In *CRC Handbook of Algorithms and Theory of Computation*,
M.J. Atallah, ed., 1998, pp. 46:1–46:19.

Algorithmic Techniques for Networks of Processors

Russ Miller and Quentin F. Stout
State University of New York at Buffalo and University of Michigan

Introduction

This chapter is concerned with designing algorithms for machines constructed from multiple processors. In particular, we discuss algorithms for machines in which the processors are connected to each other by some simple, systematic, interconnection pattern. For example, consider a chess board, where each square represents a processor (for example, a processor similar to one in a home computer) and every generic processor is connected to its 4 neighboring processors (those to the north, south, east, and west). This is an example of a *mesh computer*, a network of processors that is important for both theoretical and practical reasons.

The focus of this chapter is on algorithmic techniques. Initially, we define some basic terminology that is used to discuss parallel algorithms and parallel architectures. Following this introductory material, we define a variety of interconnection networks, including the mesh (chess board), which are used to allow processors to communicate with each other. We also define an abstract parallel model of computation, the *PRAM*, where processors communicate with memory instead of with each other. We then discuss several parallel programming paradigms, including the use of high-level data movement operations, divide-and-conquer, pipelining, and master-slave. Finally, we discuss the problem of mapping the structure of an inherently parallel problem onto a target parallel architecture. This mapping problem can arise in a variety of ways, and with a wide range of problem structures. In some cases, finding a good mapping is quite straightforward, but in other cases it is a computationally intractable NP-complete problem.

Terminology

In order to initiate our investigation, we first define some basic terminology that will be used throughout the remainder of this chapter.

Shared Memory versus Distributed Memory

In a *shared memory* machine, there is a single global image of memory that is available to all processors in the machine, typically through a common bus or switching network (see Figure 1). This model is similar to a blackboard, where any processor can read or write to any part of the board (memory), and where all communication is performed through messages placed on the board.

Each processor in a *distributed memory* machine has access only to its private (local) memory (see Figure 1). In this model, processors communicate by sending messages to each other, with the messages being sent through some form of interconnection network. This model is similar to a school in which each professor occupies a unique classroom equipped with a blackboard. For professor W to access information maintained on the board of professor X, W sends a message to X requesting the information, and X sends a message back with the information. In this classroom scenario, messages might be transmitted by students running through the halls. In such message-passing systems, the overhead and

delay can be significantly reduced if it can be arranged so that X sends the information to W without a request being sent. This is particularly useful if it can be arranged so that the data from X arrives before W needs to use it, for then W will not be delayed waiting for the data. This analogy represents an important aspect of developing efficient programs for distributed memory machines, especially general-purpose machines in which communication can take place concurrently with calculation so that the communication time is effectively hidden.

For small shared memory systems, it may be that the network is such that each processor can access all memory cells in the same amount of time. For example, many symmetric multiprocessor (SMP) systems have this property. However, since memory takes space, systems with a large number of processors are typically constructed as modules (i.e., a processor/memory pair) that are connected to each other via an interconnection network. Thus, while memory may be logically shared in such a model, in terms of performance each processor acts as if it is distributed, with some memory being "close" (fast access) to the processor and some memory being "far" (slow access) from the processor. Notice the similarity to distributed memory machines, where there is a significant difference in speed between a processor accessing its own memory versus a processor accessing the memory of a distant processor. Such shared memory machines are called NUMA (non-uniform memory access) machines, and often the most efficient programs for NUMA machines are developed by using algorithms efficient for distributed memory architectures, rather than using ones optimized for uniform access shared memory architectures.

Efficient use of the interconnection network in a parallel computer is often an important consideration for developing and tuning parallel programs. For example, in either shared or distributed memory machines, communication will be delayed if a packet of information must pass through many communication links. Similarly, communication will be delayed by contention if many packets need to pass through the same link. As an example of contention at a link, in a distributed memory machine configured as a binary tree of processors, suppose that all leaf processors on one side of the machine need to exchange values with all leaf processors on the other side of the machine. Then a bottleneck occurs at the root since the passage of information proceeds in a sequential manner through the links in and out of the root.

Both shared and distributed memory systems can also suffer from contention at the destinations. In a distributed memory system, too many processors may simultaneously send messages to the same processor, which causes a processing bottleneck. In a shared memory system, there may be memory contention, where too many processors try to simultaneously read or write from the same location.

Another common feature of both shared and distributed memory systems is that the programmer has to be sure that computations are properly synchronized, i.e., that they occur in the correct order. This tends to be easier in distributed memory systems, where each processor controls the access to its data, and the messages used to communicate data also have the side-effect of communicating the status of the sending processor. For example, suppose processor W is calculating a value, which will then be sent to processor R. If the program is constructed so that R does not proceed until the message from W arrives, then it is guaranteed of using the correct value in the calculations. In a shared memory system, the programmer needs to be more careful. For example, in the same scenario, W may write the new value to a memory location that R reads. However, if R reads before W has written, then it may proceed using the wrong value. This is known as a *race condition*, where the correctness of the calculation depends on the order of the operations. To avoid this, various locking or signaling protocols need to be enforced so that R does not read the location until after W has written to it. Race conditions are a common source of programming errors, and are often difficult to locate because they disappear when a deterministic, serial debugging approach is used.

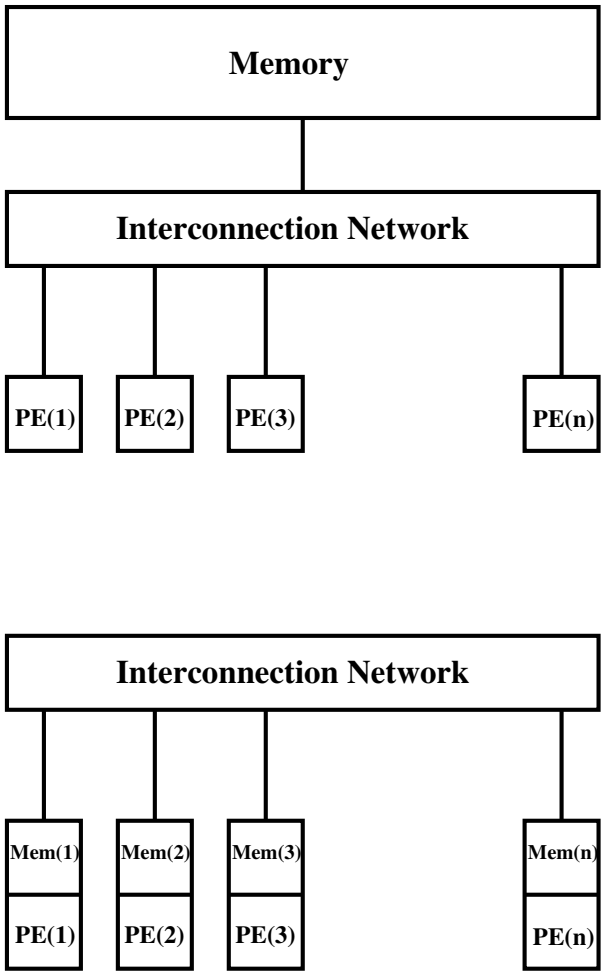


Figure 1: Shared memory (top) and distributed memory (bottom) machines. (*PE* is used represent a processing element and *Mem* is used to represent memory.)

Flynn's Taxonomy

In 1966, Michael Flynn classified computer architectures with respect to the *instruction stream*, that is, the sequence of operations performed by the computer, and the *data stream*, that is, the sequence of items operated on by the instructions [Flynn, 1966]. While extensions and modifications to Flynn's taxonomy have appeared, Flynn's original taxonomy [Flynn, 1972] is still widely used. Flynn characterized an architecture as belonging to one of the following four classes.

- Single-Instruction Stream, Single-Data Stream (SISD)
- Single-Instruction Stream, Multiple-Data Stream (SIMD)
- Multiple-Instruction Stream, Single-Data Stream (MISD)
- Multiple-Instruction Stream, Multiple Data Stream (MIMD)

Standard serial computers fall into the *single-instruction stream, single data stream (SISD)* category, in which one instruction is executed per unit time. This is the so-called "von Neumann" model of computing, in which the stream of instructions and the stream of data can be viewed as being tightly coupled, so that one instruction is executed per unit time to produce one useful result. Modern "serial" computers include various forms of modest parallelism in their execution of instructions, but most of this is hidden from the programmer and only appears in the form of faster execution of a sequential program.

