs better by a factor of over one thousand on scientific problems than the 8086-8087 chip pair of the early 1980s.

High-performance computers are increasingly in demand in the areas of structural analysis, weather forecasting, petroleum exploration, medical diagnosis, aerodynamics simulation, artificial intelligence, expert systems, genetic engineering, signal and image processing, among many other scientific and engineering applications. Without superpower computers, many of these challenges to advance human civilization cannot be made within a reasonable time period. Achieving high performance depends not only on using faster and more reliable hardware devices but also on major improvements in computer architecture and processing techniques.

There are a number of different ways to characterize the performance of both parallel computers and parallel algorithms. Usually, the peak performance of a machine is expressed in units of millions of instructions executed per second (MIPS) or millions of floating point operations executed per second (MFLOPS). However, in practice, the realizable performance is clearly a function of the match between the algorithms and the architecture.

CHAPTER 1

1 What is Parallel Computing?

Consider the problem of stacking (reshelving) a set of library books A single worker trying to stack all the books in their proper places cannot accomplish the task faster than a certain rate. We can speed up this process, however, by employing more than one worker. Assume that the books are organized into shelves and that the shelves are grouped into bays. One simple way to assign the task to the workers is to divide the books equally among them. Each worker stacks the books one at a time. This division of work may not be the most efficient way to accomplish the task, since the workers must walk all over the library to stack books. An alternate way to divide the work is to assign a fixed and disjoint set of bays to each worker. As before, each worker is assigned an equal number of books arbitrarily. If a worker finds a book that belongs to a bay assigned to him or her, he or she places that book in its assigned spot. Otherwise, he or she passes it on to the responsible for the bay it belongs to. The second approach requires less effort from individual workers.

The preceding example shows how a task can be accomplished faster by dividing it into a set of subtasks assigned to multiple workers. Workers cooperate, pass the books to each other when necessary, and accomplish the task in unison. Parallel processing works on precisely the same principles. Dividing a task among workers by assigning them a set of books is an instance of task partitioning. Passing books to each other is an example of communication between subtasks.

Problems are parallelizable to different degrees. For some problems, assigning partitions to other processors might be more time-consuming than performing the processing locally. Other problems may be completely serial. For example, consider the task of digging a post hole. Although one person can dig a hole in a certain amount of time, employing more peop'e does not reduce this time. Because it is impossible to partition this task, it is poorly suited to parallel processing. Therefore, a problem may have different parallel formulations, which result in varying benefits, and all problems are not equally amenable to parallel processing.

2 The Scope of Parallel Computing

Parallel processing is making a tremendous impact on many areas of computer application. With the high raw computing power of parallel computers, it is now possible to address many applications that were until recently beyond the capability of conventional computing techniques.

Many applications, such as weather prediction, biosphere modeling, and pollution monitoring, are modeled by imposing a grid over the domain being modeled. The entities within grid elements are simulated with respect to the influence of other entities and their surroundings. In many cases, this requires solutions to large systems of differential equations. The granularity of the grid determines the accuracy of the model. Since many such systems are evolving with time, time forms an additional dimension for these computations. Even for a small number of grid points, a three-dimensional coordinate system, and a reasonable discretized time step, this modeling process can involve trillions of operations. Thus, even moderate-sized instances of these problems take an unacceptably long time to solve on serial computers.

Parallel processing makes it possible to predict the weather not only faster but also more accurately. If we have a parallel computer with a thousand workstation-class processors, we can partition the 10^{11} segments of the domain among these processors. Each processor computes the parameters for 10^{8} segments. Processors communicate the value of the parameters in their segments to other processors. Assuming that the computing power of this computer is 100 million instructions per second, and this power is efficiently utilized, the problem can be solved in less than 3 hours. The impact of this reduction in processing time is two-fold. First, parallel computers make it possible to solve a previously unsolvable problem. Second, with the availability of even larger parallel computers, it is possible to model weather using finer grids. This enables more accurate weather prediction.

The acquisition and processing of large amounts of data from sources such as satellites and oil wells form another class of computationally expensive problems. Conventional satellites collect billions of bits per second of data relating to parameters such as pollution levels, the thickness of the ozone layer, and weather phenomena. Other applications of satellites that require processing of large amounts of data include remote sensing and telemetry. The computational rates required for handling this data effectively are well beyond the range of conventional serial computers.

Discrete optimization problems include such computationally intensive problems as planning, scheduling, VLSI design, logistics, and control. Discrete optimization problems can be solved by using state-space search techniques. For many of these problems, the size of the state-space increases exponentially with the number of variables. Problems that evaluate trillions of states are fairly commonplace in most such applications. Since processing each state requires a nontrivial amount of computation, finding solutions to large instances of these problems is beyond the scope of conventional sequential computing. Indeed, many practical problems are solved using approximate algorithms that provide suboptimal solutions.

Other applications that can benefit significantly from parallel computing are semi-conductor material modeling, ocean modeling, computer tomography, quantum chromodynamics, vehicle design and dynamics, analysis of protein structures, study of chemical phenomena, imaging, ozone layer monitoring, petroleum exploration, natural language understanding, speech recognition, neural network learning, machine vision, database query processing, and automated discovery of concepts and patterns from large databases. Many of the applications mentioned are considered grand challenge problems. A grand challenge is a fundamental problem in science or engineering that has a broad economic and scientific impact, and whose solution could be advanced by applying high performance computing techniques and resources.

3 Issues in Parallel Computing

To use parallel computing effectively, we need to examine the following issues:

3.1 Design of Parallel Computers

It is important to design parallel computers that can scale up to a large number of processors and are capable of supporting fast communication and data sharing among processors. This is one aspect of parallel computing that has seen the most advances and is the most mature.

3.2 Design of Efficient Algorithms

A parallel computer is of little use unless efficient parallel algorithms are available. The issues in designing parallel algorithms are very different from those in designing their sequential

counterparts. A significant amount of work is being done to develop efficient parallel algorithms for a variety of parallel architectures.

3.3 Methods for Evaluating Parallel Algorithms

Given a parallel computer and a parallel algorithm running on it, we need to evaluate the performance of the resulting system. Performance analysis allows us to answer questions such as How fast can a problem be solved using parallel processing? and How efficiently are the processors used?

3.4 Parallel Computer Languages

Parallel algorithms are implemented on parallel computers using a programming language. This language must be flexible enough to allow efficient implementation and must be easy to program in. New languages and programming paradigms are being developed that try to achieve these goals.

3.5 Parallel Programming Tools

To facilitate the programming of parallel computers, it is important to develop comprehensive programming environments and tools. These must serve the dual purpose of shielding users from low-level machine characteristics and providing them with design and development tools such as debuggers and simulators.

3.6 Portable Parallel Programs

Portability is one of the main problems with current parallel computers. Typically, a program written for one parallel computer requires extensive modification to make it run on another parallel computer. This is an important issue that is receiving considerable attention.

3.7 Automatic Programming of Parallel Computers

Much work is being done on the design of parallelizing compilers, which extract implicit parallelism from programs that have not been parallelized explicitly. Such compilers are expected

to allow us to program a parallel computer like a serial computer. We speculate that this approach has limited potential for exploiting the power of large-scale parallel computers.

CHAPTER 2

1 Parallelism and Computing

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. Parallel computers are interesting because they offer the potential to concentrate computational resources-whether processors, memory, or I/O bandwidth-on important computational problems.

Parallelism has sometimes been viewed as a rare and exotic subarea of computing, interesting but of little relevance to the average programmer. A study of trends in applications, computer architecture, and networking shows that this view is no longer tenable. Parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise.

2 The National Vision for Parallel Computation

The technological driving force behind parallel computing is VLSI, or very large scale integration-the same technology that created the personal computer and workstation market over the last decade. In 1980, the Intel 8086 used 50,000 transistors; in 1992, the latest Digital alpha RISC chip contains 1,680,000 transistors-a factor of 30 increase. The dramatic improvement in chip density comes together with an increase in clock speed and improved design so that the alpha performs better by a factor of over one thousand on scientific problems than the 8086-8087 chip pair of the early 1980s.

The increasing density of transistors on a chip follows directly from a decreasing feature size which is now for the alpha. Feature size will continue to decrease and by the year 2000, chips with 50 million transistors are expected to be available. What can we do with all these transistors?

With around a million transistors on a chip, designers were able to move full mainframe functionality to about of a chip. This enabled the personal computing and workstation revolutions. The next factors of ten increase in transistor density must go into some form of parallelism by replicating several CPUs on a single chip.

By the year 2000, parallelism is thus inevitable to all computers, from your children's video game to personal computers, workstations, and supercomputers. Today we see it in the larger machines as we replicate many chips and printed circuit boards to build systems as arrays of nodes, each unit of which is some variant of the microprocessor. An nCUBE parallel supercomputer with 64 identical nodes on each board-each node is a single-chip CPU with additional memory chips. To be useful, these nodes must be linked in some way and this is still a matter of much research and experimentation. Further, we can argue as to the most appropriate node to replicate; is it a ``small" node as in the nCUBE, or more powerful ``fat" nodes such as those offered in CM-5 and Intel Touchstone, where each node is a sophisticated multichip printed circuit board. However, these details should not obscure the basic point: Parallelism allows one to build the world's fastest and most cost-effective supercomputers.

Parallelism may only be critical today for supercomputer vendors and users. By the year 2000, all computers will have to address the hardware, algorithmic, and software issues implied by parallelism. The reward will be amazing performance and the opening up of new fields; the price will be a major rethinking and reimplementation of software, algorithms, and applications. This vision and its consequent issues are now well understood and generally agreed. They provided the motivation in 1981 when CP's first roots were formed. In those days, the vision was blurred and controversial. Many believed that parallel computing would not work.

President Bush instituted, in 1992, the five-year federal High Performance Computing and Communications (HPCC) Program. The activities of several federal agencies have been coordinated in this program. The Advanced Research Projects Agency (ARPA) is developing the basic technologies which is applied to the grand challenges by the Department of Energy (DOE), the National Aeronautics and Space Agency (NASA), the National Science Foundation (NSF), the National Institute of Health (NIH), the Environmental Protection Agency (EPA), and the National Oceanographic and Atmospheric Agency (NOAA). Selected activities include the mapping of the human genome in DOE, climate modeling in DOE and NOAA, coupled structural and airflow simulations of advanced powered lift and a high-speed civil transport by NASA.

More generally, it is clear that parallel computing can only realize its full potential and be commercially successful if it is accepted in the real world of industry and government applications. The clear U.S. leadership over Europe and Japan in high-performance computing offers the rest of the U.S. industry the opportunity of gaining global competitive advantage.

