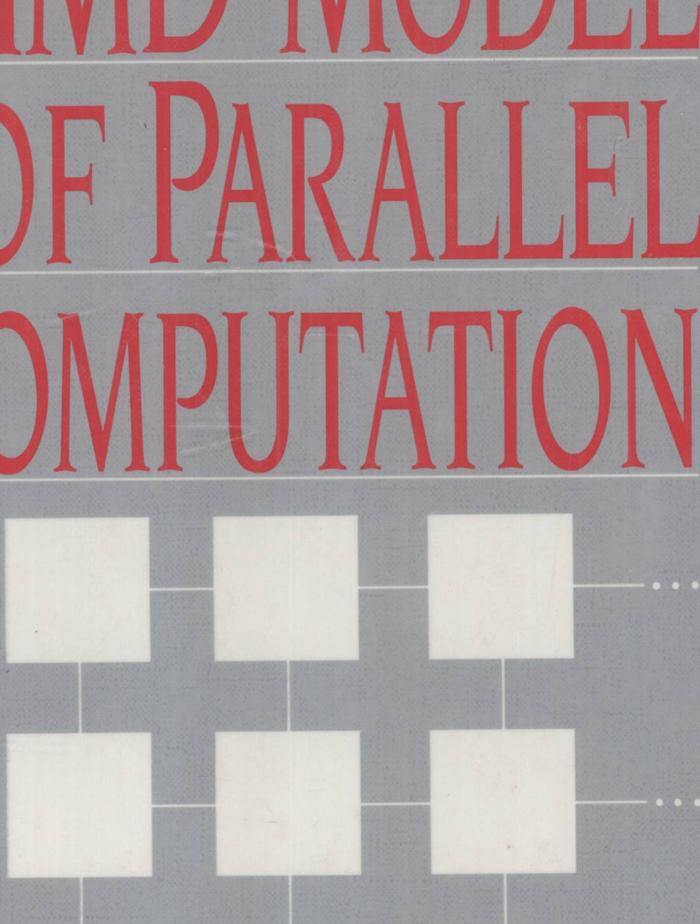
Robert Cypher
Jorge L.C. Sanz

THE SIMD MODEL OF PARALLEL COMPUTATION



Springer-Verlag

TP274
C996

Robert Cypher　　Jorge L.C. Sanz

The SIMD Model of Parallel Computation

With 13 Illustrations

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Robert Cypher
IBM T. J. Watson
Research Center
Yorktown Heights, NY 10598,
USA

Jorge L. C. Sanz
University of Illinois at
Urbana-Champaign
Department of Electrical
and Computer Engineering
Urbana, IL 61801, USA

Cover illustration: Mesh connected computer. Detail from Fig. 4.1, p. 21

Library of Congress Cataloging-in-Publication Data
Cypher, Robert.
The SIMD model of parallel computation / Robert Cypher, Jorge L. C. Sanz.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-94139-8
1. Parallel processing (Electronic computers) 2. Computer architecture. I. Sanz, J. L. C. (Jorge L. C.), 1955– , II. Title.
QA76.58.C96 1994
004′.35—dc20 93-27497

Printed on acid-free paper.

Production managed by Terry Kornak; manufacturing supervised by Vincent Scelta.
Typeset by Asco Trade Typesetting Ltd., Hong Kong.
Printed and bound by R. R. Donnelley & Sons, Inc., Harrisonburg, VA.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-94139-8 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94139-8 Springer-Verlag Berlin Heidelberg New York

The SIMD Model of Parallel Computation

Contents

CHAPTER 1

Introduction

1.1 Background

There are many paradigmatic statements in the literature claiming that this is the decade of parallel computation. A great deal of research is being devoted to developing architectures and algorithms for parallel machines with thousands, or even millions, of processors. Such massively parallel computers have been made feasible by advances in VLSI (very large scale integration) technology. In fact, a number of computers having over one thousand processors are commercially available. Furthermore, it is reasonable to expect that as VLSI technology continues to improve, massively parallel computers will become increasingly affordable and common.

However, despite the significant progress made in the field, many fundamental issues still remain unresolved. One of the most significant of these is the issue of a general purpose parallel architecture. There is currently a huge variety of parallel architectures that are either being built or proposed. The problem is whether a single parallel computer can perform *efficiently* on *all* computing applications.