A *single-instruction stream, multiple-data stream (SIMD)* machine typically consists of multiple processors, a control unit (controller), and an interconnection network, as shown in Figure 2. The

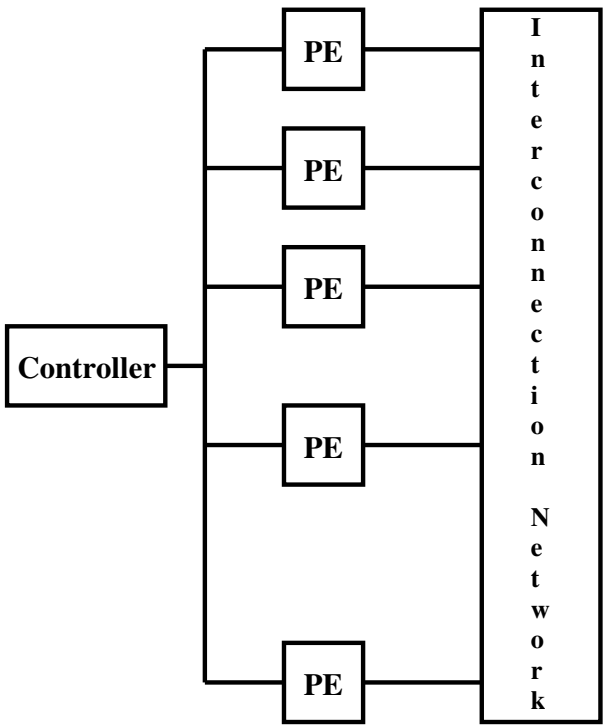


Figure 2: A SIMD machine. (*PE* is used to represent a processing element.)

control unit stores the program and broadcasts the instructions to all processors simultaneously. Active processors execute the instruction on the contents of their own local memory. Through the use of a

mask, processors may be in either an active or inactive state at any time during the execution of the program. Masks can be dynamically determined, based on local data or the processor's coordinates. Note that one side-effect of having a centralized controller is that the system is synchronous, so that no processor can execute a second instruction until all processors are finished with the first instruction. This is quite useful in algorithm design, as it eliminates many race conditions and makes it easier to reason about the status of processors and data.

Multiple-instruction stream, single-data stream (MISD) machines consist of two or more processors that perform separate instructions on the same data. This model is rarely implemented.

A *multiple-instruction stream, multiple-data stream (MIMD)* machine typically consists of multiple processors and an interconnection network. In contrast to the single-instruction stream model, the multiple-instruction stream model allows each of the processors to store and execute its own program, providing multiple instruction streams. Each processor fetches its own data on which to operate. (Thus, there are multiple data streams, as in the SIMD model.) Often, all processors are executing the same program, but may be in different portions of the program at any given instant. This is the *single-program multiple-data (SPMD)* style of programming, and is an important mode of programming because it is rarely feasible to have a large number of different programs for different processors. The SPMD style, like the SIMD architectures, also makes it somewhat simpler to reason about the status of data structures and processors.

MIMD machines have emerged as the most commonly used general-purpose parallel computers, and are available in a variety of configurations. Both shared and distributed memory machines are available, as are mixed architectures where small numbers of processors are grouped together as a shared memory symmetric multiprocessor, and these SMPs are linked together in a distributed memory fashion.

Granularity

When discussing parallel architectures, the term *granularity* is often used to refer to the relative number and complexity of the processors. A *fine-grained machine* typically consists of a relatively large number of small, simple processors (in terms of local memory and computational power), while a *coarse-grained machine* typically consists of relatively few processors, each of which is large and powerful. Fine-grained machines typically fall into the SIMD category, where all processors operate in lockstep fashion (i.e., synchronously) on the contents of their own small, local, memory. Coarse-grained machines typically fall into the shared memory MIMD category, where processors operate asynchronously on the large, shared, memory. Medium-grained machines are typically built from commodity microprocessors, and are found in both distributed and shared memory models, almost always in MIMD designs.

For a variety of reasons, medium-grained machines currently dominate the parallel computer marketplace in terms of number of installations. Because they utilize commodity processors and have the ability to efficiently perform as general-purpose (parallel) machines, medium-grained machines tend to have cost/performance advantages over systems utilizing special-purpose processors. In addition, they can also exploit much of the software written for their component processors. Fine-grained machines are difficult to use as general-purpose computers because it is often difficult to determine how to efficiently distribute the work to such simple processors. However, fine-grained machines can be quite effective in tasks such as image processing or pattern matching.

By analogy, one can also use the granularity terminology to describe data and algorithms. For example, a database is a coarse-grained view of data, while considering the individual records in the database is a fine-grained view of the same data.

Interconnection Networks

In this section, we discuss interconnection networks that are used for communication among processors in a distributed memory machine. First, we define some terminology. The *degree of processor* P is the number of other processors that P is directly connected to via bi-directional communication links. The *degree of the network* is the maximum degree of any processor in the network. The *distance* between two processors is the number of communication links on a shortest path between the processors. The *communication diameter* of the network is the maximum, over all pairs of processors, of the distance between the processors. The *bisection bandwidth* of the network corresponds to the minimum number of communication links that need to be removed (or cut) in order to partition the network into two pieces, each with the same number of processors. Goals for interconnection networks include minimizing the degree of the processors (to minimize the cost of building a processor), minimizing the communication diameter (to minimize the communication time for any single message), and maximizing the bisection bandwidth (to minimize contention when many messages are being sent concurrently). Unfortunately, these design goals are in conflict. Other important design goals include simplicity (to reduce the design costs for the hardware and software) and scalability (so that similar machines, with a range of sizes, can be produced).

Before defining some network models (i.e., distributed memory machines characterized by their interconnection networks), we briefly discuss the *parallel random access machine (PRAM)*, which is an idealized parallel model of computation, with a unit-time communication diameter. The PRAM is a shared memory machine that consists of a set of identical processors, where all processors have unit-time access to any memory location. The appeal of a PRAM is that one can ignore issues of communication when designing algorithms, focusing instead on obtaining the maximum parallelism possible in order to minimize the running time necessary to solve a given problem. The PRAM model typically assumes a SIMD strategy, so that operations are performed synchronously. If multiple processors try to simultaneously read or write from the same memory location, then a memory conflict occurs. There are several variations of the PRAM model targeted at handling these conflicts, ranging from the Exclusive Read Exclusive Write (EREW) model, which prohibits all such conflicts, to Concurrent Read Concurrent Write (CRCW) models, which have various ways of resolving the effects of simultaneous writes. One popular intermediate model is the concurrent read exclusive write (CREW) PRAM, in which there may be concurrent reads to a memory location, but not concurrent writes. For example, a classroom is usually conducted in a CREW manner. In the classroom, even if several students are writing simultaneously on the blackboard, they are doing so in different locations, and hence there are no write conflicts.

The PRAM does not use a regular interconnection scheme for communication and the unit-time memory access requirement is not scalable (i.e., it is not realistic for a large number of processors and memory). However, in creating parallel programs, it is sometimes useful to describe a PRAM algorithm and then either perform a stepwise simulation of every PRAM operation on the target machine, or perform a higher-level simulation by using global operations. In such settings, it is often useful to design the algorithm for a powerful CRCW PRAM model, since often the CRCW PRAM can solve a problem faster or more naturally than an EREW PRAM. Since one is not trying to construct an actual PRAM, objections to the difficulty of implementing CRCW are not relevant; rather, having a simpler and/or faster algorithm is the dominant consideration.

In the remainder of this section, several specific interconnection networks are defined. See Figure 3 for illustrations of these. The networks defined in this section are among the most commonly utilized networks. However, additional networks have appeared in both the literature and in real machines, and variations of the basic networks described here are numerous.