We note some interesting possibilities which include: use in the oil industry for both seismic analysis of new oil fields and the reservoir simulation of existing fields; environmental modeling of past and potential pollution in air and ground; fluid flow simulations of aircraft, and general vehicles, engines, air-conditioners, and other turbomachinery; integration of structural analysis with the computational fluid dynamics of airflow; car crash simulation; integrated design and manufacturing systems; design of new drugs for the pharmaceutical industry by modeling new compounds; simulation of electromagnetic and network properties of electronic systems-from new components to full printed circuit boards; identification of new materials with interesting properties such as superconductivity; simulation of electrical and gas distribution systems to optimize production and response to failures; production of animated films and educational and entertainment uses such as simulation of virtual worlds in theme parks and other virtual reality applications; support of geographic information systems including real-time analysis of data from satellite sensors in NASA's ``Mission to Planet Earth."

A relatively unexplored area is known as ``command and control" in the military area and ``decision support" or ``information processing" in the civilian applications. These combine large databases with extensive computation. In the military, the database could be sensor information and the processing a multitrack Kalman filter. Commercially, the database could be the nation's medicaid records and the processing would aim at cost containment by identifying anomalies and inconsistencies.

Servers in multimedia networks set up by cable and telecommunication companies. These servers will provide video, information, and simulation on demand to home, education, and industrial users. CP did not address such large-scale problems. Rather, we concentrated on major academic applications. This fit the experience of the Caltech faculty who led most of the CP teams, and further academic applications are smaller and cleaner than large-scale industrial problems. One important large-scale CP application was a military simulation and produced by Caltech's Jet Propulsion Laboratory. CP chose the correct and only computations on which to cut its parallel computing teeth. In spite of the focus on different applications, there are many

similarities between the vision and structure of CP and today's national effort. It may even be that today's grand challenge teams can learn from CP's experience.

3 Trends in Applications

As computers become ever faster, it can be tempting to suppose that they will eventually become ``fast enough'' and that appetite for increased computing power will be sated. However, history suggests that as a particular technology satisfies known applications, new applications will arise that are enabled by that technology and that will demand the development of new technology. As an amusing illustration of this phenomenon, a report prepared for the British government in the late 1940s concluded that Great Britain's computational requirements could be met by two or perhaps three computers. In those days, computers were used primarily for computing ballistics tables. The authors of the report did not consider other applications in science and engineering, let alone the commercial applications that would soon come to dominate computing. Similarly, the initial prospectus for Cray Research predicted a market for ten supercomputers; many hundreds have since been sold.

Traditionally, developments at the high end of computing have been motivated by numerical simulations of complex systems such as weather, climate, mechanical devices, electronic circuits, manufacturing processes, and chemical reactions. However, the most significant forces driving the development of faster computers today are emerging commercial applications that require a computer to be able to process large amounts of data in sophisticated ways. These applications include video conferencing, collaborative work environments, computer-aided diagnosis in medicine, parallel databases used for decision support, and advanced graphics and virtual reality, particularly in the entertainment industry. For example, the integration of parallel computation, high-performance networking, and multimedia technologies is leading to the development of video servers, computers designed to serve hundreds or thousands of simultaneous requests for real-time video. Each video stream can involve both data transfer rates of many megabytes per second and large amounts of processing for encoding and decoding. In graphics, three-dimensional data sets are now approaching volume elements (1024 on a side). At 200 operations per element, a display updated 30 times per second requires a computer capable of 6.4 operations per second.

Although commercial applications may define the architecture of most future parallel computers, traditional scientific applications will remain important users of parallel computing technology. Indeed, as nonlinear effects place limits on the insights offered by purely theoretical investigations and as experimentation becomes more costly or impractical, computational studies of complex systems are becoming ever more important. Computational costs typically increase as the fourth power or more of the "resolution" that determines accuracy, so these studies have a seemingly insatiable demand for more computer power. They are also often characterized by large memory and input/output requirements. For example, a ten-year simulation of the earth's climate using a state-of-the-art model may involve floating-point operations, ten days at an execution speed of floating-point operations per second (10 gigaflops). This same simulation can easily generate a hundred gigabytes (bytes) or more of data. Yet scientists can easily imagine refinements to these models that would increase these computational requirements 10,000 times.

In summary, the need for faster computers is driven by the demands of both data-intensive applications in commerce and computation-intensive applications in science and engineering. Increasingly, the requirements of these fields are merging, as scientific and engineering applications become more data intensive and commercial applications perform more sophisticated computations.

4 Trends in Computer Design

The performance of the fastest computers has grown exponentially from 1945 to the present, averaging a factor of 10 every five years. While the first computers performed a few tens of floating-point operations per second, the parallel computers of the mid-1990s achieve tens of billions of operations per second. Similar trends can be observed in the low-end computers of different eras: the calculators, personal computers, and workstations. There is little to suggest that this growth will not continue. However, the computer architectures used to sustain this growth are changing radically from sequential to parallel.

The performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently. The time to perform a basic operation is ultimately limited by the ``clock cycle'' of the processor, that is, the time required to perform the most primitive operation. However, clock cycle times are

decreasing slowly and appear to be approaching physical limits such as the speed of light. We cannot depend on faster processors to provide increased computational performance.

To circumvent these limitations, the designer may attempt to utilize internal concurrency in a chip, for example, by operating simultaneously on all 64 bits of two numbers that are to be multiplied. However, a fundamental result in Very Large Scale Integration (VLSI) complexity theory says that this strategy is expensive. This result states that for certain transitive computations (in which any output may depend on any input), the chip area A and the time T required to perform this computation are related so that must exceed some problem-dependent function of problem size. This result can be explained informally by assuming that a computation must move a certain amount of information from one side of a square chip to the other. The amount of information that can be moved in a time unit is limited by the cross section of the chip. This gives a transfer rate of , from which the relation is obtained. To decrease the time required to move the information by a certain factor, the cross section must be increased by the same factor, and hence the total area must be increased by the square of that factor.

This result means that not only is it difficult to build individual components that operate faster, it may not even be desirable to do so. It may be cheaper to use more, slower components. For example, if we have an area of silicon to use in a computer, we can either build components, each of size A and able to perform an operation in time T, or build a single component able to perform the same operation in time T/n. The multicomponent system is potentially n times faster. Computer designers use a variety of techniques to overcome these limitations on single computer performance, including pipelining (different stages of several instructions execute concurrently) and multiple function units (several multipliers, adders, etc., are controlled by a single instruction stream). Increasingly, designers are incorporating multiple ``computers," each with its own processor, memory, and associated interconnection logic. This approach is facilitated by advances in VLSI technology that continue to decrease the number of components required to implement a computer. As the cost of a computer is (very approximately) proportional to the number of components that it contains, increased integration also increases the number of processors that can be included in a computer for a particular cost. The result is continued growth in processor counts.

5 Trends in Networking

Another important trend changing the face of computing is an enormous increase in the capabilities of the networks that connect computers. Not long ago, high-speed networks ran at 1.5 Mbits per second; by the end of the 1990s, bandwidths in excess of 1000 Mbits per second will be commonplace. Significant improvements in reliability are also expected. These trends make it feasible to develop applications that use physically distributed resources as if they were part of the same computer. A typical application of this sort may utilize processors on multiple remote computers, access a selection of remote databases, perform rendering on one or more graphics computers, and provide real-time output and control on a workstation.

We emphasize that computing on networked computers ("distributed computing") is not just a subfield of parallel computing. Distributed computing is deeply concerned with problems such as reliability, security, and heterogeneity that are generally regarded as tangential in parallel computing. (As Leslie Lamport has observed, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.") Yet the basic task of developing programs that can run on many computers at once is a parallel computing problem. In this respect, the previously distinct worlds of parallel and distributed computing are converging.

6 Summary of Trends

This brief survey of trends in applications, computer architecture, and networking suggests a future in which parallelism pervades not only supercomputers but also workstations, personal computers, and networks. In this future, programs will be required to exploit the multiple processors located inside each computer and the additional processors available across a network. Because most existing algorithms are specialized for a single processor, this situation implies a need for new algorithms and program structures able to perform many operations at once. Concurrency becomes a fundamental requirement for algorithms and programs.

This survey also suggests a second fundamental lesson. It appears likely that processor counts will continue to increase perhaps, as they do in some environments at present, by doubling each

year or two. Hence, software systems can be expected to experience substantial increases in processor count over their lifetime. In this environment, scalability resilience to increasing processor counts is as important as portability for protecting software investments. A program able to use only a fixed number of processors is a bad program, as is a program able to execute on only a single computer.

CHAPTER 3

1 Flynn's Taxonomy

In general, digital computers may be classified into four categories, according to the multiplicity of instruction and data streams. This scheme for classifying computer organizations was introduced by Michael J. Flynn. The essential computing process is the execution of a sequence of instructions on a set of data. The term stream is used here to denote a sequence of items (instructions or data) as executed or operated upon by a single processor. Instructions or data are defined with respect to a referenced machine. An instruction stream is a sequence of instructions as executed by the machine; a data stream is a sequence of data including input, partial, or temporary results, called for the instruction stream.
Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. Listed below are Flynn's four machine organizations:

1. Single instruction stream single data stream (SISD)
2. Single instruction stream multiple data stream (SIMD)
3. Multiple instruction stream single data stream (MISD)
4. Multiple instruction stream multiple data stream (MIMD)

1.1 SISD computer organization

This organization represents most serial computers available today. Instructions are executed sequentially but may be overlapped in their execution stages.

1.2 SIMD computer organization

In this organization, there are multiple processing elements supervised by the same control unit. All PE's receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams.

1.3 MISD computer organization

There are n processor units, each receiving distinct instructions operating over the same data stream and its derivatives. The results (output) of one processor become the input (operands) of the next processor in the macropipe.

1.4 MIMD computer organization

Most multiprocessor systems and multiple computer systems can be classified in this category. MIMD computer implies interactions among the n processors because all memory streams are derived from the same data space shared by all processors. If the n data streams were from disjointed subspaces of the shared memories, then we would have the so-called multiple SISD (MSISD) operation, which is nothing but a set of n independent SISD uniprocessor systems.

The last three classes of computer organization are the classes of parallel computers.

2 A Taxonomy of Parallel Architectures

There are many ways in which parallel computers can be constructed. These computers differ along various dimensions.

2.1 Control Mechanism

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as stream, multiple data stream (SIMD), a single control unit dispatches instructions to each processing unit. Figure 2.2(a) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. Processing units can be selectively switched off

during an instruction cycle. Examples of SIMD parallel computers include the Illiac IV, MPP, DAP, CM-2, MasPar MP- I, and MasPar MP-2.

Computers in which each processor is capable of executing a different program independent of the other processors are called multiple instruction stream, multiple data stream (MIMD) computers. Figure 2.2(b) depicts a typical MIMD computer. Examples of MIMD computers include the Cosmic Cube, nCUBE 2, iPSC, Symmetry, FX-8, FX-2800, TC-2000, CM-5, KSR-l, and Paragon XP/S.

