# The Design of Efficient Parallel Algorithms

Selim G. Akl

Queen's University

**Summary.** This chapter serves as an introduction to the study of parallel algorithms, in particular how they differ from conventional algorithms, how they are designed, and how they are analyzed to evaluate their speed and cost.

## 1. INTRODUCTION

The design and analysis of parallel algorithms is a cornerstone of parallel computation. Fast parallel algorithms are necessary if a significant reduction in running time is to be achieved when solving computationally demanding problems on parallel computers. This chapter serves as an introduction to the study of parallel algorithms, in particular how they differ from conventional algorithms, how they are designed, and how they are analyzed to evaluate their speed and cost.

One of the most striking differences between the conventional and parallel approaches to computing is the wide diversity of parallel models of computation available. We explore this rich landscape and demonstrate how it leads to a variety of new computing paradigms. Four case studies are used for illustration. In each case, a sequence of increasingly more involved algorithms is developed in order to show either how a problem can be solved on several different models or how efficient solutions to fundamental problems can be used to address more difficult ones.

The chapter is organized as follows. Section 2. is devoted to explaining the concepts of parallel model of computation and parallel algorithm and providing a brief introduction to the analysis of parallel algorithms. The problems of sorting, matrix multiplication, computing the convex hull, and manipulating pointer-based data structures, are the subject of Sections 3. – 6., along with variations and applications. In Section 7. we discuss current trends and future directions, including alternative computational models and the notion of synergy in parallel computation.

## 2. PARALLEL MODELS AND ALGORITHMS

Today's conventional computers comprise a single *processor*. This processor is in charge of executing *one by one* the instructions of a *program* it stores. Each instruction may be an *arithmetic* or *logical operation* on values retrieved from *memory*, or may require the *input* or *output* of data. Because the instructions are performed in sequence, conventional computers are said to be *sequential*. A *parallel computer*, by contrast, consists of several processors. Each processor is of the same type as used on a sequential computer. It has its program and executes its instructions in sequence. Thus, the processors cooperate in solving a computational problem by executing several instructions simultaneously, that is, *in parallel*.

Given that a parallel computer has several processors, it is natural to pose the following questions: How are these computers organized? How are their programs designed? How is their performance measured? Obtaining answers to these questions is greatly simplified by introducing the notions of *parallel model of computation*, *parallel algorithm*, and *mathematical analysis of algorithms*.

## 2.1 What is a parallel model?

A *parallel model of computation* is an abstract description of a parallel computer. It ignores irrelevant implementation details and attempts to capture the most important features. As such, it allows us to focus on the issues that matter most when designing parallel solutions to computational problems. It also renders the analysis of such solutions simpler and more useful.

There are many different ways to organize the processors on a parallel computer. Consequently, a multitude of parallel models exist. One simple way to classify these models is to use two general families, namely, *shared-memory* models and *interconnection-network* models.

**Shared Memory.** In this family of parallel models of computation, the processors share a common memory from which they can all read and to which they can all write. The processors treat the shared memory as a bulletin board. They use it to receive data, exchange information, and deposit the results of their computations. We now describe one example of a model in this family.

*Example 2.1.* The *Parallel Random Access Machine* (PRAM), illustrated in Fig. 2.1, consists of $N$ identical processors, $P_1$, $P_2$, $\ldots$, $P_N$, where $N \geq 2$, sharing a memory of $M$ locations, $U_1, U_2, \ldots, U_M$, where $M \geq 1$. (Typically, $M > N$.) Each processor has a small local memory (i.e., a fixed number of *registers*) and circuitry to allow the execution of arithmetic and logical operations on data held by the registers. Each memory location and each processor register can store a datum of constant size. The processors gain access to the memory locations by means of a *memory access unit* (MAU). This unit may be implemented as a *combinational circuit* (a description of which is deferred until Section 7.2).

The processors operate *synchronously* and execute the same sequence of instructions on different data. Each step of a PRAM computation consists of three phases (executed in the following order):

1. A reading phase in which each processor (if so required) copies a datum from a memory location into one of its registers;
2. A computing phase in which each processor (if so required) performs an arithmetic or logical operation on data stored in its registers;
3. A writing phase in which each processor (if so required) copies a datum from one of its registers into a shared memory location.

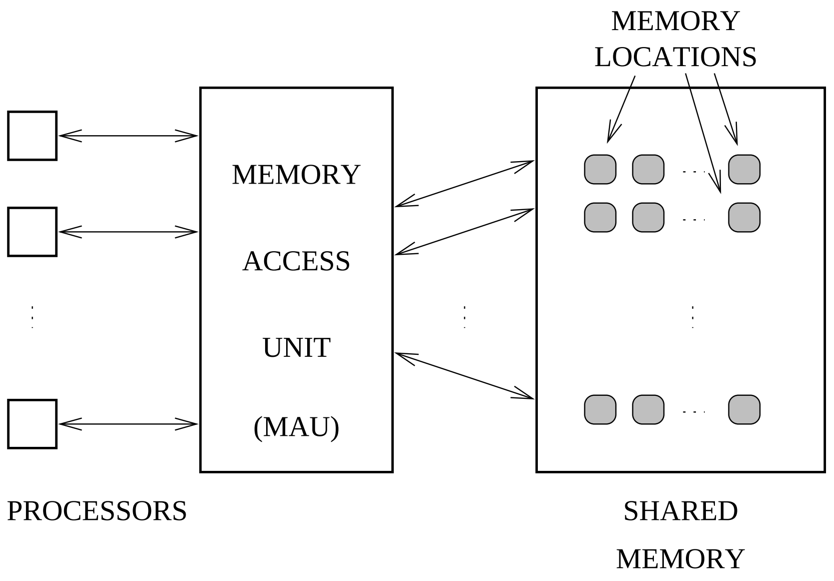


**Fig. 2.1.** A parallel random access machine.

During the reading phase any number of processors can gain access to the same memory location, if so desired. In one extreme situation each of the $N$ processors reads from a distinct memory location. In another, all $N$ processors read from the same memory location. This is also true of the writing phase, except that one must specify what ends up in a memory location when two or more processors are writing into it. Depending on the problem being solved, we may specify, for instance, that the *sum* of the values written by the processors is stored, or their *minimum*, or one of the values selected *arbitrarily*, and so on.

It is important to note that the processors use the shared memory to communicate with each other. Suppose that processor $P_i$ wishes to send a datum $d$ to processor $P_j$. During the writing phase of a step, $P_i$ writes $d$ in a location $U_k$ known to $P_j$. Processor $P_j$ then reads $d$ from $U_k$ during the reading phase of the following step. □

**Interconnection Network.** A model in this family consists of $N$ processors, $P_1, P_2, \ldots, P_N$, where $N \geq 2$, forming a connected network. Certain pairs of processors are directly connected by a two-way communication link. Those processors to which $P_i$ is directly connected, $1 \leq i \leq N$, are called $P_i$'s *neighbors*. There is no shared memory: $M$ locations are distributed among the processors. Thus, each processor has a local memory of $M/N$ locations. When two processors $P_i$ and $P_j$ wish to communicate they do so by *exchanging messages*. If $P_i$ and $P_j$ are neighbors, they use the link between them to send and receive messages; otherwise, the messages travel over a sequence of links through other processors. All processors execute the same sequence of instructions on different data. Two models in this family are distinguished from one other by the subset of processor pairs that are directly connected.

*Example 2.2.* For some integer $m \geq 2$, a *binary tree* interconnection network consists of $N = 2^m - 1$ processors connected in the shape of a complete binary tree with $2^{m-1}$ leaves. The tree has $m$ levels, numbered 0 to $m-1$, with the root at level $m-1$ and the leaves at level 0, as shown in Fig. 2.2 for $m = 4$. Each processor at level $\ell$, $0 < \ell < m-1$, is connected to three neighbors: one *parent* at level $\ell + 1$ and two children at level $\ell - 1$. The root has two children and no parent while each leaf has a parent and no children. The processor indices go from 1 to $N$ from top to bottom in the tree and from left to right on each level, such that on level $\ell$, $0 \leq \ell \leq m-1$, the processors are numbered from $2^{m-\ell-1}$ to $2^{m-\ell} - 1$. $\square$

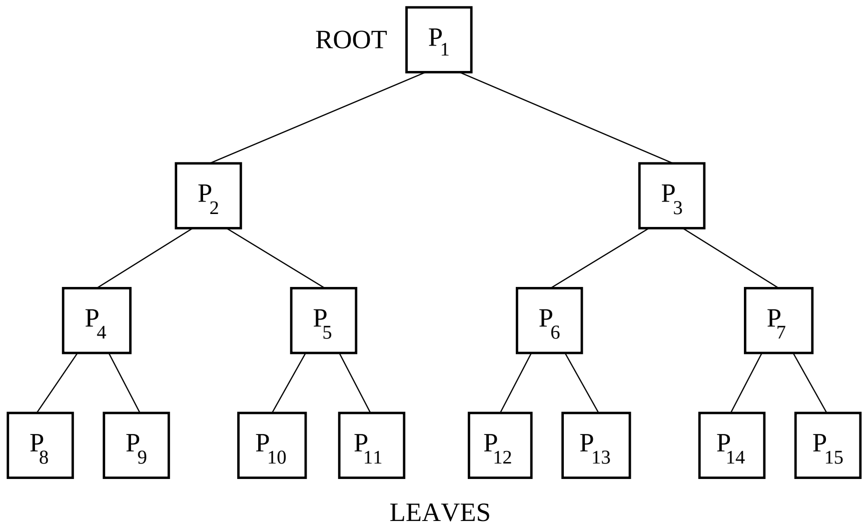


**Fig. 2.2.** A binary tree of processors.

## 2.2 What is a parallel algorithm?

A *parallel algorithm* is a method for solving a computational problem on a parallel model of computation. It allows the problem to be broken into smaller parts that are solved simultaneously. Most computer scientists are familiar with *sequential algorithms*, that is, algorithms for solving problems in a sequential fashion, one step at a time. This is not surprising since these algorithms have been around for quite a long time. In order to understand parallel algorithms, however, a new way of thinking is required.

*Example 2.3.* Suppose that we wish to construct a sequence of numbers $C = (c_1, c_2, \ldots, c_n)$ from two given sequences of numbers $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, such that $c_i = a_i \times b_i$, for $1 \leq i \leq n$. Also, let a PRAM with $n$ processors $P_1, P_2, \ldots, P_n$ and a shared memory with at least $3n$ memory locations be available. The two sequences $A$ and $B$ are initially stored in the shared memory, one number per location. The following parallel algorithm computes the sequence $C$ and stores it, one number per location, in the shared memory:

```
Algorithm PRAM SEQUENCE
  for $i = 1$ to $n$ do in parallel
    $c_i \leftarrow a_i \times b_i$
  end for. ■
```

The algorithm consists of one step, a parallel **for** loop. The difference between this 'loop' and a conventional sequential **for** loop is that all of the parallel loop's iterations are performed simultaneously. The loop instructs each processor whose index $i$ falls between 1 and $n$, inclusive, to multiply $a_i$ by $b_i$ and put the result in $c_i$. This way, all entries of $C$ are computed at once.

Now suppose that we modify the problem so that $c_i = a_i \times b_i$ when $i$ is even and $c_i = a_i / b_i$ when $i$ is odd, $1 \leq i \leq n$. In order to compute the sequence $C$ according to this new definition, we simply replace the statement $c_i \leftarrow a_i \times b_i$ in algorithm PRAM SEQUENCE with the statement:

```
if $i$ mod $2 = 0$
then $c_i \leftarrow a_i \times b_i$
else $c_i \leftarrow a_i / b_i$
end if. ■
```

In general, a processor needs only to examine its index in order to determine whether or not it is required to execute a given step of an algorithm. □

*Example 2.4.* A sequence $X = (x_1, x_2, \ldots, x_n)$ of numbers is given, where $n = 2^{m-1}$, for some integer $m \geq 2$. It is required to compute the sum $S = x_1 + x_2 + \cdots + x_n$ of this sequence. Here, we assume that a binary tree is available with $2n - 1$ processors. Each of the processors holds a number $y_j$, $1 \leq j \leq 2n - 1$. For the nonleaf processors $P_i$, $1 \leq i \leq n - 1$, $y_i$ is initialized to 0. These $y_i$ will serve to store intermediate results during the course of the computation. For leaf $P_{n+i-1}$, $y_{n+i-1} = x_i$, $1 \leq i \leq n$. In other words, the sequence $X$ is initially stored one number per leaf processor. The following algorithm computes $S$ and stores it in the root processor:

```
Algorithm TREE SUM
  for $\ell = 1$ to $m - 1$ do
    for $j = 2^{m-\ell-1}$ to $2^{m-\ell}$ do in parallel
```

```
            $y_j \leftarrow y_{2j} + y_{2j+1}$
        end for
    end for. ■
```

The outer (sequential) **for** loop is executed once per level, for levels 1 to $m-1$ of the tree. For each level, all processors on that level execute the inner (parallel) **for** loop simultaneously, with each processor replacing the number it holds with the sum of the numbers held by its children. In this way, the global sum is computed by the root in the final iteration; in other words, $y_1 = S$.

The process implemented by algorithm TREE SUM is a *halving* process: After each iteration of the outer **for** loop, the number of values to be added up is one half of the number before the iteration. Also, the data move up the tree from the leaves to the root. In a similar way, a *doubling* process can be executed, with the data moving from the root to the leaves. Suppose that the root processor holds a value that it wishes to make known to all of the remaining processors. It sends that value to its two children, who in turn send it to their respective children, and so on until the value is received by the leaf processors. □

## 2.3 How do we analyze parallel algorithms?

There are several criteria that one can use in order to evaluate the quality of a parallel algorithm. In this chapter we focus on three such criteria, namely, running time, number of processors, and cost. The presentation is simplified by the use of the following notations. Let $f(n)$, $g(n)$, and $h(n)$ be functions from the positive integers to the positive reals. Then

1. The function $f(n)$ is said to be *of order at least* $g(n)$, denoted $\Omega(g(n))$, if there are positive constants $k_1$ and $n_1$ such that $f(n) \geq k_1 g(n)$ for all $n \geq n_1$.
2. The function $f(n)$ is said to be *of order at most* $h(n)$, denoted $O(h(n))$, if there are positive constants $k_2$ and $n_2$ such that $f(n) \leq k_2 h(n)$ for all $n \geq n_2$.

The $\Omega$ and $O$ notations allow us to focus on the *asymptotic* behavior of a function $f(n)$, that is, its value *in the limit* as $n$ grows without bound. To say that $f(n)$ is $\Omega(g(n))$ is to say that, for large values of $n$, $f(n)$ is bounded from below by $g(n)$. Similarly, to say that $f(n)$ is $O(h(n))$ is to say that, for large values of $n$, $f(n)$ is bounded from above by $h(n)$. This simplifies our analyses a great deal. Thus, for example, if $f(n) = 3.5n^3 + 12n^{2.7} + 7.4n + 185$, we can say that $f(n)$ is of order $n^3$ and write either $f(n) = O(n^3)$, or $f(n) = \Omega(n^3)$, thereby ignoring constant factors and lower order terms. Note that in the special case where $f(n)$ is itself a constant, for example, $f(n) = 132$, we write $f(n) = O(1)$.

**Running Time.** The primary reason for designing a parallel algorithm is to use it (once implemented as a parallel program on a parallel computer) so that a computational task can be completed quickly. It is therefore natural for running time to be taken as an indicator of the goodness of a parallel algorithm.

A useful way of estimating what an algorithm's running time would be in practice is provided by mathematical analysis. Let a *time unit* be defined as the time required by a typical processor to execute a basic arithmetic or logical operation (such as addition, multiplication, AND, OR, and so on), to gain access to memory for reading or writing a datum (of constant size), or to route a datum (of constant size) to a neighbor. We measure a parallel algorithm's running time by counting the number of time units elapsed from the moment the algorithm begins execution until it terminates. We note here that in a parallel algorithm several basic operations are executed in one time unit. In fact, the number of such operations may be equal to the number of processors used by the algorithm, if all processors are active during that time unit.

The running time of an algorithm is intimately related to the *size* of the problem it is meant to solve, that is, the number of (constant-size) inputs and outputs. It is typical that the number of time units elapsed during the execution of a parallel algorithm is a function of the size of the problem. Therefore, we use $t(n)$ to express the running time of an algorithm solving a problem of size $n$.

A good appreciation of the speed of a parallel algorithm is obtained by comparing it to the best sequential solution. The *speedup* achieved by a parallel algorithm when solving a computational problem is equal to the ratio of two running times—that of the best possible (or best known) sequential solution divided by that of the parallel algorithm. The higher is the ratio, the better is the parallel algorithm.

*Example 2.5.* In Example 2.3, algorithm PRAM SEQUENCE computes the sequence $C$ by having each processor $P_i$, $1 \leq i \leq n$, read two values ($a_i$ and $b_i$), multiply them, and write the result in $c_i$. This requires a constant number of time units. Therefore, $t(n) = O(1)$. Sequentially, the best possible solution requires $n$ iterations and runs in $O(n)$ time. Hence, the speedup is $O(n)$. □

*Example 2.6.* In Example 2.4, algorithm TREE SUM computes the sum of $n$ numbers by performing $m-1$ iterations, one for each level of the tree (above the leaf level). Each iteration comprises a parallel **for** loop consisting of a routing operation, a multiplication, and an assignment, and thus requiring a constant number of time units. Therefore, the total number of time units is a constant multiple of $m-1$. Since $m-1 = \log n$, we have $t(n) = O(\log n)$.

Another way to see this is to note that the algorithm operates by forming $n/2$ pairs and adding up the numbers in each pair, then again pairing and adding up the results, and so on until the final sum is obtained. Since we began with $n$ inputs, the number of iterations required is $\log n$.

Sequentially, the sum requires $n-1$ constant-time iterations to be computed and, therefore, the sequential solution runs in $O(n)$ time. In this case, the speedup is $O(n/\log n)$. □

A parallel algorithm for solving a given problem on a particular model of computation is said to be *time optimal* if its running time matches (up to a constant factor) a lower bound on the time required to solve the problem on that model. For example, algorithm TREE SUM is time optimal since its running time is of $O(\log n)$ and any algorithm for adding $n$ numbers stored in the leaves of a binary tree and producing the sum from the root must require $\Omega(\log n)$ time.

**Number of processors.** Mostly for economic reasons, the number of processors required by a parallel algorithm is an important consideration. Suppose that two algorithms, designed to solve a certain problem on the same model of computation, have identical running times. If one of the two algorithms needs fewer processors, it is less expensive to run and hence this algorithm is the preferred one. The number of processors is usually expressed as a function of the size of the problem being solved. For a problem of size $n$, the number of processors is given as $p(n)$.

*Example 2.7.* In Example 2.3, algorithm PRAM SEQUENCE uses $n$ processors; thus, $p(n) = n$. □

*Example 2.8.* In Example 2.4, algorithm TREE SUM uses $2n-1$ processors; thus, $p(n) = 2n-1$. □

The number of processors often allows us to derive a lower bound on the running time of any parallel algorithm for solving a given problem, independently of the model of computation. Suppose that a lower bound of $\Omega(f(n))$ on the number of operations required to solve a problem of size $n$ is known. Then any algorithm for solving that problem in parallel using $n$ processors must require $\Omega(f(n)/n)$ time. For example, a lower bound on the number of operations required to multiply two $n \times n$ matrices is $\Omega(n^2)$. Therefore, any parallel algorithm for multiplying two $n \times n$ matrices using $n$ processors must require $\Omega(n)$ time.

The notion of *slowdown* (by contrast with speedup) expresses the *increase* in running time when the number of processors *decreases*. Specifically, if an algorithm runs in time $t_p$ using $p$ processors and in time $t_q$ using $q$ processors, where $q < p$, the slowdown is equal to $t_q/t_p$.

**Cost.** The cost of a parallel algorithm is an upper bound on the total number of basic operations executed collectively by the processors. If a parallel algorithm solves a problem of size $n$ in $t(n)$ time units with $p(n)$ processors, then its cost is given by $c(n) = p(n) \times t(n)$. Cost is useful in assessing how expensive it is to run a parallel algorithm. More importantly, this measure is often used for comparison purposes with a sequential solution. The *efficiency* of a parallel algorithm for a given problem is the ratio defined as follows: The running time of the best (possible or known) sequential solution to the problem divided by the cost of the parallel algorithm. The larger is the efficiency, the better is the parallel algorithm.

*Example 2.9.* In Example 2.3, algorithm PRAM SEQUENCE has a cost of $c(n) = p(n) \times t(n) = n \times O(1) = O(n)$. Since the best possible sequential solution runs in $O(n)$ time, the algorithm's efficiency is $O(1)$. □

*Example 2.10.* In Example 2.4, algorithm TREE SUM has a cost of $c(n) = p(n) \times t(n) = (2n-1) \times O(\log n) = O(n \log n)$. The best sequential algorithm runs in $O(n)$ time; the algorithm's efficiency in this case is $O(1/\log n)$.

The efficiency can be improved by modifying the algorithm to use fewer processors as follows. Let $q = \log n$ and $r = n/q$. For ease of presentation, we assume that $r = 2^v$, for some integer $v \geq 1$. A tree with $2r-1$ processors, of which $r$ are leaves, is available. Each processor $P_i$, $1 \leq i \leq 2r-1$, holds a value $y_i$. All $y_i$, $1 \leq i \leq 2r-1$, are initially equal to 0. The input sequence $X$ is presented to the tree with its entries organized as a table with $q$ rows and $r$ columns:

$$\begin{array}{llll}
x_1 & x_2 & \ldots & x_r \\
x_{r+1} & x_{r+2} & \ldots & x_{2r} \\
 & & \vdots & \\
x_{(q-1)r+1} & x_{(q-1)r+2} & \ldots & x_{qr}.
\end{array}$$

The new algorithm consists of two phases. The first phase has $q$ iterations: During the $j$th iteration, $1 \leq j \leq q$, leaf processor $P_{r+k-1}$, $1 \leq k \leq r$, reads $x_{(j-1)r+k}$ and adds it to $y_{r+k-1}$. The second phase is simply an application of algorithm TREE SUM with $m = v + 1$.

The first phase runs in $O(q)$, that is, $O(\log n)$ time. The second phase requires $O(v) = O(\log(n/\log n)) = O(\log n)$ time. The overall running time is therefore $t(n) = O(\log n)$. Since the number of processors used is $2r-1$, we have $p(n) = O(n/\log n)$, for a cost of $c(n) = O(n)$. As a result, the new algorithm's efficiency is $O(1)$. □

When the cost of a parallel algorithm for a given problem matches (up to a constant factor) a lower bound on the number of operations required to solve that problem, the parallel algorithm is said to be *cost optimal.* For example, if the cost of a parallel algorithm for sorting a sequence of $n$ numbers is $O(n \log n)$, then that algorithm is cost optimal in view of the $\Omega(n \log n)$ lower bound on the number of operations required to sort.

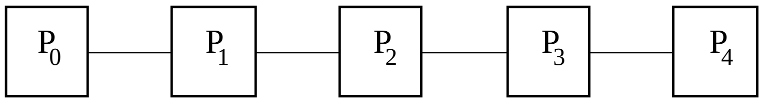


**Fig. 3.1.** A linear array of processors.

## 3. SORTING

We begin our study of the design and analysis of parallel algorithms by considering what is perhaps the most typical of problems in computer science, namely, the problem of sorting a sequence of numbers in nondecreasing order. Specifically, given a sequence of numbers $(x_0, x_1, \ldots, x_{N-1})$ listed in arbitrary order, it is required to rearrange these same numbers into a new sequence $(y_0, y_1, \ldots, y_{N-1})$ in which $y_0 \leq y_1 \leq \cdots \leq y_{N-1}$. Note here that if $x_i = x_j$ in the given sequence, then $x_i$ precedes $x_j$ in the sorted sequence if and only if $i < j$. Sequentially, a number of algorithms exist for solving the problem in $O(N \log N)$ time, and this is optimal. In this section, we demonstrate how the problem is solved on various models of parallel computation in the interconnection-network family. These models are the linear array, the mesh, the multidimensional array, the hypercube, and the star. In each case we will assume that there are as many processors on the model as there are numbers to be sorted. It is important to note here that, in order to fully define the problem of sorting, an indexing of the processors must be specified such that, when the computation terminates, $P_i$ holds $y_i$, for $0 \leq i \leq N-1$. In other words, the set of numbers has been sorted once the smallest number of the set resides in the 'first' processor, the second smallest number of the set resides in the 'second' processor, and so on.

### 3.1 Linear array

The *linear array* interconnection network is arguably the simplest of all models of parallel computation. It consists of $N$ processors $P_0, P_1, \ldots, P_{N-1}$, placed side by side in a one-dimensional arrangement and indexed from left to right. The processors are interconnected to form a linear array such that, for $1 \leq i \leq N-2$, each processor $P_i$ is connected by a two-way communication link to each of $P_{i-1}$ and $P_{i+1}$, and no other connections are present. This is illustrated in Fig. 3.1 for $N = 5$.

The sequence $(x_0, x_1, \ldots, x_{N-1})$ is sorted on a linear array of $N$ processors as follows. We begin by storing each of the numbers in a distinct processor. Thus, initially, $P_i$ holds $x_i$ and sets $y_i = x_i$, for $0 \leq i \leq N-1$. Now, the processors repeatedly perform an operation known as a 'comparison-exchange', whereby $y_i$ is compared to $y_{i+1}$ with the smaller of the two numbers ending up in $P_i$ and the larger in $P_{i+1}$. The algorithm is given next.

**Algorithm LINEAR ARRAY SORT**

```
for $j = 0$ to $N - 1$ do
  for $i = 0$ to $n - 2$ do in parallel
    if $i \bmod 2 = j \bmod 2$
    then if $y_i > y_{i+1}$
           then $y_i \leftrightarrow y_{i+1}$
           end if
    end if
  end for
end for. ■
```

It can be shown that this algorithm sorts correctly in $N$ steps, that is, $O(N)$ time. The proof is based on the observation that the algorithm is *oblivious*, that is, the sequence of comparisons it performs is predetermined. Thus, the algorithm's behavior and the number of iterations it requires are not affected by the actual set of numbers to be sorted. This property allows the proof of correctness to use the 0-1 principle: If it can be shown that the algorithm correctly sorts any sequence of 0s and 1s, it will follow that the algorithm correctly sorts any sequence of numbers.

A running time of $O(N)$ for sorting $n$ numbers in parallel is not too impressive, affording a meager speedup of $O(\log N)$ over an optimal sequential solution. However, two points are worth making here:

1. The running time achieved by algorithm LINEAR ARRAY SORT is the best that can be accomplished on the linear array. To see this note that a lower bound of $\Omega(N)$ on the time required to sort $N$ numbers on a linear array with $N$ processors is established as follows: If the largest number in the sequence to be sorted is initially stored in $P_0$, it needs $N-1$ steps to travel to its final destination in $P_{N-1}$.
2. More importantly, algorithm LINEAR ARRAY SORT will play an important role in the design of algorithms for other models as shown in what follows.

## 3.2 Mesh

In an interconnection network, the *distance* between two processors $P_i$ and $P_j$ is the number of links on the shortest path from $P_i$ to $P_j$. The *diameter* of the network is the length of the longest distance among all distances between pairs of processors in that network. Since processors need to communicate among themselves, and since the time for a message to go from one processor to another depends on the distance separating them, a network with a small diameter is better than one with a large diameter.