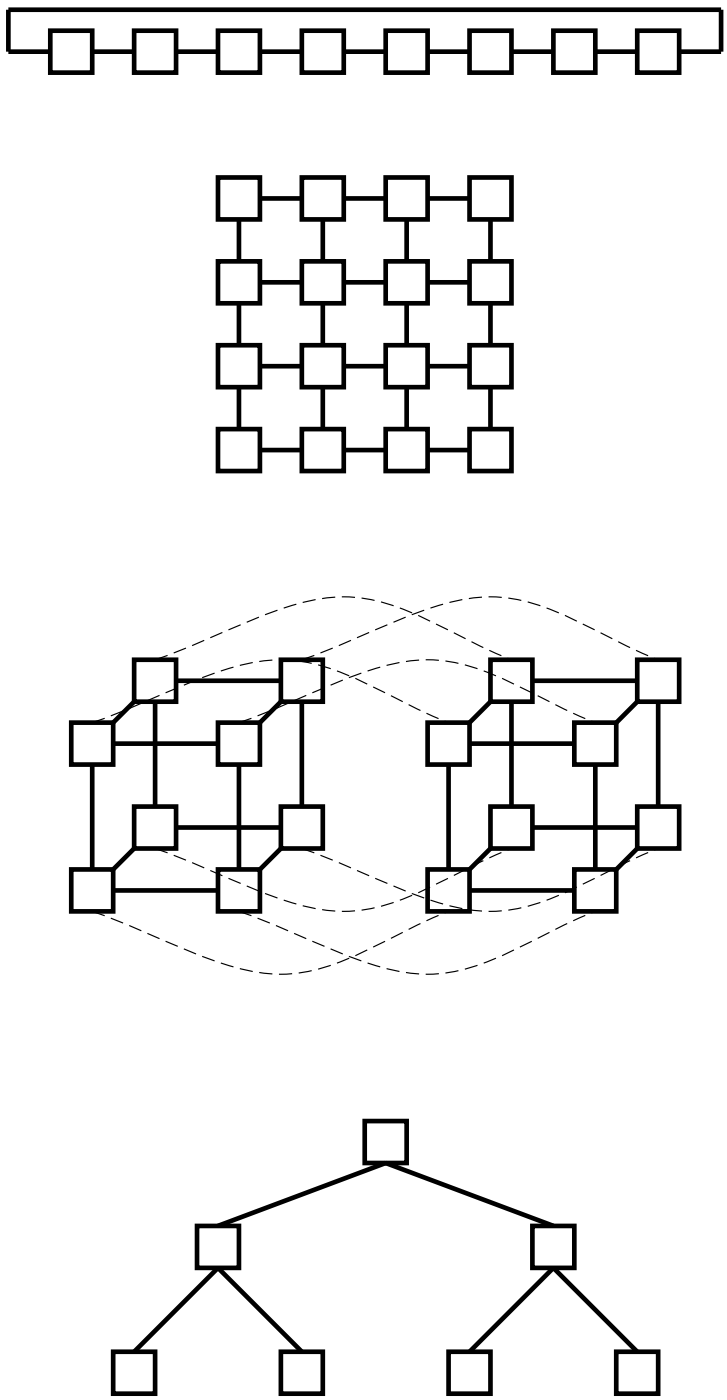


Figure 3: Sample interconnection networks (from top to bottom): ring, mesh, hypercube, and tree.

Ring

In a *ring* network, the n processors are connected in a circular fashion so that processor P_i is directly connected to processors P_{i-1} and P_{i+1} (the indices are computed modulo n, so that processors P_0 and P_{n-1} are connected). While the degree of the network is only 2, the communication diameter is $\lfloor n/2 \rfloor$, which is quite high. The bisection bandwidth is only 2, which is quite low.

Meshes and Tori

The n processors of a *2-dimensional square mesh* network are configured so that an interior processor $P_{i,j}$ is connected to its four neighbors, processors $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$, and $P_{i,j+1}$. The four corner processors are each connected to their 2 neighbors, while the remaining processors that are on the edge of the mesh are each connected to 3 neighbors. So, by increasing the degree of the network to 4, as compared to the degree 2 of the ring, the communication diameter of the network is reduced to $2(\sqrt{n} - 1)$, and the bisection bandwidth is increased to $\sqrt{n}$. The diameter is further reduced, to $2\lfloor \sqrt{n}/2 \rfloor$, and the bisection bandwidth is increased, to $2\sqrt{n}$, in a *2-dimensional torus*, which has all the connections of the 2-dimensional mesh plus connections between the first and last processors in each row and column. Meshes and tori of higher dimensions can be constructed, where the degree of a d-dimensional mesh or torus is $2d$, and, when n is a perfect d^{th} power, the diameter is either $d(n^{1/d} - 1)$ or $d\lfloor n^{1/d}/2 \rfloor$, respectively, and the bisection bandwidth is either $n^{(d-1)/d}$ or $2n^{(d-1)/d}$, respectively. Notice that the ring is a 1-dimensional torus.

For a 2-dimensional mesh, and similarly for higher-dimensional meshes, the mesh can be rectangular, instead of square. This allows a great deal of flexibility in selecting the size of the mesh, and the same flexibility is available for tori as well.

Hypercube

A *hypercube* with n processors, where n is an integral power of 2, has the processors indexed by the integers $\{0, \ldots, n-1\}$. Viewing each integer in this range as a $(\log_2 n)$-bit string, two processors are directly connected if and only if their indices differ by exactly one bit. Some advantages of a hypercube are that the communication diameter is only $\log_2 n$ and the bisection bandwidth is $n/2$. A disadvantage of the hypercube is that the number of communication links needed by each processor grows as $\log_2 n$, unlike the fixed degree for processors in ring and mesh networks. This makes it difficult to manufacture reasonably generic hypercube processors that could scale to extremely large machines, though in practice this is not a concern because the cost of an extremely large machine would be prohibitive.

Tree

A *complete binary tree* of height k, $k \geq 0$ an integer, has $n = 2^{k+1} - 1$ processors. The root node is at level 0 and the 2^k leaves are at level k. Each processor at level $1, \ldots, k-1$ has two children and one parent, the root processor does not have a parent processor, and the leaves at level k do not have children processors. Notice that the degree of the network is 3 and that the communication diameter is $2k = 2\lfloor \log_2 n \rfloor$. One severe disadvantage of a tree is that when extensive communication occurs, all messages traveling from one side of the tree to the other must pass through the root, causing a bottleneck. This is because the bisection bandwidth is only 1. Fat trees, introduced by Leiserson [Leiserson, 1985], alleviate this problem by increasing the bandwidth of the communication links near the root. This increase can come from changing the nature of the links, or, more easily, by using parallel communication links. Other generalizations of binary trees include complete t-ary trees

of height k, where each processor at level $0, \ldots, k-1$ has t children. There are $(t^{k+1}-1)/(t-1)$ processors, the maximum degree is $t+1$, and the diameter is $2k = 2\lfloor \log_t n \rfloor$.

Designing Algorithms

Viewed from the highest level, many parallel algorithms are purely sequential, with the same overall structure as an algorithm designed for a more standard "serial" computer. That is, there may be an input and initialization phase, then a computational phase, and then an output and termination phase. The differences, however, are manifested within each phase. For example, during the computational phase, an efficient parallel algorithm may be inherently different from its efficient sequential counterpart.

For each of the phases of a parallel computation, it is often useful to think of operating on an entire structure simultaneously. This is a SIMD-style approach, but the operations may be quite complex. For example, one may want to update all entries in a matrix, tree, or database, and view this as a single (complex) operation. For a fine-grained machine, this might be implemented by having a single (or few) data item per processor, and then using a purely parallel algorithm for the operation. For example, suppose an $n \times n$ array A is stored on an $n \times n$ 2-dimensional torus, so that $A(i,j)$ is stored on processor $P_{i,j}$. Suppose one wants to replace each value $A(i,j)$ with the average of itself and the four neighbors $A(i-1,j)$, $A(i+1,j)$, $A(i,j-1)$ and $A(i,j+1)$, where the indices are computed modulo n (i.e., "neighbors" is in the torus sense). This average filtering can be accomplished by just shifting the array right, left, up, and down by one position in the torus, and having each processor average the four values received along with its initial value.