When considering this question, it is important to notice that there is no unique serial architecture that dominates over all of the others. The advent of special purpose workstations and dedicated architectures has provided cost-effective solutions to many demanding computational problems. Today, nobody would perform low-level image processing without a pipeline of dedicated components performing specific image functions.[207] Signal processing chip sets and systolic arrays [140] are well-suited to many numerically intensive problems. Graphics stations are exclusively dedicated to certain design problems, and in many cases, these computers are single-user oriented. In a completely analogous manner, vector computers have been heavily used by the numerical computing community. These devices are specialized attachments to central processors. It is unlikely that people will stop using them in favor of large-scale, general purpose serial computers. Thus, the profusion of parallel architectures is not very different from the situation with regard to serial architectures.

To understand the arguments for and against a single, general purpose parallel architecture, it is important to recognize the motivation for parallel processing. Parallelism may be regarded as a vehicle to accomplish two different goals. First, it may become a viable way to provide a given amount of computation at a lower cost. Second, it may be used to extend the range of problems that can be solved by computers, regardless of the cost. Of course, these two motivations may coexist, and indeed, they are the ultimate goal of many of today's research projects. Research on general purpose parallel computers could be beneficial for future systems that will provide inexpensive computer power. These computers will be aimed at supporting a very large number of simultaneous users, much in the same way as today's powerful mainframes. Software will provide users with multiprogramming environments, time-sharing operating systems, and programming languages oriented to a rich variety of applications.

On the other hand, it is unlikely that these systems will be able to provide the amount of computing power that is demanded by applications involving numerical analysis, computer vision systems, or physics simulations. Furthermore, the environment provided by a general purpose system is not the most appropriate one for the computing needs of these users. There will be other, more specialized, parallel computers operating as backend coprocessors to satisfy the computing needs of those sophisticated users. These computers will be tailored to classes of applications and will play a role similar to today's special purpose attachments, such as vector processors. The software and hardware involved in these computers will be different from those required by general purpose parallel systems. Time and cost overheads introduced by some software and hardware features are considerable and can be justified only in the presence of a general computing environment. An insightful discussion on this topic, titled "shared memory and the corollary of modest potential", is given by L. Snyder.[230]

Overall, the parallel processing field should not be polarized. It is unlikely that a single parallel computer or architecture will satisfy the computing requirements of all possible applications. In retrospect, the short, but rich, history of computing demonstrates that there is a diversity of serial machines tailored to different applications. There is, and probably will continue to be, a large zoo of parallel computers. Ultimately, the nature of the application areas and cost considerations will make some of them more useful or appealing than others. On the other hand, it is likely that some consolidation will occur as parallel architectures become better understood and more widely used. The need for consolidation is particularly acute in the area of models for parallel programming. If a small number of models of parallel computation can be agreed upon, programmers and algorithm designers can focus on these models and create applications that will be portable across a variety of parallel architectures.

In this monograph, a tour through this zoo of parallel architectures is presented. For each architecture that is studied, algorithms that are tailored to the given architecture will be presented. Although a range of applications areas will be considered, a set of basic operations related to image processing, sorting and routing, and numerical computing will be examined for each of the architectures. These algorithms will be useful in their own right, but will also serve as a means of comparing the different types of parallel computers and will aid in selecting the correct architecture for a given problem area. The emphasis will be on the SIMD (**s**ingle **i**nstruction-stream, **m**ultiple **d**ata-stream) model of parallel computation and its implementation on both SIMD and MIMD (**m**uliple **i**nstruction-stream, **m**ultiple **d**ata-stream) architectures.[85,231]

1.2 Notation

N is used to represent the size of the input to a problem, such as the number of pixels in an image to be processed or the number of entries in each of two matrices to be multiplied. P is used to represent the number of processors in a parallel machine. A function $F(X)$ is said to be $O(G(X))$ if, for all sufficiently large X, there exists a constant C, such that $F(X) \leq C * G(X)$.

If X is a nonnegative integer, then the Y-bit representation of X will be written as $(X_{(Y-1)}, X_{(Y-2)}, \ldots, X_{(0)})$, and the i-th bit of X will be denoted by $X_{(i)}$ (where the 0-th bit is the least significant bit). Also, $X^{(i)}$ is the integer obtained by complementing the i-th bit of X.