One limitation of the linear array is its large diameter. As we saw in Section 3.1, in an array with $N$ processors, $N-1$ steps are needed to transfer

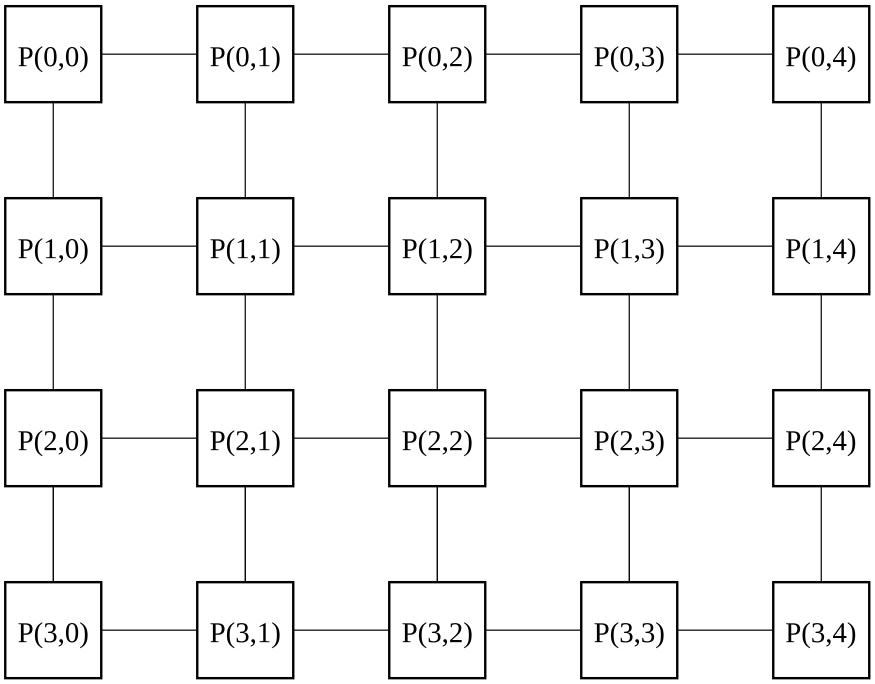


**Fig. 3.2.** A mesh of processors.

a datum from the leftmost to the rightmost processor. We therefore seek another network with a smaller diameter to solve the sorting problem.

A natural extension of the linear array is the two-dimensional array or *mesh*. Here, $N$ processors $P_0, P_1, \ldots, P_{N-1}$ are arranged into an $m \times n$ array, of $m$ rows and $n$ columns, where $N = m \times n$, as shown in Fig. 3.2 for $m = 4$ and $n = 5$.

The processor in row $j$ and column $k$ is denoted by $P(j,k)$, where $0 \leq j \leq m-1$ and $0 \leq k \leq n-1$. A two-way communication line links $P(j,k)$ to its neighbors $P(j+1,k)$, $P(j-1,k)$, $P(j,k+1)$, and $P(j,k-1)$. Processors on the boundary rows and columns have fewer than four neighbors and, hence, fewer connections.

While retaining essentially the same simplicity as the linear array, the mesh offers a significantly smaller diameter of $m+n-2$ (i.e., the number of links on the shortest path from the processor in the top left corner to the processor in the bottom right corner). When $m = n = N^{1/2}$, the diameter is $2N^{1/2} - 2$.

For the purpose of sorting the sequence of numbers $(x_0, x_1, \ldots, x_{N-1})$, the $N$ processors are indexed in *row-major* order, that is, processor $P_i$ is placed in row $j$ and column $k$ of the two-dimensional array, where $i = jn + k$ for $0 \leq i \leq N-1$, $0 \leq j \leq m-1$, and $0 \leq k \leq n-1$. Initially, $P_i$ holds $x_i$, for

$0 \leq i \leq N-1$. Since $y_i$ will be held by $P_i$, for $0 \leq i \leq N-1$, the sorted sequence will be arranged in row-major order.

The sorting algorithm consists of two operations on the rows and columns of the mesh:

1. Sorting a row or a column (or part thereof)
2. Cyclically shifting a row (or part thereof).

Sorting is performed using algorithm LINEAR ARRAY SORT. A row of the mesh is cyclically shifted $k$ positions to the right, $k \geq 1$, by moving the element in column $i$ of the row, for $i = 0, 1, \ldots, n-1$, through the row's processors, until it reaches column $j$ of the same row, where $j = (i+k) \bmod n$. Note that if $j < i$, the element in column $i$ actually moves *to the left* to reach its destination in column $j$.

Some definitions are useful. A *horizontal slice* is a submesh of $n^{1/2}$ rows and $n$ columns containing all rows $i$ such that $i = kn^{1/2}, kn^{1/2}+1, \ldots, (k+1)n^{1/2}-1$ for a nonnegative integer $k$, as shown in Fig. 3.3(a). A *vertical slice*

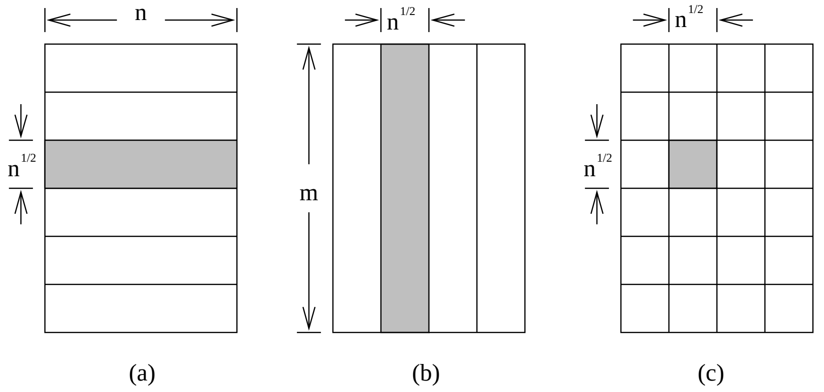


**Fig. 3.3.** Dividing a mesh into submeshes.

is a submesh of $m$ rows and $n^{1/2}$ columns containing all columns $j$ such that $j = ln^{1/2}, ln^{1/2}+1, \ldots, (l+1)n^{1/2}-1$ for a nonnegative integer $l$, as shown in Fig. 3.3(b). A *block* is a submesh of $n^{1/2}$ rows and $n^{1/2}$ columns consisting of all processors $P(i,j)$ such that $i = kn^{1/2}, kn^{1/2}+1, \ldots, (k+1)n^{1/2}-1$ and $j = ln^{1/2}, ln^{1/2}+1, \ldots, (l+1)n^{1/2}-1$, for nonnegative integers $k$ and $l$, as shown in Fig. 3.3(c).

The algorithm is as follows:

**Algorithm MESH SORT**

```
Step 1: for all vertical slices do in parallel
            (1.1) Sort each column
            (1.2) for $i = 1$ to $n$ do in parallel
```

```
                 Cyclically shift row $i$ by $i \bmod n^{1/2}$ positions to the right
                end for
            (1.3) Sort each column
          end for
Step 2: (2.1) for $i = 1$ to $n$ do in parallel
                Cyclically shift row $i$ by $in^{1/2} \bmod n$ positions to the right
              end for
        (2.2) Sort each column of the mesh
Step 3: for all horizontal slices do in parallel
            (3.1) Sort each column
            (3.2) for $i = 1$ to $n$ do in parallel
                 Cyclically shift row $i$ by $i \bmod n^{1/2}$ positions to the right
                end for
            (3.3) Sort each column
          end for
Step 4: (4.1) for $i = 1$ to $n$ do in parallel
                Cyclically shift row $i$ by $in^{1/2} \bmod n$ positions to the right
              end for
        (4.2) Sort each column of the mesh
Step 5: for $k = 1$ to 3 do
            (5.1) for $i = 1$ to $n$ do in parallel
                   if $i \bmod 2 = 0$
                   then sort row $i$ from left to right
                   else  sort row $i$ from right to left
                   end if
                end for
            (5.2) Sort each column
          end for
Step 6: for $i = 1$ to $n$ do in parallel
            Sort row $i$
          end for. ■
```

Note that in Step 4 each horizontal slice is treated as an $n \times n^{1/2}$ mesh. The algorithm is clearly oblivious and the 0-1 principle can be used to show that it sorts correctly. It is also immediate that the running time is $O(n+m)$. To see that this time is the best possible on the mesh, suppose that $P(0,0)$ initially holds the largest number in the sequence to be sorted. When the sequence is finally sorted, this number must occupy $P(m-1, n-1)$. The shortest path from $P(0,0)$ to $P(m-1, n-1)$ consists of $(m-1)+(n-1)$ links to be traversed, thus establishing an $\Omega(m+n)$ lower bound on the number of elementary operations required to sort a sequence of $mn$ elements on an $m \times n$ mesh. To derive the algorithm's cost, assume for simplicity that $m = n = N^{1/2}$. Therefore, $p(N) = N$ and $t(N) = O(N^{1/2})$, for a cost of $c(N) = O(N^{3/2})$, which is not optimal in view of the $O(N \log N)$ elementary operations sufficient for sorting sequentially.

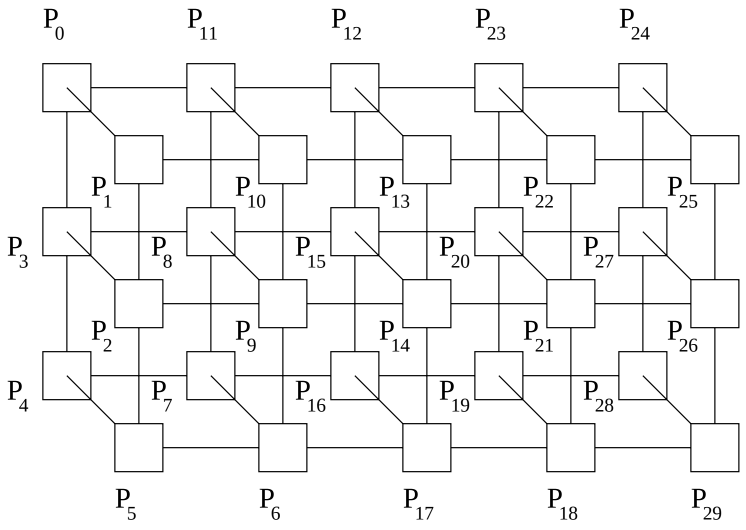


**Fig. 3.4.** A 3-dimensional array.

## 3.3 Multidimensional array

A natural extension of the mesh model is to arrange $N = n_1 n_2 \cdots n_d$ processors in a $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array, where $d \geq 3$ and $n_i \geq 2$, for $1 \leq i \leq d$. This is illustrated in Fig. 3.4 for $d = 3$, $n_1 = 2$, $n_2 = 3$, and $n_3 = 5$. Each position of the $d$-dimensional array has coordinates $I(1)$, $I(2)$, $\ldots$, $I(d)$, where $1 \leq I(k) \leq n_k$.

*Example 3.1.* When $d = 3$, $n_1 = 2$, $n_2 = 3$, and $n_3 = 5$, the coordinates of the 30 positions of the $2 \times 3 \times 5$ array are as follows:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1,1,1 | 2,1,1 | 1,1,2 | 2,1,2 | 1,1,3 | 2,1,3 | 1,1,4 | 2,1,4 | 1,1,5 | 2,1,5 |
| 1,2,1 | 2,2,1 | 1,2,2 | 2,2,2 | 1,2,3 | 2,2,3 | 1,2,4 | 2,2,4 | 1,2,5 | 2,2,5 |
| 1,3,1 | 2,3,1 | 1,3,2 | 2,3,2 | 1,3,3 | 2,3,3 | 1,3,4 | 2,3,4 | 1,3,5 | 2,3,5. □ |

In dimension $k$, where $1 \leq k \leq d$, there are

$$n_1 \times n_2 \times \cdots \times n_{k-1} \times n_{k+1} \times \cdots \times n_d$$

(i.e., $N/n_k$) groups of $n_k$ consecutive positions. The positions in each group have the same

$$I(1), I(2), \ldots, I(k-1), I(k+1), I(k+2), \ldots, I(d)$$

coordinates (in other words, they differ only in coordinate $I(k)$, with the first in the group having $I(k) = 1$, the second $I(k) = 2$, and the last $I(k) = n_k$). All the processors in a group are connected to form a linear array.

The processors are indexed in *snakelike order by dimension*. The following recursive function $snake_i$ maps the coordinates $I(1), I(2), \ldots, I(d)$ of each position of the $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array into a unique integer in the set $\{1, 2, \ldots, N\}$ according to snakelike order by dimension: $snake_1(I(1)) = I(1)$, for $1 \leq I(1) \leq n_1$, and $snake_k(I(1), I(2), \ldots, I(k)) = n_1 n_2 \cdots n_{k-1}(I(k) - 1) + K$, where $K = snake_{k-1}(I(1), I(2), \ldots, I(k-1))$ if $I(k)$ is odd, and $K = n_1 n_2 \cdots n_{k-1} + 1 - snake_{k-1}(I(1), I(2), \ldots, I(k-1))$ otherwise, for $2 \leq k \leq d$, $1 \leq I(j) \leq n_j$, and $1 \leq j \leq k$. Now the index $i$ of processor $P_i$, where $0 \leq i \leq N-1$, occupying the position whose coordinates are $I(1), I(2), \ldots, I(d)$, is thus obtained from $i = snake_d(I(1), I(2), \ldots, I(d)) - 1$.

In order to sort the sequence $(x_0, x_1, \ldots, x_{N-1})$ on this array, we store $x_i$ in $P_i$ initially, for $0 \leq i \leq N-1$. When the algorithm terminates, $P_i$ holds element $y_i$ of the sorted sequence, for $0 \leq i \leq N-1$. The algorithm consists of a number of iterations. During an iteration, the numbers are sorted either in the forward or reverse direction along each of the $d$ dimensions in turn. Here, "sorting in the forward (reverse) direction in dimension $k$" means sorting the numbers in nondecreasing (nonincreasing) order from the processor with $I(k) = 1$ to the processor with $I(k) = n_k$. This sorting is performed using algorithm LINEAR ARRAY SORT.

Henceforth, we refer to each group of consecutive positions in a given dimension of the multidimensional array, as a 'row'. Thus, a row in dimension $k$ consists of $n_k$ consecutive positions. Let $D(k) = \sum_{r=k+1}^{d}(I(r) - 1)$, for $k = 1, 2, \ldots, d$, where the empty sum (when $k = d$) is equal to 0 by definition, and let $M = \sum_{k=2}^{d}\lceil \log n_k \rceil$. The algorithm is as follows:

**Algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT**

```
for $i = 1$ to $M$ do
  for $k = 1$ to $d$ do
    for each row in dimension $k$ do in parallel
      if $D(k)$ is even
      then sort in the forward direction
      else  sort in the reverse direction
      end if
    end for
  end for
end for. ■
```

*Example 3.2.* Let $d = 4$, $n_1 = 2$, $n_2 = 3$, $n_3 = 4$, $n_4 = 5$, and let the sequence to be sorted be $(120, 119, \ldots, 3, 2, 1)$. The numbers are initially stored in the 120 processors of the 4-dimensional array, one number per processor (such that $P_0$ holds 120, $P_1$ holds 119, and so on). Once algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT has been applied, the following arrangement results:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 12 | 11 | 13 | 14 | 24 | 23 |
| 4 | 3 | 9 | 10 | 16 | 15 | 21 | 22 |
| 5 | 6 | 8 | 7 | 17 | 18 | 20 | 19 |
| | | | | | | | |
| 48 | 47 | 37 | 38 | 36 | 35 | 25 | 26 |
| 45 | 46 | 40 | 39 | 33 | 34 | 28 | 27 |
| 44 | 43 | 41 | 42 | 32 | 31 | 29 | 30 |
| | | | | | | | |
| 49 | 50 | 60 | 59 | 61 | 62 | 72 | 71 |
| 52 | 51 | 57 | 58 | 64 | 63 | 69 | 70 |
| 53 | 54 | 56 | 55 | 65 | 66 | 68 | 67 |
| | | | | | | | |
| 96 | 95 | 85 | 86 | 84 | 83 | 73 | 74 |
| 93 | 94 | 88 | 87 | 81 | 82 | 76 | 75 |
| 92 | 91 | 89 | 90 | 80 | 79 | 77 | 78 |
| | | | | | | | |
| 97 | 98 | 108 | 107 | 109 | 110 | 120 | 119 |
| 100 | 99 | 105 | 106 | 112 | 111 | 117 | 118 |
| 101 | 102 | 104 | 103 | 113 | 114 | 116 | 115. □ |

As a minor implementation detail, note that each processor computes $D(k)$ over its coordinates $I(1), I(2), \ldots, I(d)$ using two variables $C$ and $D$ as follows. Initially, it computes $C = \sum_{r=1}^{d}(I(r) - 1)$. Then, prior to the second **for** loop, it computes $D(1)$ as $D = C - (I(1) - 1)$. Subsequently, at the end of the $k$th iteration of the second **for** loop, where $k = 1, 2, \ldots, d-1$, it computes $D(k+1)$ as $D = D - (I(k+1) - 1)$.

**Analysis.** We now show that $M$ iterations, each of which sorts the rows in dimensions $1, 2, \ldots, d$, suffice to sort correctly $N = n_1 n_2 \cdots n_d$ numbers stored in a $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array. Because algorithm LINEAR ARRAY SORT is oblivious, then so is algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT. This property allows us to use the 0-1 principle. Suppose then that the input to algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT consists of an arbitrary sequence of 0s and 1s. Once sorted in nondecreasing order, this sequence will consist of a (possibly empty) subsequence of 0s followed by a (possibly empty) subsequence of 1s. In particular, the sorted sequence will contain at most one (0, 1) pattern, that is, a 0 followed by a 1. (If the input consists of all 0s or all 1s, then of course the output contains no such (0, 1) pattern.) If present, the (0, 1) pattern will appear either in a row in dimension 1, or at the boundary between two adjacent dimension 1 rows (consecutive in snakelike order) such that the 0 appears in one row and the 1 in the next.

Now consider the rows in dimension $k$, $1 \leq k \leq d$. There are $N/n_k$ such rows. A row in dimension $k$ is said to be *clean* if it holds all 0s (all 1s) and there are no 1s preceding it (no 0s following it) in snakelike order; otherwise, the row is *dirty*.

Suppose that an iteration of the algorithm has just sorted the contents of the rows in dimension $d-1$, and let us focus on two such rows, adjacent in dimension $d$. For example, for $d = 3$, $n_1 = 2$, $n_2 = 3$, and $n_3 = 4$, the coordinates of two dimension 2 rows, adjacent in dimension 3 of the $2 \times 3 \times 4$ array, are

$$\begin{array}{ccc} 1,1,1 & & 1,1,2 \\ 1,2,1 & \text{and} & 1,2,2 \\ 1,3,1 & & 1,3,2, \end{array}$$

respectively. Because these two dimension $d-1$ rows were sorted in snakelike order, they yield at least one clean dimension $d-1$ row when the dimension $d$ rows are sorted. To illustrate, suppose that the two preceding dimension 2 rows contained

$$\begin{array}{ccc} 0 & & 1 \\ 0 & \text{and} & 0 \\ 1 & & 0, \end{array}$$

respectively, after they were sorted. Then (regardless of what the other dimension 2 rows contained) a clean dimension 2 row containing all 0s is created after the dimension 3 rows are sorted. Thus, after $\lceil \log n_d \rceil$ iterations all dimension $d-1$ rows are clean, except possibly for $N/(n_{d-1}n_d)$ dimension $d-1$ rows contiguous in dimension $d-1$. All dimension $d$ rows are now permanently sorted.

By the same argument, $\lceil \log n_{d-1} \rceil$ iterations are subsequently needed to make all dimension $d-2$ rows clean (except possibly for $N/(n_{d-2}n_{d-1}n_d)$ dimension $d-2$ rows contiguous in dimension $d-2$). In general, $\lceil \log n_k \rceil$ iterations are needed to make all dimension $k-1$ rows clean (except possibly for $N/(n_{k-1}n_k \cdots n_d)$ dimension $k-1$ rows contiguous in dimension $k-1$) after all dimension $k$ rows have been permanently sorted, for $k = d, d-1, \ldots, 2$. This leaves possibly one dirty dimension 1 row holding the $(0,1)$ pattern. Since that row may not be sorted in the proper direction for snakelike ordering, one final iteration completes the sort. The algorithm therefore sorts correctly in

$$\begin{aligned} M &= \lceil \log n_d \rceil + \lceil \log n_{d-1} \rceil + \cdots + \lceil \log n_2 \rceil + 1 \\ &= O(\log N) \end{aligned}$$

iterations.

Finally, observe that algorithm LINEAR ARRAY SORT is used in dimensions $1, 2, \ldots, d$, and hence each iteration of the outer **for** loop in algorithm

MULTIDIMENSIONAL ARRAY SIMPLE SORT requires $\sum_{k=1}^{d} O(n_k)$ time. Since the outer loop is executed $O(\log N)$ times, the algorithm has a running time of $O((n_1 + n_2 + \cdots + n_d) \log N)$. When $n_i = n$ for $1 \leq i \leq d$, we have $N = n^d$. In this case, the number of iterations is $O(d \log n)$ and the time per iteration is $O(dn)$, for a total running time of $O(d^2 n \log n)$.

The algorithm is fairly straightforward and relatively efficient. It is nonrecursive and does not require that data be exchanged between pairs of distant processors: The only operation used, namely, 'comparison-exchange', applies to neighboring processors. Thus, because no routing is needed, the algorithm has virtually no control overhead. However, it is not time optimal. Indeed, it is easy to see that a $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array has a diameter of $\sum_{k=1}^{d}(n_k - 1)$. It follows that a lower bound of $\Omega(n_1 + n_2 + \cdots + n_d)$ holds on the time required to sort on such an array. This suggests that the running time of algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT is not the best that can be obtained on a $d$-dimensional array. We now show that this is indeed the case by exhibiting an algorithm whose running time is $O(n_1 + n_2 + \cdots + n_d)$.

For each $n_i$, let $u_i$ and $v_i$ be positive integers such that $u_i v_i = n_i$, $1 \leq i \leq d$. The $d$-dimensional lattice is viewed as consisting of $u_1 u_2 \cdots u_d$ $d$-dimensional $v_1 \times v_2 \times \cdots \times v_d$ sublattices, called *blocks*, numbered in snakelike order by dimension from 1 to $u_1 u_2 \cdots u_d$. Alternatively, the $d$-dimensional lattice consists of $u_1 u_2 \cdots u_d$ $(d-1)$-dimensional *hyperplanes* of $v_1 v_2 \cdots v_d$ processors each, numbered from 1 to $u_1 u_2 \cdots u_d$. The algorithm is given in what follows.

**Algorithm MULTIDIMENSIONAL ARRAY FAST SORT**

```
Step 1: Sort the contents of block $i$, $1 \leq i \leq u_1 u_2 \cdots u_d$.
Step 2: Move the data held by the processors in block $i$
        to the corresponding processors in hyperplane $i$,
        $1 \leq i \leq u_1 u_2 \cdots u_d$.
Step 3: Sort the contents of all $d$-dimensional
        $v_1 \times v_2 \times \cdots \times n_d$ 'towers' of blocks.
Step 4: Move the data held by the processors in hyperplane $i$
        to the corresponding processors in block $i$,
        $1 \leq i \leq u_1 u_2 \cdots u_d$.
Step 5: Sort the contents of
        (5.1) All pairs of consecutive blocks $i$ and $i+1$,
              for all odd $i$, $1 \leq i \leq u_1 u_2 \cdots u_d$.
        (5.2) All pairs of consecutive blocks $i-1$ and $i$,
              for all even $i$, $2 \leq i \leq u_1 u_2 \cdots u_d$. ■
```

The algorithm requires $O(n_1 + n_2 + \cdots + n_d)$ elementary steps (in which a datum is sent from a processor to its neighbor), and this is time optimal. However, note that, while appealing in theory due to its time optimality, algorithm MULTIDIMENSIONAL ARRAY FAST SORT is significantly more complex than algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT. Deriving a simple time-optimal algorithm for sorting on a $d$-dimensional array remains an interesting open problem.

## 3.4 Hypercube

A special case of a $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array occurs when $n_i = 2$ for $1 \leq i \leq d$. This array is commonly known in the literature as the *hypercube*, a popular model among theoreticians and practitioners of parallel computation alike. The hypercube has $N = 2^d$ processors, each of which is directly connected to $d$ other processors. The diameter of the hypercube is also equal to $d$. A closer look at the hypercube is provided in Section 4.1. Our purpose here is to exhibit a very simple algorithm for sorting $N$ numbers on the hypercube. The algorithm is readily derived from algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT and is as follows:

**Algorithm HYPERCUBE SIMPLE SORT**

```
for $i = 1$ to $d$ do
  for $k = 1$ to $d$ do
    for each row in dimension $k$ do in parallel
      if $D(k)$ is even
      then sort in the forward direction
      else  sort in the reverse direction
      end if
    end for
  end for
end for. ■
```

We note here that a row consists of two processors and that sorting a row is simply a 'compare-exchange' operation. The algorithm executes $d^2$ such operations. Since $d = \log N$, the algorithm has a running time of $O(\log^2 N)$. This running time is larger than the lower bound of $\Omega(\log N)$ imposed by the hypercube's diameter only by a factor of $O(\log N)$. It is not optimal, however, as there exist slightly faster (but considerably more complicated) algorithms for sorting on the hypercube. We also note here that algorithm MULTIDIMENSIONAL ARRAY FAST SORT does not appear to lead to a more efficient algorithm for the hypercube.

## 3.5 Star

For a positive integer $n > 1$, an *n-star interconnection network*, denoted $\mathcal{S}_n$, is defined as follows:

1. $\mathcal{S}_n$ is an undirected graph with $n!$ vertices, each of which is a processor;

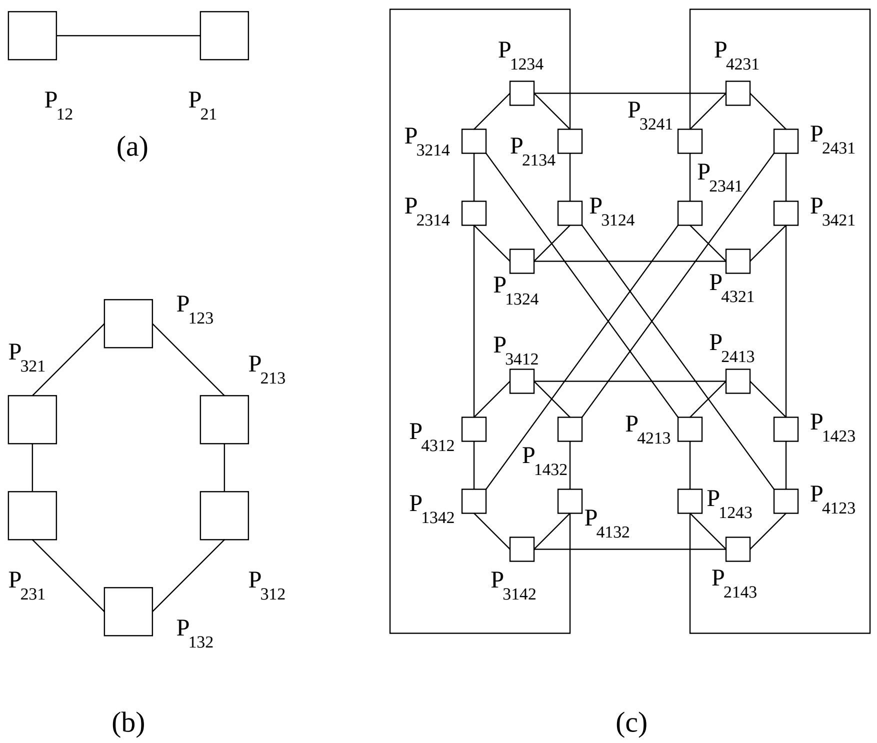


**Fig. 3.5.** A 2-star, a 3-star, and a 4-star interconnection networks.

2. The label $v$ of each processor $P_v$ is a distinct permutation of the symbols $\{1, 2, \ldots, n\}$;
3. Processor $P_v$ is directly connected by an edge to each of $n-1$ processors $P_u$, where $u$ is obtained by interchanging the first and $i$th symbols of $v$, that is, if
$$v = v(1)v(2)\ldots v(n),$$
where $v(j) \in \{1, 2, \ldots, n\}$ for $j = 1, 2, \ldots, n$, then
$$u = v(i)v(2)\ldots v(i-1)v(1)v(i+1)\ldots v(n),$$
for $i = 2, 3, \ldots, n$.