For a medium- or coarse-grained machine, operating on entire structures is most likely to be implemented by blending serial and parallel approaches. On such an architecture, each processor uses an efficient serial algorithm applied to the portion of the data in the processor, and communicates with other processors in order to exchange critical data. For example, suppose the $n \times n$ array of the previous paragraph is stored in a $p \times p$ torus, where p evenly divides n, so that $A(i,j)$ is stored in processor $P_{\lfloor ip/n \rfloor, \lfloor jp/n \rfloor}$. Then, to do the same average filtering on A, each processor $P_{k,l}$ still needs to communicate with its torus neighbors $P_{k\pm1,l}$, $P_{k,l\pm1}$, but now sends them either the leftmost or rightmost column of data, or the topmost or bottommost row. Once a processor receives its boundary set of data from its neighboring processors, it can then proceed serially through its subsquare of data and produce the desired results. To maximize efficiency, this can be performed by having each processor send the data needed by its neighbors, then perform the filtering on the part of the array that it contains that does not depend on data from the neighbors, and then finally perform the filtering on the elements that depend on the data from neighbors. Unfortunately, while this maximizes the possible overlap between communication and calculation, it also complicates the program because the order of computations within a processor needs to be rearranged.

Global Operations

To manipulate entire structures in one step, it is useful to have a collection of operations that perform such manipulations. These *global operations* may be very problem-dependent, but certain ones have been found to be widely useful. For example, the average filtering example above made use of shift operations to move an array around. *Broadcast* is another common global operation, used to send data from one processor to all other processors. Extensions of the broadcast operation include simultaneously performing a broadcast within every (predetermined and distinct) subset of processors. For example, suppose matrix A has been partitioned into submatrices allocated to different processors, and one needs to broadcast the first row of A so that if a processor contains any elements of column i then

it obtains the value of $A(1, i)$. In this situation, the more general form of a subset-based broadcast can be used.

Besides operating within subsets of processors, many global operations are defined in terms of a commutative, associative, semigroup operator $\otimes$. Examples of such semigroup operators include `minimum`, `maximum`, `or`, `and`, `sum`, and `product`. For example, suppose there is a set of values $V(i)$, $1 \leq i \leq n$, and the goal is to obtain the maximum of these values. Then $\otimes$ would represent maximum, and the operation of applying $\otimes$ to all n values is called *reduction*. If the value of the reduction is broadcast to all processors, then it is sometimes known as *report*. A more general form of the reduction operation involves labeled data items, i.e., each data item is embedded in a record that also contains a label, where at the end of the reduction operation the result of applying $\otimes$ to all values with the same label will be recorded in the record.

Global operations provide a useful way to describe major actions in parallel programs. Further, since several of these operations are widely useful, they are often made available in highly optimized implementations. The language APL provided a model for several of these operations, and some parallel versions of APL have appeared. Languages such as C* [Thinking Machines Corporation, 1991] and FORTRAN 90 [Brainerd, Goldberg, and Adams, 1990] also provide for some forms of global operations, as do message-passing systems such as MPI [Snir, Otto, Huss-Lederman, Walker, and Dongarra, 1995]. Reduction operations are so important that most parallelizing compilers detect them automatically, even if they have no explicit support for other global operations.

Besides broadcast, reduction, and shift, other important global operations include the following.

Sort: Let $X = \{x_0, x_1, \ldots, x_{n-1}\}$ be an ordered set such that $x_i < x_{i+1}$, for all $0 \leq i < n-1$. (That is, X is a subset of a linearly ordered data type.) Given that the n elements of X are arbitrarily distributed among a set of p processors, the sort operation will (re)arrange the members of X so that they are ordered with respect to the processors. That is, after sorting, elements $x_0, \ldots, x_{\lfloor n/p \rfloor}$ will be in the first processor, elements $x_{\lfloor n/p \rfloor + 1}, \ldots, x_{\lfloor 2n/p \rfloor}$ will be in the second processor, and so forth. Note that this assumes an ordering on the processors, as well as on the elements.

Merge: Suppose that sets D_1 and D_2 are subsets of some linearly ordered data type, and D_1 and D_2 are each distributed in an ordered fashion among disjoint sets of processors $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Then the merge operation combines D_1 and D_2 to yield a single sorted set stored in ordered fashion in the entire set of processors $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$.

Associative Read/Write: These operations start with a set of *master* records indexed by unique keys. In the associative read, each processor specifies a key and ends up with the data in the master record indexed by that key, if such a record exists, or else a flag indicating that there is no such record. In the associative write, each processor specifies a key and a value, and each master record is updated by applying $\otimes$ to all values sent to it. (Master records are generated for all keys written.)

These operations are extensions of the CRCW PRAM operations. They model a PRAM with associative memory and a powerful combining operation for concurrent writes. On most distributed memory machines the time to perform these more powerful operations is within a multiplicative constant of the time needed to simulate the usual concurrent read and concurrent write, and the use of the more powerful operations can result in significant algorithmic simplifications and speedups.

Compression: Compression moves data into a region of the machine where optimal interprocessor communication is possible. For example, compressing k items in a fine-grain two-dimensional mesh will move them to a $\sqrt{k} \times \sqrt{k}$ subsquare.

Parallel Prefix (Scan): Given a set of values a_i, $1 \leq i \leq n$, the *parallel prefix* computation determines $p_i = a_1 \otimes a_2 \otimes \cdots \otimes a_i$, for all i. This operation is available in APL, where it is called *scan*. Note that the hardware feature known as "fetch-and-op" implements a variant of parallel prefix, where "op" is $\otimes$ and the ordering of the processors is not required to be deterministic.

All-to-all broadcast: Given data $D(i)$ in processor i, every processor receives a copy of $D(i)$, for all i.

All-to-all personalized communication: Every processor P_i has a data item $D(i, j)$ that is sent to processor P_j, for all $i \neq j$.

Example: Maximal Point Problem

As an example of the use of global operations, consider the following problem from computational geometry. Let S be a finite set of planar (i.e., 2-dimensional) points. A point $p = (p_x, p_y)$ in S is a *maximal point* of S if $p_x > q_x$ or $p_y > q_y$, for every point $(q_x, q_y) \neq p$ in S. The *maximal point problem* is to determine all maximal points of S. See Figure 4. The following parallel algorithm for the maximal point problem was apparently first noted by Atallah and Goodrich [Atallah and Goodrich, 1986].

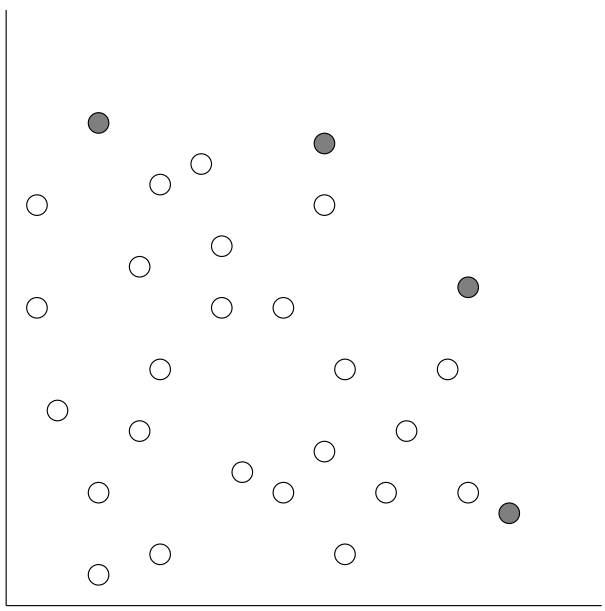


Figure 4: The maximal points of the set are shaded.

1. Sort the n planar points in reverse order by x-coordinate, with ties broken by reverse order by y-coordinate. Let (i_x, i_y) denote the coordinates of the ith point after the sort is complete. Therefore, after sorting, the points will be ordered so that if $i < j$ then either $i_x > j_x$ or $i_x = j_x$ and $i_y > j_y$.

2. Use parallel prefix on the i_y values, where the operation $\otimes$ is taken to be maximum. The resulting values $\{L_i\}$ are such that L_i is the largest y-coordinate of any point with index less than i.