The notation $\log X$ will denote the base-2 logarithm of X. The function $\log^{(0)} X = X$, and for all integers $i > 0$, $\log^{(i)} X = \log(\log^{(i-1)} X)$. The function $\log^* X$ equals the smallest nonnegative integer i, such that $\log^{(i)} X \leq 1$.

1.3 Outline

The remaining chapters are organized as follows. Chapters 2 and 3 present an overview of parallel architectures and programming methodologies, respectively. Chapters 4 through 12 provide a critical survey of various parallel architectures and algorithms, based on the topology of the connections between the processors. Specifically, Chapters 4 and 5 look at mesh connected computers, Chapters 6 and 7 focus on pyramid computers, and Chapters 8 through 11 are devoted to hypercube and related computers. For each topology, several existing and proposed parallel machines are discussed and compared. Also, an analysis of parallel algorithms for image processing and scientific and symbolic tasks is presented. The effects of architectural decisions on algorithm design are examined in detail. Finally, some conclusions are outlined in Chapter 12.

CHAPTER 2

Parallel Computer Architectures

The basic types of parallel computer architectures are examined in this chapter. The focus here will be on the physical design of the computer. In Chapter 3, the different high-level models that can be presented to a programmer of a parallel machine will be studied.

2.1 Memory Organization

The physical location of the memory in a parallel computer can be classified as being either *shared* or *distributed*. In a shared (also called "centralized") memory parallel architecture, there is a set of memory locations that are not local to any processor. In order for the processors to access these shared memory locations, they must issue read or write requests that are routed to the memory via a bus or a switching network. In addition to these shared memory locations, each processor in a shared memory architecture has a local private memory in which it can store private data, copies of shared data, and pointers to the shared memory.

In a distributed memory parallel computer, each memory location is local to some processor. A processor can access its own local memory directly. However, to access another processor's local memory, it must send a message to the processor that owns the memory. A processor and its local memory will sometimes be referred to as a *processing element* (PE).

2.2 Communication Medium

In both shared memory and distributed memory computers, a processor must communicate in order to access data that are not stored in its local memory. In shared memory computers, this communication occurs between processors and the shared memory, while in distributed memory computers, it occurs between pairs of processors. There are three techniques that are used for performing this communication, namely, busses, switching networks, and direct processor-to-processor links. Computers that use these communication techniques are called *bus-based, switch-based,* and *processor-based,* respectively. These distinctions are not always sharp, since a single computer can have multiple communication media. For example, some computers have both busses and direct links between pairs of processors.

Both switch-based and processor-based architectures can use either *packet routing* or *circuit switching* to deliver messages. In packet routing, messages are divided into packets that are routed to their destinations. Packets compete with other packets for resources (such as wires and buffers) in a dynamic manner. In circuit switching, an entire path between the message sender and receiver is established. All of the communication links along this path are reserved for the given sender-receiver pair. They cannot be accessed by other senders. Once the path is established, the sender may transmit messages without fear of interference.

There are also several switching modes for packet routing. In store-and-forward routing, the packet "hops" between buffers, and the head of the packet waits until the tail of the packet has been stored in the buffer. In *wormhole routing*[63] and *virtual cut-through routing,*[129] each packet is divided into small units called *flits.* The flits follow one another in a snakelike manner from the sender to the receiver. Thus, if a packet consists of only one flit, these techniques store the entire packet after each hop, and a store-and-forward implementation is obtained. On the other hand, if a message contains a very large number of flits, the first flits will arrive at the receiver before the later flits have even been sent. As a result, the entire path between the sender and receiver will be occupied, as is the case with circuit switching. Wormhole and virtual cut-through routing behave differently when a packet encounters congestion. In wormhole routing, the entire packet is stopped in place, thus, blocking all of the communication links that it occupies. In virtual cut-through routing, the tail of the packet continues to advance, and the entire packet is stored in the node where the congestion was encountered.

In a processor-based distributed memory computer, only certain pairs of processors are connected by direct communication links. Thus, access to another processor's memory may require that a message be routed to the other processor via several intermediate processors. Some processor-based computers allow data to be transferred in both directions simultaneously along a single communication link, while others require that data be transferred in one direction at a time. Some processor-based machines, called *strong communication* machines, allow a single processor to send a different data item over each of its communication links simultaneously. Other processor-based machines, called *weak communication* machines, limit each processor to sending a single data item over a single communication link at a time.