Networks $\mathcal{S}_2$, $\mathcal{S}_3$, and $\mathcal{S}_4$ are shown in Fig. 3.5(a)–(c).

In a given interconnection network, the maximum number of neighbors per processor is called the *degree* of the network. In practice, a small degree is usually a good property for a network to possess. As pointed out earlier, a small diameter is also desirable for an interconnection network. In this regard, the $n$-star has a degree of $n-1$ and a diameter of $\lceil 3(n-1)/2 \rceil$. This makes it a much more attractive network than a hypercube with $O(n!)$ processors which, by comparison, has a degree and a diameter of $O(n \log n)$.

We now consider the problem of sorting on the $n$-star. Specifically, given $n!$ numbers, held one per processor of the $n$-star, it is required to sort these numbers in nondecreasing order. The central idea of the algorithm we are about to describe is to map the processors of the $n$-star onto an $(n-1)$-dimensional array and then apply either of the two algorithms described in Section 3.3.

**Ordering the Processors.** As we did with previous networks, we impose an order on the processors, so that the problem of sorting is properly defined. For a given permutation $v = v(1)v(2)\ldots v(n)$, let

(a) $\mathit{vmax}(n) = \max\{v(k) \mid v(k) < v(n),\ 1 \leq k \leq n-1\}$, if such a $v(k)$ exists; otherwise, $\mathit{vmax}(n) = \max\{v(k) \mid 1 \leq k \leq n-1\}$.
(b) $\mathit{vmin}(n) = \min\{v(k) \mid v(k) > v(n),\ 1 \leq k \leq n-1\}$, if such a $v(k)$ exists; otherwise, $\mathit{vmin}(n) = \min\{v(k) \mid 1 \leq k \leq n-1\}$.

Beginning with the processor label $v = 12\ldots n$, the remaining $n! - 1$ labels are arranged in the order produced by (sequential) algorithm LABELS $(n, j)$ given in what follows. The algorithm operates throughout on the array $v = v(1)v(2)\ldots v(n)$. Initially, $v(i) = i$, $1 \leq i \leq n$, and the algorithm is called with $j = 1$. A new permutation is produced every time two elements of the array $v$ are swapped.

```
Algorithm LABELS($n, j$)
  Step 1: $i \leftarrow 1$
  Step 2: while $i \leq n$ do
            (2.1) if $n > 2$
                    then LABELS($n-1, i$)
                  end if
            (2.2) if $i < n$
                    then if $j$ is odd
                           then $v(n) \leftrightarrow \mathit{vmax}(n)$
                           else $v(n) \leftrightarrow \mathit{vmin}(n)$
                         end if
                  end if
            (2.3) $i \leftarrow i + 1$
          end while. ■
```

The order produced by the preceding algorithm satisfies the following properties:

1. All $(n-1)!$ permutations with $v(n) = \ell$ precede all permutations with $v(n) = \ell'$, for each $\ell$ and $\ell'$ such that $1 \leq \ell' < \ell \leq n$.
2. Consider the $j$th group of $(m-1)!$ consecutive permutations in which the symbols $v(m)v(m+1)\ldots v(n)$ are fixed, where $1 \leq j \leq n!/(m-1)!$ and $2 \leq m \leq n-1$. Within this group, the $m-1$ elements of the set $\{1, 2, \ldots, n\} - \{v(m), v(m+1), \ldots, v(n)\}$ take turns in appearing as

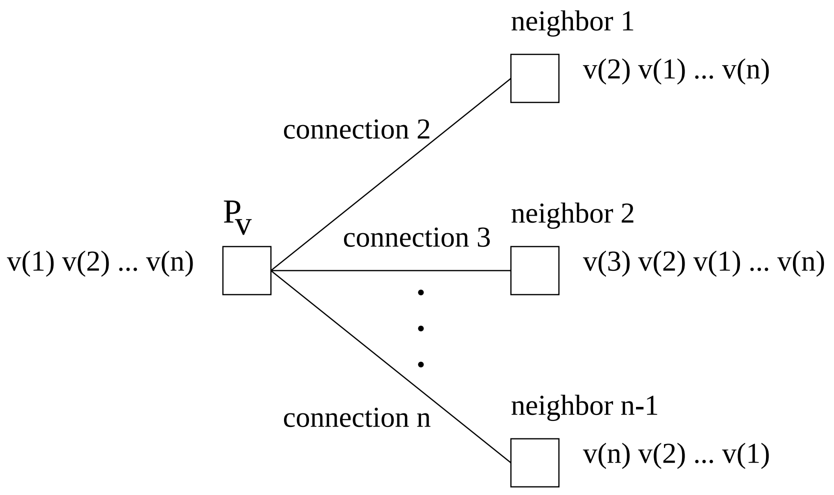


**Fig. 3.6.** Rearranging the data.

$v(m-1)$, each repeated $(m-2)!$ times consecutively. They appear in decreasing order if $j$ is odd and in increasing order if $j$ is even.

*Example 3.3.* For $n = 4$, the 4! labels, that is, the 24 permutations of $\{1, 2, 3, 4\}$, ordered according to algorithm LABELS, are as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1. | 1234 | 7. | 4213 | 13. | 1342 | 19. | 4321 |
| 2. | 2134 | 8. | 2413 | 14. | 3142 | 20. | 3421 |
| 3. | 3124 | 9. | 1423 | 15. | 4132 | 21. | 2431 |
| 4. | 1324 | 10. | 4123 | 16. | 1432 | 22. | 4231 |
| 5. | 2314 | 11. | 2143 | 17. | 3412 | 23. | 3241 |
| 6. | 3214 | 12. | 1243 | 18. | 4312 | 24. | 2341. □ |

**Rearranging the Data.** An important operation in our sorting algorithms is *data rearrangement* whereby pairs of processors exchange their data. We now define this operation. Recall that a processor can send a datum of constant size to a neighbor in constant time. In $\mathcal{S}_n$ each processor $P_v$ has $n-1$ neighbors $P_u$. The label $u$ of the $k$th neighbor of $P_v$ is obtained by swapping $v(1)$ with $v(k+1)$, for $k = 1, 2, \ldots, n-1$. The edges connecting $P_v$ to its neighbors are referred to as *connections* $2, 3, \ldots, n$. If each processor of $\mathcal{S}_n$ holds a datum, then the phrase "rearranging the data over connection $i$" means that all processors directly connected through connection $i$ exchange their data. Specifically, each processor $P_v$, where $v = v(1)v(2)\ldots v(i-1)v(i)v(i+1)\ldots v(n)$ sends its datum to processor $P_u$, where $u = v(i)v(2)\ldots v(i-1)v(1)v(i+1)\ldots v(n)$. This is illustrated in Fig. 3.6.

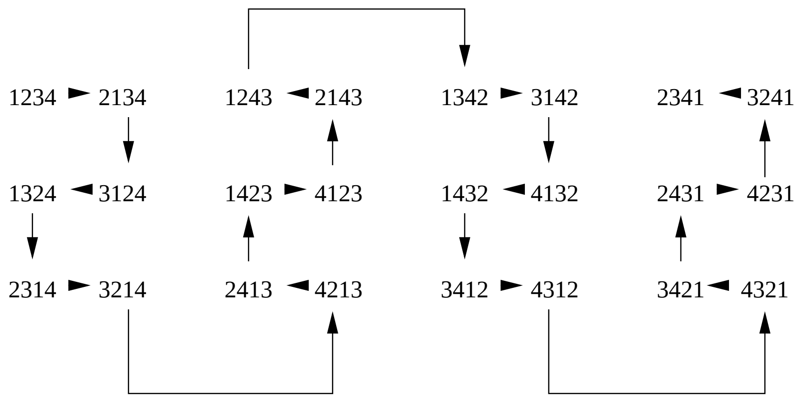


**Fig. 3.7.** Snakelike order by dimension.

**The Star Viewed As a Multidimensional Array.** The $n!$ processors of $\mathcal{S}_n$ are thought of as being organized in a virtual $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ orthogonal array. This organization is achieved in two steps: The processors are first listed according to the order obtained from algorithm LABELS, they are then placed (figuratively) in this order in the $(n-1)$-dimensional array according to snakelike order by dimension.

*Example 3.4.* For $n = 4$, the 24 processor labels appear in the $2 \times 3 \times 4$ array as shown in Fig. 3.7, where the arrows indicate snakelike order by dimension. A three-dimensional drawing is given in Fig. 3.8. $\square$

As a result of the arrangement just described, processor $P_v$, such that $v = v(1)v(2)\dots v(n)$, occupies that position of the $(n-1)$-dimensional array whose coordinates $I(1), I(2), \dots, I(n-1)$ are given by $I(k) = k+1-\sum_{j=1}^{k}[v(k+1) > v(j)]$, for $1 \leq k \leq n-1$, where $[v(k+1) > v(j)]$ equals 1 if $v(k+1) > v(j)$ and equals 0 otherwise.

It should be stressed here that two processors occupying adjacent positions in some dimension $k$, $1 \leq k \leq n-1$, on the $(n-1)$-dimensional array are not necessarily directly connected on the $n$-star. Now suppose that in dimension $k$, each processor in a group of $k+1$ processors occupying consecutive positions holds a datum. Our purpose in what follows is to show that after rearranging the data over connection $k+1$, these $k+1$ data are stored in $k+1$ processors forming a linear array (i.e., the first of the $k+1$ processors is directly connected to the second, the second is directly connected to the third, and so on). To simplify the presentation, we assume in what follows that the $k+1$ consecutive positions (of the multidimensional array) occupied by the $k+1$ processors begin at the position with $I(k) = 1$ and end at that with $I(k) = k+1$ (the argument being symmetric for the case where the $k+1$ processors are placed in the opposite direction, that is, from the position with

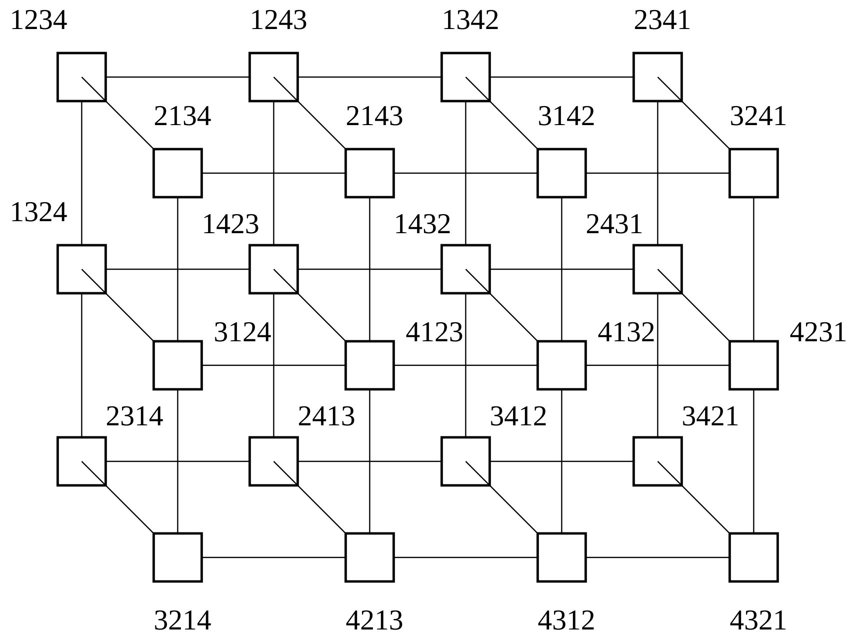


**Fig. 3.8.** A 4-star viewed as a 3-dimensional array.

$I(k) = k + 1$ to that with $I(k) = 1$). We proceed in two steps as described in the next two paragraphs.

First we show how the labels of these $k+1$ processors, which are permutations of $\{1, 2, \dots, n\}$, can be obtained from one another. Let $v(1)v(2)\dots v(k+1)\dots v(n)$ be the label of the last processor in the group, and let $d_m$, $1 \leq m \leq k+1$, be the position of the $m$th smallest symbol among $v(1)v(2)\dots v(k+1)$, where $d_1 = k+1$. Then, the label of the $(k+1-i)$th processor in the group, $1 \leq i \leq k$, is obtained from the label of the $(k+1-i+1)$st processor by exchanging the symbols in positions $k+1$ and $d_{i+1}$. For example, for $n = 4$, consider the four processors, occupying consecutive positions over dimension 3, whose labels are 1324, 1423, 1432, and 2431. The second of these (i.e., 1423) can be obtained from the third (i.e., 1432) by exchanging the symbol in position 4 (i.e., 2) with the third smallest symbol among 2, 4, 3, and 1 (i.e., 3). This property follows directly from the definition of the LABEL ordering.

Now consider a group of $k+1$ processors occupying consecutive positions over dimension $k$, with each processor holding a datum and the label of the $(k+1)$st processor in the group being $v(1)v(2)\dots v(k+1)\dots v(n)$. From the property established in the previous paragraph, the label of the $(k+1-i)$th processor has the $(i+1)$st smallest symbol among $v(1)v(2)\dots v(k+1)$ in position $k+1$, for $0 \leq i \leq k$. Rearranging the data over connection $k+1$ moves the data from this group to another group of $k+1$ processors in which the label of the $(k+1-i)$th processor has the $(i+1)$st smallest symbol among $v(1)v(2)\dots v(k+1)$ in position 1, for $0 \leq i \leq k$. In this new group, the label of the $(k+1-i)$th processor can be obtained from that of the $(k+1-i+1)$st by exchanging the symbol in position 1 with that in position $d_{i+1}$, for $1 \leq i \leq k$. It follows that the processors in the new group are connected to form a linear array. For example, consider once again the four processors, occupying consecutive positions over dimension 3, whose labels are 1324, 1423, 1432, and 2431 as illustrated in Fig. 3.8. Rearranging the data held by these processors over connection 4 moves them to the processors whose labels are 4321, 3421, 2431, and 1432, respectively.

**Sorting Algorithms for $\mathcal{S}_n$.** In order to sort the sequence of numbers $(x_0, x_1, \dots, x_{N-1})$, where $N = n!$, on an $n$-star, we view the processors of the $n$-star as an $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ virtual array. Either of the two sorting algorithms described in Section 3.3 is now applied.

Note that in dimension $k$ the $k+1$ processors in each group are not necessarily connected on the $n$-star to form a linear array. Thus, each time a linear array operation is to be performed (such as applying algorithm LINEAR ARRAY SORT), the contents of the $k+1$ processors are copied in constant time to another set of $k+1$ processors which *are* connected on the $n$-star as a linear array. This is done simply by rearranging the numbers to be sorted over connection $k+1$. After the linear array operation is performed, the numbers are brought back to the original processors, also in constant time.

As shown in Section 3.3, when sorting $N = n_1 n_2 \cdots n_d$ numbers on a $d$-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array, algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT has a running time of $O((n_1+n_2+\cdots+n_d)\log N)$, while algorithm MULTIDIMENSIONAL ARRAY FAST SORT runs in $O(n_1+n_2+\cdots+n_d)$ time. Since the number of dimensions $d$ of the multidimensional array (to which the processors of the star are mapped) is $n-1$, and $n_i = i+1$ for $1 \leq i \leq n-1$, we obtain running times of $O(n^3 \log n)$ and $O(n^2)$, respectively, when using the aforementioned algorithms.

Because $n!$ numbers are sorted optimally using one processor in time $O(n! \log n!)$, a parallel algorithm using $n!$ processors must have a running time of $O(\log n!)$, that is, $O(n \log n)$, in order to be time optimal. In that sense, the $O(n^2)$-time $n$-star algorithm obtained in the preceding is suboptimal by a factor of $O(n/\log n)$. However, it is not known at the time of this writing whether an $O(n \log n)$ running time for sorting $n!$ numbers is achievable on the $n$-star.

A related open question is formulated as follows. Let $G$ be a graph whose set of vertices is $V$ and set of edges is $E$. A *Hamilton cycle* in $G$ is a cycle that starts at some vertex $v$ of $V$, traverses the edges in $E$, visiting each vertex in $V$ exactly once, and finally returns to $v$. Not all graphs possess a Hamilton cycle; those that do are said to be *Hamiltonian*. As it turns out, it is often useful in parallel computation to determine whether the graph underlying an interconnection network is Hamiltonian. When this is the case, the processors can be viewed as forming a *ring*, that is, a linear array with an additional link connecting the first and last processors. Several useful operations can thus be performed efficiently on the data held by the processors. For example, one such operation is a circular shift. Now, while the $n$-star can be shown to be Hamiltonian, it is easily verified that the order defined on the processors by algorithm LABELS does not yield a Hamilton cycle for $n > 3$. Thus, for $n = 4$, $P_{1234}$ and $P_{2341}$, the first and last processors, respectively, are not neighbors. Does there exist an efficient sorting algorithm for the $n$-star when the processors are ordered to form a Hamilton cycle?

## 4. MATRIX MULTIPLICATION

We now turn our attention to the problem of computing the product of two matrices. This is a fundamental computation in many fields, including linear algebra and numerical analysis. Its importance to parallel algorithms, however, goes well beyond its immediate applications. A great many computations are performed efficiently in parallel by expressing each of them as a matrix multiplication.

The problem of matrix multiplication is defined as follows. Let $A$ and $B$ be two $n \times n$ matrices of real numbers. It is required to compute a third $n \times n$ matrix $C$ equal to the product of $A$ and $B$. The elements of $C$ are obtained from

$$c_{jk} = \sum_{i=0}^{n-1} a_{ji} \times b_{ik} \qquad 0 \le j, k \le n-1.$$

A lower bound on the number of operations required to perform this computation is $\Omega(n^2)$. This is easy to see: The result matrix $C$ consists of $n^2$ elements, and that many steps are therefore needed simply to produce it as output. While this is what one might call an *obvious* lower bound, as it is based solely on the size of the output, it turns out to be the highest lower bound known for this problem. Despite concerted efforts, no one has been able to show that more than $n^2$ steps are required to compute $C$.

On the other hand, since $C$ has $n^2$ entries, each of which, by definition, equals the sum of $n$ products, a straightforward sequential algorithm requires $O(n^3)$ time. However, the asymptotically fastest known sequential algorithm for multiplying two $n \times n$ matrices runs in $O(n^\epsilon)$ time, where $2 < \epsilon < 2.38$. Because of the evident gap between the $\Omega(n^2)$ lower bound and the lowest available upper bound of $O(n^\epsilon)$, it is not known whether this algorithm is time-optimal, or whether a faster algorithm exists (but has not yet been discovered).

In the remainder of this section a parallel algorithm is described for computing the product of two matrices on a hypercube interconnection network. Applications of this algorithm in graph theory and numerical computation are then studied. We assume in what follows that $n = 2^q$, for a positive integer $q$.

## 4.1 Hypercube algorithm

The hypercube was briefly introduced in Section 3.4. We now provide a formal specification of the model.

**The Hypercube Model.** Let $N = 2^d$ processors $P_0$, $P_1$, …, $P_{N-1}$ be available, for $d \geq 1$. Further, let $i$ and $i^{(b)}$ be two integers, $0 \leq i, i^{(b)} \leq N-1$, whose binary representations differ only in position $b$, $0 \leq b < d$. Specifically, if

$$i_{d-1}i_{d-2}\dots i_{b+1}i_b i_{b-1}\dots i_1 i_0$$

is the binary representation of $i$, then

$$i_{d-1}i_{d-2}\dots i_{b+1}i'_b i_{b-1}\dots i_1 i_0$$

is the binary representation of $i^{(b)}$, where $i'_b$ is the binary complement of bit $i_b$. A *d-dimensional hypercube interconnection network* is formed by connecting each processor $P_i$, $0 \leq i \leq N-1$, to $P_{i^{(b)}}$ by a two-way link, for all $0 \leq b < d$. Thus, each processor has $d$ neighbors. This is illustrated in Fig. 4.1(a)–(d), for $d = 1, 2, 3$ and 4, respectively. The indices of the processors are given in binary notation. Note that the hypercube in dimension $d$ is obtained by connecting corresponding processors in two $(d-1)$-dimensional hypercubes (a 0-dimensional hypercube being a single processor). It is also easy to see that the diameter of a $d$-dimensional hypercube is precisely $d$.

**Matrix Multiplication on the Hypercube.** The hypercube algorithm for computing the product of two matrices is essentially an implementation of the straightforward sequential algorithm. In order to multiply the two $n \times n$ matrices $A$ and $B$, where $n = 2^q$, we use a hypercube with $N = n^3 = 2^{3q}$ processors. It is helpful to visualize the processors as being arranged in an $n \times n \times n$ array, with processor $P_r$ occupying position $(i, j, k)$, where $r = in^2 + jn + k$ and $0 \leq i, j, k \leq n-1$. Thus, if the binary representation of $r$ is

$$r_{3q-1}r_{3q-2}\dots r_{2q}r_{2q-1}\dots r_q r_{q-1}\dots r_0,$$

then the binary representations of $i$, $j$, and $k$ are

$$r_{3q-1}r_{3q-2}\dots r_{2q}, \quad r_{2q-1}r_{2q-2}\dots r_q, \quad \text{and} \quad r_{q-1}r_{q-2}\dots r_0,$$

respectively. Note that in the $n \times n \times n$ array, all processors agreeing on one or two of the coordinates $(i, j, k)$ form a hypercube. Specifically, all processors with the same index value in one of the three coordinates form a hypercube with $n^2$ processors; similarly, all processors with the same index values in two fixed coordinates form a hypercube with $n$ processors.

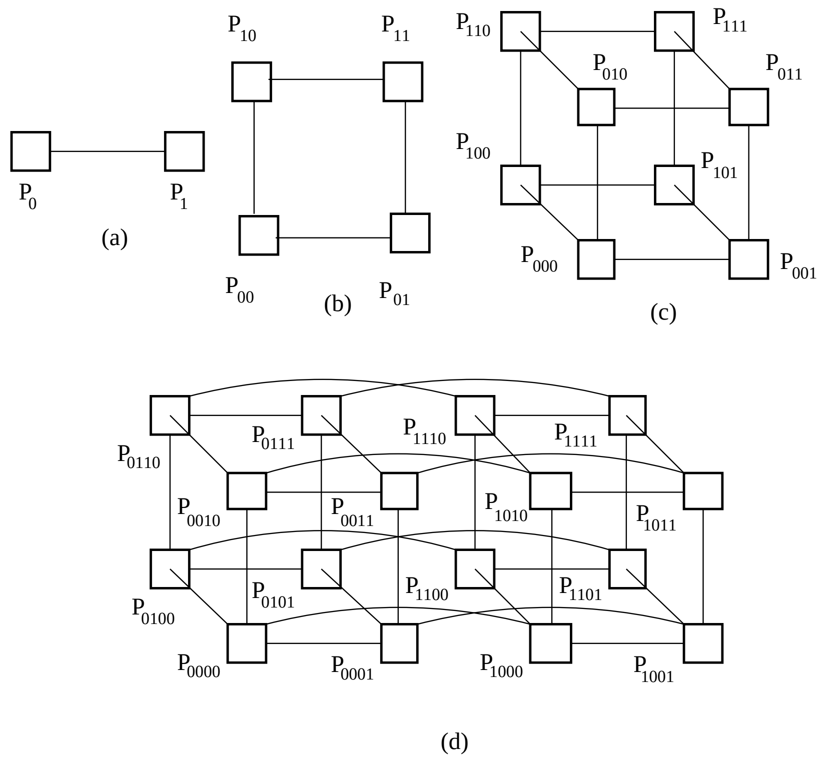


**Fig. 4.1.** A $d$-dimensional hypercube for $d = 1, 2, 3$ and 4.

We assume in what follows that each processor $P_r$ has three registers $A_r$, $B_r$, and $C_r$. For notational convenience, the same three registers are also denoted $A(i,j,k)$, $B(i,j,k)$, and $C(i,j,k)$, respectively. Initially, processor $P_s$ in position $(0,j,k)$, $0 \leq j \leq n-1$, $0 \leq k \leq n-1$, contains element $a_{jk}$ of matrix $A$ and element $b_{jk}$ of matrix $B$ in registers $A_s$ and $B_s$, respectively. The registers of all other processors are initialized to 0. At the end of the computation, register $C_s$ of $P_s$ should contain element $c_{jk}$ of the product matrix $C$. The algorithm is designed to perform the $n^3$ multiplications involved in computing the $n^2$ entries of matrix $C$ simultaneously. It is given in what follows.

**Algorithm HYPERCUBE MATRIX MULTIPLICATION**

**Step 1:** The elements of matrices $A$ and $B$ are distributed over the $n^3$ processors so that the processor in position $(i,j,k)$ contains $a_{ji}$ and $b_{ik}$. This is done as follows:

(1.1) Copies of data initially in $A(0, j, k)$ and $B(0, j, k)$, are sent to the processors in positions $(i, j, k)$, where $1 \le i \le n-1$. As a result, $A(i, j, k) = a_{jk}$ and $B(i, j, k) = b_{jk}$, for $0 \le i \le n-1$. Formally,

```
for $m = 3q - 1$ downto $2q$ do
  for all $r$ such that $r_m = 0$ do in parallel
    (i) $A_{r^{(m)}} \leftarrow A_r$
    (ii) $B_{r^{(m)}} \leftarrow B_r$
  end for
end for
```

(1.2) Copies of the data in $A(i, j, i)$ are sent to the processors in positions $(i, j, k)$, where $0 \le k \le n-1$. As a result, $A(i, j, k) = a_{ji}$ for $0 \le k \le n-1$. Formally,

```
for $m = q - 1$ downto $0$ do
  for all $r$ such that $r_m = r_{2q+m}$ do in parallel
    $A_{r^{(m)}} \leftarrow A_r$
  end for
end for
```

(1.3) Copies of the data in $B(i, i, k)$ are sent to the processors in positions $(i, j, k)$, where $0 \le j \le n-1$. As a result, $B(i, j, k) = b_{ik}$ for $0 \le j \le n-1$. Formally,

```
for $m = 2q - 1$ downto $q$ do
  for all $r$ such that $r_m = r_{q+m}$ do in parallel
    $B_{r^{(m)}} \leftarrow B_r$
  end for
end for
```

**Step 2:** Each processor in position $(i, j, k)$ computes the product

$$C(i, j, k) \leftarrow A(i, j, k) \times B(i, j, k).$$

Thus, $C(i, j, k) = a_{ji} \times b_{ik}$ for $0 \le i, j, k \le n-1$. Formally,

```
for $r = 0$ to $N - 1$ do in parallel
  $C_r \leftarrow A_r \times B_r$
end for
```

**Step 3:** The sum

$$C(0, j, k) \leftarrow \sum_{i=0}^{n-1} C(i, j, k)$$

is computed for $0 \le j, k \le n-1$. Formally,

```
for $m = 2q$ to $3q - 1$ do
  for all $r \in N(r_m = 0)$ do in parallel
    $C_r \leftarrow C_r + C_{r^{(m)}}$
  end for
end for. ■
```

**Analysis.** Each of Steps (1.1), (1.2), (1.3), and 3 consists of $q$ constant-time iterations. Step 2 requires constant time. Therefore, the algorithm has a running time of $O(q)$—that is, $t(n) = O(\log n)$. Since $p(n) = n^3$, the algorithm's cost is $c(n) = O(n^3 \log n)$. This cost is not optimal since $O(n^3)$ basic operations suffice to multiply two $n \times n$ matrices by the straightforward sequential algorithm based on the definition of the matrix product.