3. The point (i_x, i_y) is an extreme point if and only if $i_y > L_i$.

The running time $T(n)$ of this algorithm is given by

$$T(n) = Sort(n) + Prefix(n) + O(1) \ , \tag{1}$$

where $Sort(n)$ is the time to sort n items and $Prefix(n)$ is the time to perform parallel prefix. On all parallel architectures known to the authors, $Prefix(n) = O(Sort(n))$, and hence on such machines the time of the algorithm is $\Theta(Sort(n))$. It is worth noting that for the sequential model, [Kung, Luccio, and Preparata, 1975] have shown that the problem of determining maximal points is as hard as sorting.

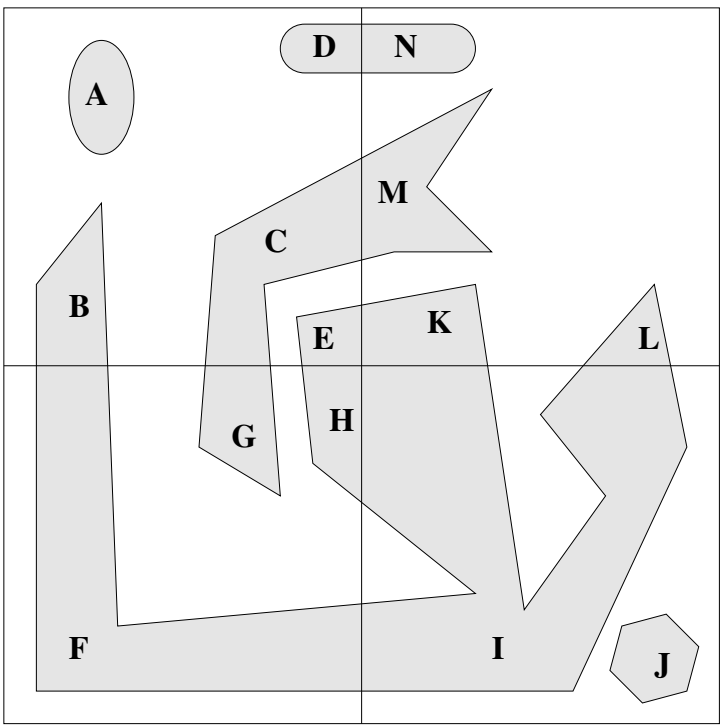
Divide-and-Conquer

Divide-and-conquer is a powerful algorithmic paradigm that exploits the repeated subdivision of problems and data into smaller, similar problems. It is quite useful in parallel computation because the logical subdivisions into subproblems can correspond to physical decomposition among processors, where eventually the problem is broken into subproblems that are each contained within a single processor. These small subproblems are typically solved by an efficient sequential algorithm within each processor.

As an example, consider the problem of labeling the figures of a black/white image, where the interpretation is that of black objects on a white background. Two black pixels are defined to be *adjacent* if they are vertical or horizontal neighbors, and *connected* if there is a path of adjacent black pixels between them. A *figure* (i.e., *connected component*) is defined to be a maximally connected set of black pixels in the image. The figures of an image are said to be *labeled* if every black pixel in the image has a label, with two black pixels having the same label if and only if they are in the same figure.

We utilize a generic parallel divide-and-conquer solution for this problem, given, for example, in [Miller and Stout, 1996]. Suppose that the $n \times n$ image has been divided into p subimages, as square as possible, and distributed one subimage per processor. Each processor labels the subimage it contains, using whatever serial algorithm is best and using labels that are unique to the processor (so that no two different figures can accidentally get the same label). For example, often the label used is a concatenation of the row and column coordinates of one of the pixels in the figure. Notice that so as long as the global row and column coordinates are used, the labels will be unique. After this step, the only figures that could have an incorrect global label are those that lie in two or more subimages, and any such figures must have a pixel on the border of each subimage it is in (see Figure 5). To resolve these labels, a record is prepared for each black pixel on the border of a subimage, where the record contains information about the pixel's position in the image, and its current label. There are far fewer such records than there are pixels in the original image, yet they contain all of the information needed to determine the proper global labels for figures crossing subimages. The problem of reconciling the local labels may itself be solved via divide-and-conquer, repeatedly merging results from adjacent regions, or may be solved via other approaches. Once these labels have been resolved, information is sent back to the processors generating the records, informing them of the proper final label.

One useful feature of many of the networks described in the section on Interconnection Networks is that they can be divided into similar subnetworks, in a manner that matches the divide-and-conquer paradigm. For example, if the component labeling algorithm just described were performed on a mesh computer, then each subregion of the image would correspond to a subsquare of the mesh. As another example, consider an implementation of quicksort on a hypercube. Suppose a pivot is chosen and that the data is partitioned into items smaller than the pivot and items larger than the pivot. Further, suppose that the hypercube is logically partitioned into two subcubes, where all of the small items are moved into one subcube and all of the large items are moved into the other subcube. Now, the quicksort routine may proceed recursively within each subcube. Because the recursive divide-and-conquer occurs within subcubes, all of the communication will occur within the subcubes and not cause contention with the other subcube.



The 14 labels shown were generated after each quadrant performed its own, local, labeling algorithm. While the labels are unique, they need to be resolved globally. Notice that once the labels are resolved (not shown), the image will have only 5 unique labels, one corresponding to each of the 5 figures.

Figure 5: Divide-and-Conquer for Labeling Figures

Master-Slave

One algorithmic paradigm based on real-world organization paradigms is the master-slave (sometimes referred to as manager-worker) paradigm. In this approach, one processor acts as the master, directing all of the other slave processors. For example, many branch-and-bound approaches to optimization problems keep track of the best solution found so far, as well as a list of subproblems that need to be explored. In a master-slave implementation, the master maintains both of these items and is responsible for parceling out the subproblems to the slaves. The slaves are responsible for processing the subproblems and reporting the result to the master (which will determine if it is the current best solution), reporting new subproblems that need to be explored to the master, and notifying the master when it is free to work on a new subproblem. There are many variations on this theme, but the basic idea is that one processor is responsible for overall coordination, and the other processors are responsible for solving assigned subproblems. Note that this is a variant of the SPMD style of programming, in that there are two programs needed, rather than just one.

Pipelining and Systolic Algorithms

Another common parallel algorithmic technique is based on models that resemble an assembly line. A large problem, such as analyzing a number of images, may be broken into a sequence of steps that must be performed on each image (e.g., filtering, labeling, scene analysis). If one had three processors, and if each step takes about the same amount of time, one could start the first image on the first processor that does the filtering. Then the first image is passed on to the next processor for labeling, while the first processor starts filtering the second image. In the third time step, the initial image is at the third processor for scene analysis, the second image is at the second processor for labeling, and the third image is at the first processor for filtering. This form of processing is called *pipelining*, and it maps naturally to a parallel computer configured as a linear array (i.e., a 1-dimensional mesh or, equivalently, a ring without the wraparound connection).

This simple scenario can be extended in many ways. For example, as in a real assembly line, the processors need not all be identical, and may be optimized for their task. Also, if some task takes longer to perform than others, then more than one processor can be assigned to it. Finally, the flow may not be a simple line. For example, an automobile assembly process may have one line working on the chassis, while a different line is working on the engine, and eventually these two lines are merged. Such generalized pipelining is called *systolic processing*. For example, some matrix and image-processing operations can be performed in a two-dimensional systolic manner (see [Ullman, 1984]).

Mappings

Often, a problem has a natural structure to be exploited for parallelism, and this needs to be mapped onto a target machine. Several examples follow.

- The average filtering problem, discussed in the section on Designing Algorithms, has a natural array structure that can easily be mapped onto a mesh computer. If, however, one had the same problem, but a tree computer, then the mapping might be much more complicated.
- Some artificial intelligence paradigms exploit a blackboard-like communication mechanism that naturally maps onto a shared memory machine. However, a blackboard-like approach is more difficult to map onto a distributed-memory machine.

- Finite-element decompositions have a natural structure whereby calculations at each grid point depend only on values at adjacent points. A finite-element approach is frequently used to model automobiles, airplanes, and rocket exhaust, to name a few. However, the irregular (and perhaps dynamic) structure of such decompositions might need to be mapped onto a target parallel architecture that bears little resemblance to the finite-element grid.