2.3 Topology

Whether busses, switches, or direct links between processors are used for the communication, the communication network can have a wide range of topologies. In bus-based systems, a single bus or multiple busses may be used. When multiple busses are present, every bus may be connected to every processor and/or memory bank, or different busses may be connected to different subsets of processors and memory banks.

One of the most common topologies for switch-based computers is the Omega network.[145] An Omega network connects N inputs to N outputs by means of $\log N$ stages of switches. Each stage consists of $N/2$ switches, each of which has two inputs and two outputs. An Omega network can perform many useful permutations of the inputs to the outputs, but it cannot perform every permutation. As a result, it is possible that some collisions will occur within the network, even though each input is accessing a different output.

Another switching network topology that has been used in a parallel computer is the Benes network.[19,20] Benes networks can be defined for various switch sizes. When it is composed of switches with two inputs and two outputs, the Benes network has $2(\log N) - 1$ stages, each of which consists of $N/2$ switches. A Benes network is capable of performing every permutation of the inputs to the outputs. However, it is time-consuming to calculate how the switches should be set in order to implement a given permutation without having collisions. Therefore, the Benes network is typically used only if the patterns of communication are known in advance and the switch settings can be calculated by the compiler.

A rich class of topologies has been proposed for processor-based parallel computers. These include trees, two- and three-dimensional meshes, pyramids, hypercubes, shuffle-exchanges, and cube-connected cycles. Processor-based architectures will be emphasized in this monograph, and the topologies for these architectures will be studied in depth in the remaining chapters.

2.4 Control

An important characteristic of a parallel machine is whether it operates in an SIMD or an MIMD mode. In an MIMD architecture, different processors can perform different operations at a single time. As a result, each processor in an MIMD machine must have its own copy of its program as well as instruction fetching and decoding logic in order to interpret its program.

In an SIMD architecture, all of the processors are forced to execute the same instruction at the same time. Thus, in an SIMD machine, it is possible to have only one copy of the program and a single controller that fetches the instructions, decodes them, and broadcasts the control signals to all of the processors. However, most SIMD machines offer some degree of processor autonomy by allowing a subset of the processors to ignore the current instruction while the remaining processors execute it. This is accomplished by placing a binary register, called a *mask register*, in each processor and designating certain instructions as being *maskable*. When a maskable instruction is executed, those processors that have a 1 in their mask register perform the instruction, while those processors that have a 0 in their mask register are idle.

Most SIMD architectures do not have a direct data connection from the controller to the processors. However, there are situations in which the controller must broadcast a data value to all of the processors. This can be accomplished by having the processors calculate the number one bit at a time. For instance, if each processor has the ability to calculate arbitrary boolean functions, then they can be directed to calculate the function that always returns TRUE for those bit positions of the broadcast value that contain a 1 and to calculate the function that always returns FALSE for the remaining bit positions.

In addition to distinguishing between SIMD and MIMD control, there are two other ways in which the type of control may be classified. First, in a distributed memory machine, the local memory addresses of the operands and results can be the same for every processor at a given time or they can be different in separate processors. The former case will be referred to as *uniform addressing*, while the latter will be referred to as *independent addressing*. It is possible to have independent addressing even when all of the processors are operating under the direction of a single controller; this can be accomplished by using indirection.

Second, processor-based distributed memory computers with weak communication can be separated into two categories. Assume that the communication links leaving each processor are numbered. If every processor must send data along the same communication link (such as the third one) at a given time, the machine will be said to operate with *uniform communication*. If, instead, separate processors can send messages along different communication links at a given time, the machine will be said to operate with *independent communication*. Independent communication can be implemented when there is a single controller by using indirection to choose the communication port from which data will be sent.

2.5 Clocking

Parallel computers use several different clocking schemes. One option is the use of a single global clock that is broadcast to all of the processors. This option is particularly natural for SIMD machines, but it can be used in MIMD machines as well. A difficulty with using only a single clock is that the clock signal may reach different processors at different times. This phenomenon is called *clock skew*. Clock skew puts a limit on the cycle time of the clock, because the skew between any two communicating processors must be kept below the cycle time to guarantee that the processors are operating on the same cycle. Fortunately, it appears that clock skew can be effectively controlled by careful design of the clock lines. For example, clock skew in the 4096 processor J-machine is kept below 2 ns.[175]