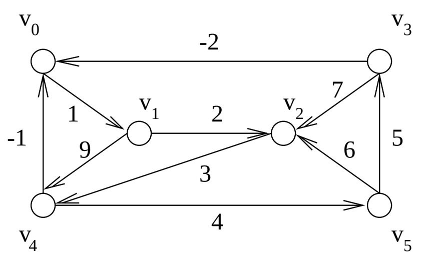
## 4.2 Graph theoretical problems

Graphs are important modeling tools in many fields of knowledge, particularly in science and engineering. Graph theory is that branch of discrete mathematics whose purpose is the study of graphs and their algorithms. In this section we show how two problems defined on graphs can be solved efficiently in parallel through the use of matrix multiplication. We begin with some definitions.

A *graph* $G = (V, E)$ is a set of vertices $V$ connected by a set of edges $E$. If the edges have no orientation, then the graph is *undirected*; thus, the edge connecting two vertices $v_1$ and $v_2$ can be traversed in either way, from $v_1$ to $v_2$ or from $v_2$ to $v_1$. By contrast, a graph is *directed* if each edge has an orientation; thus, an edge connecting $v_1$ to $v_2$ can only be traversed from $v_1$ to $v_2$. A *path* in a graph is an ordered list of edges of the form $(v_i, v_j), (v_j, v_k), (v_k, v_l)$, and so on. If, for every pair of vertices $v_p$ and $v_q$ in an undirected graph, there is a path leading from $v_p$ to $v_q$, then the graph is said to be *connected.* A directed graph is connected if the undirected graph obtained from it by ignoring the edge orientations is connected. A *cycle* in a graph is a path that begins and ends at the same vertex.

When each edge of a graph is associated with a real number, called its *weight,* the graph is said to be *weighted.* A weighted graph may be directed or undirected. The meaning of an edge's weight varies from one application to another; it may represent distance, cost, time, probability, and so on. A *weight matrix* $W$ is used to represent a weighted graph. Here, entry $w_{ij}$ of $W$ represents the weight of edge $(v_i, v_j)$. If $v_i$ and $v_j$ are not connected by an edge, then $w_{ij}$ may be equal to zero, infinity, or any appropriate value, depending on the application.

**All-Pairs Shortest Paths.** Suppose that we are given a directed and weighted graph $G = (V, E)$, with $n$ vertices, as shown in Fig. 4.2(a). The graph is defined by its weight matrix $W$, as shown in Fig. 4.2(b). We assume that $W$ has positive, zero, or negative entries, as long as there is no cycle in $G$ such that the sum of the weights of the edges on the cycle is negative.



(a)

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & 2 & \infty & 9 & \infty \\ 2 & \infty & \infty & 0 & \infty & 3 & \infty \\ 3 & -2 & \infty & 7 & 0 & \infty & \infty \\ 4 & -1 & \infty & \infty & \infty & 0 & 4 \\ 5 & \infty & \infty & 6 & 5 & \infty & 0 \end{array}$$

(b)

**Fig. 4.2.** A directed and weighted graph and its weight matrix.

The problem that we address here is known as the *all-pairs shortest paths problem* and is stated as follows: For every pair of vertices $v_i$ and $v_j$ in $V$, it is required to find the length of the shortest path from $v_i$ to $v_j$ along edges in $E$. Specifically, a matrix $D$ is to be constructed such that $d_{ij}$ is the length of the shortest path from $v_i$ to $v_j$ in $G$, for all $i$ and $j$. Here, the length of a path (or cycle) is the sum of the weights of the edges forming it. In Fig. 4.2, the shortest path from $v_0$ to $v_4$ is along edges $(v_0, v_1)$, $(v_1, v_2)$, $(v_2, v_4)$ and has length 6. It may be obvious now why we insisted that $G$ have no cycle of negative length: If such a cycle were to exist within a path from $v_i$ to $v_j$, then one could traverse this cycle indefinitely, producing paths of ever shorter length from $v_i$ to $v_j$.

Let $d_{ij}^k$ denote the length of the shortest path from $v_i$ to $v_j$ that goes through at most $k-1$ intermediate vertices. Thus, $d_{ij}^1 = w_{ij}$ — that is, the weight of the edge from $v_i$ to $v_j$. In particular, if there is no edge from $v_i$ to $v_j$, where $i \neq j$, then $d_{ij}^1 = w_{ij} = \infty$. Also, $d_{ii}^1 = w_{ii} = 0$. Given that $G$ has no cycles of negative length, there is no advantage in visiting any vertex more than once in a shortest path from $v_i$ to $v_j$. It follows that $d_{ij} = d_{ij}^{n-1}$, since there are only $n$ vertices in $G$.

In order to compute $d_{ij}^k$ for $k > 1$, we can use the recurrence

$$d_{ij}^k = \min_l (d_{il}^{k/2} + d_{lj}^{k/2}).$$

The validity of this relation is established as follows: Suppose that $d_{ij}^k$ is the length of the *shortest* path from $v_i$ to $v_j$ and that two vertices $v_r$ and $v_s$ are on this shortest path (with $v_r$ preceding $v_s$). It must be the case that the edges from $v_r$ to $v_s$ (along the shortest path from $v_i$ to $v_j$) form a shortest path from $v_r$ to $v_s$. (If a shorter path from $v_r$ to $v_s$ existed, it could be used to obtain a shorter path from $v_i$ to $v_j$, which is absurd.) Therefore, to obtain $d_{ij}^k$, we can compute all combinations of *optimal subpaths* (whose concatenation is a path from $v_i$ to $v_j$) and then choose the shortest one. The fastest way to do this is to combine pairs of subpaths with at most $k/2$ vertices each. This guarantees that a recursive computation of $d_{ij}^k$ can be completed in $O(\log k)$ steps.

Let $D^k$ be the matrix whose entries are $d_{ij}^k$, for $0 \leq i, j \leq n-1$. In accordance with the discussion in the previous two paragraphs, the matrix $D$ can be computed from $D^1$ by evaluating $D^2$, $D^4$, ..., $D^m$, where $m$ is the smallest power of 2 larger than or equal to $n-1$ (i.e., $m = 2^{\lceil \log(n-1) \rceil}$), and then taking $D = D^m$. In order to obtain $D^k$ from $D^{k/2}$, we use a special form of matrix multiplication in which the operations '+' and 'min' replace the standard operations of matrix multiplication—that is, '$\times$' and '+', respectively. Hence, if a matrix multiplication algorithm is available, it can be modified to generate $D^m$ from $D^1$. Exactly $\lceil \log(n-1) \rceil$ such matrix products are required.

In particular, algorithm HYPERCUBE MATRIX MULTIPLICATION, appropriately modified, can be used to compute the shortest path matrix $D$. The resulting algorithm, to which we refer as algorithm HYPERCUBE SHORTEST PATHS, has a running time of $t(n) = O(\log^2 n)$. Since $p(n) = n^3$, the algorithm's cost is $c(n) = O(n^3 \log^2 n)$.

**Minimum Spanning Tree.** An undirected graph is a *tree* if it is connected and contains no cycles. Let $G = (V, E)$ be an undirected, connected, and weighted graph with $n$ vertices. A *spanning tree* of $G$ is a connected subgraph $G' = (V, E')$ that contains no cycles. A *minimum-weight spanning tree* (MST) of $G$ is a spanning tree of $G$ with the smallest edge-weight sum.

We now show how algorithm HYPERCUBE SHORTEST PATHS can be used to compute an MST for a given graph $G$. Let $w_{ij}$ be the weight of edge $(v_i, v_j) \in E$. In what follows, we assume that all $w_{ij}$ are distinct. (If, for two edges $(v_i, v_j)$ and $(v_{i'}, v_{j'})$, $w_{ij} = w_{i'j'}$, then $w_{ij}$ is considered smaller than $w_{i'j'}$ if $in + j < i'n + j'$; otherwise, $w_{i'j'}$ is considered smaller than $w_{ij}$.) Our algorithm uses the following property of MST edges: Edge $(v_i, v_j) \in$ MST if and only if, on every path from $v_i$ to $v_j$ consisting of two or more edges, there is an edge $(v_{i'}, v_{j'})$ such that $w_{i'j'} > w_{ij}$.

*Example 4.1.* A graph $G$ is illustrated in Fig. 4.3, with its MST shown by thick lines. Note that edge $(v_1, v_3) \in$ MST, since every path of length 2 or more from $v_1$ to $v_3$ contains an edge whose weight is larger than $w_{13}$. On the other hand, $(v_0, v_5) \notin$ MST, since every edge on the path from $v_0$ to $v_5$ in MST has weight less than $w_{05}$. $\square$

Given $G$, an algorithm for computing MST is now obtained as follows: Algorithm HYPERCUBE SHORTEST PATHS is modified so that the length of a path is now equal to the longest edge it contains (as opposed to the sum

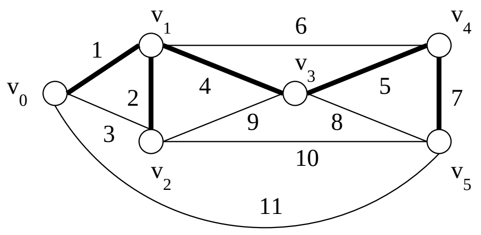


**Fig. 4.3.** A graph and its MST

of the lengths of the edges). With this new definition, all shortest paths are computed by replacing the statement

$$d_{ij}^k \leftarrow \min_l (d_{il}^{k/2} + d_{lj}^{k/2})$$

in algorithm HYPERCUBE SHORTEST PATHS with

$$d_{ij}^k \leftarrow \min_l (\max(d_{il}^{k/2}, d_{lj}^{k/2})).$$

This means that, for every path from $v_i$ to $v_j$, we find the longest edge in the path, and then we determine the shortest of these edges, whose length is $d_{ij}^{n-1}$. It is now possible to determine those edges of $E$ which belong in the minimum-weight spanning tree. This is done using the condition

$$(v_i, v_j) \in \text{ MST} \quad \text{if and only if} \quad w_{ij} = d_{ij}^{n-1}.$$

It follows that the MST of a graph $G$ with $n$ vertices can be computed in $O(\log^2 n)$ time on a hypercube with $n^3$ processors.

## 4.3 Numerical computations

Matrix operations arise most naturally in numerical computations, such as the solution of a set of linear equations, to cite a well-known example. In this section we illustrate how matrix multiplication can be used as a powerful tool to address a fundamental problem in linear algebra.

Let $A$ be an $n \times n$ matrix whose *inverse* $A^{-1}$ we seek to compute, such that $AA^{-1} = A^{-1}A = I$. Here $I$ is an $n \times n$ identity matrix (whose main diagonal elements are 1, and all the rest are 0). Our derivation of a parallel algorithm for computing $A^{-1}$ proceeds in stages. We begin by developing a parallel algorithm for inverting $A$ in the special case where $A$ is a lower triangular matrix.

**Inverting a Lower Triangular Matrix.** Let $A$ be an $n \times n$ *lower triangular matrix*—that is, a matrix all of whose entries above the main diagonal are 0. It is desired to compute the inverse $A^{-1}$ of $A$. We begin by writing

$$A = \begin{pmatrix} B & Z \\ C & D \end{pmatrix}$$

where $B$, $C$, and $D$ are $(n/2) \times (n/2)$ submatrices of $A$ and $Z$ is an $(n/2) \times (n/2)$ zero matrix. It follows that

$$A^{-1} = \begin{pmatrix} B^{-1} & Z \\ -D^{-1}CB^{-1} & D^{-1} \end{pmatrix}.$$

If $B^{-1}$ and $D^{-1}$ are computed recursively and simultaneously, a parallel algorithm for $A^{-1}$ would have a running time of $t(n) = t(n/2) + 2r(n/2)$, where $r(n/2)$ is the time it takes to multiply two $(n/2)\times(n/2)$ matrices. Using algorithm HYPERCUBE MATRIX MULTIPLICATION, we have $r(n) = O(\log n)$. Therefore, $t(n) = O(\log^2 n)$.

**The Characteristic Equation of a Matrix.** In order to take advantage of the algorithm just developed for inverting a lower triangular matrix, we need two definitions and a theorem from linear algebra.

Let $A$ be an $n\times n$ matrix. When $n > 1$, we denote by $A_{ij}$ the $(n-1)\times(n-1)$ matrix obtained from $A$ by deleting row $i$ and column $j$. The *determinant* of $A$ denoted $det(A)$, is defined as follows: For $n = 1$, $det(A) = a_{11}$, and for $n > 1$, $det(A) = a_{11}\,det(A_{11}) - a_{12}\,det(A_{12}) + \cdots + (-1)^{1+j}a_{1j}\; det(A_{1j}) + \cdots + (-1)^{n+1}a_{1n}det(A_{1n})$. If $det(A) \neq 0$, then $A$ is said to be *nonsingular*; in this case, the inverse $A^{-1}$ is guaranteed to exist (and, conversely, if $A^{-1}$ exists, then $det(A) \neq 0$).

The *characteristic polynomial* $\phi(x)$ of an $n \times n$ matrix $\boldsymbol{A}$ is defined to be

$$\begin{aligned} \phi(x) &= det(xI - A) \\ &= x^n + h_1x^{n-1} + h_2x^{n-2} + \cdots + h_n. \end{aligned}$$

Note, in particular, that for $x = 0$, $det(-A) = h_n$. Therefore, $det(A) = (-1)^n h_n$.

The *Cayley-Hamilton Theorem* states that every $n \times n$ matrix $A$ satisfies its own characteristic polynomial. Thus, taking $x = A$ yields the *characteristic equation*

$$A^n + h_1A^{n-1} + h_2A^{n-2} + \cdots + h_{n-1}A + h_nI \;=\; 0.$$

Multiplying by $A^{-1}$ yields

$$A^{n-1} + h_1A^{n-2} + h_2A^{n-3} + \cdots + h_{n-1}I + h_nA^{-1} \;=\; 0.$$

Therefore,

$$A^{-1} \;=\; -\frac{1}{h_n}(A^{n-1} + h_1A^{n-2} + \cdots + h_{n-2}A + h_{n-1}I).$$

Now, the coefficients $h_1, h_2, \ldots, h_n$ satisfy

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ s_1 & 2 & 0 & \dots & 0 \\ s_2 & s_1 & 3 & \dots & 0 \\ & & \vdots & & \\ s_{n-1} & \dots & s_2 & s_1 & n \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ \vdots \\ h_n \end{pmatrix} = - \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_n \end{pmatrix},$$

or $Sh = -s$, where $s_i$ denotes the *trace* of $A^i$—that is, the sum of the diagonal elements of $A^i$, for $1 \leq i \leq n$. The system of equations $Sh = -s$ can be solved for $h$ by computing the inverse of $S$ and writing: $h = -S^{-1}s$. Since $S$ is lower triangular, it can be inverted using the algorithm previously developed for that purpose.

**Inverting an Arbitrary Matrix.** The algorithm for inverting an arbitrary $n \times n$ matrix $A$ is now straightforward and is given next:

**Step 1:** Compute the powers $A^2$, $A^3$, ..., $A^n$ of the matrix $A$ (taking $A^1 = A$).
**Step 2:** Compute $s_i$ by summing the diagonal elements of $A^i$ for $i = 1, 2, \dots, n$.
**Step 3:** Compute $h_1$, $h_2$, ..., $h_n$ from the equation $h = -S^{-1}s$.
**Step 4:** Compute $A^{-1} = (-1/h_n)(A^{n-1} + h_1 A^{n-2} + \cdots + h_{n-2}A + h_{n-1}I)$.
■

**Analysis.** Each of the powers $A^i$ in Step 1 can be computed in $O(\log n) \times O(\log i)$ time with $n^3$ processors by repeated squaring and multiplication of matrices, using algorithm HYPERCUBE MATRIX MULTIPLICATION. Step 1 therefore requires $O(n^4)$ processors and $O(\log^2 n)$ time if all matrices $A^i$, $2 \leq i \leq n$, are to be computed simultaneously.

Step 2 is performed as follows: The elements of the matrix $A^i$ reside in the base layer of a hypercube with $n^3$ processors, viewed as an $n \times n \times n$ array. In particular, the diagonal elements are held by the processors in positions $(0, j, j)$ of the array, for $j = 0, 1, \dots, n-1$. These processors do not form a hypercube. However, because each row of the base layer is a hypercube of $n$ processors, the diagonal elements can be routed to the processors in positions $(0,0,0)$, $(0,1,0)$, ..., $(0, n-1, 0)$, respectively, in $O(\log n)$ time. The latter processors, in turn, form a hypercube and can compute the sum of the diagonal elements in $O(\log n)$ time. Thus, each of the $s_i$ can be computed in $O(\log n)$ time using $n$ processors.

Inverting the lower triangular matrix $S$ in Step 3 requires $O(\log^2 n)$ time and $n^3$ processors. This is followed by a matrix-by-vector product to obtain $h_1$, $h_2$, ..., $h_n$. The latter computation is easy to execute on the hypercube as a special case of matrix-by-matrix multiplication and requires $O(\log n)$ time and $n^2$ processors.

Finally, in Step 4, the simple operation of multiplying a matrix by a real number is immediate, while computing the sum of a collection of matrices consists in applying the method used in Step 3 of algorithm HYPERCUBE MATRIX MULTIPLICATION. Thus, Step 4 can be performed in $O(\log n)$ time using $O(n^3)$ processors.

The overall running time is therefore $t(n) = O(\log^2 n)$, while the number of processors used is $p(n) = O(n^4)$.

## 5. COMPUTING THE CONVEX HULL

Computational geometry is that branch of computer science whose focus is the design and analysis of efficient algorithms for the solution of geometric problems, that is, problems involving points, lines, polygons, circles, and the like. Solutions to such problems have applications in graphics, image processing, pattern recognition, operations research, and computer-aided design and manufacturing, to name just a few. A typical problem in computational geometry is that of computing the convex hull of a set of points in the plane. In this section, we develop a parallel algorithm for solving the convex hull problem on the PRAM model of computation. Our solution illustrates the algorithm design method of *divide and conquer*. In this method, a problem to be solved is broken into a number of subproblems of the same form as the original problem; this is the *divide* step. The subproblems are then solved independently, usually recursively; this is the *conquer* step. Finally, the solutions to the subproblems are combined to provide the answer to the original problem. In conjunction with the divide and conquer method, our solution to the convex hull problem uses a number of techniques that are interesting in their own right due to their wide applicability, namely, computing prefix sums, array packing, and sorting by merging. We begin by describing these techniques.

### 5.1 Prefix sums on the PRAM

We are given a sequence $X = (x_0, x_1, \ldots, x_{n-1})$ of $n$ numbers. For simplicity, let $n$ be a power of 2. It is required to compute the following sums, known as the *initial* or *prefix* sums: $s_0 = x_0$, $s_1 = x_0 + x_1$, $s_2 = x_0 + x_1 + x_2$, $\ldots$, $s_{n-1} = x_0 + x_1 + \cdots + x_{n-1}$. A sequential algorithm to solve this problem reads one number at a time and executes $n - 1$ additions in sequence. The time required is $O(n)$, which is optimal in view of the $\Omega(n)$ operations required to compute just $s_{n-1}$.

On a PRAM with $n$ processors $P_0, P_1, \ldots, P_{n-1}$, the problem can be solved much faster. Initially, with all processors operating in parallel, $P_i$ reads $x_i$, sets $s_i = x_i$, and stores $s_i$ in the shared memory, for $i = 0, 1, \ldots, n-1$. The sequence $S = (s_0, s_1, \ldots, s_{n-1})$ is assumed to be stored in an array of contiguous locations. The parallel algorithm now consists of $\log n$ steps. During each step, two numbers are added whose indices are twice the distance in the previous step. The algorithm is as follows:

**Algorithm PRAM PREFIX SUMS**

```
for $j = 0$ to $(\log n) - 1$ do
  for $i = 2^j$ to $n - 1$ do in parallel
    $s_i \leftarrow s_{i-2^j} + s_i$
  end for
end for. ■
```

It is easy to see that each iteration yields twice as many final values $s_i$ as the previous one. The algorithm therefore runs in $O(\log n)$ time. Since $p(n) = n$, the algorithm's cost is $c(n) = O(n \log n)$. This cost is not optimal, in view of the $O(n)$ operations that are sufficient to solve the problem sequentially.

We now show how to obtain a cost-optimal algorithm for computing the sequence $S = (s_0, s_1, \ldots, s_{n-1})$. The algorithm is based on the design technique of divide and conquer and proceeds as follows. Initially, $s_i = x_i$ for $i = 0, 1, \ldots, n-1$.

1. The sequence $S$ is divided into $n^{1/2}$ subsequences of size $n^{1/2}$ elements each. The prefix sums of each subsequence are now computed recursively, using $n^{1/2}$ processors. This requires $t(n^{1/2})$ time. Let the result of this computation over the $i$th sequence be $(s(i,1), s(i,2), \ldots, s(i,n^{1/2}))$.
2. The prefix sums $(s'(1,n^{1/2}), s'(2,n^{1/2}), \ldots, s'(n^{1/2}-1,n^{1/2}))$ of the sequence $(s(1,n^{1/2}), s(2,n^{1/2}), \ldots, s(n^{1/2}-1,n^{1/2}))$, are computed using $n$ processors. This can be done in constant time by assigning $k$ processors, denoted by $P(k,1), P(k,2), \ldots, P(k,k)$, to the computation of the $k$th prefix sum $s'(k,n^{1/2}) = s(1,n^{1/2}) + s(2,n^{1/2}) + \cdots + s(k,n^{1/2})$, for $1 \leq k \leq n^{1/2} - 1$. With all processors operating in parallel, $P(k,j)$ writes $s(j,n^{1/2})$ into $s'(k,n^{1/2})$, such that the *sum* of all the values written ending up in $s'(k,n^{1/2})$.
3. The sum $s'(i,n^{1/2})$ is now added to each element of the sequence $(s(i+1,1), s(i+1,2), \ldots, s(i+1,n^{1/2}))$ for $i = 1, 2, \ldots, n^{1/2}-1$, using $n^{1/2}$ processors per sequence. This requires constant time. ■

The total number of processors used is therefore $O(n)$, and the running time is $t(n) = t(n^{1/2}) + a$, where $a$ is a positive constant. Thus, $t(n) = O(\log \log n)$, for a cost of $O(n \log \log n)$.

This cost can be reduced to $O(n)$ by using only $O(n/\log \log n)$ processors. Initially, each processor is assigned $O(\log \log n)$ elements of the input sequence. With all processors operating in parallel, each processor computes the prefix sums of its assigned subsequence sequentially in $O(\log \log n)$ time. The sequence formed by the last prefix sum computed by each processor is now fed as input to the algorithm described in the preceding Steps 1 - 3. Each prefix sum thus computed is now added to $O(\log \log n)$ elements obtained in the initial step, as done in Step 3.

The technique described in the previous paragraph can be also be applied to obtain a cost-optimal algorithm with a lower processor requirement, namely, $O(n/\log n)$, and a higher running time, namely, $O(\log n)$.

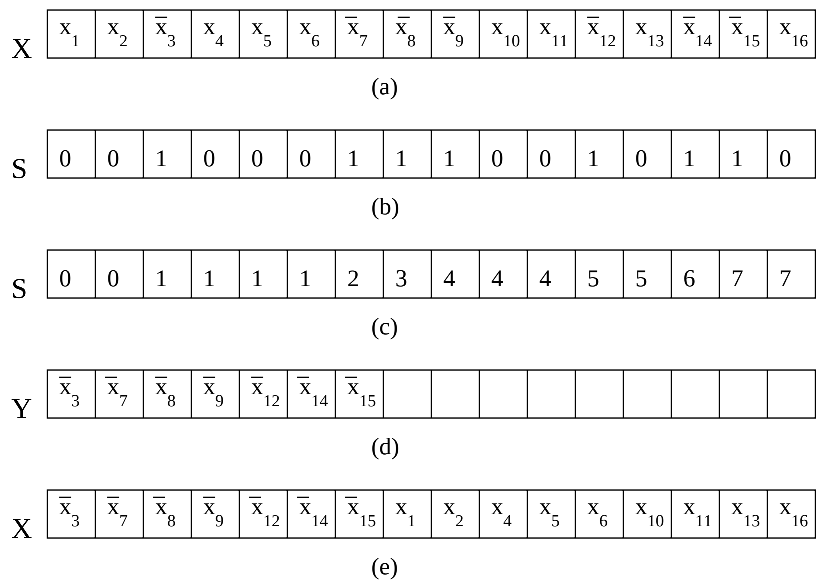

X $x_1$ $x_2$ $\bar{x}_3$ $x_4$ $x_5$ $x_6$ $\bar{x}_7$ $\bar{x}_8$ $\bar{x}_9$ $x_{10}$ $x_{11}$ $\bar{x}_{12}$ $x_{13}$ $\bar{x}_{14}$ $\bar{x}_{15}$ $x_{16}$
(a)
S 0 0 1 0 0 0 1 1 1 0 0 1 0 1 1 0
(b)
S 0 0 1 1 1 1 2 3 4 4 4 5 5 6 7 7
(c)
Y $\bar{x}_3$ $\bar{x}_7$ $\bar{x}_8$ $\bar{x}_9$ $\bar{x}_{12}$ $\bar{x}_{14}$ $\bar{x}_{15}$
(d)
X $\bar{x}_3$ $\bar{x}_7$ $\bar{x}_8$ $\bar{x}_9$ $\bar{x}_{12}$ $\bar{x}_{14}$ $\bar{x}_{15}$ $x_1$ $x_2$ $x_4$ $x_5$ $x_6$ $x_{10}$ $x_{11}$ $x_{13}$ $x_{16}$
(e)

**Fig. 5.1.** Packing labeled and unlabeled data in an array.

## 5.2 Array packing

Of all the applications of prefix sums computation to the design of efficient parallel algorithms, one of the most common is packing an array. Suppose that an array $X$ of $n$ elements is given, some of whose entries are labeled. The labeled entries are scattered arbitrarily throughout the array, as illustrated in Fig. 5.1(a) for the array $X = (x_1, x_2, \ldots, x_{16})$, where a label is denoted by a bar over $x_i$. Such an array may arise in an application in which a set of data is given and it is required to identify those data satisfying a certain condition. A datum satisfying the condition is labeled.

Once the data have been labeled, it is required to bring the labeled elements into contiguous positions in the same or another array. This may be necessary for a proper output of the results, or because the labeled elements are to undergo further processing and they are expected to appear in adjacent positions in the next stage of the computation.