- A more traditional example consists of porting a parallel algorithm designed for one parallel architecture onto another parallel architecture.

In all of these examples, one starts with a source structure that needs to be mapped onto a target machine. The goal is to map the source structure onto the target architecture so that calculation and communication steps on the source structure can be efficiently performed by the target architecture. Usually, the most critical aspect is to map the calculations of the source structure onto the processors of the target machine, so that each processor performs the same amount of calculations. For example, if the source is an array, and each position of the array represents calculations that need to be performed, then one tries to map the array onto the machine so that all processors contain the same number of entries. If the source model is a shared-memory paradigm with agents reading from a blackboard, then one would map the agents to processors, trying to balance the computational work.

Besides trying to balance the computational load, one must also try to minimize the time spent on communication. The approaches used for these mappings depend on the source structure and target architecture, and some of the more widely used approaches are discussed in the following subsections.

Simulating Shared Memory

If the source structure is a shared memory model, and the target architecture is a distributed memory machine, then besides mapping the calculations of the source onto the processors of the target, one must also map the shared memory of the source onto the distributed memory of the target.

To map the memory onto the target machine, suppose that there are memory locations $0 \ldots n-1$ in the source structure, and p processors in the target. Typically one would map locations $0 \ldots \lfloor n/p - 1 \rfloor$ to processor 0 of the target machine, locations $\lfloor n/p \rfloor \ldots \lfloor 2n/p - 1 \rfloor$ to processor 1, and so forth. Such a simple mapping balances the amount of memory being simulated by each target processor, and makes it easy to determine where data is located. For example, if a target processor needs to read from shared memory location i, it sends a message to target processor $\lfloor ip/n \rfloor$ asking for the contents of simulated shared memory location i.

Unfortunately, some shared memory algorithms utilize certain memory locations far more often than others, which can cause bottlenecks in terms of getting data in and out of processors holding the popular locations. If popular memory locations form contiguous blocks, then this congestion can be alleviated by stripping (mapping memory location i to processor $i \bmod p$) or other mappings. Replication (having copies of frequently read locations in more than one processor) or adaptive mapping (dynamically moving simulated memory locations from heavily loaded processors to lightly loaded ones) are occasionally employed to relieve congestion, but such techniques are more complicated and involve additional overhead.

Simulating Distributed Memory

It is often useful to view distributed memory machines as graphs. Processors in the machine are represented by vertices of the graph, and communication links in the machine are represented by edges in the graph. Similarly, it is often convenient to view the structure of a problem as a graph, where vertices represent work that needs to be performed, and edges represent values that need to be communicated

in order to perform the desired work. For example, in a finite-element decomposition, the vertices of a decomposition might represent calculations that need to be performed, while the edges correspond to flow of data. That is, in a typical finite-element problem, if there is an edge from vertex p to vertex q, then the value of q at time t depends on the values of q and p at time $t-1$. (Most finite-element decompositions are symmetric, so that p at time t would also depend on q at time $t-1$.) Questions about mapping the structure of a problem onto a target architecture can then be answered by considering various operations on the related graphs.

The best situation is when the graph representing the structure of a problem is a subgraph of the graph representing the target architecture. For example, if the structure of a problem was represented as a connected string of p vertices and the target architecture was a ring of p processors, then the mapping of the problem onto the architecture would be straightforward and efficient. In graph terms, this is described through the notion of embedding. An *embedding* of an undirected graph $G = (V, E)$ (i.e., G has vertex set V and edges E) into an undirected graph $G' = (V', E')$ is a mapping ϕ of V into V' such that

- every pair of distinct vertices $u, v \in V$, map to distinct vertices $\phi(u)$, $\phi(v) \in V'$, and
- for every edge $\{u, v\} \in E$, $\{\phi(u), \phi(v)\}$ is an edge in E'.

Let G represent the graph corresponding to the structure of a problem (i.e., the *source structure*) and let G' represent the graph corresponding to the target architecture. Notice that if there is an embedding of G into G', then values that need to be communicated may be transmitted by a single communication step in the target architecture represented by G'. The fact that embeddings map distinct vertices of G to distinct vertices of G' ensures that a single calculation step for the problem can be simulated in a single calculation step of the target architecture.

One reason that hypercube computers were quite popular is that many graphs can be embedded into the hypercube (graph). An embedding of the one-dimensional ring of size 2^d into a d-dimensional hypercube is called a *d-dimensional Gray code*. In other words, if $\{0,1\}^d$ denotes the set of all d-bit binary strings, then the d-dimensional Gray code G_d is a 1-1 map of $0 \ldots 2^d - 1$ onto $\{0,1\}^d$, such that $G_d(j)$ and $G_d((j+1) \bmod 2^d)$ differ by a single bit, for $0 \le j \le 2^d - 1$. The most common Gray codes, called *reflected binary* Gray codes, are recursively defined as follows: $\mathcal{G}_d$ is a 1–1 mapping from $\{0, 1, \ldots, 2^d - 1\}$ onto $\{0,1\}^d$, given by $\mathcal{G}_1(0) = 0$, $\mathcal{G}_1(1) = 1$, and for $d \ge 2$,

$$\mathcal{G}_d(x) \quad = \quad \begin{cases} 0\mathcal{G}_{d-1}(x) & 0 \le x \le 2^{d-1} - 1 \\ 1\mathcal{G}_{d-1}(2^d - 1 - x) & 2^{d-1} \le x \le 2^d - 1. \end{cases} \tag{2}$$

Alternatively, the same Gray code can be defined in a non-recursive fashion as $\mathcal{G}_d(x) = x \oplus \lfloor x/2 \rfloor$, where x and $\lfloor x/2 \rfloor$ are interpreted as d-bit strings. Further, the inverse of the reflected binary Gray code can be determined by

$$\mathcal{G}_d^{-1}(y_0 \ldots y_{d-1}) = x_0 \ldots x_{d-1}, \tag{3}$$

where $x_{d-1} = y_{d-1}$, and $x_i = y_{d-1} \oplus \cdots \oplus y_i$ for $0 \le i < d-1$.

Meshes can also be embedded into hypercubes. Let M be a d-dimensional mesh of size $m_1 \times m_2 \times \cdots \times m_d$, and let $r = \sum_{i=1}^{d} \lceil \log_2 m_i \rceil$. Then M can be embedded into the hypercube of size 2^r. To see this, let $r_i = \lceil \log_2 m_i \rceil$, for $1 \le i \le d$. Let ϕ be the mapping of mesh node $(a_1, \ldots, a_d)$ to the hypercube node which has as its label the concatenation $G_{r_1}(a_1) \cdot \ldots \cdot G_{r_d}(a_d)$, where G_{r_i} denotes any r_i-bit Gray code. Then ϕ is an embedding. Wrapped dimensions can be handled using reflected Gray codes rather than arbitrary ones. (A mesh M is *wrapped* in dimension j if, in addition to the normal mesh adjacencies, vertices with indices of the form $(a_1, \ldots, a_{j-1}, 0, a_{j+1}, \ldots, a_d)$ and $(a_1, \ldots, a_{j-1}, m_j - 1, a_{j+1}, \ldots, a_d)$ are adjacent. A torus is a mesh wrapped in all dimensions.) If

dimension j is wrapped and m_j is an integral power of 2, then the mapping ϕ suffices. If dimension j is wrapped and m_j is even, but not an integral power of 2, then to ensure that the first and last nodes in dimension j are mapped to adjacent hypercube nodes, use ϕ, but replace $G_{r_j}(a_j)$ with

$$\begin{cases} \mathcal{G}_{r_j}(a_j) & \text{if } 0 \leq a_j \leq m_j/2 - 1 \\ \mathcal{G}_{r_j}(a_j + 2^{r_j} - m_j) & \text{if } m_j/2 \leq a_j \leq m_j - 1, \end{cases} \tag{4}$$

where $\mathcal{G}_{r_j}$ is the r_j-bit reflected binary Gray code. This construction ensures that $\mathcal{G}_{r_j}(m_j/2 - 1)$ and $\mathcal{G}_{r_j}(2^{r_j} - m_j/2)$ differ by exactly one bit (the highest order one), which in turns ensures that the mapping takes mesh nodes neighboring in dimension j to hypercube neighbors.