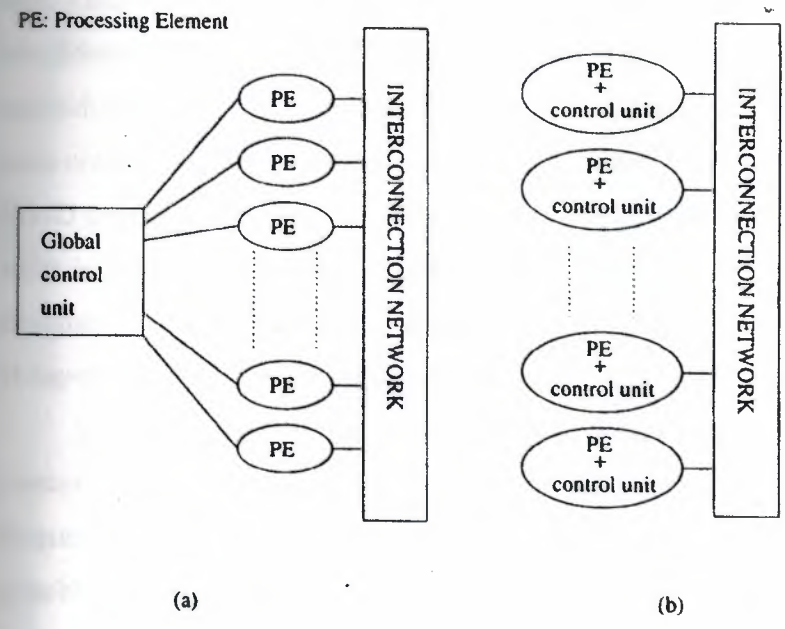

Figure 2.2 A typical SIMD architecture (a) and a typical MIMD architecture (b).

SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. SIMD computers are naturally suited for data-parallel programs; that is, programs in which the same set of instructions are executed on a large data set. Furthermore, SIMD computers require less startup time for communicating with neighboring

processors. This is because the communication of a word of data is just like a register transfer (due to the presence of a global clock) with the destination register in the neighboring processor. A drawback of SIMD computers is that different processors cannot execute different instructions in the same clock cycle. For instance, in a conditional statement, the code for each condition must be executed sequentially. This is illustrated in Figure 2.3. The conditional statement in Figure 2.3(a) is executed in two steps. In the first step, all processors that have B equal to zero execute the instruction C = A. All other processors are idle. In the second step, the 'else' part of the instruction (C = A/B) is executed. The processors that were active in the first step now become idle. Data-parallel programs in which significant parts of the computation are contained in conditional statements are therefore better suited to MIMD computers than to SIMD computers. Individual processors in an MIMD computer are more complex, because each processor has its own control unit. It may seem that the cost of each processor must be higher than the cost of a SIMD processor. However, it is possible to use general-purpose microprocessors as processing units in MIMD computers. In contrast, the CPU used in SIMD computers has to be specially designed. Hence, due to the economy of scale, processors in MIMD computers may be both cheaper and more powerful than processors in SIMD computers.

SIMD computers offer automatic synchronization among processors after each instuction execution cycle. Hence, SIMD computers are better suited to parallel programs that require frequent synchronization. Many MIMD computers have extra hardware to provide fast synchronization, which enables them to operate in SIMD mode as well. Examples of such computers are the DADO and CM-5.

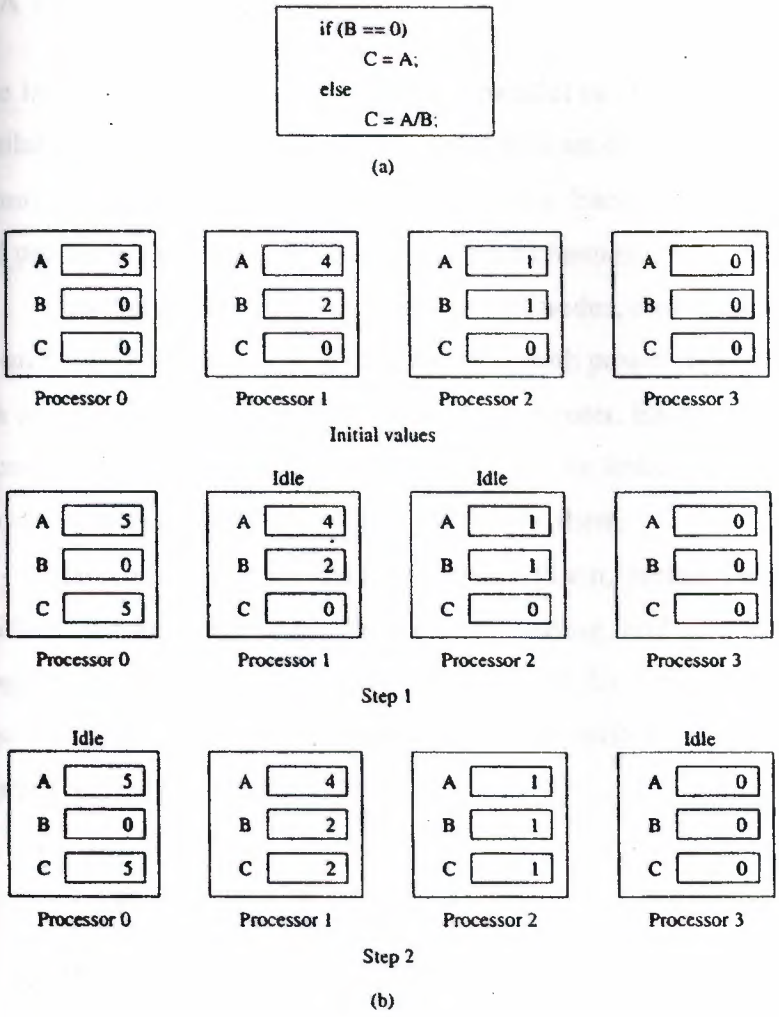

Figure 2.3 executing a conditional statement on an SIMD computer with four processors: (a) The conditional statement; (b) The execution of the statement in two steps.

3. A Parallel Machine

The Intel Paragon is a particular form of parallel machine, which makes concurrent computation available at relatively low cost. It consists of a set of independent processors, each with its own memory, capable of operating on its own data. Each processor has its own program to execute and processors are linked by communication channels.

The hardware consists of a number of nodes, disk systems, communications networks all mounted together in one or several cabinets with power supply for the whole system. Each node is a separate board, rather like a separate computer. Each node has memory, network interface, expansion port, cache and so on. The nodes are linked together through a back plane, which provides high-speed communications between them.

Each node has its own operating system, which can be considered as permanently resident. It takes care of all the message passing, and also allows more than one executable program, or process as they will be called, to be active on each node at any time. Strictly speaking, it is node processes that communicate with other node processes rather than the nodes themselves.

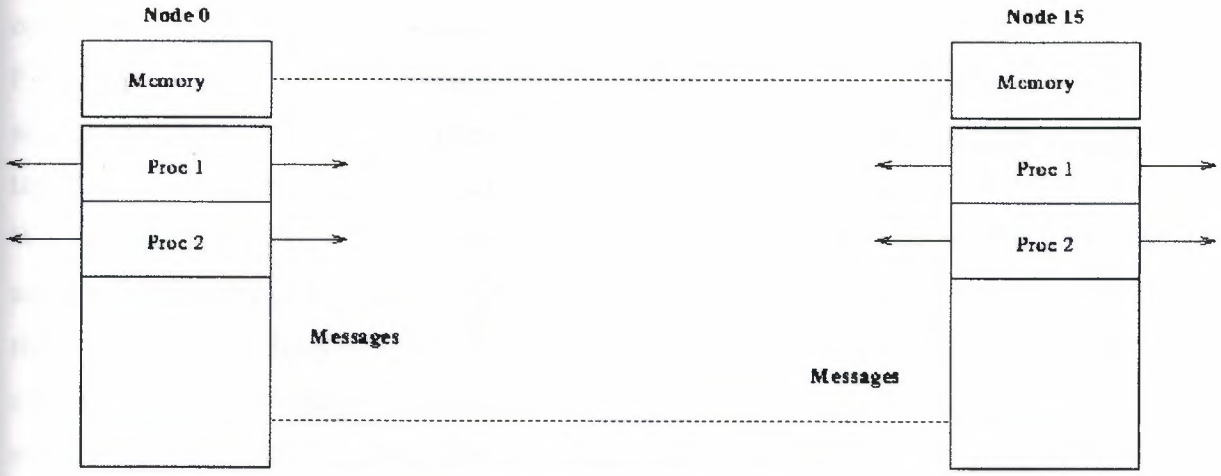

Figure 1.1: Process communication on the nodes

Remember, nodes use their own copy of the program and have their own memory allocation. No variables are shared between nodes or even between processes on the same node. Data can only be shared by sending them as messages between processes.

The Paragon supercomputer is a distributed-memory multicomputer. The system can accommodate more than a thousand heterogeneous nodes connected in a two-dimensional rectangular mesh. A lightweight MACH 3.0 based microkernel is resident on each node, which provides core operating system functions. Transparent access to file systems is also provided. Nodes communicate by passing messages over a high-speed internal interconnect network. A general-purpose MIMD (Multiple Instruction, Multiple Data) architecture supports a choice of programming styles and paradigms, including true MIMD and Single Program Multiple Data (SPMD).

We will adopt the SPMD programming paradigm (Single Program Multiple Data) i.e. each process is the same program executing on different processors. Each program executes essentially the same algorithms, but different branches of the code may be active in different processors. The general architecture of the machine is illustrated in figure 1.2.In the illustration, nodes are arranged in a 2D mesh. Each compute node consists of two i860XP processors. One of these is an application processor and the other a dedicated communication processor. User applications will normally run using the application processor. The figure illustrates that each compute node may pass messages to neighbouring nodes through a bi-directional communication channel. When messages are to be passed indirectly between non-neighbouring processors, the operating system will handle routing the message between intermediate processors.

File system support and high-speed parallel file access is provided through the nodes labelled service and I/O in the diagram. Access to the parallel file system is made through standard OSF library routines (open(), close(), read(), write(), etc.,).

When a user is logged in to the Paragon system, the operating system will allocate the login session to one of the service nodes. Exactly which service node is in use is totally transparent to the user. The user will usually edit files, and compile, link and run applications while logged in to one of the service nodes. Note also that most sites will have available a so-called cross-environment which allows most of the program development stages - editing, compiling, linking and debugging - to be carried out on a workstation away from the paragon system. Using the cross-environment is highly recommended, as the available capacity for such operations is usually greater on a workstation than on the service nodes. Consult your local system administrator to find out how to use this facility.