Sequentially, the problem is solved by "burning the candle at both ends:" Two pointers $q$ and $r$ are used, where $q = 1$ and $r = n$ initially. The pointers are moved in opposite directions, $q$ to the right and $r$ to the left; $q$ advances if $x_q$ is a labeled element, whereas $r$ advances if $x_r$ is an unlabeled element. If at any time $x_q$ is unlabeled and $x_r$ is labeled, these two elements switch positions. When $q \geq r$, the labeled elements appear in adjacent positions in the first part of the array. This takes $O(n)$ time, which is optimal.

In parallel, we use a secondary array of $n$ elements $S = (s_1, s_2, \ldots, s_n)$ to compute the destination of each labeled element of $X$. Initially, $s_i = 1$, provided that $x_i$ is labeled, and $s_i = 0$, otherwise, as illustrated in Fig. 5.1(b). A prefix sums computation is then performed on the elements of $S$, as shown in Fig. 5.1(c). Finally, each labeled element $x_i$ of $X$ is copied into position $s_i$ of an array $Y$. This is illustrated in Fig. 5.1(d). Note that the labeled elements of $X$ occupy the first $s_n$ positions of $Y$. The array $S$ can be created with $n$ processors in $O(1)$ time, each processor filling one positions of $S$. The prefix sums are computed by algorithm PRAM PREFIX SUMS in $O(\log n)$ time using $n$ processors. Copying the labeled elements of $X$ into $Y$ can be done by $n$ processors in $O(1)$ time. Therefore, this algorithm, which we call PRAM ARRAY PACKING, runs in $O(\log n)$ time and uses $O(n)$ processors. The algorithm's time and processor requirements can be reduced in a straightforward manner as described in the Section 5.1. It is interesting to observe here that, unlike the sequential solution described in the preceding, the parallel algorithm packs the labeled elements while maintaining their order (i.e., their original positions relative to one another).

In some cases, it may be necessary to pack the labeled elements of $X$ within $X$ itself: Once their destinations are known, the labeled elements can be copied directly into the first $s_n$ positions of $X$. However, this would overwrite some unlabeled elements of $X$. Should this be undesirable, it can be avoided as follows:

1. The destinations of the $s_n$ labeled elements are first computed.
2. The same procedure is then applied to the $n - s_n$ unlabeled elements.
3. Simultaneously, the labeled and unlabeled elements are copied into the first $s_n$ and last $n - s_n$ positions of $X$, respectively.

This is shown in Fig. 5.1(e).

### 5.3 Sorting on the PRAM

As pointed out in Section 3., sorting is fundamental to computer science. For many problems, sorting a sequence of data is an inherent part of the solution, both in sequential as well as in parallel algorithms. The interconnection-network algorithms for sorting described in Section 3. can be easily simulated on the PRAM to run within the same time and processor requirements: Whenever two processors need to communicate they do so through the shared memory in constant time. In this section, however, we present a sorting algorithm that is specifically designed for the PRAM. We do so for two reasons. First, the algorithm is based on merging and hence offers a new paradigm for parallel sorting. Second, and more important, the algorithm is significantly faster than any simulation of the interconnection-network algorithms, and in addition is cost optimal.

Suppose that a sequence of $n$ numbers (in arbitrary order) $X = (x(1), x(2), \ldots, x(n))$ is to be sorted in nondecreasing order. The sequence $X$ is stored in an array in shared memory, one number per array element. The array elements are thought of as being the leaves of a virtual complete binary tree $T$. Conceptually, the algorithm moves the numbers up the tree: At each node, the sequences received from the left and right children of the node are merged and the resulting sorted sequence is sent to the node's parent. When the algorithm terminates, the initial sequence of $n$ values emerges from the root of the tree in sorted order. Because our aim is to achieve a total running time of $O(\log n)$, the merging process is pipelined. Thus, the algorithm works at several levels of the tree at once, overlapping the merging at different nodes over time. Henceforth, we refer to this as algorithm PRAM SORT.

**Algorithm PRAM SORT.** Let $L_v$ be the sequence of values stored in the leaves of the subtree of $T$ whose root is an internal (i.e., nonleaf) node $v$. At the $j$th time step, $v$ contains a *sorted* sequence $Q_v(j)$ whose elements are selected from $L_v$. Furthermore, the sequence $Q_v(j)$ is an *increasing* subsequence of $L_v$. Eventually, when $Q_v(j) = L_v$, node $v$ is said to be *complete.* Also, all leaves are said to be complete by definition. During the algorithm, a node $v$ whose parent is not complete at the $j$th step sends a sorted subsequence $R_v(j)$ of $Q_v(j)$ to its parent.

How is $Q_v(j)$ created? Let $w$ and $z$ be the children of $v$. Node $v$ merges $R_w(j)$ and $R_z(j)$ to obtain $Q_v(j)$, where $R_w(j)$ and $R_z(j)$ are themselves formed as follows:

1. If $w$ is not complete during the $(j-1)$st step, then $R_w(j)$ consists of every fourth element of $Q_w(j-1)$.
2. If $w$ becomes complete during the $j$th step, then:
   (i) $R_w(j+1)$ consists of every fourth element of $Q_w(j)$.
   (ii) $R_w(j+2)$ consists of every second element of $Q_w(j)$.
   (iii) $R_w(j+3) = Q_w(j)$.

Consequently, if $w$ and $z$ become complete during the $j$th step, then $v$ becomes complete during step $j+3$. It follows that the root of $T$ becomes complete during step $3 \log n$. At this point, the root contains the input sequence of $n$ values in sorted order, and the algorithm terminates.

The only detail left is to show how the sequences $R_w(j)$ and $R_z(j)$ are merged in constant time. Given two sequences of values, the *predecessor* of an element in one sequence is the largest element in the other sequence that is smaller than it (if such an element exists). Suppose that each element of $R_w(j)$ $(R_z(j))$ knows the position of its predecessor in $R_z(j)$ $(R_w(j))$. In that case, $R_w(j)$ and $R_z(j)$ can be merged in constant time using $|R_w(j)| + |R_z(j)|$ processors, each processor directly placing one element in its final position in $Q_v(j)$. To make this possible, certain necessary information about predecessors is maintained. To wit, after step $j-1$:

1. The elements of $R_w(j-1)$ "know" their predecessors in $R_z(j-1)$, and vice versa, these two sequences having just been merged to form $Q_v(j-1)$.

2. Each element of $R_w(j-1)$ finds its predecessor in $Q_w(j-1)$ in constant time. Consequently, all elements in $R_w(j-1)$ can determine their predecessor in $R_w(j)$, also in constant time. Note that no more than four elements of $R_w(j-1)$ have the same predecessor in $R_w(j)$. Now each element in $R_w(j)$ can determine its predecessor in $R_w(j-1)$ in constant time.
3. Each element of $R_z(j-1)$ finds its predecessor in $Q_z(j-1)$ in constant time. Consequently, all elements in $R_z(j-1)$ can determine their predecessors in $R_z(j)$, also in constant time. Note that no more than four elements of $R_z(j-1)$ have the same predecessor in $R_z(j)$. Now each element in $R_z(j)$ can determine its predecessor in $R_z(j-1)$ in constant time.
4. With the preceding "knowledge," the elements of $R_w(j)$ can determine their predecessors in $R_z(j)$, and vice versa, in constant time.

Thus, obtaining the information about predecessors required in the current step, merging, and obtaining the information about predecessors for the following step all require constant time. As mentioned before, there are $3 \log n$ steps in all, and therefore, the algorithm runs in $O(\log n)$ time. Since the sequences involved at each step contain a total of $O(n)$ elements, the number of processors needed is $O(n)$.

## 5.4 Divide and conquer

Let $Q = (q_1, q_2, \ldots, q_n)$ be a finite sequence representing $n$ points in the plane, where $n \geq 4$. Each point $q_i$ of $Q$ is given by a pair of Cartesian coordinates $(x_i, y_i)$. The *convex hull* of $Q$, denoted $CH(Q)$, is the convex polygon with the smallest area containing all the points of $Q$. Thus, each $q_i \in Q$ either lies inside $CH(Q)$ or is a corner of $CH(Q)$. A set of points and its convex hull are shown in Fig. 5.2(a) and (b), respectively.

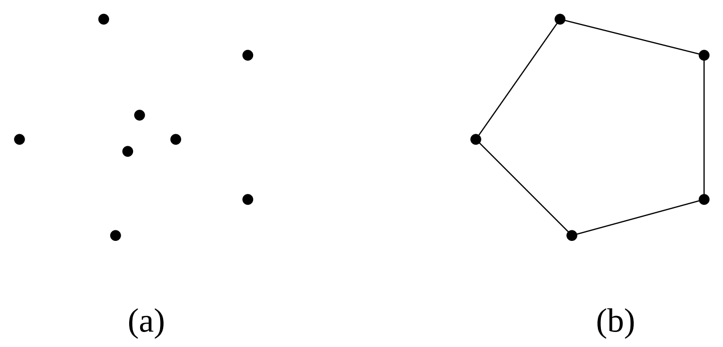


**Fig. 5.2.** A set of points and its convex hull.

Given $Q$, the problem we wish to solve is to compute $CH(Q)$. Since $CH(Q)$ is a polygon, an algorithm for this problem must produce the corners of $CH(Q)$ in the order in which they appear on the boundary of the polygon (in clockwise order, for example). Sequentially, the problem is solved in $O(n \log n)$ time, and this is optimal in light of an $\Omega(n \log n)$ lower bound on the number of operations required to compute $CH(Q)$, obtained by showing that any convex hull algorithm must be able to sort a sequence of $n$ numbers in nondecreasing order.

We now describe a parallel algorithm for computing $CH(Q)$ that uses the divide and conquer approach. The algorithm consists of four steps and is given next as algorithm PRAM CONVEX HULL:

```
Algorithm PRAM CONVEX HULL ($n, Q, CH(Q)$)
  Step 1: Sort the points of $Q$ by their $x$-coordinates.
  Step 2: Partition $Q$ into $n^{1/2}$ subsets $Q_1, Q_2, \ldots, Q_{n^{1/2}}$,
          of $n^{1/2}$ points each, separated by vertical lines,
          such that $Q_i$ is to the left of $Q_j$ if $i < j$.
  Step 3: for $i = 1$ to $n^{1/2}$ do in parallel
            if $|Q_i| \leq 3$
            then $CH(Q_i) \leftarrow Q_i$
            else PRAM CONVEX HULL ($n^{1/2}, Q_i, CH(Q_i)$)
            end if
          end for
  Step 4: $CH(Q) \leftarrow CH(Q_1) \cup CH(Q_2) \cup \cdots \cup CH(Q_{n^{1/2}})$. ■
```

Step 1 is performed using algorithm PRAM SORT. Step 2 is immediate: Since the points are sorted by their $x$-coordinates, it suffices to take the $i$th set of $n^{1/2}$ points in the sorted list and call it $Q_i$, $i = 1, 2, \ldots, n^{1/2}$. Step 3 applies the algorithm recursively to all the $Q_i$ simultaneously. Finally, Step 4 merges the convex hulls obtained in Step 3 to compute $CH(Q)$. We now show how this step is implemented.

**Merging a Set of Disjoint Polygons.** The situation at the end of Step 3 is illustrated in Fig. 5.3, where $u$ and $v$ are the points of $Q$ with the smallest and largest $x$-coordinates, respectively. The convex hull $CH(Q)$ consists of two parts, namely, the *upper hull* (i.e., the sequence of corners of $CH(Q)$ in clockwise order, beginning with $u$ and ending with $v$), and the *lower hull* (i.e., the sequence of corners of $CH(Q)$ in clockwise order, beginning with $v$ and ending with $u$). This is depicted in Fig. 5.4, in which the upper hull consists of the corners $u$, $a$, $b$, $c$, and $v$, while $v$, $d$, $e$, $f$, and $u$ are the corners of the lower hull. Step 4 computes the upper hull and lower hull separately and then concatenates them. Since computing the lower hull is symmetric to computing the upper hull, we only explain how the latter is performed.

**Identifying the Upper Hull.** Given two convex polygons, an *upper common tangent* to the two polygons is an infinite straight line that touches exactly one corner of each polygon; all remaining corners of the two polygons

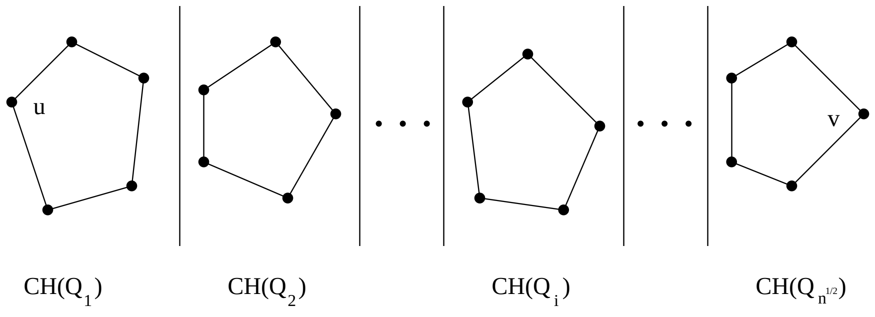


**Fig. 5.3.** Convex polygons to be merged in the computation of $CH(Q)$.

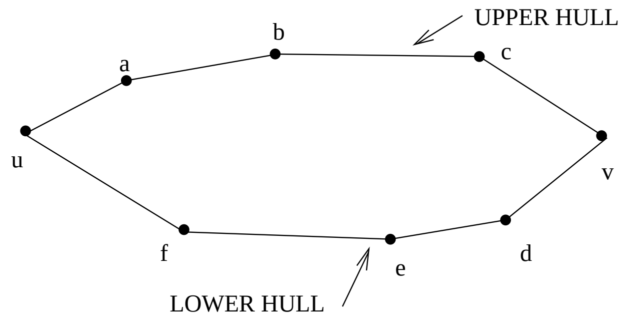


**Fig. 5.4.** The upper and lower hulls of $CH(Q)$.

fall below the tangent. Suppose that $n$ processors are available. We assign $n^{1/2} - 1$ processors to $CH(Q_i)$, for $i = 1, 2, \ldots, n^{1/2}$. Each processor assigned to $CH(Q_i)$ finds the upper tangent common to $CH(Q_i)$ and one of the remaining $n^{1/2} - 1$ convex polygons $CH(Q_j)$, $j \neq i$. Among all tangents to polygons to the left of $CH(Q_i)$, let $L_i$ be the one with the smallest slope and tangent to $CH(Q_i)$ at corner $l_i$. Similarly, among all tangents to polygons to the right of $CH(Q_i)$, let $R_i$ be the one with the largest slope and tangent to $CH(Q_i)$ at point $r_i$. If the angle formed by $L_i$ and $R_i$ is smaller than 180 degrees, then none of the corners of $CH(Q_i)$ is on the upper hull. Otherwise, all corners from $l_i$ to $r_i$ are on the upper hull.

These computations are done simultaneously for all $CH(Q_i)$, each yielding a (possibly empty) list of points of $Q$ on the upper hull. The lists are then compressed into one list using algorithm PRAM ARRAY PACKING. This way, all upper-hull points (from $u$ to $v$) occupy contiguous locations of an array. A similar computation yields all the lower-hull points (from $v$ to $u$) in contiguous positions of an array. By putting the two arrays side by side (and omitting $v$ and $u$ from the second array), we obtain $CH(Q)$.

One detail still requires some attention, namely, the way in which the tangents are found. This computation is a major component of the merge step, and it is important that we perform it efficiently, as explained next.

**Computing Tangents.** Given two convex polygons $CH(Q_i)$ and $CH(Q_j)$, each with $O(n^{1/2})$ corners, it is required to find their upper common tangent, that is, an infinite straight line tangent to $CH(Q_i)$ at some point $k$ and to $CH(Q_j)$ at some point $m$. Here, $k$ and $m$ are on the upper hulls of $CH(Q_i)$ and $CH(Q_j)$, respectively. We now show how the points $k$ and $m$ can be obtained by one processor in $O(\log n^{1/2})$—that is, $O(\log n)$—time. The approach is based on the same idea as binary search. Consider the sorted sequence of corners forming the upper hull of $CH(Q_i)$, and let $s$ be the corner in the middle of the sequence. Similarly, let $w$ be the corner in the middle of the sorted sequence of corners forming the upper hull of $CH(Q_j)$. For illustration purposes, suppose that $CH(Q_j)$ is to the right of $CH(Q_i)$. There are two possibilities:

1. Either both $s$ and $w$ lie on the upper common tangent of $CH(Q_i)$ and $CH(Q_j)$, that is, $k = s$ and $m = w$, as shown in Fig. 5.5(a), in which case we are done;
2. Or one half of the (remaining) corners of $CH(Q_i)$ and/or $CH(Q_j)$ can be removed from further consideration as upper tangent points. An example of such a situation is illustrated in Fig. 5.5(b), in which those parts of a polygon removed from consideration are highlighted. The process is now repeated by finding the corners $s$ and $w$ in the middle of the remaining sequence of corners in $CH(Q_i)$ and $CH(Q_j)$, respectively.

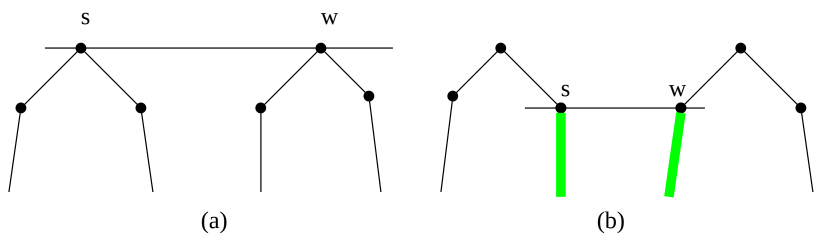


**Fig. 5.5.** Computing the upper tangent.

**Analysis.** As mentioned earlier, Step 1 of algorithm PRAM CONVEX HULL is performed using algorithm PRAM SORT. This requires $O(n)$ processors and $O(\log n)$ time. If $\{q'_1, q'_2, \ldots, q'_n\}$ represents the sorted sequence, then points $q'_j$, $j = (i-1)n^{1/2}+1, (i-1)n^{1/2}+2, \ldots, in^{1/2}$, belong to $Q_i$, for $i = 1, 2, \ldots, n^{1/2}$. Therefore, with $n$ processors, Step 2 requires constant time: Processor $P_j$ reads $q'_j$, then uses $j$ and $n^{1/2}$ to compute $i$, and finally assigns $q'_j$ to $Q_i$, $j = 1, 2, \ldots, n$. If $t(n)$ is the running time of algorithm PRAM CONVEX HULL, then Step 3 requires $t(n^{1/2})$ time, with $n^{1/2}$ processors computing $CH(Q_i)$. In Step 4, each of $(n^{1/2}-1)n^{1/2}$ processors computes one tangent in $O(\log n)$ time. For each $CH(Q_i)$, the tangent $L_i$ is found in constant time: The slopes of the tangents to polygons to the left of

$CH(Q_i)$ are written simultaneously into some memory location, each slope written by a different processor, such that the *minimum* of all these slopes ends up in the memory location. The tangent $R_i$ is found similarly. It also takes constant time to determine whether the corners from $l_i$ to $r_i$ belong to the upper hull.

Finally, array packing is done in $O(\log n)$ time using $n$ processors. Putting all the pieces together, the overall running time of the algorithm is given by $t(n) = t(n^{1/2}) + \beta \log n$, for some constant $\beta$. Thus, $t(n) = O(\log n)$. Since $p(n) = n$, the algorithm's cost is $c(n) = O(n \log n)$ and that is optimal.

## 6. POINTER-BASED DATA STRUCTURES

The PRAM algorithms described in Section 5. manipulate data stored in *arrays* in shared memory. Specifically, the inputs to the prefix sums, array packing, sorting, and convex hull problems are known to occupy contiguous locations in memory. Thus, element $x_i$ of an array $X$ can be accessed directly using its index $i$ in the array. In this section, we continue to use the PRAM model while focusing on problems whose inputs are stored in data structures that use pointers to relate their elements to one another. These data structures include linked lists, trees, and general graphs.

We begin in Section 6.1 by describing *pointer jumping*, a basic tool at the heart of all efficient parallel algorithms for pointer-based data structures. The problem of computing prefix sums on a linked list is then addressed in Section 6.2. The importance of this problem stems from the fact that a fast parallel algorithm for its solution allows a host of other computations to be performed efficiently. In fact, the scope of this algorithm extends beyond computations defined strictly on linked lists. Many computational problems occurring in the context of other data structures can be solved by transforming those structures into linked lists and then applying the prefix sums algorithm. Such a transformation is performed efficiently through the use of a powerful approach known as the *Euler tour* method, presented in Section 6.3. Finally, we show in Section 6.4 how a number of problems defined on trees are solved in parallel using the Euler tour and pointer jumping techniques.

### 6.1 Pointer jumping

Consider a data structure in the form of a rooted tree where each node (except the root $R$) has a pointer to its parent. A certain node $E$, at a distance $n$ from the root, holds a datum $d$. It is desired to copy $d$ into every node on the path from $E$ to $R$. Sequentially, there are $n$ pointers to traverse and hence the computation requires $\Omega(n)$ time. In a parallel setting, we assume that each node of the tree is directly accessible to one (and only one) processor. The computation is performed in a number of iterations, with all processors operating in parallel. Let some node $A$ point to some node $B$ and node $B$ point to some node $C$. The processor in charge of node $A$ executes the following two steps:

1. It copies the datum $d$, if held by $A$, into node $B$, then
2. Modifies the pointer of node $A$ to point to node $C$.

The same two steps are now repeated using the new pointers. After $O(\log n)$ iterations, all nodes on the path from $E$ to $R$ hold the datum $d$.

## 6.2 Prefix sums computation

Consider the singly linked list $L$ shown in Fig. 6.1 and composed of *nodes* linked by *pointers*. Each node $i$ consists of a number of fields, two of which

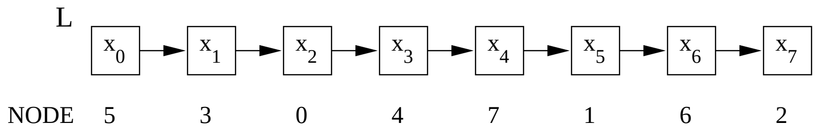


**Fig. 6.1.** A linked list.

in particular are illustrated in Fig. 6.1:

1. A value field $val(i)$ holding a number $x_j$.
2. A pointer field $succ(i)$ pointing to the successor of node $i$ in $L$.

Since the last node in the list, the *tail*, has no successor, its pointer is equal to *nil*. It is important to note here that the index $i$ of a node is essentially meaningless as the nodes occupy arbitrary locations in memory. The same is true of the index $j$ of $x_j$ since the latter is just a number. We use indices only for ease of exposition.

It is required to compute the prefix sums $x_0$, $x_0 + x_1$, $x_0 + x_1 + x_2$, $\ldots$, as shown in Fig. 6.2, where the symbol $x_{ij}$ is used to denote $x_i + x_{i+1} + \cdots + x_j$. On a PRAM, we assume that processor $P_i$ is assigned node $i$ and is in charge of that node throughout the computation: $P_i$ "knows" the address of node $i$ and can gain access to its contents in $O(1)$ time. However, $P_i$ "knows" neither its position relative to the other processors nor the number of nodes in $L$. A processor assignment for the list of Fig. 6.1 is shown in Fig. 6.3(a).

The prefix sums of $L$ are computed using pointer jumping by the following algorithm:

```
Algorithm PRAM LINKED LIST PREFIX
  Step 1: for all $i$ do in parallel
              $next(i) \leftarrow succ(i)$
          end for
```

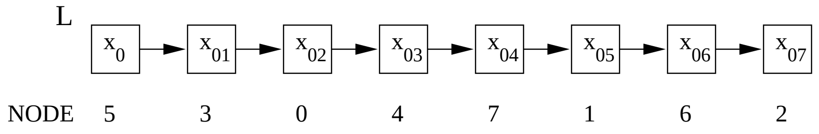


**Fig. 6.2.** Linked list after prefix computation on *val* field.

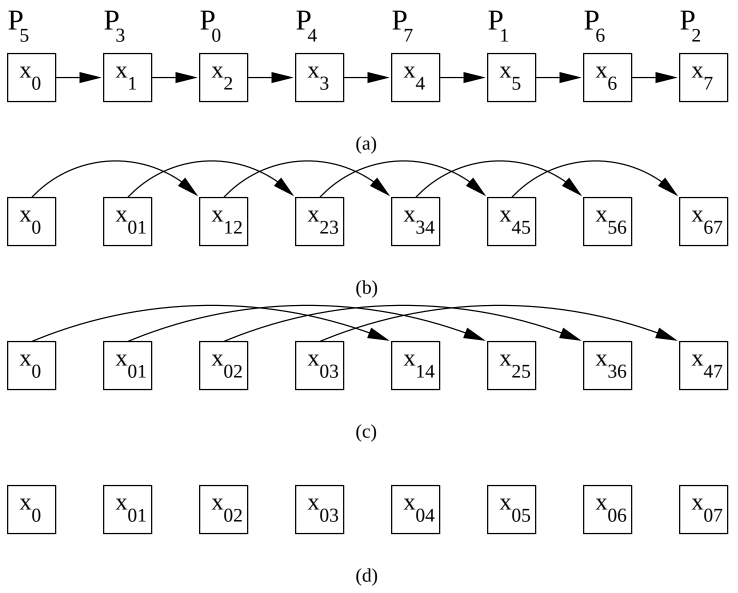

$P_5$ $P_3$ $P_0$ $P_4$ $P_7$ $P_1$ $P_6$ $P_2$
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$
(a)
$x_0$ $x_{01}$ $x_{12}$ $x_{23}$ $x_{34}$ $x_{45}$ $x_{56}$ $x_{67}$
(b)
$x_0$ $x_{01}$ $x_{02}$ $x_{03}$ $x_{14}$ $x_{25}$ $x_{36}$ $x_{47}$
(c)
$x_0$ $x_{01}$ $x_{02}$ $x_{03}$ $x_{04}$ $x_{05}$ $x_{06}$ $x_{07}$
(d)

**Fig. 6.3.** Prefix computation by pointer jumping.

```
Step 2: finished ← false
Step 3: while not finished do
            (3.1) finished ← true
            (3.2) for all i do in parallel
                      (i) if next(i) ≠ nil
                          then (a) val(next(i)) ← val(i) + val(next(i))
                               (b) next(i) ← next(next(i))
                          end if
                      (ii) if next(i) ≠ nil
                          then finished ← false
                          end if
                  end for
        end while. ■
```

The algorithm is illustrated in Fig. 6.3(b)–(d). Note that in order to avoid destroying the pointers of $L$ during the pointer jumping, the contents of *succ*$(i)$ are copied during Step 1 into another field *next*$(i)$ of node $i$. Also, in

Step 3(ii) since several processors may be writing simultaneously into *finished*, one of them, chosen arbitrarily, succeeds.

**Analysis.** Steps 1 and 2 and each iteration of Step 3 require constant time. Since the number of final answers doubles after each iteration of Step 3, the number of iterations is the logarithm of the number of nodes in $L$. Assuming that this number is $n$, algorithm PRAM LINKED LIST PREFIX runs in $t(n) = O(\log n)$ time and uses $p(n) = n$ processors.