Another clocking scheme is the use of a separate clock for each processor. This avoids the problem of clock skew, but it creates problems associated with communicating between separate clocked regions. When two clocked regions exchange data, arbitration between the clocks is required. The time required by this arbitration depends on the relative phases of the clocks, but it can be significant. If the architecture is processor-based, then each communication between connected processors requires a separate clock arbitration. If the architecture is switch-based, then the switching network can be asynchronous. In this case, both clock skew and clock arbitration problems are minimized. Furthermore, an asynchronous network has the potential to run more quickly than a synchronous one. This is because in an asynchronous network, data are passed on as soon as they are ready, while in a synchronous network, the cycle time is set to the time required by the slowest operation. However, asynchronous networks often require some overhead, both in terms of wires and time, to perform handshaking.

Yet another clocking scheme is possible when the routing is circuit-switched. In this scheme, each communication link consists of data wires and a strobe wire. Once a path has been established between a source and destination node, successive data and strobe wires along the path are electrically connected to one another, forming parallel data and strobe paths. Then, data are placed on the data path, and the strobe path is used to clock the associated data. Specifically, each transition on the strobe path indicates that new data are present on the data path. In this scheme, each communication path operates synchronously, although separate data paths have separate clocks. The advantage of this technique is that it avoids the handshaking required by asynchronous techniques and the clock arbitration delays required by other synchronous techniques. The disadvantages are the need for extra wires (for the strobe signal and for status information that is sent from the destination processor to the source processor) and the requirement that circuit switching be used. This type of clocking scheme is used in the Intel iPSC/2 computer.[178]

2.6 Processor Design

Although it would be desirable to use powerful processors, cost and technological considerations force a tradeoff between the number of processors and processor power. Commercial machines with 1K or more 32-bit processors are available, where each processor occupies a single chip or board. These processors could be either general purpose reduced instruction set (RISC) processors or custom processors that have been optimized for parallel processing.[115,173] As the number of processors increases into the tens of thousands, multiple processors are placed on a single chip and the word size of the processors decreases. MasPar has recently introduced a machine with up to 16K processors, each of which operates on 4-bit quantities.[159]

Other massively parallel machines have used bit-serial (1-bit word size) processors.[17,106,172] A bit-serial processor can typically perform an arbitrary boolean function of two 1-bit inputs. Also, bit-serial processors usually have the power of a full adder, which is a unit that takes three 1-bit operands, $A1$, $A2$, and $A3$, and provides two 1-bit outputs, S and C, where $S = A1$ XOR $A2$ XOR $A3$ and $C = (A1$ AND $A2)$ OR $(A1$ AND $A3)$ OR $(A2$ AND $A3)$. That is, S is the sum bit and C is the carry bit resulting from adding the three operands. A full adder may be used to add two B bit numbers by setting $A1$ and $A2$ to the least significant bits of the addends and setting $A3$ to 0. The resulting sum bit is the least significant bit of the answer. The resulting carry bit is used as the next value of $A3$, and the next bits of the addends are used as the values of $A1$ and $A2$. Repeating this process B times yields the desired sum. Two B bit numbers can be multiplied by performing, at most, $B - 1$ additions of shifted versions of the multiplicand and can be accomplished by using $B(B - 1)$ full adds.

Support for floating-point operations varies greatly. Some powerful custom processors include full floating-point support, while some bit-serial processors perform floating-point operations in software, one bit at a time. In between these extremes, some machines offer limited hardware support for floating-point operations, such as a barrel shifter, per processor, and others have standard floating-point coprocessors that are shared by several processors.

2.7 Selection of a Parallel Architecture

The best choice of a parallel architecture depends on the applications to be run, the programming model to be supported, and the costs to be considered. Distributed memory architectures typically offer better performance than shared memory architectures, because they improve the likelihood that a memory request can be satisfied locally. On the other hand, shared memory machines provide a separation between the processes that are running and the memory that they are accessing. As a result, load balancing can be accomplished in a shared memory architecture by moving processes from heavily loaded processors to lightly loaded ones. Load balancing is more difficult in a distributed memory architecture if the instruction set differentiates between accesses to local memory and to nonlocal memory. If such a differentiation exists, the local memory of a process must be moved with the process, thus, greatly increasing the overhead.

Although it might seem that shared memory architectures are better suited to providing the programmer with a high-level shared memory programming model, Chapter 3 will show that this is not necessarily the case.