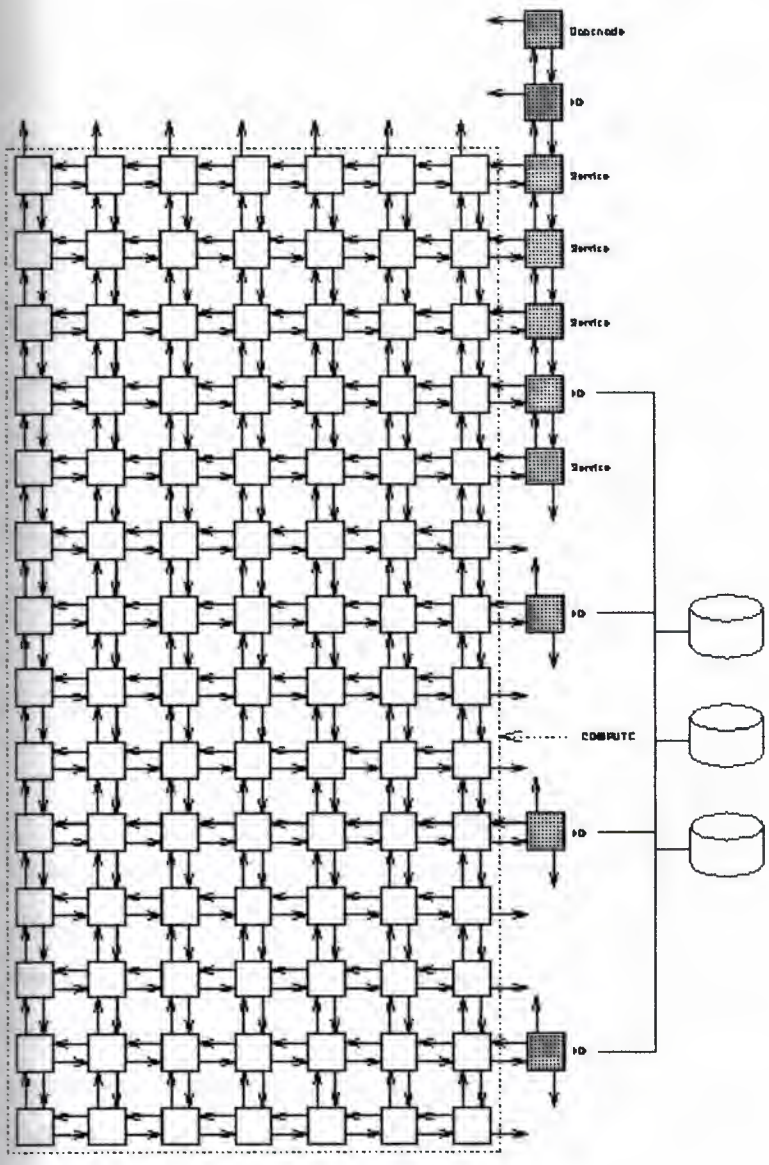

Figure 1.2: Overview of the Paragon system

CHAPTER 4

1 Parallel Programming

To run the algorithms on a parallel computer, we need to implement them in a programming language. In addition to providing all the functionality of a sequential language, a language for programming parallel computers must provide mechanisms for sharing information among processors. It must do so in a way that is clear, concise, and readily accessible to the programmer. A variety of parallel programming paradigms have been developed. This chapter discusses the strengths and weaknesses of some of these paradigms, and illustrates them with examples.

2 Parallel Programming Paradigms

Different parallel programming languages enforce different programming paradigms. The variations among paradigms are motivated by several factors. First, there is a difference in the amount of effort invested in writing parallel programs. Some languages require more work from the programmer, while others require less work but yield less efficient code. Second, one programming paradigm may be more efficient than others for programming on certain parallel computer architectures. Third, various applications have different types of parallelism, so different programming languages have been developed to exploit them. This section discusses these factors in greater detail.

2.1 Explicit versus Implicit Parallel Programming

One way to develop a parallel program is to code an explicitly parallel algorithm. 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. The compiler's task is straightforward. It simply generates code for the instructions specified by the programmer. The programmer's task, however, is quite difficult.

Another way to develop parallel programs is to use a sequential programming language and have the compiler insert the constructs necessary to run the program on a parallel computer. This approach, called implicit parallel programming, is easier for the programmer because it places a majority of the burden of parallelization on the compiler.

Unfortunately, the automatic conversion of sequential programs to efficient parallel ones is very difficult because the compiler must analyze and understand the dependencies in different parts of the sequential code to ensure an efficient mapping onto a parallel computer. The compiler must partition the sequential program into blocks and analyze dependencies between the blocks. The blocks are then converted into independent tasks that are executed on separate processors. Dependency analysis is complicated by control structures such as loops, branches, and procedure calls. Furthermore, there are often many ways to write a sequential program for a given application. Some sequential programs make it easier than others for the compiler to generate efficient parallel code. Therefore, the success of automatic parallelization also depends on the structure of the sequential code. Some recent languages, such as Fortran D, allow the programmer to specify the decomposition and placement of data among processors. This makes the job performed by parallelizing compilers somewhat simpler.

2.2 Shared-Address-Space versus Message-Passing

In the shared-address-space programming paradigm, programmers view their programs as a collection of processes accessing a central pool of shared variables. The shared-address-space programming style is naturally suited to shared-address-space computers. A parallel program on a shared-address-space computer shares data by storing it in globally accessible memory. Each processor accesses the shared data by reading from or writing to shared variables. However, more than one processor might access the same shared variable at a time, leading to unpredictable and undesirable results. For example, assume that x initially contains the value 5 and that processor P_1 increases the value of x by one while processor P_2 decreases it by one. Depending on the sequence in which the instructions are executed, the value of x can become 4, 5, or 6. For example, if P_1 reads the value of x before P_2 decreases it, and stores the increased value after P_2 stores the decreased value, x will become 6. We can conrect the situation by preventing the second processor from decreasing x while it is being increased by the first processor.

Shared-address-space programming languages must provide primitives to resolve such mutual-exclusion problems.

In the message-passing programming paradigm, programmers view their programs as a collection of processes with private local variables and the ability to send and receive data between processes by passing messages. In this paradigm, there are no shared variables among processors. Each processor uses its local variables, and occasionally sends or receives data from other processors. The message-passing programming style is naturally suited to message-passing computers.

Shared-address-space computers can also be programmed using the message-passing paradigm. Since most practical shared-address-space computers are nonuniform memory access architectures, such emulation exploits data locality better and leads to improved performance for tnany applications. On shared-address-space computers, in which the local memory of each processor is globally accessible to all other processors (Figure 2.5(a)), this emulation is done as follows. Part of the local memory of each processor is designated as a communication buffer, and the processors read from or write to it when they exchange data. On shared-address-space computers in which each processor has local memory in addition to global memory, message passing can be done as follows. The local memory becomes the logical local memory, and a designated area of the global memory becomes the communication buffer for message passing.

Many parallel programming languages for shared-address-space or message-passing MIMD computers are essentially sequential languages augmented by a set of special system calls. These calls provide low-level primitives for message passing, process synchronization, process creation, mutual exclusion, and other necessary functions. Extensions to C, Fortran, and C++ have been developed for various parallel computers including nCUBE2, iPSC 860, Paragon XP/S CM-5, TC 2000, KSR- 1, and Sequent Symmetry. In order for these programming languages to be used on a parallel computer, information stored on different processors must be explicitly shared using these primitives. As a result, programs may be efficient, but tend to be difficult to understand, debug, and maintain. Moreover, the lack of standards in many of the languages makes programs difficult to port between architectures. Parallel programming libraries, such as PVM, Parasoft EXPRESS, P4, and PICL, try to address some of these problems by offering vendor-independent low-level primitives. These libraries offer better code portability compared

to earlier vendor-supplied programming languages. However, programs are usually still difficult to understand, debug, and maintain.

2.3 Data Parallelism versus Control Parallelism

In some problems, many data items are subject to identical processing. Such problems can be parallelized by assigning data elements to various processors, each of which performs identical computations on its data. This type of parallelism is called data parallelism. An example of a problem that exhibits data parallelism is matrix multiplication. When multiplying two n x n matrices A and B to obtain matrix C = ($c_{i,j}$), each element $c_{i,j}$ is computed by performing a dot product of the i^{th} row of A with the j^{th} column of B. Therefore, each element $c_{i,j}$ is computed by performing identical operations on different data, which is data parallel.

Several programming languages have been developed that make it easy to exploit data parallelism. Such languages are called data-parallel programming languages and programs written in these languages are called data-parallel programs. A data-parallel program contains a single sequence of instructions, each of which is applied to the data elements in lockstep. Data-parallel programs are naturally suited to SIMD computers.

A global control unit broadcasts the instructions to the processors, which contain the data. Processors execute the instruction stream synchronously. Data-parallel programs can also be executed on MINID computers. However, the strict synchronous execution of a data-parallel program on an MIMD computer results in inefficient code since it requires global synchronization after each instructions. One solution to this problem is to relax the synchronous execution of instructions. In this programming model, called single program, multiple data or SPMD, each processor executes the same program asynchronously. Synchronization takes place only when processors need to exchange data. Thus, data parallelism can be exploited on an MINID computer even without using an explicit data-parallel programming language.

Control parallelism refers to the simultaneous execution of different instruction streams. Instructions can be applied to the same data stream, but more typically they are applied to different data streams. An example of control parallelism is pipelining. In pipelining, computation is parallelized by executing a different program at each processor and sending

intermediate results to the next processor. The result is a pipeline of data owing between processors. Algorithms for problems requiring control parallelism usually map well onto MIMD parallel computers because control parallelism requires multiple instruction streams. In contrast, SIMD computers support only a single instruction stream and are not able to exploit control parallelism efficiently.

Many problems exhibit a certain amount of both data parallelism and control parallelism. The amount of control parallelism available in a problem is usually independent of the size of the problem and is thus limited. In contrast, the amount of data parallelism in a problem increases with the size of the problem. Therefore, in order to use a large umber of processors efficiently, it is necessary to exploit the data parallelism inherent in an application.

Note that not all data-parallel applications can be implemented using data-parallel programming languages nor can all data-parallel applications be executed on SIMD computers. In fact, many of them are more suited for MIMD computers. For example, the search problem has data parallelism, since successors must eventually be generated for all the nodes in the tree. However, the actual code for generating successor nodes contains many conditional statements. Thus, depending upon the code being generated, different instructions are executed. As shown in Figure 2.3, such programs perform poorly on SIMD computers. In some data-parallel applications, the data elements are generated dynamically in an unstructured manner, and distribution of data to processors must be done dynamically. For example, in the tree-search problem, nodes in the tree are generated during the execution of the search algorithm, and the tree grows unpredictably. To obtain a good load balance, the search space must be divided dynamically among processors. Data-parallel programs can perform data redistribution only on a global scale; that is, they do not allow some processors to continue working while other processors redistribute data among themselves. Hence, problems requiring dynamic distribution are harder to program in the data-parallel paradigm.