## 6.3 The Euler tour

A cycle in a directed and connected graph is said to be an *Euler tour* if every edge of the graph appears in the cycle exactly once. It is easy to see that an Euler tour is guaranteed to exist in a directed and connected graph if the number of edges "entering" each vertex is equal to the number of edges "leaving" that vertex. Let $T$ be an undirected (and unrooted) tree on $n$ vertices, as shown in Fig. 6.4(a) for $n = 7$. Clearly, $T$ has $n - 1$ edges. Suppose that we replace each edge $(v_i, v_j)$ in $T$ with two *oriented* edges $(v_i, v_j)$ and $(v_j, v_i)$, as shown in Fig. 6.4(b). Then the resulting directed graph is guaranteed to have an Euler tour. In what follows, we denote a directed tree such as the one in Fig. 6.4(b) by $DT$. An Euler tour defined on $DT$ is denoted $ET$. Note that a $DT$ with $n$ vertices has $2n - 2$ edges, and hence $ET$ is a sequence of $2n - 2$ edges.

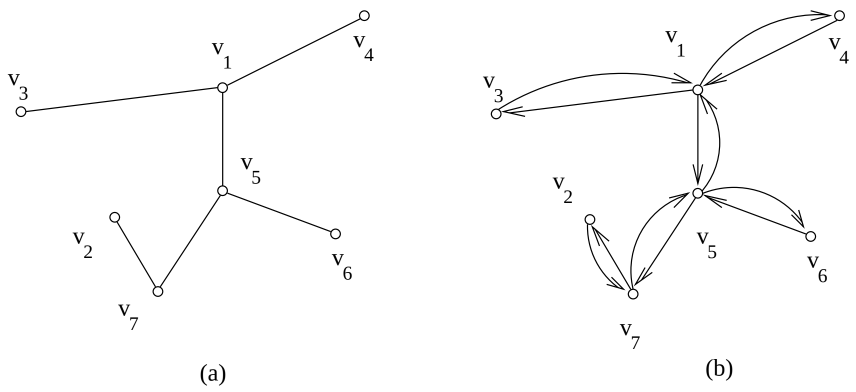


**Fig. 6.4.** A tree $T$ and its directed version $DT$.

We now describe an algorithm for computing an Euler tour $ET$ of a directed tree $DT$. The algorithm receives $DT$ as input, stored in the form of $n$ linked lists, each of which containing the edges "leaving" a particular vertex. Node $< ij >$ in the linked list for vertex $v_i$ consists of two fields, namely, a field *edge* holding the edge $(v_i, v_j)$ and a field *next* holding a pointer to the next node in the list. A pointer $head(v_i)$ gives access to the first node in the linked list for vertex $v_i$. For example, the linked list for vertex $v_5$ of Fig. 6.4(b) is as follows:

$$head(v_5) \longrightarrow (v_5, v_1) \longrightarrow (v_5, v_6) \longrightarrow (v_5, v_7).$$

An algorithm for computing $ET$ arranges all the edges of $DT$ into a *single* linked list in which each edge $(v_i, v_j)$ is followed by an edge $(v_j, v_k)$. Also, if the first edge in $ET$ "leaves" some vertex $v_l$, then the last edge in $ET$ "enters" $v_l$. Such a linked list for the $DT$ of Fig. 6.4(b) is shown in Fig. 6.5 (in which the pointer from $(v_5, v_1)$ to $(v_1, v_3)$ is omitted). Note that each node in the linked list $ET$ consists of two fields, namely, a field *edge* containing an edge and a field *succ* containing a pointer to the successor of the node.

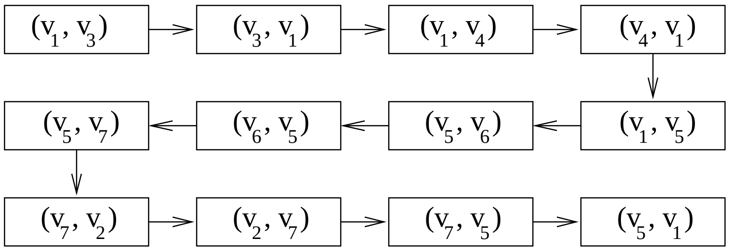


**Fig. 6.5.** The Euler tour as a linked list.

On a PRAM we assume that $2n - 2$ processors are available. Processor $P_{ij}$ is in charge of edge $(v_i, v_j)$ and determines the position in $ET$ of node $< ij >$ holding that edge. It does so in constant time by determining the successor of $< ij >$ in the linked list as follows:

```
Successor of $(v_i, v_j)$
  if $next(< ji >) = jk$
  then $succ(< ij >) \leftarrow jk$
  else $succ(< ij >) \leftarrow head(v_j)$
  end if. ■
```

Note that $P_{ij}$ needs $next(< ji >)$ to compute $succ(< ij >)$, and obtains it from $P_{ji}$ through shared memory. It follows that $ET$ is computed in constant time.

Suppose now that $T$ is a *rooted* tree. A *depth-first* (or *preorder*) traversal of $T$ visits the vertices of the tree in the following order: First, we go to the root, then each of the subtrees of the root is traversed (recursively) in depth-first order. We can make $ET$ correspond to a depth-first traversal of $T$ by choosing an edge "leaving" the root as the first edge of $ET$. Thus, for example, if $v_1$ is the root of $T$ in Fig. 6.4(a), then $v_1 v_3 v_4 v_5 v_6 v_7 v_2$ is a depth first traversal of $T$, and it corresponds to an Euler tour of $DT$ whose first edge is $(v_1, v_3)$.

Now let $v_r$ be the root of a tree $T$, $(v_r, v_j)$ be the first edge in an Euler tour $ET$ of $DT$, and $(v_k, v_r)$ be the last edge in $ET$. In Fig. 6.5, the last edge in $ET$ is $(v_5, v_1)$. By setting $succ(<kr>) = nil$ and letting each node $<ij>$ of $ET$ have an additional field *val*, we turn $ET$ into the linked list $L$ of Fig. 6.1. This allows us to apply algorithm PRAM LINKED LIST PREFIX to $ET$.

In particular, let a prefix sums computation be applied to $ET$ with all *val* fields initialized to 1. This gives the position in $ET$ of each edge $(v_i, v_j)$, denoted $pos(v_i, v_j)$ and stored in the *val* field of node $<ij>$. Thus, if $(v_r, v_j)$ and $(v_k, v_r)$ are the first and last edges in $ET$, respectively, then $pos(v_r, v_j) = 1$ and $pos(v_k, v_r) = 2n - 2$. In Fig. 6.5, $pos(v_1, v_3) = 1$ and $pos(v_5, v_1) = 12$. If $pos(v_i, v_j) < pos(v_j, v_i)$, then we refer to edge $(v_i, v_j)$ of $DT$ as an *advance* edge; otherwise, edge $(v_i, v_j)$ is a *retreat* edge. Clearly, for a tree $T$ of $n$ vertices, labeling the edges of $DT$ as advance or retreat edges, by computing their positions in $ET$, requires $O(\log n)$ time and $O(n)$ processors.

## 6.4 Applications

The Euler tour, combined with prefix computation on linked lists, provides a powerful tool for solving problems on graphs. We provide three examples of parallel algorithms for problems defined on trees.

**Determining Levels of Vertices.** The *level* of a vertex in an undirected rooted tree $T$ is the number of (unoriented) edges on a shortest path from the root to the vertex. In an $ET$ of $DT$, the level of a vertex $v_j$ equals the number of retreat edges minus the number of advance edges following the first occurrence of $v_j$ in $ET$. We compute this level as follows: First, we assign values to the *val* fields of $ET$. If $(v_k, v_l)$ is a retreat edge, then $val(<lk>) = 1$; otherwise, $val(<lk>) = -1$. A *suffix sums* computation is now performed over the *val* fields using algorithm PRAM LINKED LIST PREFIX, with two changes: Each occurrence of $i$ is replaced with $<ij>$, and Step (3.2)(i)(a) modified to be

$$val(<ij>) \leftarrow val(<ij>) + val(next(<ij>)).$$

Finally, if $(v_i, v_j)$ is an advance edge, then

$$level(v_j) \leftarrow val(<ij>) + 1.$$

Note that if $v_r$ is the root, then $level(v_r) \leftarrow 0$. If $T$ has $n$ vertices, then the computation requires $O(n)$ processors and runs in $O(\log n)$ time.

**Numbering the Vertices.** The *inorder* traversal of a rooted binary tree $T$ visits the vertices in the following order: The left subtree of the root is traversed recursively in inorder, then the root is visited, and finally, the right subtree of the root is traversed recursively in inorder. Given a binary tree $T$

with $n$ vertices, the Euler tour technique can be used to number the vertices of $T$ in the order of the inorder traversal of $T$.

Let $v_r$ be the root of $T$ and $ET$ be an Euler tour defined on $T$ such that $(v_r, v_s)$ is the first (advance) edge in $ET$, for some child $v_s$ of the root. Values are assigned to the *val* fields of the nodes in $ET$ as follows:

1. Let $(v_i, v_j)$ be an advance edge; if $(v_i, v_j)$ is immediately followed by a retreat edge, then $val(< ij >) = 1$; otherwise $val(< ij >) = 0$.
2. Let $(v_k, v_l)$ be a retreat edge; if $(v_k, v_l)$ is immediately followed by an advance edge, then $val(< kl >) = 1$; otherwise $val(< kl >) = 0$.

A prefix sums computation is now performed over the *val* fields. Finally,

1. If $(v_i, v_j)$ is an advance edge immediately followed by a retreat edge, then $inorder(v_j) \leftarrow val(< ij >)$.
2. If $(v_k, v_l)$ is a retreat edge immediately followed by an advance edge, then $inorder(v_l) \leftarrow val(< kl >)$.

The algorithm runs in $O(\log n)$ time using $O(n)$ processors.

**Computing Lowest Common Ancestors.** Let $T$ be a rooted complete binary tree and $v_1$ and $v_2$ two of its vertices. The *lowest common ancestor* of $v_1$ and $v_2$ is a vertex $u$ that is an ancestor of both $v_1$ and $v_2$ and is farthest away from the root. Suppose that $T$ has $n$ leaves. Given $n$ pairs of vertices $(v_i, v_j)$, it is required to compute the lowest common ancestor of each pair. We now show that $n$ processors can find the required $n$ lowest common ancestors in constant time, provided that the vertices of $T$ have been numbered in the order of an inorder traversal.

Let $v_i$ and $v_j$ be two vertices of $T$, whose inorder numbers are $i$ and $j$, respectively. Further, let $i_1 i_2 \ldots i_k$ and $j_1 j_2 \ldots j_k$ be the binary representations of $i$ and $j$, respectively. Now, let $l$ be the leftmost position where $i_1 i_2 \ldots i_k$ and $j_1 j_2 \ldots j_k$ differ; in other words, $i_1 i_2 \ldots i_{l-1} = j_1 j_2 \ldots j_{l-1}$ and $i_l \neq j_l$.

The lowest common ancestor of $v_i$ and $v_j$ is that vertex $v_m$ whose inorder number $m$ has the following binary representation: $i_1 i_2 \ldots i_{l-1} 100 \ldots 0$. A single processor can obtain $m$ in constant time.

## 7. CONCLUSION

A number of approaches to the design of efficient parallel algorithms were illustrated in the previous five sections. Our purpose in this final section is to probe some of these concepts a little further. We begin by extending the mesh interconnection network with the addition of *buses* and demonstrate the new model's flexibility and efficiency. Then it is the PRAM 's turn to be augmented with a special instruction for broadcasting. This results in *broadcasting with selective reduction*, an elegant and effective environment for designing parallel algorithms. Finally, we reflect on the power of parallelism by developing the notion of *parallel synergy*.

## 7.1 Meshes that use buses

The mesh of processors introduced in Section 3.2 is one of the most attractive models of computation in theory and in practice mainly because of its simplicity. In it, the maximum degree of a processor is four. The network is *regular*, as all rows (and columns) are connected to their successors in exactly the same way. It is also *modular*, in the sense that any of its regions can be implemented with the same basic component. These properties allow the mesh to be easily extended by the simple addition of a row or a column. However, the mesh is inadequate for those computational problems in which its diameter is considered too large. This limits its usefulness in many applications. In an attempt to retain most of the advantages of the mesh, an extension to the basic model is sought to reduce its diameter. In the most promising approach, the mesh is augmented with a certain number of *buses*. Here, a bus is simply a communication link to which some or all of the processors of the mesh are attached. Here, two processors that are not neighbors on the mesh can communicate directly through the bus to which they are both connected.

Writing a (fixed-size) datum on a bus and reading a (fixed-size) datum from the bus are considered basic operations requiring constant time. But how long does it take to traverse a bus? Let $B(L)$ represent a bus of length $L$, and let $\tau_{B(L)}$ be the time taken by a datum of fixed size to go from one end of the bus to the other. Clearly, $\tau_{B(L)}$ is a function of $L$. There are many choices for this function, depending on the technology used to implement the bus. In theoretical analyses of algorithms for meshes enhanced with buses, it is best, therefore, to leave $\tau_{B(L)}$ as a parameter when expressing the running time of an algorithm. Thus, if $S$ steps of an algorithm use the bus, the time consumed by these steps would be $S \times \tau_{B(L)}$. However, for simplicity, we take $\tau_{B(L)} = O(1)$ in what follows. Specifically, we take $\tau_{B(L)}$ to be smaller than or equal to the time required by a basic computational operation, such as adding or comparing two numbers.

Buses can be implemented in a variety of ways. Thus, for example, they may be *fixed* (i.e., their number, shape, and length are static throughout the computation), or they may be *reconfigurable* (i.e., the buses are created dynamically while a problem is being solved). There may be one (or several) *global* bus(es) spanning all rows and columns of the mesh, or there may be one (or several) bus(es) for each row and one (or several) bus(es) for each column. Also, a bus may be *electronic* or *optical*. On an electronic bus, *one* processor can *broadcast* a datum by placing it on the bus; the datum is now available for reading by *all* the remaining processors on the bus. An *optical bus*, by contrast, allows several processors to inject data on the bus simultaneously, each datum destined for one or several processors. This capability leads to an entirely new range of techniques for parallel algorithm design. In this section, we sketch the design and analysis of algorithms for electronic reconfigurable buses and fixed optical buses.

**Reconfigurable Buses.** In a mesh-of-processors interconnection network a typical processor has four links connecting it to its neighbors. Each of these links is attached to the processor itself via an interface, commonly referred to as a *port*. A mesh processor therefore has four ports, called its north (N), south (S), west (W), and east (E) ports. Suppose that a processor is capable of connecting its ports internally in pairs in any one of the 10 configurations depicted in Fig. 7.1. These internal connections, combined with

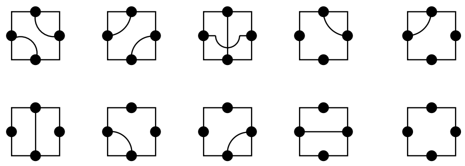


**Fig. 7.1.** Possible internal connections of a processor's ports.

the standard mesh links, allow for paths of arbitrary lengths and shapes to be created in the mesh, as shown in Fig. 7.2 for a $4 \times 4$ mesh in which three paths have been established. These paths are *dynamic*: They are created in constant time by the processors (and then modified if necessary) as specified by the algorithm. Each path created by a set of processors among themselves is viewed as a *bus* to which these processors are connected. It therefore possesses all the properties of (electronic) buses. Specifically, at any given time only *one* processor can place (i.e., write) on the bus a datum that takes constant time to travel from one end of the bus to the other and can be obtained (i.e., read) by *all* processors connected to the bus simultaneously. These specifications yield a new model of computation, called the *mesh with reconfigurable buses*.

We now show how this model can be used to solve a problem on linked lists, known as the *list ranking* problem. Given a linked list $L$ of $n$ nodes, it is required to determine the distance of each node from the end of the list. For example, in Fig. 6.1, node 7 is at distance 3 from the end of the list. Specifically, for each node $i$ for which $succ(i) \neq nil$, we wish to compute $rank(i) = rank(succ(i)) + 1$. Of course, if $succ(i) = nil$, then $rank(i) = 0$.

The list $L$ is mapped onto an $n \times n$ mesh with reconfigurable buses. In doing so we seek to shape a bus that reflects the topology of the list. Thus, node $i$ is assigned to $P(i,i)$. A subbus is then implemented connecting $P(i,i)$ to $P(succ(i), succ(i))$ as follows: The bus goes vertically from $P(i,i)$ to $P(succ(i), i)$ and then horizontally to $P(succ(i), succ(i))$. Also, if $i = succ(j)$, for some node $j$, then node $j$ simultaneously creates a bus from $P(j,j)$ to $P(i,i)$. Finally, $P(i,i)$ establishes a connection internally to join these two buses together. The algorithm for computing the rank of node $i$ in $L$ is as follows:

**Step 1:** $P(i,i)$ removes its internal connection thus splitting the bus in two.

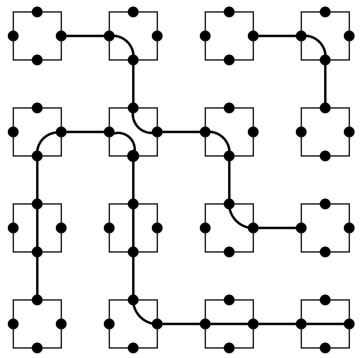


**Fig. 7.2.** A mesh with three configured buses.

**Step 2:** $P(i,i)$ sends a signal '$*$' on the bus connecting it to $P(succ(i), succ(i))$.

**Step 3:** If a processor $P(k,k)$, $1 \leq k \leq n$, receives '$*$' then it produces a 1 as output; otherwise, it produces a 0.

**Step 4:** The rank of node $i$ in $L$ is the sum of the 1s and 0s produced in Step 3 and is computed as follows:

- **(4.1)** All processors connect their W and E ports, thus creating a bus on each row. Processor $P(k,k)$, $1 \leq k \leq n$, broadcasts its 0 or 1 on the bus in row $k$. All processors in row $k$ store the received value (i.e., 0 or 1).
- **(4.2)** If a processor contains a 0, it connects its N and S ports; otherwise, it connects its W and N ports and its S and E ports.
- **(4.3)** Processor $P(n,1)$ places a '$*$' on the bus to which its S port is connected. If processor $P(1,j)$ receives that symbol then the the rank of node $i$ is $j-1$. ■

*Example 7.1.* Let $n = 5$, and assume that for node $i$, the output of $P(k,k)$ in Step 3 is 0, 1, 1, 0, 1, for $k = 1,2,3,4$, and 5, respectively. Step 4 is illustrated in Fig. 7.3, showing that the rank of node $i$ is 3. □

By using $n$ such $n \times n$ meshes, and running the preceding algorithm simultaneously for all nodes of $L$, the ranks of all nodes can be computed in $O(1)$ time. This requires a mesh of size $n^2 \times n$. This algorithm can be modified to use a mesh of size $O(mn \times n)$ and run in $O(\log n / \log m)$ time.

**Optical Buses.** An alternative to electronic buses is provided by *optical buses*. Here, *light signals* are used instead of electrical signals. This allows the bus to have two properties, namely, *unidirectionality*, which means that a datum placed by any processor on an optical bus travels in only one (always the same) direction, and *predictability of the propagation delay per unit length*, which means that the time it takes a light signal to travel a certain distance along the bus is directly proportional to that distance. These

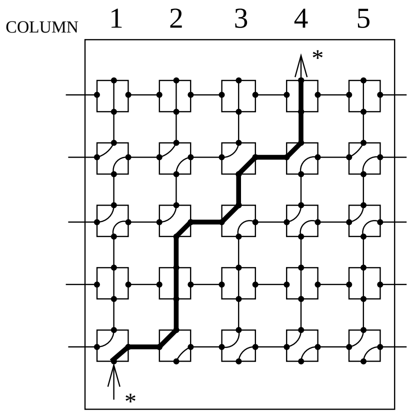


**Fig. 7.3.** Computing the rank of node $i$.

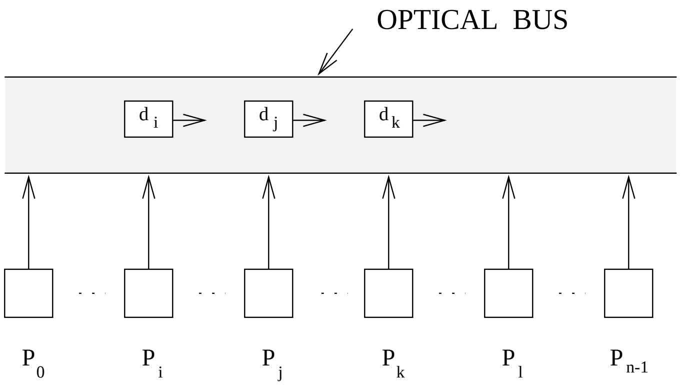


**Fig. 7.4.** A pipeline of data on an optical bus.

two simple, yet important, properties lead to the definition of entirely new computational models and open up rich avenues for algorithmic design. To see this, let $n$ processors $P_0$, $P_1$, $\ldots$, $P_{n-1}$ be connected to an optical bus. It is possible for *several* processors to place data on the bus simultaneously, one datum per processor. The data form a *pipeline* and travel down the bus, all in the same direction, as shown in Fig. 7.4, where $d_i, d_j$, and $d_k$ are the data placed at the same time on the bus by processors $P_i$, $P_j$, and $P_k$, respectively. The difference between the arrival times of two data $d_i$ and $d_j$ at some processor $P_l$ can be determined by the distance separating $P_i$ and $P_j$.

The time taken by a light signal to traverse the optical bus from one end (at $P_0$) to the other (at $P_{n-1}$)—that is, $\tau_{B(L)}$—is known as the *bus cycle*. Time is divided into bus-cycle intervals. As stated at the beginning of Section 7.1, we take $\tau_{B(L)} = O(1)$. All processors *write* their data on the bus synchronously at the beginning of each bus cycle. (Processors without any real message to send place a dummy message on the bus.) When is a processor $P_j$ to *read* a message $d_i$ from the bus? Since $P_j$ "knows" the identity of the sender $P_i$, it skips $j - i - 1$ messages and reads the $(j - i)$th message that passes by it. In what follows we express $j - i$ using the function $wait(i, j)$.

The system in Fig. 7.4 permits data to travel only in one direction. In order to allow messages to be sent in both directions, *two* optical buses are used. Data travel from left to right on one bus and from right to left on the other. Each processor can read and write to either of the two buses, as required by the algorithm. The two buses are completely independent from one another and accommodate separate pipelines. The *wait* function is now interpreted as follows: For $i \neq j$, if $wait(i, j) > 0$ then $P_j$ reads from the left-to-right bus, otherwise it reads from the right-to-left bus.

It is now clear that any communication pattern among the processors (such as, for example, broadcasting a datum from one processor to all others, or executing an arbitrary permutation of the data) can be defined simply by specifying the *wait* function. To illustrate, consider the following general form of data distribution: For each $i$, $0 \leq i \leq n-1$, it is required to send datum $d_i$, held by $P_i$, to one or more processors or to no processor at all. Let $s(j)$ be the index of the processor from which $P_j$ receives a datum, where $0 \leq j, s(j) \leq n-1$. This allows for the possibility that $s(j) = s(k) = i$, for $j \neq k$, meaning that the function $s$ is not necessarily a permutation. In order to perform this operation, we define the waiting function as $wait(s(j), j) = j - s(j)$, for all $j$. Thus, the entire data distribution operation is completed in one bus cycle and hence requires constant time.

One interesting consequence of the foregoing is that the system of Fig. 7.4, consisting of $n$ processors and $O(1)$ memory locations per processor, can simulate in constant time all (but one) of the forms of memory access allowed on a PRAM with $n$ processors and $O(n)$ shared memory locations. This is true because memory access can be viewed as a data distribution operation. The only exception is that form of PRAM memory access (known as *concurrent writing*) where several processors write to the same memory location simultaneously (see Example 2.1). This case cannot be simulated in constant time. To see this, recall that $\tau_{B(L)}$ is assumed to be smaller than or equal to the time required by a basic operation, such as adding or comparing two numbers, whereas concurrent writing typically involves an arbitrary number of such operations. It follows that any PRAM algorithm (which does not require concurrent writing) can be performed in the same amount of time on the system of Fig. 7.4 as on the PRAM (the processors used in the two models being identical in kind and in number).

In an attempt to reduce the bus length $L$ when the number of processors $n$ is large, the system of Fig. 7.4 is modified so that the $n$ processors are placed in a two-dimensional pattern with $n^{1/2}$ rows and $n^{1/2}$ columns. In this arrangement, each processor $P(i, j)$ is connected to four optical buses:

Two buses on its row to send data horizontally and two buses on its column to send data vertically. A message can be sent from $P(i,j)$ to $P(k,l)$ in two bus cycles: First the message is sent from $P(i,j)$ to $P(i,l)$ and then from $P(i,l)$ to $P(k,l)$. We conclude our discussion of optical buses by showing how $n$ numbers can be sorted on this model.

Recall that algorithm MESH SORT uses two basic operations to sort a sequence of numbers stored in a two-dimensional array, namely, sorting a row (or column) and cyclically shifting a row. Now suppose that the $n$ numbers to be sorted are stored one number per processor in a two-dimensional array with row and column optical buses. The steps of algorithm MESH SORT are simulated as follows:

1. Whenever a row (or column) is to be sorted, the processors in that row (or column) and their two horizontal (vertical) optical buses simulate algorithm PRAM SORT.
2. Whenever a row is to be cyclically shifted, this is done using function *wait*, since a cyclic shift is a special case of data distribution.

This requires $O(\log n)$ time and $n$ processors, for an optimal cost of $O(n \log n)$.

## 7.2 Broadcasting with selective reduction

In this section we describe a model of computation called *broadcasting with selective reduction* (BSR). This model is more powerful than the PRAM yet requires no more resources than the PRAM for its implementation. It consists of $N$ processors, $M$ shared-memory locations, and a memory access unit (MAU). Thus, BSR can be fully described by the diagram in Fig. 2.1 depicting the PRAM. In fact, all the components of the PRAM shown in that figure are the same for BSR (including the MAU, as we demonstrate later in this section). The only difference between the two models is in the way the MAU is exploited on BSR. Specifically, BSR's repertoire of instructions is that of the PRAM, augmented with one instruction called BROADCAST. This instruction allows *all* processors to write to *all* shared-memory locations simultaneously. It consists of three phases:

1. A *broadcasting* phase, in which each processor $P_i$ broadcasts a *datum* $d_i$ and a *tag* $g_i$, $1 \leq i \leq N$, destined to all $M$ memory locations.
2. A *selection* phase, in which each memory location $U_j$ uses a *limit* $l_j$, $1 \leq j \leq M$, and a *selection rule* $\sigma$ to test the condition $g_i \ \sigma \ l_j$. Here, $g_i$ and $l_j$ are variables of the same type (e.g., integers), and $\sigma$ is a *relational operator* selected from the set $\{<, \leq, =, \geq, >, \neq\}$. If $g_i \ \sigma \ l_j$ is **true**, then

$d_i$ is selected for reduction in the next phase; otherwise $d_i$ is rejected by $U_j$.

3. A *reduction* phase, in which all data $d_i$ selected by $U_j$ during the selection phase are combined into one datum that is finally stored in $U_j$. This phase uses an appropriate binary associative *reduction operator* $\mathcal{R}$ selected from the set $\{\sum, \prod, \bigwedge, \bigvee, \bigoplus, \bigcap, \bigcup\}$, whose elements denote the operations *sum, product, and, or, exclusive-or, maximum*, and *minimum*, respectively.

All three phases are performed simultaneously for all processors $P_i$, $1 \leq i \leq N$, and all memory locations $U_j$, $1 \leq j \leq M$.