Several shared memory architectures with small numbers of processors (fewer than 100) use busses to communicate between the processors and the memory. However, as the number of processors increases, bus-based systems usually become impractical. This is because a bus is typically connected to a large number of processors. Pin limitation (fan-out) considerations prevent any one processor from being connected to too many busses. Therefore, there are typically far fewer busses than processors, and a high bandwidth of communication cannot be supported. Busses may be useful if the application that is being solved involves far more computation than communication or if the communication that is required consists of broadcasting a small amount of data to a large number of locations. Also, several hybrid distributed memory architectures, with both busses and direct links between processors, have been proposed.[31,193,234] In these hybrid systems, the direct links are used to transfer large amounts of data between a small number of PEs, and the busses are used to transmit small amounts of data between large numbers of PEs.

The topology of the communication network is closely related to the applications that will be implemented. For example, a two-dimensional mesh interconnection supports low-level image processing applications very efficiently, while a hypercube interconnection supports symbolic, pointer-based data structures very well. The types of topologies that have been proposed and the applications that they support will be studied in detail in the following chapters.

The efficiency of SIMD or MIMD control is also very dependent on the application being implemented. For example, SIMD control is well-suited to those problems in which the granularity of computing is fine (that is, each PE is assigned only a few data items). An SIMD machine will perform efficiently in this case, since the small amount of data in each PE cannot have a rich structure, which could be exploited by an MIMD machine. In addition, the small amount of data per PE often indicates that the processing to be performed is intrinsically synchronous. Of course, SIMD control requires every processor to execute the same program, but this is a fairly common characteristic among numerical analysis and physics computations in which the PEs carry out a parallel algorithm cooperatively. In fact, even in some shared memory MIMD computers, an SPMD (**s**ingle-**p**rogram, **m**ultiple-**d**ata) paradigm has been proposed.[67] In this paradigm, a powerful coarse-grain MIMD architecture allows data-dependent processing to be performed efficiently.

By studying many examples from different application domains, it is seen that problems for which SIMD computing has been successfully used assign only a small number of data items to each PE. Specifically, the quantum-chromodynamics simulations carried out in the GF11 computer involve a few lattice points per processor;[19] in image processing operations,[156] each

PE handles a few image pixels;[1] in circuit simulations and layout optimization problems,[28,266] each processor handles a few devices, nets, or components; in parallel Fast Fourier Transform (FFT) algorithms,[183] each processor contains a sample of the signal; in sorting problems,[232] each PE typically holds one key.

Finally, the selection of a clocking scheme is very dependent on the other architecural decisions that have been made. For example, if SIMD control has been selected, then a single global clock is almost required. A single global clock may also be preferred because it provides a deterministic operation. On the other hand, the randomness caused by an asynchronous communication network could actually be desirable for performance reasons.[134]

[1] However, from a theoretical viewpoint, the product of the elapsed time and the number of processors may be improved significantly for some problems by using fewer processors.[8] In fact, in some cases, the elapsed time actually decreases when fewer processors are used.[161]

CHAPTER 3

High-Level Models

In this chapter, some of the different high-level models of parallel computers that have been proposed will be examined. A high-level model of a computer has two main purposes. First, it should simplify programming and algorithm design by providing a set of powerful, easy-to-compose, basic operations. Second, it should aid portability, so that a program or algorithm designed for one machine may be used on other machines. However, to be useful, it should accurately reflect the costs of the basic operations. This is essential because the wrong algorithm could be chosen if the costs of different algorithms cannot be accurately judged. The subject of high-level parallel models has been examined by other authors.[117, 155, 230]

High-level parallel models have been defined to capture the important features of both shared memory and distributed memory architectures. Although these high-level models are abstractions of particular architectures, it is important to realize that they are separate from the architectures. For example, it is possible to program a distributed memory architecture using a shared memory model. All that is required is systems software that implements the shared memory abstraction on the distributed memory hardware. This systems software is analogous to virtual memory support in a sequential machine, since in both cases, the software supports a model of the machine that is different from the underlying hardware. Similarly, it is possible to implement an SIMD model on MIMD hardware.[199]

3.1 Shared vs. Distributed Memory Models

3.1.1 Shared Memory Models

In a shared memory model,[128] there is a single global memory that all of the processors can access (write to or read from) in unit time. In addition to the global shared memory, each processor has a local private memory in which data and pointers to the global memory may be stored.