Any tree T can be embedded into a $(|T|-1)$-dimensional hypercube, where $|T|$ denotes the number of vertices in T, but this result is of little use since the target hypercube is exponentially larger than the source tree. Often one can map the tree into a more reasonably sized hypercube, but it is a difficult problem to determine the minimum dimension needed, and there are numerous papers on the subject.

In general, however, one cannot embed the source structure into the target architecture. For example, a complete binary tree of height 2, which contains 7 processors, cannot be embedded into a ring of any size. Therefore, one must consider weaker mappings, which allow for the possibility that the target machine has fewer processors than the source, and does not contain the communication links of the source. A *weak embedding* of a directed source graph $G = (V, E)$ into a directed target graph $G' = (V', E')$ consists of

- a map ϕ_v of V into V', and
- a map ϕ_e of E onto *paths* in G', such that if $(u, v) \in E$ then $\phi_e((u, v))$ is a path from $\phi_v(u)$ to $\phi_v(v)$.

(Note that if G is undirected, each edge becomes two directed edges that may be mapped to different paths in G'. Most machines that are based on meshes, tori, or hypercubes have the property that a message from processor P to processor Q may not necessarily follow the same path as a message sent from processor Q to processor P, if P and Q are not adjacent.) The map ϕ_v shows how computations are mapped from the source onto the target, and the map ϕ_e shows the communication paths that will be used in the target.

There are several measures that are often used to describe the quality of a weak embedding (ϕ_v, ϕ_e) of G into G', including the following.

Processor Load: the maximum, over all vertices $v' \in V'$, of the number of vertices in V mapped onto v' by ϕ_v. Note that if all vertices of the source structure represent the same amount of computation, then the processor load is the maximum computational load by any processor in the target machine. The goal is to make the processor load as close as possible to $|V|/|V'|$. If vertices do not all represent the same amount of work, then one should use labeled vertices, where the label represents the amount of work, and try to minimize the maximum, over all vertices $v' \in V'$, of the sum of the labels of the vertices mapped onto v'.

Link Load (Link Congestion): the maximum, over all edges $(u', v') \in E'$, of the number of edges $(u, v) \in E$ such that (u', v') is part of the path $\phi_e((u, v))$. If all edges of the source structure represent the same amount of communication, then the link load represents the maximum amount of communication contending for a single communication link in the target architecture. As for processor load, if edges do not represent the same amount of communication, then weights should be balanced instead.

Dilation: the maximum, over all edges $(u, v) \in E$, of the path length of $\phi_e((u, v))$. The dilation represents the longest delay that would be needed to simulate a single communication step along an edge in the source, if that was the only communication being performed.

Expansion: the ratio of the number of vertices of G' divided by the number of vertices of G. As was noted in the example of trees embedding into hypercubes, large expansion is impractical. In practice, usually the real target machine has far fewer processors than the idealized source structure, so expansion is not a concern.

In some machines, dilation is an important measure of communication delay, but in most modern general-purpose machines it is far less important because each message has a relatively large start-up time that may be a few orders of magnitude larger than the time per link traversed. Link contention may still be a problem in such machines, but some solve this by increasing the bandwidth on links that would have heavy contention. For example, as noted earlier, *fat-trees* [Leiserson, 1985] add bandwidth near the root to avoid the bottlenecks inherent in a tree architecture. This increases the bisection bandwidth, which reduces the link contention for communication that poorly matches the basic tree structure.

For machines with very large message start-up times, often the number of messages needed becomes a dominant communication issue. In such a machine, one may merely try to balance calculation load and minimize the number of messages each processor needs to send, ignoring other communication effects. The number of messages that a processor needs to send can be easily determined by noting that processors p and q communicate if there are adjacent vertices u and v in the source structure such that ϕ_v maps u to p and v to q.

For many graphs that cannot be embedded into a hypercube, there are nonetheless useful weak embeddings. For example, keeping the expansion as close to 1 as is possible (given the restriction that a hypercube has a power of 2 processors), one can map the complete binary tree onto the hypercube with unit link congestion, dilation two, and unit processor contention. See, for example, [Leighton, 1992].

In general, however, finding an optimal weak embedding for a given source and target is an NP-complete problem. This problem, sometimes known as the *mapping problem*, is often solved by various heuristics. This is particularly true when the source structure is given by a finite-element decomposition or other approximation schemes for real entities, for in such cases the sources are often quite large and irregular. Fortunately, the fact that such sources often have an underlying geometric basis makes it easier to find fairly good mappings rather quickly.

For example, suppose the source structure is an irregular grid representing the surface of a 3-dimensional airplane, and the target machine is a 2-dimensional mesh. One might first project the airplane onto the x-y plane, ignoring the z-coordinates. Then one might locate a median x-coordinate, call it $\bar{x}$, where half of the plane's vertices lie to the left of $\bar{x}$ and half to the right. The vertices may then be mapped so that those that lie to the left of $\bar{x}$ are mapped onto the left half of the target machine, and those vertices that lie to the right of $\bar{x}$ are mapped to the right half of the target. In the left half of the target, one might locate the median y-coordinate, denoted $\bar{y}$, of the points mapped to that half, and map the points above $\bar{y}$ to the top-left quadrant of the target, and map points below $\bar{y}$ to the bottom-left. On the right half a similar operation would be performed for the points mapped to that side. Continuing in this recursive, divide-and-conquer manner, eventually the target machine would have been subdivided down into single processors, at which point the mapping would have been determined. This mapping is fairly straightforward, balances the processor load, and roughly keeps points adjacent in the grid near to each other in the target machine, and hence it does a reasonable approximation of minimizing communication time.

Note that if message start-up time is very high then this probably would not minimize the number of messages sent by each processor, and in such a situation it may be better to partition the plane by cutting along only, say, the x-axis at each step. Each processor would end up with a cross-sectional

slab, with all of the source vertices in given range of x-coordinates. If grid edges are not longer than the width of such a slab, then each processor would have to send messages to only two processors, namely the processor with the slab to the left and the processor with the slab to the right.

Other complications can arise because the nodes or edges of such sources may not all represent the same amount of computation or calculation, respectively, in which case weighted mappings are appropriate. A variety of programs are available that perform such mappings, and over time the quality of the mapping achieved, and the time to achieve it, has significantly improved. For irregular source structures, such packages are generally superior to what one would achieve without considerable effort.

A more serious complication is that the natural source structure may be dynamic, adding nodes or edges over time. In such situations one often needs to dynamically adjust the mapping to keep the computational load balanced and keep communication minimal. This introduces additional overhead, which one must weigh against the costs of not adjusting the imbalance. Often the dynamical remappings are made incrementally, moving only a little of the data to correct the worst imbalances. Deciding how often to check for imbalance, and how much to move, typically depends quite heavily on the problem being solved.

Research Issues and Summary

The development of parallel algorithms and efficient parallel programs lags significantly behind that of standard serial computers. This is perhaps due to the fact that only recently have parallel computers become commercially available. Therefore, parallel computing is in a rapidly growing phase, with important research and development still needed in almost all areas. Extensive theoretical and practical work continues in discovering parallel programming paradigms, in developing a wide range of efficient parallel algorithms, in developing ways to describe and manage parallelism, in developing techniques to automatically detect parallelism, and in developing libraries of parallel routines.

Another factor that has hindered parallel algorithm development is the fact that there are many different parallel computing models. As noted earlier, architectural differences can significantly affect the efficiency of an algorithm, and hence parallel algorithms have traditionally been tied to specific parallel models. One advance is that various hardware and software approaches are being developed to help hide some of the architectural differences. Thus, one may have, say, a distributed memory machine, but have a software system that allows the programmer to view it as a shared memory machine. While it is true that a programmer will usually only be able to achieve the highest performance by directly optimizing the code for a target machine, in many cases acceptable performance can be achieved without tying the code to excessive details of an architecture. This, then, allows code to be ported to a variety of machines, which encourages code development. In the past, extensive code revision was needed every time the code was ported to a new parallel machine, and this strongly discouraged many users who did not want to plan for an unending parade of changes.