The instruction BROADCAST of BSR is written as follows:

$$\underset{1 \leq j \leq M}{U_j} \leftarrow \underset{\substack{g_i \ \sigma \ l_j \\ 1 \leq i \leq N}}{\mathcal{R}} d_i.$$

This notation is interpreted as saying that, for each memory location $U_j$ associated with the limit $l_j$, the proposition $g_i \ \sigma \ l_j$ is evaluated over all broadcast *tag* and *datum* pairs $(g_i, d_i)$. In every case where the proposition is **true**, $d_i$ is *accepted* by $U_j$. The set of all *data* accepted by $U_j$ is reduced to a single value by means of the binary associative operation $\mathcal{R}$ and is stored in that memory location. If no *data* are accepted by a given memory location, the value (of the shared variable) held by that location is not affected by the BROADCAST instruction. If only one *datum* is accepted, $U_j$ is assigned the value of that datum.

As we show at the end of this section, the BROADCAST instruction is executed in one access to memory, that is, in one traversal of the MAU. Since the BSR's MAU is identical to that of the PRAM, one access to memory on BSR takes constant time (as it does on the PRAM). Therefore, BROADCAST requires $O(1)$ time.

**Computing A Maximum-Sum Subsequence.** BSR is now used to solve a nontrivial computational problem. The power and elegance of the model are demonstrated by the efficiency and conciseness of the algorithm it affords. Given a sequence of numbers $X = (x_1, x_2, \ldots, x_n)$, it is required to find two indices $u$ and $v$, where $u \leq v$, such that the subsequence $(x_u, x_{u+1}, \ldots, x_v)$ has the largest possible sum $x_u + x_{u+1} + \cdots + x_v$, among all such subsequences of $X$. In case of a tie, the leftmost subsequence with the largest sum is to be returned as the answer. Sequentially, the problem can be solved optimally in $O(n)$ time (by an algorithm which is by no means obvious).

The BSR algorithm uses $n$ processors $P_1, P_2, \ldots, P_n$, and consists of four steps:

1. The prefix sums $s_1, s_2, \ldots, s_n$ of $x_1, x_2, \ldots, x_n$ are computed.

2. For each $j$, $1 \leq j \leq n$, the maximum prefix sum to the right of $s_j$, beginning with $s_j$, is found. The value and index of this prefix sum are stored in $m_j$ and $a_j$, respectively. To compute $m_j$, a BROADCAST instruction is used, where the *tag* and *datum* pair broadcast by $P_i$ is $(i, s_i)$, while $U_j$ uses $j$ as *limit*, '$\geq$' for selection, and '$\bigcap$' for reduction. Similarly, to compute $a_j$, a BROADCAST is used, where $P_i$ broadcasts $(s_i, i)$ as its *tag* and *datum* pair, while $U_j$ uses $m_j$, '$=$', and '$\bigcap$' as *limit*, selection rule, and reduction operator, respectively.
3. For each $i$, the sum of a maximum-sum subsequence, beginning with $x_i$, is computed as $m_i - s_i + x_i$. This step is implemented using a simple PRAM write, with $P_i$ writing in $b_i$.
4. Finally, the sum $L$ of the overall maximum-sum subsequence is found. Here, a PRAM concurrent-write operation suffices (with the *maximum* of the values written by the processors chosen for storing in $L$). Similarly, the starting index $u$ of the overall maximum-sum subsequence is computed using a PRAM concurrent-write operation (with the *minimum* of the values written by the processors chosen for storing in $u$). The index at which the maximum-sum subsequence ends is computed as $v = a_u$.

The algorithm is given in what follows:

```
Algorithm BSR MAXIMUM SUM SUBSEQUENCE
Step 1: for $j = 1$ to $n$ do in parallel
            for $i = 1$ to $n$ do in parallel
                $s_j \leftarrow \sum_{i \leq j} x_i$
            end for
        end for
Step 2: (2.1) for $j = 1$ to $n$ do in parallel
                  for $i = 1$ to $n$ do in parallel
                      $m_j \leftarrow \bigcap_{i \geq j} s_i$
                  end for
              end for
        (2.2) for $j = 1$ to $n$ do in parallel
                  for $i = 1$ to $n$ do in parallel
                      $a_j \leftarrow \bigcap_{s_i = m_j} i$
                  end for
              end for
Step 3: for $i = 1$ to $n$ do in parallel
            $b_i \leftarrow m_i - s_i + x_i$
        end for
Step 4: (4.1) for $i = 1$ to $n$ do in parallel
```

```
            (i) $L \leftarrow \bigcap_{i \geq 1} b_i$
           (ii) if $b_i = L$
                then $u \leftarrow \bigcup_{i \geq 1} i$
                end if
        end for
    (4.2) $v \leftarrow a_u$. ■
```

Each step of the algorithm runs in $O(1)$ time. Thus, $p(n) = n$, $t(n) = O(1)$, and $c(n) = O(n)$, which is optimal.

**A MAU for the PRAM and BSR.** We conclude our description of the BSR model by illustrating how the same MAU implemented as a combinational circuit can serve both the PRAM and BSR. A *combinational circuit* is a device consisting of components arranged in *stages*. A datum travels through the circuit from one end to the other in one direction: It is never allowed to backtrack midway. Once it has reached the opposite end, then (and only then) the datum is allowed (if necessary) to travel in the other direction (thus returning where it came from). Each component is a simple processor capable of performing in constant time some elementary operations such as adding or comparing two numbers. It receives a constant number of inputs and produces a constant number of outputs. The inputs to the components in each stage are received from the outside world or from components in a previous stage using direct links. The outputs from a component are sent to components in a subsequent stage or to the outside world. The *width* of a combinational circuit is the maximum number of components forming a stage. The *depth* of a combinational circuit is the maximum number of components forming a path from one end of the circuit to the other. The *size* of a combinational circuit is the total number of components it uses.

*Example 7.2.* The combinational circuit of Fig. 7.5 computes the prefix sums of four numbers (received as input by its leftmost stage) and produces them as output (through its rightmost stage). It can also compute the suffix sums of its inputs. As well, the circuit may be used to distribute a value received on one input line to several output lines. The circuit has a width of 4, a depth of 3, and a size of 12. □

The combinational circuit depicted in Fig. 7.6 serves as a memory access unit for both the PRAM and the BSR models. It is made up of two circuits:

1. A sorting circuit (the box labeled SORT) of width $O(N + M)$, depth $O(\log(N + M))$, and size $O((N + M)\log(N + M))$, and
2. A circuit for computing prefix and suffix sums (the box labeled PREFIX). This is the circuit of Fig. 7.5 rotated 90 degrees clockwise and capable of receiving $N + M$ inputs. It has a width of $N + M$, a depth of $1 + \log(N + M)$, and a size of $N + M + ((N + M)\log(N + M))$.

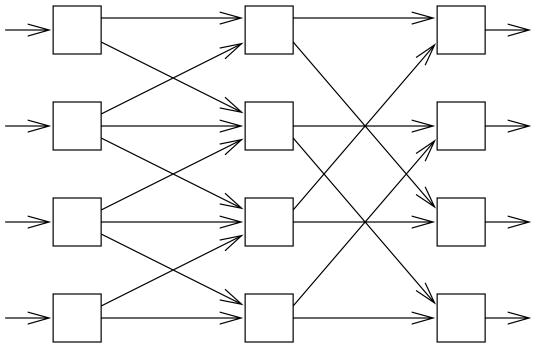


**Fig. 7.5.** A combinational circuit for computing prefix and suffix sums.

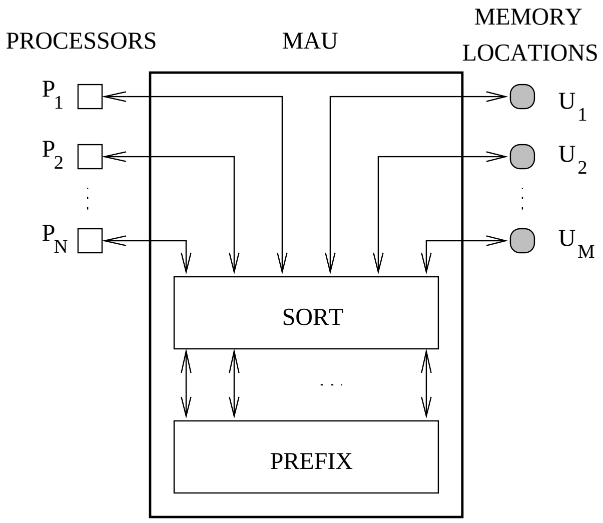


**Fig. 7.6.** A MAU for the PRAM and BSR models.

On the PRAM, when a memory access operation is to be executed, each processor $P_i$ submits a record (INSTRUCTION, $a_i, d_i, i$) to the MAU, where INSTRUCTION is a field containing the type of memory access operation, $a_i$ is the address of the memory location $U_{a_i}$ to which $P_i$ is requesting access, $d_i$ is a variable whose value is to be read from or written into the memory location $U_{a_i}$, and $i$ is $P_i$'s unique index. Similarly, each memory location $U_j$ submits a record (INSTRUCTION, $j, h_j$) to the MAU, where INSTRUCTION is an empty field to be filled later if required, $j$ is the unique address of $U_j$, and $h_j$ carries the contents of $U_j$. These records are sorted collectively by their second fields (i.e., the memory addresses) in the SORT circuit. All transfers of information from processor to memory records (including all computations required by a concurrent write), and vice versa, take place in the PREFIX circuit. Each record then returns to its source (processor or memory location) by retracing its own path through the MAU. On the BSR model, the memory access operations of the PRAM instruction repertoire are executed in the same way as just described. When a BROADCAST instruction is to be performed, each processor $P_i$ submits a record $(i, g_i, d_i)$ to the MAU, where $i$ is the processor's index, $g_i$ its *tag*, and $d_i$ its *datum*. Similarly, each memory location $U_j$ submits a record $(j, l_j, v_j)$ to the MAU, where $j$ is $U_j$'s *index*, $l_j$ its *limit*, and $v_j$ a *variable* that holds the datum to be stored in $U_j$. These records are now sorted by their second fields (i.e., the *tags* and *limits*) in the SORT circuit. Again, all transfers of information among records, including selection and reduction, take place in the PREFIX circuit. All records are then routed back to their sources.

*Example 7.3.* Assume for the purpose of this example that $N = M = 4$, and let the **BROADCAST** instruction be

$$\underset{1 \leq j \leq M}{v_j} \leftarrow \sum_{\substack{g_i < l_j \\ 1 \leq i \leq N}} d_i .$$

Suppose that the four processor records $(i, g_i, d_i)$ are

$$(1, 15, 9), \quad (2, -4, -5), \quad (3, 17, -2), \quad (4, 11, 10),$$

while the four memory records $(j, l_j, v_j)$, are

$$(1, 16, v_1), \quad (2, 12, v_2), \quad (3, 18, v_3), \quad (4, -6, v_4).$$

Initially, $v_j = 0$, for $1 \leq j \leq 4$. When the **BROADCAST** instruction is complete, we want $v_1 = 9-5+10 = 14$ (since 15, $-4$, and 11 are less than 16), $v_2 = -5+10 = 5$ (since $-4$ and 11 are less than 12), $v_3 = 9-5-2+10 = 12$ (since all *tags* are less than 18), and $v_4 = 0$ (since no *tag* is less than $-6$).

The processor and memory records are fed as input to the SORT circuit. They exit from it sorted (from left to right) on their second fields (i.e., the $g_i$ and the $l_j$) as $\{(4, -6, v_4), (2, -4, -5), (4, 11, 10), (2, 12, v_2), (1, 15, 9), (1, 16, v_1), (3, 17, -2), (3, 18, v_3)\}$. Note that in case of equality, a memory record precedes a processor record (since the selection rule is '$<$' and we don't want any datum $d_i$ to be included in the computation of $v_j$ unless its tag $g_i$ is strictly smaller than $l_j$).

The sorted sequence of records now enters a PREFIX circuit whose width is $N + M = 8$ and whose depth is $1 + \log(N + M) = 4$. This circuit computes the values of the $v_j$. It performs *selection* and *reduction* by computing the sum of the $d_i$ entries of all processor records preceding $v_j$'s record in the sorted sequence. This computation uses only the links going 'down' and 'to the right' from one stage to the next in the PREFIX circuit (again because the selection rule is '$<$'). Depending on whether it has one or two inputs and one or two outputs, a component of the PREFIX circuit behaves in one of the four ways illustrated in Fig 7.7.

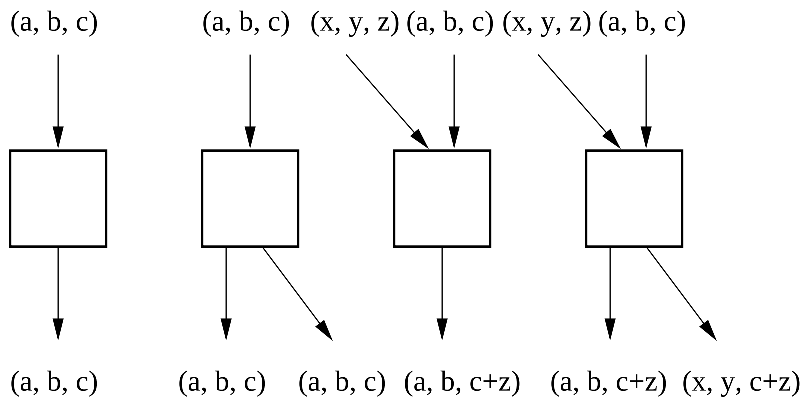


**Fig. 7.7.** Behavior of components in PREFIX circuit.

After the last stage of the PREFIX circuit has performed its computation, the records have become $\{(4,-6,0)$, $(2,-4,-5)$, $(4,11,5)$, $(2,12,5)$, $(1,15,14)$, $(1,16,14)$, $(3,17,12)$, $(3,18,12)\}$. Note in particular that $v_4 = 0$, $v_1 = 14$, $v_2 = 5$, and $v_3 = 12$, as required. The four memory records $(4,-6,0)$, $(2,12,5)$, $(1,16,14)$, and $(3,18,12)$ now retrace their paths through the MAU to return to their respective memory locations. □

When $N = O(M)$, the MAU of Fig 7.6 has a width of $O(M)$, a depth of $O(\log M)$, and a size of $O(M \log M)$. That this circuit is optimal for all three measures is seen by deriving $\Omega(M)$, $\Omega(\log M)$, and $\Omega(M \log M)$ lower bounds on the width, depth, and size, respectively, of any combinational circuits capable of implementing the simplest of the memory access operations allowed on the PRAM (namely, when each processor gains access to a distinct memory location). Finally, note that although the MAU's depth is $O(\log M)$ the time to gain access to memory on both the PRAM and BSR models is taken to be constant.

## 7.3 Synergy in parallelism

A central 'belief' in the theory and practice of parallel computation is stated as follows:

**Claim:** If a certain computation can be performed with $p$ processors in time $t_p$ on a given model and with $q$ processors in time $t_q$ on the same model, where $q < p$, then for a positive constant $\alpha \geq 1$,

$$1 \leq \frac{t_q}{t_p} \leq \alpha \frac{p}{q}. \square$$

The preceding statement essentially puts a limit on how well one can do by increasing the number of processors solving a problem. It also puts a limit on how badly an algorithm can perform when the number of processors is decreased. Thus, when $q = 1$, we have $t_1/t_p \leq \alpha p$. This is known as the 'speedup theorem', which says that when $p$ processors are used to solve a problem requiring time $t_1$ sequentially, the parallel time $t_p$ can be *at most* (on the order of) $p$ times smaller than $t_1$. The converse situation covered by the claim occurs when the number of processors assigned to a problem is *reduced* from $p$ to $q$. This is known as the 'slowdown theorem', according to which a time of $t_p$ achieved by $p$ processors when solving a problem decreases *at most* by a factor (on the order) of $p/q$ when using $q$ processors, $1 < q < p$.

The speedup and slowdown 'theorems' are true for many standard computations of the type described so far in this chapter. These include sorting, searching, matrix computations, and so on. However, there is ample evidence to contradict these 'theorems'. Examples abound even in everyday life. Many tasks done by one person in time $t_1$ are often completed by $p$ people in time $t_p$, where $t_p$ is significantly *smaller* than $t_1/p$. By contrast, $p/2$ people will perform those tasks in time $t_{p/2}$, where $t_{p/2}$ is a lot *greater* than $2t_p$. Huge construction jobs provide a perfect illustration.

Our purpose in this final section is to show that for a large class of computational problems, the claim that $t_q/t_p \leq \alpha p/q$, for $q < p$, does *not* hold. Specifically, we exhibit computations satisfying a computational phenomenon called *parallel synergy* and stated as follows:

**Parallel Synergy:** There exist computations performed with $p$ processors in time $t_p$ on a given model and by $q$ processors in time $t_q$ on the same model, where $q < p$, and for which

$$\frac{t_q}{t_p} \text{ is } \textit{asymptotically} \text{ greater than } \frac{p}{q}. \ \square$$

When parallel synergy applies, $p$ processors can perform a computation *asymptotically* faster than $t_1/p$. In other words, the running time is reduced by a factor larger than the number of processors used. On the other hand, $q$ processors, $1 < q < p$, perform such a computation *asymptotically* slower than $pt_p/q$. This means that, when full parallelism is not used, the running time increases by a factor larger than the processor ratio. In both cases, 'asymptotically' is used to indicate that $(t_q/t_p)/(p/q)$ is not just a constant.

Parallel synergy manifests itself in nonconventional computations arising, for example, in time-dependent applications in which a computer receives its input in real time and has to produce an output by a certain deadline, in applications where inputs vary with time, or where outputs affect subsequent inputs, and so on. We illustrate parallel synergy with the help of the following examples.

**Independent Input Streams.** Suppose that a computer receives $n$ independent and simultaneous streams of data as input. Each stream is of the form $< v_1, v_2, \ldots, v_n >$ and represents a distinct cyclic permutation of the values in the sequence $X = (x_1, x_2, \ldots, x_n)$. For example, when $n = 5$, there are five input streams as follows: $< x_1, x_2, x_3, x_4, x_5 >$, $< x_5, x_1, x_2, x_3, x_4 >$, $< x_4, x_5, x_1, x_2, x_3 >$, $< x_3, x_4, x_5, x_1, x_2 >$, and $< x_2, x_3, x_4, x_5, x_1 >$. It is also the case that within each stream value $v_{i+1}$ arrives $n$ time units after value $v_i$, $1 \leq i \leq n-1$. Receiving a value from a stream requires one time unit. Thus, a single processor can monitor the values in only one stream: By the time the processor reads and stores the first value of a selected stream, it is too late to turn and process the remaining $n-1$ values from the other streams, which arrived at the same time. Furthermore, a stream remains active if and only if its first value has been read and stored by a processor.

An application in which the situation just described occurs is when $n$ sensors are used to make certain measurements from an environment. Each $x_i$ is measured in turn by a different sensor and relayed to the computer. It takes $n$ time units to make a measurement. It is important to use several sensors to make the same set of measurements (in different orders): This increases the chances that the measurements will be made, even if some sensors break down; furthermore, it allows parallelism to be exploited advantageously.

Suppose that the application calls for finding the smallest value in $X$. If the computer used is a sequential one, its single processor selects a stream and reads the consecutive values it receives, keeping track of the smallest encountered so far. By our assumption, a processor can read a value, compare it to the smallest so far, and update the latter if necessary, all in one time unit. It therefore takes $n$ time units to process the $n$ inputs, plus $n(n-1)$ time units of waiting time in between consecutive inputs. Therefore, after exactly $n^2$ time units, the minimum value is known, and $t_1 = n^2$.

Now consider the situation when a parallel computer is used. For instance, let the computer be a complete binary tree of processors with $n$ leaves (and a total of $p = 2n-1$ processors). Each leaf processor is connected to an input stream. As soon as the first $n$ values arrive at the leaves, with each leaf receiving one value from a different stream, computation of the minimum can commence. Each leaf reads its input value and sends it to its parent. The latter finds the smaller of the two values received from its children and sends it to its parent. This continues up the tree, and after $t_p = \log n$ time units, the minimum emerges from the root. It follows that $t_1/t_p = n^2/\log n$, and this is asymptotically greater than $p$.

What if the tree has fewer than $n$ leaves? For example, suppose that a tree with $n^{1/2}$ leaves (and a total of $q = 2n^{1/2} - 1$ processors) is used. The best we can do is assign the $n^{1/2}$ leaf processors to monitor $n^{1/2}$ streams whose first $n^{1/2}$ values are distinct from one another. Since consecutive data in each stream are separated by $n$ time units, each leaf processor requires $(n^{1/2}-1)n$ time to see the first $n^{1/2}$ values in its stream and determine their minimum. The $n^{1/2}$ minima thus identified now climb the tree, and the overall minimum is obtained in $\log n^{1/2}$ time. Therefore, $t_q = (n^{1/2}-1)n + (\log n)/2$. Clearly, $t_q/t_p$ is asymptotically greater than $p/q$.

**Accumulating Data.** In a certain application, a set of $n$ data is received every $k$ time units and stored in a computer's memory. Here $2 < k < n$; for example, let $k = 5$. The $i$th data set received is stored in the $i$th row of a two-dimensional array $A$. In other words, the elements of the $i$th set occupy locations $A(i,1)$, $A(i,2)$, $\ldots$, $A(i,n)$. At most $2^n$ such sets may be received. Thus, $A$ has $2^n$ rows and $n$ columns. Initially, $A$ is empty. The $n$ data forming a set are received and stored simultaneously: One time unit elapses from the moment the data are received from the outside world to the moment they settle in a row of $A$. Once a datum has been stored in $A(i,j)$, it requires one time unit to be processed; that is, a certain operation must be performed on it which takes one time unit. This operation depends on the application. For example, the operation may simply be to replace $A(i,j)$ by $(A(i,j))^2$. The computation terminates once all data currently in $A$ have been processed, *regardless of whether more data arrive later.*

Suppose that we use a sequential computer to solve the problem. It receives the first set of $n$ data in one time unit. It then proceeds to update it. This requires $n$ time units. Meanwhile, $n/5$ additional data sets would have arrived in $A$, and must be processed. The computer does not catch up with the arriving data until they cease to arrive. Therefore, it must process $2^n \times n$ values. This requires $2^n \times n$ time units. Hence, $t_1 = 1 + 2^n \times n$.

Now consider what happens when a PRAM with $p = n$ processors is used. It receives the first data set, stores it in $A(1,1)$, $A(1,2)$, $\ldots$, $A(1,n)$, and updates it to $(A(1,1))^2$, $(A(1,2))^2$, $\ldots$, $(A(1,n))^2$, all in two time unit. Since all data currently in $A$ have been processed and no new data have been received, the computation terminates. Thus, $t_p = 2$. Clearly, $t_1/t_p$ is asymptotically greater than $p$.

Finally, let the PRAM have only $q$ processors, where $q < n$, and assume that $(n/q) > 5$. The first set of data is processed in $n/q$ time units. Meanwhile, $(n/q)/5$ new data sets would have been received. This way, the PRAM cannot catch up with the arriving data until the data cease to arrive. Therefore, $2^n \times n$ data must be processed, and this requires $(2^n \times n)/q$ time units. It follows that $t_q = 1 + (2^n \times n)/q$, and once again, $t_q/t_p$ is asymptotically greater than $p/q$.

**Varying Data.** An array $A$ of size $p$ resides in the memory of a computer, such that $A(i) = 0$ for $1 \leq i \leq p$. It is required to set $A(i)$ to the value $p^x$, for all $i$, $1 \leq i \leq p$, where $x$ is some positive integer constant, such that $1 < x \leq p$. One condition of this computation is that at no time during the update two elements of $A$ differ by more than a certain constant $w < p$.

A sequential computer updates each element of $A$ by $w$ units at a time, thus requiring $t_1 = p \times (p^x/w)$ time to complete the task. Now, let a PRAM with $p$ processors be available. The processors compute $p^x$ using a concurrent-write operation in one time unit: Each of the first $x$ processors writes the value $p$ in some memory location $X$; the *product* of all such values written is stored in $X$ as $p^x$. All $p$ processors now read $X$ and prite $p^x$ in all positions of $A$

simultaneously in another time unit. Thus, $t_p = 2$, and $t_1/t_p$ is asymptotically greater than $p$. Finally, suppose that $q$ processors are available, where $q < p$. The PRAM now updates the elements of $A$ in groups of $q$ elements by $w$ units at a time. The total time required is $t_q = (p/q) \times (p^x/w)$. It follows that $t_q/t_p$ is asymptotically greater than $p/q$.

The preceding examples dramatically illustrate the tremendous potential of parallel computation. In each case, when moving from a sequential to a parallel computer, the benefit afforded by parallel synergy was significantly greater than the investment in processors. Also, any reduction in processing power resulted in a disproportionate loss. Yet, in more than one sense, these examples barely scratch the surface. Indeed, each of the aforementioned computations can in principle be performed, albeit considerably slowly, on a sequential computer. There are situations, however, were using a parallel computer is not just a sensible approach—it is the *only* way to complete successfully the computation involved. This is the case, for instance, in applications where the input data cease to exist or become redundant after a certain period of time, or when the output results are meaningful only by a certain deadline. Similarly, in each of the examples where parallel synergy occurred, the models of computation used were conventional static engines. One of the main research challenges is to study the role played by parallelism in a paradigm of computation where processors are *dynamic agents* capable of manifesting themselves within all aspects of the physical universe.

## 8. BIBLIOGRAPHICAL REMARKS

A good introduction to the design and analysis of sequential algorithms is provided in Cormen et al. [CLR90]. The design and analysis of parallel algorithms are covered in Akl [A85, A89, A97], Akl and Lyons [AL93], JáJá [J92], Kumar et al. [KGGK94], Leighton [L92], and Reif [R93]. Combinational circuits implementing the memory access unit of the PRAM are described in Akl [A97], Fava Lindon and Akl [FA93], and Vishkin [V84].

The sequential complexity of sorting and the 0-1 principle are studied in Knuth [K73] along with the design and analysis of several optimal sorting algorithms. A proof of correctness and an analysis of algorithm LINEAR ARRAY SORT are provided in Akl [A97]. Algorithm MESH SORT is due to Marberg and Gafni [MG88]. Algorithm MULTIDIMENSIONAL ARRAY SIMPLE SORT was first proposed by Akl and Wolff [AW97]. It can be seen as a generalization of an algorithm designed by Scherson et al. [SSS86] for sorting on a mesh. Furthermore, its idea of repeatedly going through the dimensions of the multidimensional array during each iteration of the outer **for** loop is reminiscent of the ASCEND paradigm (originally proposed for the hypercube and related interconnection networks [PV81]). Algorithm MULTIDIMENSIONAL ARRAY FAST SORT was discovered by Kunde [K87].