Data-parallel languages offer the programmer high-level constructs for sharing information and managing concurrency. Programs using these high-level constructs are easier to write and understand. Some examples of languages in this category are Dataparallel C and C*. However, code generated by these high-level constructs is generally not as efficient as handcrafted code that uses low-level primitives. In general, if the communication patterns

required by the parallel algorithm are not supported by the data-parallel language, then the data-parallel program will be less efficient.

3 Primitives for the Message-Passing Programming Paradigm

Existing sequential languages can easily be augmented with library calls to provide message-passing services. This section presents the basic extensions that a sequential language must have in order to support the message-passing programming paradigm.

Message passing is often associated with MIMD computers, but SIMD computers can be programmed using explicit message passing as well. However, due to the synchronous execution of a single instruction stream by SIMD computers, the explicit use of message passing sometimes results in inefficient programs.

3.1 Basic Extensions

The message-passing paradigm is based on just two primitives: SEND and RECEIVE. SEND transmits a message from one processor to another, and RECEIVE reads a message from another processor.

The general form of the SEND primitive is

SEND(message, messagesize, target, type, flag)

Message contains the data to be sent, and messagesize is its size in bytes. Target is the label of the destination processor. Sometimes, target can also specify a set of processors as the recipient of the message. For example, in a hypercube-connected computer, target may specify certain subcubes, and in a mesh-connected computer it may specify certain submeshes, rows, or columns of processors.

The parameter type is a user-specified constant that distinguishes various types of messages. For example, in the matrix multiplication algorithm described in Section there are at least two distinct types of messages.

Usually there are two forms of SEND. One allows processing to continue immediately after a message is dispatched, whereas the other suspends processing until the message is received by the target processor. The latter is called a blocking SEND, and the former a nonblocking SEND. The flag parameter is sometimes used to indicate whether the SEND operation is blocking or nonblocking.

When a SEND operation is executed, the operating system performs the following steps. It copies the data stored in message to a separate area in the memory, called the communication buffer. It adds an operating-system-specific header to the message that includes type, flag, and possibly some routing information. Finally, it sends the message. In newer parallel computers, these operations are performed by specialized routing hardware. When the message arrives at the destination processor, it is copied into this processor's communication buffer and a system variable is set indicating that a message has arrived. In some systems, however, the actual transfer of data does not occur until the receiving processor executes the corresponding RECEIVE operation.

The RECEIVE operation reads a message from the communication buffer into user memory. The general form of the RECEIVE primitive is

RECEIVE(message, messagesize, source, type, flag)

There is a great deal of similarity between the RECEIVE and SEND operations because they perform complementary operations. The message parameter specifies the location at which the data will be stored and messagesize indicates the maximum number of bytes to be put into message. At any time, more than one message may be stored in the communication buffer. These messages may be from the same processor or different processors. The source parameter specifies the label of the processor whose message is to be read. The source parameter can also be set to special values, indicating that a message can be read from any processor or a set of processors. After successfully completing the RECEIVE operation, source holds the actual label of the processor that sent the message.

The type parameter specifies the type of the message to be received. There may be more than one message in the communication buffer from the source processor(s). The type parameter selects a particular message to read. It can also take on a special value to indicate that any type of message can be read. After the successful completion of the RECEIVE operation, type will store the actual type of the message read.

As with SEND, the RECEIVE operation can be either blocking or nonblocking. In a blocking RECEIVE, the processor suspends execution until a desired message arrives and is read from the communication buffer. In contrast, nonblocking RECEIVE returns control to the program even if the requested message is not in the communication buffer. The flag parameter can be used to specify the type of RECEIVE operation desired.

Both blocking and nonblocking RECEIVE operations are useful. If a specific piece of data from a specific processor is needed before the computation can proceed, a blocking RECEIVE is used. Otherwise, it is preferable to use a nonblocking receive. For example, if a processor must receive data from several processors, and the order in which these data arrive is not predetermined, nonblocking RECEIVE is usually better.

Most message-passing extensions provide other functions in addition to SEND and RECEIVE. These functions include system status querying, global synchronization, and setting mode for communication. Another important function is WHOAMI. The WHOAMI function returns information about the system and the processor itself. The general form of the WHOAMI function is:

WHOAMI (processorid, numofprocessor s)

Processorid returns the label of the processor, and numofprocessor s returns the total number of processors in the parallel computer. The processarid is the value used for the target and source parameters of the RECEIVE and SEND operations. The total number of processors helps determine certain characteristics of the topology of the parallel computer (such as the number of dimensions in a hypercube or the number of rows and columns in a mesh).

Most message-passing parallel computers are programmed using either a host--node model or a hostless model. In the host-node model, the host is a dedicated processor in charge of loading the program onto the remaining processors (the nodes). The host also performs

housekeeping tasks such as interactive input and output, termination detection, and process termination. In contrast, the hostless model has no processor designated for such housekeeping tasks. However, the programmer can program one of the processors to perform these tasks as required.

The following sections present the actual functions used by message passing for some commercially-available parallel computers.

3.2 nCUBE 2

The nCUBE 2 is an MIMD parallel computer developed by nCUBE Corporation. Its processors are connected by a hypercube interconnection network. A fully configured nCUBE 2 can have up to 8192 processors. Each processor is a 32-bit RISC processor with up to 64MB of local memory. Early versions of the nCUBE 2's system software supported the host-node programming model. A recent release of the system software primarily supports the hostless model.

The nCUBE 2's message-passing primitives are available for both the C and Fortran languages. The nCUBE 2 provides nonblocking SEND with the use of the nwrite function.

```
C           int nwrite (char *message, int messagesize, int target, int type, int *fiag)

Fortran     integer function nwrite(message, messagesize, target, type, flag)
            dimension message (*)
            integer messagesize, target, type, flag
```

The functions of nwrite's parameters are similar to those of the SEND operation. The main difference is that the flag parameter is unused. The nCUBE 2 does not provide a blocking SEND operation.

The blocking RECEIVE operation is performed by the nread function.

```
C           int nread(char *message, int messagesize, int *source, int *type, int  *flag)

Fortran     integer function nread (message, messsgesize, source, type, flag)
```

```
dimension reasage (*)
integer messagesize, source, type, flag
```

The nread function's parameters are similar to those of RECEIVE with the exception of the flag parameter, which is unused. The nCUBE 2 emulates a nonblocking RECEIVE by calling a function to test for the existence of a message in the communication buffer. If the message is present, nread can be called to read it. The ntest function tests for the presence of messages in the communication buffer.

```
C           int ntest ( int *source, int  *type)

Fortran     integer function ntest (source, type)
            integer source, type
```

The ntest function checks to see if there is a message in the communication buffer from processor source of type type. If such a message is present, ntest returns a positive value, indicating success; otherwise it returns a negative value. When the value of source or type (or both) is set
to-1, ntest checks for the presence of a message from any processor or of any type. After the function is executed, type and source contain the actual source and type of the message in the communication buffer.

The functions npid and ncubesize implement the WHOAMI function.

```
C           int npid()
            int ncubesize()

Fortran     integer function npid()
            integer function ncubesize()
```

The npid function returns the processor's label, and ncubesize returns the number of processors in the hypercube.

3.3 iPSC 860

Intel's iPSC 860 is an MIMD message-passing computer with a hypercube interconnection network. A fully configured iPSC 860 can have up to 128 processors. Each processor is a 32-bit i860 RISC processor with up to 16MB of local memory. One can program the iPSC using either the host-node or the hostless programming model. The iPSC provides message~passing extensions for the C and Fortran languages. The same message-passing extensions are also available for Intel Paragon XP/S, which is a mesh-connected computer.

The iPSC's nonblocking SEND operation is called csend.

```
C          csend (long type, char *message, long messagesize, long target,
           long flag)

Fortran    subroutine csend (type, message, messagesize, target, flag)
           integer type
           integer message (*)
           integer messagesize, target, flag
```

The parameters of csend are similar to those of SEND. The flag parameter holds the process identification number of the process receiving the message. This is useful when there are multiple processes running on the target processor. The IPSC does not provide a blocking SEND operation. We can perform blocking RECEIVE by using the crecv function.

```
C          crecv (long type, char  *rnessage, long messagesize)

Fortran    subroutine crecv (type, message, messagesize)
           integer type
```

integer message (*)
integer messagesize

Comparing the crecv function with the RECEIVE operation, we see that the source and flag parameters are not available in crecv. However, crecv allows information about the source processor to be encoded in the type parameter. The iPSC provides nonblocking RECEIVE by using a function called irecv. The arguments of irecv are similar to crecv, with the exception that irecv returns a number that is used to check the status of the receive operation. The program can wait for a nonblocking receive to complete by calling the msgwait function. It takes the number returned by irecv as its argument and waits until the nonblocking RECEIVE operation has completed.

The iPSC functions rnynade and numnodes are similar to WHOAMI. They return the label of the calling processor and the number of processors in the hypercube, respectively.

C	long mynode()
	long numnodes()
Fortran	integer function mynode()
	integer function numnodes()

3.4 CM-5

The CM-5, developed by Thinking Machines Corporation, supports both the MIMD and SIMD models of computation. A fully configured CM-5 can have up to 16384 processors connected by a fat tree interconnection network. The CM-5 also has a control network, used for operations involving many or all processors. Each CM-5 node has a SPARC RISC processor and four vector units with up to 32MB of local memory. One can program the CM-5 using either the host-node or hostless programming models.

When the CM-5 is used in MIMD mode, it is programmed with the use of message-passing primitives that are available for the C, Fortran, and C++ languages.

The CM-5's blocking SEND function is CMMD_send_lack.

C int CMMD_send_block (int target, int type, void *message,
int messagesize)

Fortran integer function CMMD_send_block (target, type, message, messagesize)
integer target, type
integer message (*)
integer messagesize

The parameters of CMMD_send_block are similar to those for the generic SEND primitive. The CM-5's nonblocking SEND operation is CMMD_send_async.

C CMMD_mcb CMMD_send_async (int target, int type, void *message,
int messagasize, void (*handler) (CMMD_mcb))

Fortran integer funotion CMMD_send_asyno (target, type, message,
messagesize, handler)
integer target, type
integer message (*)
integer messagesize, handler

Most of the parameters required by CMMD_send_async are similar to those required by the SEND operation. The CMMD_send_async function returns a pointer to a message control block (CMMD_mcb) after it has queued the message for transmission. The programmer is responsible for preserving the data in the buffer pointed to by message, and for freeing the CMMD_rncb when the message has been sent. The parameter handler allows the programmer to define a handler routine that is invoked automatically when the message has been sent.