Another factor that has limited parallel algorithm development is that most computer scientists were not trained in parallel computing. As the field matures, more courses will incorporate parallel computing and the situation will improve. However, many thousands of small shared-memory systems have already been purchased, often to be used as departmental or corporate compute servers. Unfortunately, do to the dearth of parallel programmers, many of these systems are used only to run concurrent serial programs, or to run turnkey parallel programs (such as databases). There is a serious need for professionals who are able to utilize the full power of these parallel machines, since there are a great many problems which are beyond the power of single processors.

Defining Terms

Distributed memory: Each processor only has access to only its own private (local) memory, and communicates with other processors via messages.
Divide-and-conquer: A programming paradigm whereby large problems are solved by decomposing them into smaller, yet similar, problems.
Global operations: Parallel operations that affect system-wide data structures.
Interconnection network: The communication system that links together all of the processors and memory of a parallel machine.
Master-slave: A parallel programming paradigm whereby a problem is broken into a collection of smaller problems, with a master processor keeping track of the subproblems and assigning them to the slave processors.
Parallel Random Access Machine (PRAM): A theoretical shared-memory model, where typically the processors all execute the same instruction synchronously, and access to any memory location occurs in unit time.
Pipelining: A parallel programming paradigm that abstracts the notion of an assembly line. A task is broken into a sequence of fixed subtasks corresponding to the stations of an assembly line. A series of similar tasks is solved by starting one task through the subtask sequence, then starting the next task through as soon as the previous task has finished its first subtask. At any point in time, several tasks are in various stages of completion.
Shared memory: All processors have the same global image of (and access to) all of the memory.
Single Program Multiple Data (SPMD): The dominant style of parallel programming, where all of the processors utilize the same program, though each has its own data.

References

[Akl and Lyon, 1993] Akl, S.G. and Lyon, K.A. 1993. *Parallel Computational Geometry*, Prentice-Hall, Englewood Cliffs, NJ.

[Atallah and Goodrich, 1986] Atallah, M.J. and Goodrich, M.T. 1986. Efficient parallel solutions to geometric problems, *Journal of Parallel and Distributed Computing* **3** (1986), pp. 492-507.

[Brainerd, Goldberg, and Adams, 1990] Brainerd, W.S., Goldberg, C., and Adams, J.C. 1990. *Programmers Guide to FORTRAN 90*, McGraw-Hill Book Company, New York, NY.

[Flynn, 1966] Flynn, M.J. 1966. Very high-speed computing systems, *Proc. of the IEEE*, 54(12):1901-1909.

[Flynn, 1972] Flynn, M.J. 1972. Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, C-21:948-960.

[JáJá, 1992] JáJá, J. 1992. *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.

[Kung, Luccio, and Preparata, 1975] Kung, H.T., Luccio, F., and Preparata, F.P. 1975. On finding the maxima of a set of vectors, *Journal of the ACM* 22(4):469-476.

[Leighton, 1992] Leighton, F.T. 1992. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA.

[Leiserson, 1985] Leiserson, C.E. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing, *IEEE Transactions on Computers*, C-34(10):892-901.

[Li and Stout, 1991] Li, H. and Stout, Q.F. 1991. *Reconfigurable Massively Parallel Computers*, Prentice Hall, Englewood Cliffs, NJ.

[Miller and Stout, 1996] Miller, R. and Stout, Q.F. 1996. *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, The MIT Press, Cambridge, MA.

[Quinn, 1994] Quinn, M.J. 1994. *Parallel Computing Theory and Practice*, McGraw-Hill, Inc., New York, NY.

[Reif, 1993] Reif, J., ed. 1993. *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA.

[Snir, Otto, Huss-Lederman, Walker, and Dongarra, 1995] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., and Dongarra, J. 1995. *MPI: The Complete Reference*, The MIT Press, Cambridge, MA.

[Thinking Machines Corporation, 1991] Thinking Machines Corporation. 1991. *C* Programming Guide*, Version 6.0.2, Cambridge, MA.

[Ullman, 1984] Ullman, J.D. 1984. *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD.

Further Information

A good introduction to parallel computing at the undergraduate level is *Parallel Computing: Theory and Practice* by Michael J. Quinn. This book provides a nice introduction to parallel computing, including parallel algorithms, parallel architectures, and parallel programming languages. *Parallel Algorithms for Regular Architectures: Meshes and Pyramids* by Russ Miller and Quentin F. Stout focuses on fundamental algorithms and paradigms for fine-grained machines. It advocates an approach of designing algorithms in terms of fundamental data movement operations, including sorting, concurrent read, and concurrent write. Such an approach allows one to port algorithms in an efficient manner between architectures. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* is a comprehensive book by F. Thomson Leighton that also focuses on fine-grained algorithms for several traditional interconnection networks. Finally, for the reader interested in algorithms for the PRAM, *An Introduction to Parallel Algorithms* by J. JáJá covers fundamental algorithms in geometry, graph theory, and string matching. It also includes a chapter on randomized algorithms.

There are several professional societies that sponsor conferences, publish books, and publish journals in the area of parallel algorithms. These include the *Association for Computing Machinery (ACM)*, which can be found at http://www.acm.org, *The Institute for Electrical and Electronics Engineers, Inc. (IEEE)*, which can be found at http://www.ieee.org, and the *Society for Industrial and Applied Mathematics (SIAM)*, which can be found at http://www.siam.org.

Since parallel computing has become so pervasive, most computer science journals cover work concerned with parallel and distributed systems. For example, one would expect a journal on programming languages to publish articles on languages for shared-memory machines, distributed memory machines, networks of workstations, and so forth. For several journals, however, the primary focus is on parallel algorithms. These journals include the *Journal for Parallel and Distributed Computing*, published by Academic Press (http://www.apnet.com), the *IEEE Transactions on Parallel and Distributed Systems* (http://computer.org/pubs/tpds), and for results that can be expressed in a condensed form, *Parallel Processing Letters*, published by World Scientific. Finally, several comprehensive journals should be mentioned that publish a fair number of articles on parallel algorithms. These include the *IEEE Transactions on Computers*, *Journal of the ACM*, and *SIAM Journal on Computing*.

Unfortunately, due to very long delays from submission to publication, most results that appear in journals (with the exception of *Parallel Processing Letters*) are actually quite old. (A delay of 3-5 years from submission to publication is no uncommon.) Recent results appear in a timely fashion in conferences, most of which are either peer reviewed or panel reviewed. The first conference devoted primarily to parallel computing is the *International Conference on Parallel Processing (ICPP)*, which had its inaugural conference in 1972. Many landmark papers have been presented at ICPP, especially during the 1970s and 1980s. Proceedings from this conference have been published in recent years by the IEEE Computer Society and CRC. In recent years, the *International Parallel Processing Symposium (IPPS)* (http://www.ippsxx.org) has emerged as the premier conference devoted to parallel computing. IPPS is quite comprehensive in that in addition to the conference, it offers a wide variety of workshops and tutorials. Another conference that has matured nicely in recent years is the *ACM Symposium on Parallel and Distributed Processing (SPDP)*. It is interesting to note that both IPPS and SPDP initially started as regional conferences. In fact, in 1998, IPPS and SPDP will hold a combined conference. A conference that focuses on very theoretical, primarily PRAM-based, algorithms is the *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. This conference is an offshoot of the premier theoretical conferences in computer science, *ACM Symposium on Theory of Computing (STOC)* and *IEEE Symposium on Foundations of Computer Science (FOCS)*. A conference which focuses on very large parallel systems is *SC 'XY* (http://www.supercomp.org), where XY represents the last two digits of the year. This conferences includes the presentation of the Gordon Bell Prize for best parallelization. Awards are given in

various categories, such as highest sustained performance and best price/performance.

Finally, a variety of sites exist that can be used to effectively navigate the web, including the *IEEE Technical Committee on Parallel Processing (IEEETCPP)*, which can be found at http://www.cs.buffalo.edu/tcpp. This site contains links to conferences, journals, people in the field, bibliographies on parallel processing, on-line course material, books, and so forth. Another nice site (http://www.computer.org/parascope) is currently maintained by David A. Bader (UMIACS) with support from the IEEE. Finally, several newsgroups cater to parallel computing, including comp.parallel and comp.arch.