The CM-5 provides blocking RECEIVE with the CMMD_receive_block function

C int CMMD_receive_blook(int source, int type, void *message,
int messagesize)

```
Fortran     integer  function CMMD_receive_block (source, type, message,
            messagesize)
            integer source, type
            integer message (*)
            integer messagesize
```

A nonblocking RECEIVE operation is provided by the function CMMD_receive_async.

```
C           CMMD_mcb CMMD_receive_async (int source, int type, void *message,
            int messagesize, void (*handler) (CMMD_mcb))
```

```
Fortran     integer     function CMMD_receive_async (source, type, message,
            messagesize, handler)
            integer source, type
            integer message (*)
            integer messagesize, handler
```

The parameters of the CMMD_receive_lock and CMMD_receive_async operations are similar to those for the corresponding CMMD_send_lock and CMMD_send_async operations.

On the CM-5, the send function does not actually send the message until the destination node invokes a receive function, indicating that it is ready to receive a message. Furthermore, the CMMD send functions send no more data than the receiver has signaled it can accept. Thus, the number of bytes sent is the smaller of the number of bytes requested (that is, the messagesize of the send function) and the number of bytes the receive function allows (that is, the messagesize of the receive function).

The CM-5 provides the functionality of WHOAMI with the functions CMMD_self_address and CMMD_partition_size. These functions return the label of the calling processor and the total number of processors.

```
C           int CMMD_self_address()
            int CMMD_partition_size()
```

Fortran int function CMMD_self_address
int function CMMD_partition_size

4 Data-Parallel Languages

The main emphasis of data-parallel languages is to make it easier for the programmer to express the data parallelism available within a program in a manner that is independent of the architectural characteristics of a given parallel computer. A data-parallel language has the following characteristics:

(1) It generates only a single instruction stream.

(2) It implies the synchronous execution of instructions. Hence, it is much easier to write and debug data-parallel programs, since race conditions and deadlocks are impossible.
It requires the programmer to develop code that explicitly specifies parallelism.

(3) It associates a virtual processor with the fundamental unit of parallelism. The programmer expresses computation in terms of operations performed by virtual processors. The advantage of virtual processors is that programmers need not be concerned with the number of physical processors available on a parallel computer. They simply specify how many processors they need. However, using virtual processors inappropriately may result in inefficient parallel programs.

(4) It allows each processor to access memory locations in any other processor. This characteristic creates the illusion of a shared address-space and simplifies programming since programmers do not have to perform explicit message passing.

Since data-parallel languages hide many architectural characteristics from the programmer, writing data-parallel programs is generally easier than writing programs for explicit message passing. However, the ease of programming comes at the expense of increased compiler complexity. Compilers for data-parallel languages must map virtual processors onto physical processors, generate code to communicate data, and enforce synchronous instruction execution.

4.1 Data Partitioning and Virtual Processors

In a data-parallel language, data are distributed among virtual processors. The virtual processors must be mapped onto the physical processors at some point. If the number of virtual processors is greater than the number of physical processors, then several virtual processors are emulated by each physical processor. In that case, each physical processor partitions its memory into blocks-one for each virtual processor it emulates-and executes each instruction in the program once for each of the virtual processors. For example, assume that an instruction increments the value of a variable by one and that three virtual processors are emulated by each physical processor. The physical processors execute the instruction by performing three consecutive increment operations, one for each virtual processor. These operations affect the memory blocks of each virtual processor.

The amount of work done by each physical processor depends on the number of virtual pmcessors it emulates. If VPR is the ratio of virtual to physical processors. then the work performed by each physical processor for each program instruction is greater by a factor of VPR. This is because each physical processor has to execute VPR instructions for each program instruction. However, the amount of communication performed may be smaller or larger than VPR. For instance, if the virtual processors are mapped so that neighboring virtual processors reside on physical processors that are farther away, the communication requirements will be higher than VPR. In most cases, however, it is possible to map virtual processors onto physical processors so that nearest-neighbor communication is preserved. If this is the case. some virtual processors may need to communicate with virtual processors mapped onto the same physical processor. Depending on how smart the emulation is, this may lead to lower communication requirements.

Some data-parallel languages contain primitives that allow the programmer to specify the desired mapping of virtual processors onto physical processors. This is essential in developing efficient parallel programs. The efficiency of a mapping depends on both the data communication

patterns of the algorithm, and the interconnection network of the target computer. For example, a mapping suited to a hypercube-connected parallel computer may not be suited to a mesh-connected parallel computer.

4.2 C*

C* is a data-parallel programming language that is an extension of the C programming language. C* was designed by Thinking Machines Corporation for the CM-2 parallel computer. The CM-2 is a fine-grain SIMD computer with up to 65,536 processors. Each CM-2 processor is one bit wide, and supports up to 1 Mbit of memory. C* is also available for the CM-5.

C* adheres to the ANSI standard for C, so programs written in ANSI C compile and run correctly under C*. In addition, C* provides new features for specifying data parallelism. The features of C* include the following

(1) A method to describe the size and the shape of parallel data and to create parallel variables.

(2) Operators and expressions for parallel data that provide functionality such as data broadcasting and reduction. Some of these operators require communication.

(3) Methods to specify data points within selected parallel variables on which C* code is to operate.

4.2.1 Parallel Variables

C* has two types of variables. A scalar variable is identical to an ordinary C variable; scalar variables are allocated in the host processor. A parallel variable is allocated on all node processors. A parallel variable has as many elements as the number of processors.

A parallel variable has a shape in addition to a type. A shape is a template for parallel data-a way to configure data logically. It defines how many parallel elements exist and how they are organized. A shape has a specific number of dimensions, referred to as its rank, with a given number of processors or positions in each dimension. A dimension is called an axis. For example,

the following statement declares a shape called mesh, of rank two and having 1,048,576 positions:

shape [1024] [lO24] mesh;

Similarly, the following statement declares a shape of rank four with two positions along each axis:

shape [2] [2] [2] [2] fourcube;

The fourcube shape declaration declares a template containing a total of 2 x 2 x 2 x 2 = 16 positions. A shape should reflect the most logical organization of the problem's data. For example, a graphics program might use the mesh shape to represent the two-dimensional images that it is going to process. However, not all possible configurations can be declared using the shape primitive. For example, shape does not allow us to declare a triangular-shaped or a diamond-shaped mesh. However, we can do this by declaring a larger shape and using only a portion of it. For example, we can obtain a triangular shape by declaring a square shape and using only half of it.

C* does not allow the programmer to specify virtual-to-physical processor mappings explicitly. C* maps virtual processors onto physical processors so that neighboring virtual processors are mapped onto neighboring physical processors. However, C* allows us to specify across which dimensions of the shape communication will. be performed more frequently. The compiler uses this information to reduce communication cost.

After a shape is specified, parallel variables of that shape can be declared. Parallel variables have a type, a storage class, and a shape. The following statement declares the parallel variable count of type int and shape ring:

shape [8192] ring;
int: ring count;

This declaration creates a parallel variable count with 8192 positions each of which is allocated to a different processor. We can access individual elements of the parallel variable count by using left indexing. For example, [1] count accesses the value of the count that resides on the second

processor (numbering is from 0 to 8191). Figure 13.1 illustrates the differences between scalar and parallel variables.

Any standard or user-defined data type can be used with parallel variables. For example, an entire C structure can be a parallel variable. As another example, int: fourcube a [1000] declares the 16-position parallel variable a, in which each element is an array of 1000 integers.

4.2.2 Parallel Operations

C* supports all standard C operations and a few new operations for data-parallel programming. In addition, C* defines additional semantics for standard C operations when they are used with parallel variables.

If the operands of an operation are scalar, then C* code behaves exactly like standard C code and the operation is performed on the host computer. The situation is different when one or more operands are parallel variables. For example, consider a simple assignment statement of the form x + = y, where both x and y are parallel variables. This assignment adds the value of y at each shape position to the value of x at the corresponding shape position. All additions take place in parallel. Note that an expression that evaluates to a parallel variable must contain parallel variables of the same shape as the resulting parallel variable. Hence, in this example, x and y must be of the same shape. In a statement of the form x = a, where a is a scalar variable, the value of a is stored in each position of x. This is similar to a broadcast operation.

A more interesting situation arises when the left side of an assignment operation is a scalar variable and the right side is a parallel variable. There are two cases in which this assignment makes sense. In the first case, the parallel variable is fully left indexed. For instance, if a is a scalar variable and x is a parallel variable of rank one, then a = [4]x is a valid statement and assigns to a the value of x at the fifth position of the shape. In the second case, the operation is one of those shown in Table 13.1. The result of these operations is a reduction. For instance, a + = x sums all the values of x and stores the result in a.

shape [1024] ring
shape [1024] [1024] mesh

int ring : a
int mesh : b
int flag

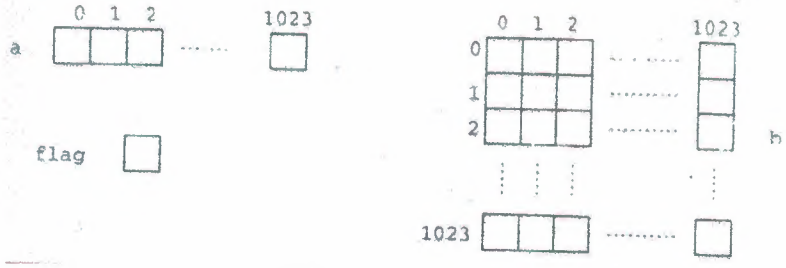

Figure 13.1 Examples of parallel and scalar variables. a and b are parallel variables of different shapes, and flag is a scalar variable. Courtesy of Thinking Machines Corporation.

Table 13.1 C* reduction operations.

Operator	Meaning
+=	Sum of values of parallel variable elements
-=	Negative of the sum of values
&=	Bitwise AND of values
^=	Bitwise XOR of values
|=	Bitwise OR of values
<?=	Minimum of values
>?=	Maximum of values

4.2.3 Choosing a Shape

The with statement enables operations on parallel data by setting the current shape. Operations are performed on parallel variables of the current shape. In the following example, the With statement is required for performing the parallel addition:

```
shape [8192] ring;
int: ring x, y,z
 with (ring)
    x= y+z;
```

4.2.4 Setting the Context

C* has a where statement that restricts the positions of a parallel variable on which operations are performed. The positions to be operated on are called active positions. Selecting the active positions of a shape is called setting the context. For example, the where statement in the following code avoids division by zero:

```
with (ring) {
   where (z != 0)
     x  = y / z;
}
```

The where statement can include an else clause. The else clause complements the set of active positions. Specifically, the positions that were active when the where statement was executed are deactivated, and the inactive positions are activated. For example,

```
with (ring) {
  where (z != 0)
     x  = y / z;
  else
     x  = y;
}
```

On the CM-2 (since it is an SIMD machine) the where and else clauses are executed serially. One should limit the use of the where-else clause because multiple context setfings degrade performance substantially.

4.2.5 Communication

C* supports two methods of interprocessor communication. The first is called grid communication, in which parallel variables of the same type can communicate in regular patterns. The second method is called general communication, in which the value of any element of a parallel variable can be sent to any other element, whether or not the parallel variables are of the same shape. The regularity of grid communication makes it considerably faster than general communication on many architectures. In particular, on CM-2, grid communication can be mapped onto the underlying interconnection network quite efficiently.

Data communication in C* uses left indexing, but instead of using a scalar value to left-index a parallel variable, a parallel variable is used. This operation is called parallel left indexing. A parallel left index rearranges the elements of the parallel variable based on the values stored in the elements of the parallel index. The index must be of the current shape.

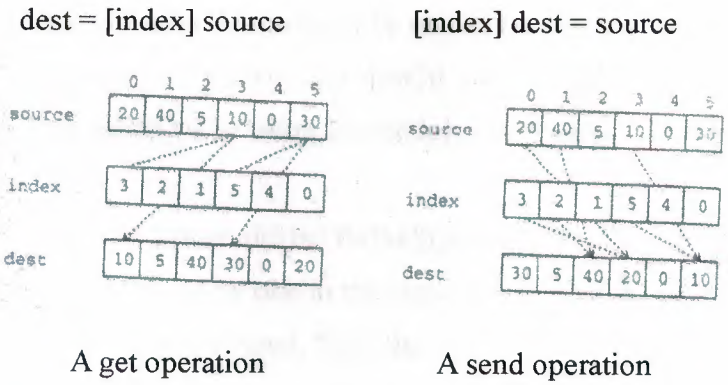

Figure 13.2 Examples of the send and get general communication operations. Courtesy of Thinking Machines Corporation.

C* allows both send and get operations. If index, dest, and source are parallel variables of rank one, the general form of the send operation is

```
[index]dest = source;
```

and the general form of the get operation is

```
dest = [index]source;
```

These operations are illustrated in Figure 13.2.

For general communication, the values of the index variable can he arbitrary. For grid communication, C* uses a new function called pcoord to provide a self-index for a parallel variable along a specified axis. In grid communication, data can be sent only a fixed distance along each dimension. For example,

```
destid = [pcoord(0)+1] source1d;
```

shifts the elements stored in sourceld by one to the right,

```
destid = [pcoord(0)-2]source1d;
```

shifts the elements by two to the left, and

```
dest2d = [pcoord(0)+1] [pcoord(1)+1]source2d;
```

shifts the elements of source2d by one to the left and up. Note that dest1d and source1d are one-dimensional shapes, whereas dest2d and source2d are two-dimensional shapes. Wraparound shifts are achieved by using the modulus operation. For example,

```
dest2d = [(pcoord(0)+1)%%4][(pcoord(1)+1)%%3]source2d;
```

shifts the elements by one to the right and down. The elements that fall off the two-dimensional shape are wrapped around. Note that the numbers 4 and 3 used in the modulus operation, are the number of positions along the corresponding axis. The operator '% %' is similar to C's '%' operator but works with negative values as well.

To summarize, in general we can say that data-parallel programs tend to be smaller than explicit message-passing programs. Furthermore, prograras that use the virtual-processor paradigm tend to be simpler to implement.

4.3 CM Fortran

Thinking Machines Corporation has developed a Fortran-based data-parallel language that runs on both the CM-2 and CM-5 parallel computers. The CM Fortran language is an extension of Fortran 77 supplemented with array-processing extensions from Fortran 90. The array-processing features map easily onto the SIMD architecture of the CM-2, as well as the CM-5 running in SIMD mode.

The essence of the Fortran 90 array-processing features is that they treat arrays as first-class objects. An array can be referenced by name in an expression or assignment and passed as an argument to any Fortran intrinsic function; the operation is performed on every element of the array. For example,

```
REAL A(40,40,40)
....
A = 8.0          ! Sets all 64,000 elements to 8.0
A = A*2.0        ! All 64,000 elements Contain 16.0
A = SQRT(A)      !  All 64,000 elements Contain 4.0
```

The serial implementation of Fortran 90 treats arrays as objects but generates serial code. The Connection Machine stores each array element in the memory of a separate virtual processor and operates on all elements simultaneously.

In contrast to C*, CM Fortran does not provide new data types to support parallelism; thus, we need not take any special action to use the processors of the CM-2 or the CM-5. The CM Fortran compiler allocates arrays on either the host or the processors, depending on how they are used. The rules are as follows:

(1) Arrays that are used only in Fortran 77 constructs in a program, as well as all scalar data, reside on the host. Essentially, the host executes all of the Fortran 77 code.

(2) Arrays that are used in array operations anywhere in a program reside on the processors. The processors execute all of the Fortran 90 code.

CM Fortran supplies a rich set of intrinsic functions for transforming arrays. The array transformations that can be performed are data movement. array reduction. array construction, and array multiplication.

4.3.1 Conformable Arrays

When an expression or an assignment involves two or more arrays, the arrays must be conformable; that is, they must be of the same size and shape. Scalars can be used freely in array assignments and array-valued expressions, since Fortran 90 defines a scalar as conformable with any array. Arrays of different sizes and shapes can coexist in the memory of the processors (in different sets of virtual processors), but conformable arrays are always stored in the same set of processors in the same order. For instance, if A, B, and C are one-dimensional arrays of size 10, then elements A(1), B(1), and C(1) all reside in the memory ofthe same processor, as doA(2), B(2), and C(2), and so forth. Each processor executes operations on its own set of array elements; no data motion occurs between processors. If arrays are not conformable, then corresponding array elements do not reside in the memory of the same processor. This fact suggests one of the basic principles of CM Fortran programming: operations on corresponding elements of conformable arrays use the system the most efficiently.

4.3.2 Selecting Array Elements

Data parallelism requires that some operations be performed only on certain data elements. An array assignment can be made conditional on an array's values by enclosing the assignment in a WHERE statement. WHERE is the array-processing extension of the Fortran IF statement and is

similar to the where statement in C*. For example, the following statement prevents division by zero:

```
WHERE (A. NE. 0)        C = B/A
```

4.3.3 Communication

Like C*, CM Fortran provides two types of communication: grid communication, which moves data in a regular fashion, and general communication, which allows arbitrary data movement. Fortran 90 defines triplet subscripts, a new construct for specifying a sequence of subscripts in an array reference. The subscript sequence indicates the subset, or section, of the array to be operated on. The triplet subscript has the following form:

```
array-name (first: 1ast: stride)
```

A triplet indicates the first element, the last element, and an increment interval. The first and last subscripts default to the declared bounds of the array, and stride defaults to one. In CM Fortran, array sections are particularly useful for moving data in regular grid patterns. Data elements are moved by assigning one section of an array to another section of the same array or another array. As with all array operations, array sections must be conformable. For example, if A and B are one-dimensional arrays, then the statement

```
A(2:9:1) = B(3:10:1)
```

shifts the first eight elements of B after position two by one position to the left, and assigns them to A.

A vector-valued subscript is a form of array section that uses a vector as a subscript. The values of the vector need not be ordered, and there is no fixed stride. This construct specifies an arbitrary selection of array values along a dimension. Vector-valued subscripts are useful for vector permutations and for indexing into a vector or an array. This type of communication is similar to C* 's get operation.

Besides single-dimension communication, CM Fontran also provides general communication between dimensions by using the FORALL statement. A FORALL statement defines one or more index variables and uses them in an assignment, thus indicating an action that depends on the positions of the target array elements. FORALL is a powerful feature for expressing data motion. For example, FORALL can perform, in parallel, arbitrary permutations of multidimensional arrays. The following statement indexes into matrix H, using index arrays X and Y:

```
FORALL      (I=1:N,J=1:M)      G(I,J) = H(X(I,J) , Y(I,J))
```

5 Primitives for the Shared-Address-Space Programming Paradigm

The primitives required for the shared-address-space programming paradigm fall into three categories: (1) primitives to allocate shared variables, (2) primitives for mutual exclusion and synchronization, and (3) primitives for creating processes.

5.1 Primitives to Allocate Shared Variables

The shared-address-space programming paradigm has two kinds of variables: shared and local. Shared variables are accessible to all processes, but local variables can be accessed only by the process that declared them. In the following discussion we use two keywords, shared and private, to specify the type of variables. For example, the following code fragment declares a shared array b and a private integer variable i:

```
shared int b [10000]
private int i;
```

5.2 Primitives for Mutual Exclusion and Synchronization

When separate processes access shared variables, it is important that the operations do not conflict with each other. A critical section contains code that must be executed by only one process at a time. Languages using the shared-address-space programming paradigm provide locks to help the programmer enforce mutual exclusion for critical section execution. These locks are usually implemented by using a special type of shared variable that can be manipulated only by atomic operations. An operation is atomic if it cannot be interrupted by another process. Each lock can be owned by at most one process at any time. The process that owns the lock can execute the corresponding critical section. When a process leaves the critical section, it releases the lock. If a process tries to acquire a lock that is currently owned by another process, it enters a busy-wait cycle or suspends execution. In a busy-wait cycle, a processor continuously checks the lock, waiting for it to be released. When a process suspends execution, it joins a queue associated with the lock it tried to acquire. The process becomes active again when the lock has been released and the process is at the front of the queue.

Besides locks, other mechanisms, such as semaphores and monitors, are also provided by languages using the shared-address-space programming paradigm to help the programmer enforce mutual exclusion in more complicated situations.

During the concurrent execution of processes, there are situations in which the processes need to synchronize before the end of the execution. Synchronization is achieved by a barrier synchronization primitive. This primitive is executed when the processes need to synchronize, and operates as follows. Each processes wait at the barrier for every other processes. After synchronization, all processes proceed with their execution.

5.3 Primitives for Creating Processes

In the shared-address-space programming paradigm, process creation is done by a system call that creates processes identical to the parent process. These processes share the variables declared shared by the parent process, as well as the locks declared and initialized by the parent process. Once created, subprocesses can perform independent computations. After the subprocesses have been created, the single execution thread is partitioned into several threads

with access to shared data. Creating subprocesses is similar to spawning new processes in any multitasking operating system. On most shared-address-space computers, this operation is called fork (after a similar system call in the UNIX operating system). The parent process continues execution after creating its subprocesses. When the subprocesses terminate, they merge by using another primitive, typically called join.

Instead of creating processes at the beginning of the execution. we can also have a single process, usually called the master process. When the master process needs to perform a task in parallel it can create a predetermined number of other processes that work in parallel to perform the task. These processes are usually called slave processes. When the task is complete, the slave processes terminate and control returns to the master process. Later, the master can create slave processes again. For example. consider a program that consists of a sequence of computationally intensive loops separated by some bookkeeping computations. Using the master-slave approach, the master performs the bookkeeping and creates slave processes to perform the computations required by each loop. We assume that there is a function called slave-spawn (func) that creates a predetermined number of slave processes, each executing the function func. Slave processes partition the work done by func in a predetermined way.

5.4 Sequent Symmetry

The Sequent Symmetry is a shared-address-space computer that runs the DYNIX operating system, a multiprocessor version of UNIX. The Symmetry is a bus-based system that can use up to thirty 32-bit processors. Even if no effort is made to parallelize a user application, all DYNIX processes potentially run in parallel. In addition, users can specify explicit parallelism. The entire main memory is equally accessible to all processors. Each processor also has some local cache memory.

The Symmetry is programmed in languages such as C and Fortran. These languages include extensions that allow programs to specify explicit parallelism. Shared variables are declared in C by using the shared data-type modifier. Shared variables are declared in Fortran by putting all the shared data into the same COMMON block, then giving compiler directives indicating that the COMMON block is to reside in shared memory. All other variables are private.

The Symmetry supports two different ways of creating processes. The first is called multitasking and the other is called microtasking. We create multitasks by using the fork function and microtasks by using the m_fork function. Fork is the standard UNIX process creation function that creates an identical copy of the calling process. M_fork takes the function that will be executed by all the processes as an argument. For microtasking, the user specifies how many processes to create with the m_set_procs function. The processes take significantly less time to be created compared to the time required for the fork function. The number of processes that can be created is limited by the number of available physical processors in the system. However, this does not impose any limit on the computational granularity assigned to each process. Processes can be assigned small amounts of computation and request more work when they run out.

The Symmetry provides two new data types for C to allow locks and barrier synchronization. These data types are slock_t for locks and sbarrier_t for barrier synchronization. The operations defined on these types depend on whether microtasking or multitasking is used. The operations are shown in Table 13.2.

6 Fortran D

Several parallel programming languages have been proposed that lie between explicit and implicit parallel languages. These languages provide facilities that enable the compiler to generate efficient parallel code. Fortran D is one such language.

Fortran D programs specify how the data (such as Fortran arrays) are assigned to processors. The compiler uses this information to generate parallel programs. Although Fortran D exploits data parallelism, it is not a data-parallel programming language. In pure data-parallel programming languages, the programmer specifies the code to be executed by each processor, and the code is replicated in all the processors. In Fortran D, the programmer describes the computation as a single program. The programmer does not specify how data elements are moved between processors; the compiler uses Imowledge about how the data are distributed to generate code for each processor. The facilities provided by Fortran D for data decomposition and distribution are significantly richer than those provided by conventional data-parallel languages (such as the shape primitive of

C*).

The problem of mapping arrays onto processors is approached at two levels: problem mapping and machine mapping. Problem mapping determines the alignment of arrays with respect to each other. The problem mapping is influenced by the structure of the underlying computation. If a computation requires corresponding elements from two separate arrays, it is better to align the arrays. This ensures that, after the distribution of arrays on different processors, these elements will belong to the same processor. As an example, consider the problem of matrix addition. In order to avoid any communication, the matrices to be added must be aligned with each other.
Machine mapping determines the distribution of the data on the actual parallel machine. This distribution is influenced by hardware characteristics such as the topology and communication mechanism.
Fortran D supports problem mapping with the DECOMPOSITION and ALIGN statements, and machine mapping with the DISTRIBUTE statement. These capabilities are described in the following subsections. In addition to these statements, Fortran D also provides irregular data distribution and dynamic data decomposition (that is, the ability to change the alignment or distribution of a decomposition at any point in the program).

6.1 Problem Mapping

The DECOMPOSITION statement declares the name, dimensionality, and size of a decomposition. A decomposition is simply an abstract layout or index domain. For example, the following statements declare a one-dimensional decomposition of size N called A, and a two-dimensional decomposition of size NxN called B.

DECOMPOSITION A (N)
DECOMPOSITION B (N, N)

The ALIGN statement maps arrays with respect to a decomposition. There are a variety of possible alignments in Fortran D. The simplest alignment occurs when the array is mapped exactly onto the decomposition. For example, if we want to align a two-dimensional array X (N, N) with the two-dimensional decomposition B, the syntax is

ALIGN X(I,J) with B(I,J)

The indices I and J play the role of placeholders, indicating that element (I , J) of matrix X is mapped to element (I, J) of the decomposition. Exact matching may be a good way to align the data with the decomposition. One such algorithm is matrix multiplication, in which a two-dimensional decomposition is a natural choice, and exact matching of the arrays to the decomposition provides the data distribution.

The user can specify an alignment offset for any dimension of an array. For example, to shift the elements of matrix X in the previous example by two in the first dimension and by three in the second dimension, the syntax of the ALIGN statement is

ALIGN X(I,J) with B(I+2, J+3)

In general, any positive or negative constants can be added to the placeholders to indicate the desired offset.

The ALIGN statement can be specified with a stride to achieve a mapping in which consecutive array elements are mapped onto nonconsecutive decomposition elements. For example, if we want to map a one-dimensional array Y (N) onto the one-dimensional decomposition A such that there is a stride of three for successive array elements, then the syntax of the ALIGN statement is

ALIGN Y(I) with A(3*I)

The ALIGN construct also allows the user to permute the dimensions between arrays and decompositions. For example, this ability can be used to map the transpose of an array onto a decomposition:

ALIGN X(I,J), with B(J,I)

It is sometimes convenient to ignore certain dimensions of the array when mapping an array onto a decomposition. All data elements in the unassigned dimensions are mapped onto the same location in the decomposition.

Conversely, it may be necessary to map arrays with fewer dimensions onto the decomposition. In this case, it is necessary to specify the mapping for each dimension of the array and the actual position of the array in the unmapped dimensions of the decomposition. One example is mapping a vector onto a two-dimensional decomposition in matrix-vector multiplication. This can be done by using the following statement:

ALIGN Y(I) with B(I,N)

This statements aligns the vector Y with the last column of the decomposition.

In many of the ALIGN statements presented, some of the array elements do not fit within the decomposition. In this case, Fortran D allows the programmer to specify whether the elements should be truncated or wrapped around.

6.2 Machine Mapping

The DISTRIBUTE statement takes a decomposition and distributes it to available processors. Each dimension of a decomposition can be distributed differently. Three types of distributions are supported by Fortran D: BLOCK, CYCLIC, and BLOCK_CYCLIC. If p is the number of processors and n is the size of a dimension in the decomposition, the distributions are described as follows:

(1) BLOCK distribution divides the decomposition into contiguous chunks of size n / P, assigning one block to each processor. For example, if B (N, N) is a two-dimensional decomposition, then the statement DISTRIBUTE B (BLOCK, BLOCK) assigns blocks of ($N/\sqrt{p}$ x ($N/\sqrt{p}$) elements to each processor. The BLOCK decomposition can be used to map elements onto processors in a blocked style. If the decomposition is one-dimensional, then this distribution

corresponds to block-striped mapping. If the decomposition is two-dimensional, then this distribution corresponds to block-checkerboard mapping.

(2) CYCLIC distribution specifies a round-robin division of the decomposition, assigning every p^{th} element to the same processor. The statement DISTRIBUTE B (CYCLIC, CYCLIC) distributes the elements of the decomposition in a manner similar to thcyclic mapping. If the decomposition is one-dimensional, then this distribution corresponds to cyclic-striped mapping . If the decomposition is two-dimensional, then this distribution corresponds to cyclic-checkerboard mapping.

(3) BLOCK_CYCLIC is similar to CYCLIC but takes a parameter m. It first divides the dimension into contiguous chunks of size m and then assigns these chunks in the same fashion as CYCLIC. If the decomposition is one-dimensional, then this distribution corresponds to block-cyclic-striped mapping. If the decomposition is two-dimensional, then this distribution corresponds to block-cyclic checkerboard mapping. For example, DISTRIBUTE B (BLOCK_CYCLIC (2), BLOCK_CYCLIC (2)) distributes the elements in a block-cyclic-checkerboard mapping mapping with block size of two.

Any one of these three types of distributions can be assigned for each dimension of the decomposition. For multidimensional decompositions, different combinations of distribution patterns may be assigned to distinct dimensions. The asterisk (*) denotes dimensions that are assigned locally ; these dimensions are not distributed.

For example, to implement Dijkstra's single-source shortest-paths algorithm, we must map the rows of the adjacency matrix onto the processors. If x (N, N) is the adjacency matrix and A (N,N) is the two-dimensional decomposition, this can he done by the following statements:

ALIGN X(I,J) with A(I,.J)
DISTRIBUTE A (BLOCK, *)

Fortran D also allows us to specify processor allocations. where the allocation gives the number of processors assigned to each dimension of the decomposition. This allows the user to change

the default assignment of the compiler, which is to assign an equal number of processors to each dimension.

CONCLUSION

The increasing density of transistors on a chip follows directly from a decreasing feature size, which is now for the alpha. Feature size will continue to decrease and by the year 2000, chips with 50 million transistors are expected to be available. What can we do with all these transistors? With around a million transistors on a chip, designers were able to move full mainframe functionality to about of a chip. This enabled the personal computing and workstation revolutions. The next factors of ten increase in transistor density must go into some form of parallelism by replicating several CPUs on a single chip.

By the year 2000, parallelism is thus inevitable to all computers, from your children's video game to personal computers, workstations, and supercomputers. Today we see it in the larger machines as we replicate many chips and printed circuit boards to build systems as arrays of nodes, each unit of which is some variant of the microprocessor. Parallelism allows one to build the world's fastest and most cost-effective supercomputers.

Parallelism may only be critical today for supercomputer vendors and users. By the year 2000, all computers will have to address the hardware, algorithmic, and software issues implied by parallelism. The reward will be amazing performance and the opening up of new fields; the price will be a major rethinking and re-implementation of software, algorithms, and applications.

REFERANCES

[(Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis / Univ. of Minnesota)
" Introduction to Parallel Computing " The Benjamin/Cummings Publishing company Inc.
Copyright© 1994 by The Benjamin/Cummings Publishing Company Inc.]

[(M.E.C. Hull / Univ. of Ulster, D. Crookes / The Queen's Univ., Belfast, P.J. Sweeney / Univ. of Ulster) " Parallel Processing " Addison-Wesley Publishing Company
Copyright© 1994 Addison-Wesley Publishers Ltd.
Copyright© Addison-Wesley Publishing Company Inc.

[(Rza E. Bashirov/Eastern Mediterranean Univ.) " Lecture Notes on Parallel Processing "