Algorithm HYPERCUBE SORT follows directly from the work of Akl and Wolff [AW97]. A different derivation of this algorithm appears in [G91] where it is shown to be equivalent to the *odd-even-merge* [B68] and to the *balanced* [DPRS83] sorting circuits. Other algorithms for sorting on the hypercube, including those with the same running time (i.e., $O(\log^2 M)$) as well as asymptotically faster ones, are considerably more complicated [A85, L92]. Because its degree and diameter are sublogarithmic in the number of its vertices, the star graph interconnection network has received a good deal of attention lately (see, for example, Akers et al. [AHK87], Akers and Krishnamurthy [AK87, AK89], Akl [A97], Akl et al. [ADF94], Akl and Qiu [AQ92, AQ93], Akl et al. [AQS92, AQS93], Akl and Wolff [AW97], Chiang and Chen [CC95], Dietzfelbinger et al. [MS91], Fragopoulou [F95], Fragopoulou and Akl [FA94, FA95a, FA95b, FA95c, FA95d, FA96a], Fragopoulou et al. [FA96b], Gordon [G91], Jwo et al. [J90], Menn and Somani [MS90], Nigam et al. [NSK90], Qiu [Q92], Qiu and Akl [QA94a, QA94b], Qiu et al. [QAM94, QMA91a, QMA91b, QMA93], Rajasekaran and Wei [RW93, RW97], and Sur and Srimani [SS91]. Algorithms for sorting on the $n$-star in $O(n^3 \log n)$ time are described in Akl and Wolff [AW97], Menn and Somani [MS90] and Rajasekaran and Wei [RW93]. Another algorithm for sorting on the $n$-star is described in [G91], and a lower bound of $\Omega((\log n!)^2)$ on its running time is given, but no upper bound.

A fast sequential algorithm for multiplying two matrices is described in Coppersmith and Winograd [CW87]. Several parallel algorithms for matrix multiplication on the hypercube, including the algorithm of Section 4.1, together with their applications, are presented in Dekel et al. [DNS81]. References to, and descriptions of, other parallel algorithms for matrix multiplication and their applications to graph theoretic and numerical problems, can be found in Akl [A89], Leighton [L92], Reif [R93], and Ullman [U84]. Miscellaneous hypercube algorithms are provided in Akl [A85, A89], Akl and Lyons [AL93], Blelloch [B90], Das et al. [DDP90], Ferreira [F96b], Fox et al. [FJLOSW88], Hatcher and Quinn [H91], Hillis [H85], Leighton [L92], Qiu and Akl [QA97], Ranka and Sahni [RS90], Seitz [S84, S85], and Trew and Wilson [TW91].

The algorithms of Section 5.1 for computing prefix sums can be used in a more general setting where the operation '$+$' is replaced with any binary associative operation '$\circ$', such as '$\times$', MIN, MAX, AND, OR, and so on. Prefix computation plays a central role in the design of efficient parallel algorithms in a variety of applications ranging from fundamental computations such as array packing, sorting, searching, and selection, to numerical computations, graph theory, picture processing, and computational geometry. A number of such applications are described in Akl [A97] and Lakshmivarahan and Dhall [LD94]. Algorithm PRAM SORT was originally proposed by Cole [C88]. Lower bounds and sequential algorithms for fundamental problems in computational geometry are provided in Preparata and Shamos [PS85]; see also Mulmuley [M93] and O'Rourke [O94]. Parallel algorithms for computational geometric problems are described in Akl and Lyons [AL93] and Goodrich [G97]. Algorithm PRAM CONVEX HULL was first described in Aggarwal et al. [ACGOY88] and Atallah and Goodrich [AG86].

The pointer-jumping method and algorithm PRAM LINKED LIST PREFIX were originally introduced in Wyllie [W79]. It is interesting to note that prefix computation can be done sequentially on a linked list of $n$ nodes in $O(n)$ time and, consequently, the parallel algorithm's cost of $O(n \log n)$ is not optimal. However, the main advantage of the algorithm, besides its elegance and simplicity, is the fact that it adheres faithfully to the definition of a linked list data structure, namely, that the list is made up of nodes stored in arbitrary locations in memory with a pointer linking each node to its successor in the list. It is in fact possible, once this definition is modified, to design algorithms that use $O(n/\log n)$ processors and run in $O(\log n)$ time. Such algorithms are based on the assumption that the linked list is stored in an *array*, in which each node is not necessarily adjacent to its successor in the linked list. The array is divided into subarrays of $O(\log n)$ elements, with each processor receiving one such array to process. See, for example, Anderson and Miller [AM91] and Cole and Vishkin [CV86a, CV88, CV89]. Algorithms that use even fewer processors, but whose running time is higher, are given in Cole and Vishkin [CV86b, CV86c], Kruskal et al. [KRS85], Snir [S86], and Wagner and Han [WH86]. As is the case with prefix computation on a linear array, algorithm PRAM LINKED LIST PREFIX can be used in a variety of contexts whenever '+' is replaced by any binary associative operation. Numerous applications of this algorithm to problems defined on pointer-based data structures appear in the literature; see, for example, Abrahamson et al. [ADKP89], Chen et al. [CDA91], Eppstein and Galil [EG88], Kruskal et al. [KRS90], and Lin and Olariu [LO91]. The Euler tour method was first used effectively in Tarjan and Vishkin [TV85] and Vishkin [V85]. A survey of its applications is given in Karp and Ramachandran [KR90].

A tutorial survey of algorithmic aspects of meshes enhanced with buses is provided in Akl [A97]. There are many algorithms in the literature for meshes, and more generally multidimensional arrays, augmented with fixed electronic buses; see, for example, Aggarwal [A86], Bokhari [B84], and Stout [S83] for global buses, and Bhagavathi et al. [BOSW94], Chen et al. [CCCS90], and Prasanna Kumar and Raghavendra [PR87] for row and column buses. Meshes with reconfigurable buses have been extensively studied; see, for example, Alnuweiri [AC94], Ben-Asher and Shuster [BS91], and Nigam and Sahni [NS94]. Algorithms for arrays augmented with optical buses are given in Chiarulli et al. [CML87], Hamdi [H95], Pavel [P96] and Pavel and Akl [PA96a, PA96b, PA96c, PA96d, PA98].

Broadcasting with selective reduction was first proposed by Akl and Guenther [AG89]. It is shown in Akl [A97] that BSR is strictly more powerful than the PRAM: A certain computation can be performed in constant time on BSR using $n$ processors, while requiring $\Omega(n)$ time on an $n$-processor PRAM. Several algorithms have been proposed for solving various computational problems on this model; see, for example, Akl and Chen [AC96], Akl and Guenther [AG91], Akl and Lyons [AL93], Chen [C97], Gewali and Stojmenović [GS94], Melter and Stojmenović [MS95], Semé and Myoupo [SM97], Springsteel and Stojmenović [SS89], Stojmenović [S96], and Xiang and Ushijima [XU98, XU99]. Different implementations of BSR appear in Akl et al. [AFG91], Akl and Guenther [AG89], and Fava Lindon and Akl [FA93]. A generalization of BSR to allow for multiple selection criteria is proposed in Akl and Stojmenović [AS94]. An implementation of the MAU for this generalization and several algorithms for solving problems on it are described in Akl and Stojmenović [AS96].

Parallel synergy was first introduced in a restricted form in Akl [A93]. The concept was later generalized and extended in Akl [A97], Akl and Bruda [AB99], Akl and Fava Lindon [AF94, AF97], Bruda and Akl [BA98a, BA98b, BA99], Fava Lindon [F92, F96a], Luccio and Pagli [LP92a, LP92b], and Luccio et al. [LPP92].

## References

[ADKP89] Abrahamson, K., Dadoun, N., Kirkpatrick, D., Prztycka, T., A simple parallel tree contraction algorithm, *Journal of Algorithms*, Vol. 10, 1989, 187–302.

[A86] Aggarwal, A., Optimal bounds for finding maximum on array of processors with $k$ global buses, *IEEE Transactions on Computers*, Vol. 35, 1986, 62–64.

[ACGOY88] Aggarwal, A., Chazelle, B., Guibas, L.J., Ó'Dúnlaing, C., Yap, C.K., Parallel computational geometry, *Algorithmica*, Vol. 3, 1988, 293–327.

[AHK87] Akers, S.B., Harel, D., Krishnamurthy, B., The star graph: An attractive alternative to the $n$-cube, *Proceedings of the International Conference on Parallel Processing*, 1987, 393–400.

[AK87] Akers, S.B., Krishnamurthy, B., The fault tolerance of star graphs, *Proceedings of the International Conference on Supercomputing*, Vol. 3, 1987, 270–276.

[AK89] Akers, S.B., Krishnamurthy, B., A group theoretic model for symmetric interconnection networks, *IEEE Transactions on Computers*, Vol. 38, 1989, 555–566.

[A85] Akl, S.G., *Parallel Sorting Algorithms*, Academic Press, Orlando, Florida, 1985.

[A89] Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[A93] Akl, S.G., Parallel synergy, *Parallel Algorithms and Applications*, Vol. 1, 1993, 3–9.

[A97] Akl, S.G., *Parallel Computation: Models and Methods*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[AB99] Akl, S.G., Bruda, S.D., Parallel real-time optimization: Beyond speedup, Technical Report No. 1999-421, Department of Computing and Information Science, Queen's University, Kingston, Ontario, January 1999.

[AC96] Akl, S.G., Chen, L., Efficient parallel algorithms on proper circular arc graphs, *IEICE Transactions on Information and Systems, Special Issue on Architecture, Algorithms and Networks for Massively Parallel Computing*, Vol. E79-D, 1996, 1015–1020.

[ADF94] Akl, S.G., Duprat, J., Ferreira, A.G., Hamiltonian circuits and paths in star graphs, in: I. Dimov, O. Tonev (eds.), *Advances in Parallel Algorithms*, IOS Press, Sofia, Bulgaria, 1994, 131–143.

[AG89] Akl, S.G., Guenther, G.R., Broadcasting with selective reduction, *Proceedings of the IFIP Congress*, 1989, 515–520.

[AG91] Akl, S.G., Guenther, G.R., Applications of broadcasting with selective reduction to the maximal sum subsegment problem, *International Journal of High Speed Computing*, Vol. 3, 1991, 107–119.

[AF94] Akl, S.G., Fava Lindon, L., Paradigms admitting superunitary behavior in parallel computation, *Proceedings of the Joint Conference on Vector and Parallel Processing (CONPAR)*, Lecture Notes in Computer Science, No. 854, Springer-Verlag, Berlin, 1994, 301–312.

[AF97] Akl, S.G., Fava Lindon, L., Paradigms for superunitary behavior in parallel computations, *Parallel Algorithms and Applications*, Vol. 11, 1997, 129–153.

[AFG91] Akl, S.G., Fava Lindon, L., Guenther, G.R., Broadcasting with selective reduction on an optimal PRAM circuit, *Technique et Science Informatiques*, Vol. 10, 1991, 261–268.

[AL93] Akl, S.G., Lyons, K.A., *Parallel Computational Geometry*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[AQ92] Akl, S.G., Qiu, K., Les réseaux d'interconnexion star et pancake, in: M. Cosnard, M. Nivat, Y. Robert (eds.), *Algorithmique parallèle*, Masson, Paris, 1992, 171–181.

[AQ93] Akl, S.G., Qiu, K., A novel routing scheme on the star and pancake networks and its applications, *Parallel Computing*, Vol. 19, 1993, 95–101.

[AQS92] Akl, S.G., Qiu, K., Stojmenović, I., Computing the Voronoi diagram on the star and pancake interconnection networks, *Proceedings of the Canadian Conference on Computational Geometry*, 1992, 353–358.

[AQS93] Akl, S.G., Qiu, K., Stojmenović, I., Fundamental algorithms for the star and pancake interconnection networks with applications to computational geometry, *Networks*, *Special Issue on Interconnection Networks and Algorithms*, Vol. 23, 1993, 215–226.

[AS94] Akl, S.G., Stojmenović, I., Multiple criteria BSR: An implementation and applications to computational geometry problems, *Proceedings of the Hawaii International Conference on System Sciences*, Vol. 2, 1994, 159–168.

[AS96] Akl, S.G., Stojmenović, I., Broadcasting with selective reduction: A powerful model of parallel computation, in: A.Y. Zomaya (ed.), *Parallel and Distributed Computing Handbook*, McGraw-Hill, New York, 1996, 192–222.

[AW97] Akl, S.G., Wolff, T., Efficient sorting on the star graph interconnection network, *Proceedings of the Annual Allerton Conference*, 1997.

[AC94] Alnuweiri, H.M., Constant-time parallel algorithm for image labeling on a reconfigurable network of processors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 1994, 321–326.

[AM91] Anderson, R., Miller, G., Deterministic parallel list ranking, *Algorithmica*, Vol. 6, 1991, 859–868.

[AG86] Atallah, M.J., Goodrich, M.T., Efficient parallel solutions to some geometric problems, *Journal of Parallel and Distributed Computing*, Vol. 3, 1986, 492–507.

[B68] Batcher, K.E., Sorting networks and their applications, *Proceedings of the AFIPS Spring Joint Computer Conference*, 1968, 307–314. (Reprinted in: C.L. Wu, T.S. Feng (eds.), *Interconnection Networks for Parallel and Distributed Processing*, IEEE Computer Society, 1984, 576–583.)

[BS91] Ben-Asher, Y., Shuster, A., Ranking on reconfigurable networks, *Parallel Processing Letters*, Vol. 1, 1991, 149–156.

[BOSW94] Bhagavathi, D., Olariu, S., Shen, W., Wilson, L., A unifying look at semigroup computations on meshes with multiple broadcasting, *Parallel Processing Letters*, Vol. 4, 1994, 73–82.

[B90] Blelloch, G.E., *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, Massachusetts, 1990.

[B84] Bokhari, S.H., Finding maximum on an array processor with a global bus, *IEEE Transactions on Computers*, Vol. 33, 1984, 133–139.

[BA98a] Bruda, S.D., Akl, S.G., On the data-accumulating paradigm, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, 1998, pp. 150–153.

[BA98b] Bruda, S.D., Akl, S.G., A case study in real-time parallel computation: Correcting algorithms, Technical Report No. 1998-420, Department of Computing and Information Science, Queen's University, Kingston, Ontario, December 1998.

[BA99] Bruda, S.D., Akl, S.G., The characterization of data-accumulating algorithms, *Proceedings of the International Parallel Processing Symposium*, 1999.

[CDA91] Chen, C.C.Y., Das, S.K., Akl, S.G., A unified approach to parallel depth-first traversals of general trees, *Information Processing Letters*, Vol. 38, 1991, 49–55.

[C97] Chen, L., Optimal bucket sorting and overlap representations. *Parallel Algorithms and Applications*, Vol. 10, 1997, 249–269.

[CCCS90] Chen, Y.C., Chen, W.T., Chen, G.H., Sheu, J.P., Designing efficient parallel algorithms on mesh-connected computers with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, 1990, 241–245.

[CC95] Chiang, W.K., Chen, R.J., The $(n, k)$-star graph: A generalized star graph, *Information Processing Letters*, Vol. 56, 1995, 259–264.

[CML87] Chiarulli, D.M., Melhem, R.G., Levitan, S.P., Using coincident optical pulses for parallel memory addressing, *The Computer Journal*, Vol. 30, 1987, 48–57.

[C88] Cole, R., Parallel merge sort, *SIAM Journal on Computing*, Vol. 17, 1988, 770–785.

[CV86a] Cole, R., Vishkin, U., Approximate and exact parallel scheduling with applications to list, tree, and graph problems, *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1986, 478–491.

[CV86b] Cole, R., Vishkin, U., Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms, *Proceedings of the ACM Symposium on Theory of Computing*, 1986, 206–219.

[CV86c] Cole, R., Vishkin, U., Deterministic coin tossing with applications to optimal parallel list ranking, *Information and Control*, Vol. 70, 1986, 32–53.

[CV88] Cole, R., Vishkin, U., Approximate parallel scheduling, Part 1: The basic technique with applications to optimal list ranking in logarithmic time, *SIAM Journal on Computing*, Vol. 17, 1988, 128–142.

[CV89] Cole, R., Vishkin, U., Faster optimal parallel prefix sums and list ranking, *Information and Control*, Vol. 81, 1989, 334–352.

[CW87] Coppersmith, D., Winograd, S., Matrix multiplication via arithmetic progressions, *Proceedings of the ACM Symposium on Theory of Computing*, 1987, 1–6.

[CLR90] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[DDP90] Das, S.K., Deo, N., Prasad, S., Parallel graph algorithms for hypercube computers, *Parallel Computing*, Vol. 13, 1990, 143–158.

[DNS81] Dekel, E., Nassimi, D., Sahni, S., Parallel matrix and graph algorithms, *SIAM Journal on Computing*, Vol. 10, 1981, 657–675.

[MS91] Dietzfelbinger, M., Madhavapeddy, S., Sudborough, I.H., Three disjoint path paradigms in star networks, *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1991, 400–406.

[DPRS83] Dowd, M., Perl, Y., Rudolph, L., Saks, M., The balanced sorting network, *Proceedings of the Conference on Principles of Distributed Computing*, 1983, 161–172.

[EG88] Eppstein, D., Galil, Z., Parallel algorithmic techniques for combinatorial computation, *Annual Review of Computer Science*, Vol. 3, 1988, 233–283.

[F92] Fava Lindon, L., Discriminating analysis and its application to matrix by vector multiplication on the CRCW PRAM, *Parallel Processing Letters*, Vol. 2, 1992, 43–50.

[F96a] Fava Lindon, L., *Synergy in Parallel Computation*, Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1996.

[FA93] Fava Lindon, L., Akl, S.G., An optimal implementation of broadcasting with selective reduction, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, 256–269.

[F96b] Ferreira, A.G., Parallel and communication algorithms on hypercube multiprocessors, in: A.Y. Zomaya (ed.), *Parallel and Distributed Computing Handbook*, McGraw-Hill, New York, 1996, 568–589.

[FJLOSW88] Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K., Walker, D.W., *Solving Problems on Concurrent Processors*, Vol. 1. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[F95] Fragopoulou, P., *Communication and Fault Tolerance Algorithms on a Class of Interconnection Networks*, Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.

[FA94] Fragopoulou, P., Akl, S.G., A parallel algorithm for computing Fourier transforms on the star graph, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 1994, 525–531.

[FA95a] Fragopoulou, P., Akl, S.G., Optimal communication algorithms on star graphs using spanning tree constructions, *Journal of Parallel and Distributed Computing*, Vol. 24, 1995, 55–71.

[FA95b] Fragopoulou, P., Akl, S.G., Fault tolerant communication algorithms on the star network using disjoint paths, *Proceedings of the Hawaii International Conference on System Sciences*, Vol. 2, 1995, 5–13.

[FA95c] Fragopoulou, P., Akl, S.G., A framework for optimal communication on a subclass of Cayley graph based networks, *Proceedings of the International Conference on Computers and Communications*, 1995, 241–248.

[FA95d] Fragopoulou, P., Akl, S.G., Efficient algorithms for global data communication on the multidimensional torus network, *Proceedings of the International Parallel Processing Symposium*, 1995, 324–330.

[FA96a] Fragopoulou, P., Akl, S.G., Edge-disjoint spanning trees on the star network with applications to fault tolerance, *IEEE Transactions on Computers*, Vol. 45, 1996, 174–185.

[FA96b] Fragopoulou, P., Akl, S.G., Meijer, H., Optimal communication primitives on the generalized hypercube network, *Journal of Parallel and Distributed Computing*, Vol. 32, 1996, 173–187.

[GS94] Gewali, L.P., Stojmenović, I., Computing external watchman routes on PRAM, BSR, and interconnection models of parallel computation, *Parallel Processing Letters*, Vol. 4, 1994, 83–93.

[G97] Goodrich, M.T., Parallel algorithms in geometry, in: J.E. Goodman, J. O'Rourke (eds.), *Discrete and Computational Geometry*, CRC Press, New York, 1997, 669–681.

[G91] Gordon, D.M., Parallel sorting on Cayley graphs, *Algorithmica*, Vol. 6, 1991, 554–564.

[H95] Hamdi, M., Communications in optically interconnected computer systems, in: D.F. Hsu, A.L. Rosenberg, D. Sotteau (eds.), *Interconnection Networks and Mapping and Scheduling Parallel Computations*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 21, 1995, 181–200.

[H91] Hatcher, P.J., Quinn, M.J., *Data-Parallel Programming on MIMD Computers*, MIT Press, Cambridge, Massachusetts, 1991.

[H85] Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge, Massachusetts, 1985.

[J92] JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, Massachusetts, 1992.

[J90] Jwo, J.S., Lakshmivarahan, S., Dhall, S.K., Embedding of cycles and grids in star graphs, *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1990, 540–547.

[KR90] Karp, R.M., Ramachandran, V., A survey of parallel algorithms for shared memory machines, in: Vol. A, J. van Leeuwen (ed.) *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990, 869–941.

[K73] Knuth, D.E., *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, Massachusetts, 1973.

[KRS85] Kruskal, C.P., Rudolph, L., Snir, M., The power of parallel prefix, *IEEE Transactions on Computers*, Vol. 34, 1985, 965–968.

[KRS90] Kruskal, C.P., Rudolph, L., Snir, M., Efficient parallel algorithms for graph problems, *Algorithmica*, Vol. 5, 1990, 43–64.

[KGGK94] Kumar, V., Grama, A., Gupta, A., Karypis, G., *Introduction to Parallel Computing*, Benjamin-Cummings, Menlo Park, California, 1994.

[K87] Kunde, M., Optimal sorting on multi-dimensionally mesh-connected computers, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science No. 247, Springer-Verlag, Berlin, 1987, 408–419.

[LD94] Lakshmivarahan, S., Dhall, S.K., *Parallel Computing Using the Prefix Problem*, Oxford University Press, New York, 1994.

[L92] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Mateo, California, 1992.

[LO91] Lin, R., Olariu, S., A simple optimal parallel algorithm to solve the lowest common ancestor problem, *Proceedings of the International Conference on Computing and Information*, Lecture Notes in Computer Science, No. 497, Springer-Verlag, Berlin, 1991, 455–461.

[LP92a] Luccio, F., Pagli, L., The $p$-shovelers problem (computing with time-varying data), *SIGACT News*, Vol. 23, 1992, 72–75.

[LP92b] Luccio, F., Pagli, L., The $p$-shovelers problem (computing with time-varying data), *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1992, 188–193.

[LPP92] Luccio, F., Pagli, L., Pucci, G., Three non conventional paradigms of parallel computation, *Proceedings of the Heinz Nixdorf Symposium*, Lecture Notes in Computer Science, No. 678, Springer-Verlag, Berlin, 1992, 166–175.

[MG88] Marberg, J.M., Gafni, E., Sorting in constant number of row and column phases on a mesh, *Algorithmica*, Vol. 3, 1988, 561–572.

[MS95] Melter, R.A., Stojmenović, I., Solving city block metric and digital geometry problems on the BSR model of parallel computation, *Journal of Mathematical Imaging and Vision*, Vol. 5, 1995, 119–127.

[MS90] Menn, A., Somani, A.K., An efficient sorting algorithm for the star graph interconnection network, *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1990, 1–8.

[M93] Mulmuley, K., *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[NS94] Nigam, M., Sahni, S., Sorting $n$ numbers on $n \times n$ reconfigurable meshes with buses, *Journal of Parallel and Distributed Computing*, Vol. 23, 1994, 37–48.

[NSK90] Nigam, M., Sahni, S., Krishnamurthy, B., Embedding Hamiltonians and hypercubes in star interconnection graphs, *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1990, 340–343.

[O94] O'Rourke, J., *Computational Geometry in C*, Cambridge University Press, Cambridge, England, 1994.

[P96] Pavel, S., *Computation and Communication Aspects of Arrays with Optical Pipelined Buses*, Ph.D. thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1996.

[PA96a] Pavel, S., Akl, S.G., Matrix operations using arrays with reconfigurable optical buses, *Journal of Parallel Algorithms and Applications*, Vol. 8, 1996, 223–242.

[PA96b] Pavel, S., Akl, S.G., Area-time trade-offs in arrays with optical pipelined buses, *Applied Optics*, Vol. 35, 1996, 1827–1835.

[PA96c] Pavel, S., Akl, S.G., On the power of arrays with reconfigurable optical buses, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996, 1443–1454.

[PA96d] Pavel, S., Akl, S.G., Efficient algorithms for the Hough transform on arrays with reconfigurable optical buses, *Proceedings of the International Parallel Processing Symposium*, 1996, 697–701.

[PA98] Pavel, S., Akl, S.G., Integer sorting and routing in arrays with reconfigurable optical buses, to appear in *International Journal of Foundations of Computer Science*, *Special Issue on Interconnection Networks*, 1998.

[PR87] Prasanna Kumar, V.K., Raghavendra, C.S., Array processor with multiple broadcasting, *Journal of Parallel and Distributed Computing*, Vol. 4, 1987, 173–190.

[PS85] Preparata, F.P., Shamos, M.I., *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[PV81] Preparata, F.P., Vuillemin, J.E., The cube-connected cycles: A versatile network for parallel computation, *Communications of the ACM*, Vol. 24, 1981, 300–309.

[Q92] Qiu, K., *The Star and Pancake Interconnection Networks: Properties and Algorithms*, Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1992.

[QA94a] Qiu, K., Akl, S.G., Load balancing, selection and sorting on the star and pancake interconnection networks, *Parallel Algorithms and Applications*, Vol. 2, 1994, 27–42.

[QA94b] Qiu, K., Akl, S.G., On some properties of the star graph, *Journal of VLSI Design*, *Special Issue on Interconnection Networks*, Vol. 2, 1994, 389–396.

[QA97] Qiu, K., Akl, S.G., Parallel point location algorithms on hypercubes, *Proceedings of the Tenth International Conference on Parallel and Distributed Computing*, 1997, 27–30.

[QAM94] Qiu, K., Akl, S.G., Meijer, H., On some properties and algorithms for the star and pancake interconnection networks, *Journal of Parallel and Distributed Computing*, Vol. 22, 1994, 16–25.

[QMA91a] Qiu, K., Meijer, H., Akl, S.G., Parallel routing and sorting on the pancake network, *Proceedings of the International Conference on Computing and Information*, Lecture Notes in Computer Science, No. 497, Springer-Verlag, Berlin, 1991, 360–371.

[QMA91b] Qiu, K., Meijer, H., Akl, S.G., Decomposing a star graph into disjoint cycles, *Information Processing Letters*, Vol. 39, 1991, 125–129.

[QMA93] Qiu, K., Meijer, H., Akl, S.G., On the cycle structure of star graphs, *Congressus Numerantium*, Vol. 96, 1993, 123–141.

[RW93] Rajasekaran, S., Wei, D.S.L., Selection, routing and sorting on the star graph, *Proceedings of the International Parallel Processing Symposium*, 1993, 661–665.

[RW97] Rajasekaran, S., Wei, D.S.L., Selection, routing, and sorting on the star graph, *Journal of Parallel and Distributed Computing*, Vol. 41, 1997, 225–233.

[RS90] Ranka, S., Sahni, S., *Hypercube Algorithms*, Springer-Verlag, New York, 1990.

[R93] Reif, J.H. (ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, California, 1993.

[SSS86] Scherson, I., Sen, S., Shamir, A., Shear-sort: A true two-dimensional sorting technique for VLSI networks, *Proceedings of the International Conference on Parallel Processing*, 1986, 903–908.

[S84] Seitz, C.L., Concurrent VLSI architectures, *IEEE Transactions on Computers*, Vol. 33, 1984, 1247–1265.

[S85] Seitz, C.L., The cosmic cube, *Communications of the ACM*, Vol. 28, 1985, 22–33.

[SM97] Semé, D., Myoupo, J.-F., A parallel solution of the sequence alignment problem using BSR model, *Proceedings of the International Conference on Parallel and Distributed Computing*, 1997, 357–362.

[S86] Snir, M., Depth-size tradeoffs for parallel prefix computation, *Journal of Algorithms*, Vol. 7, 1986, 185–201.

[SS89] Springsteel, F., Stojmenović, I., Parallel general prefix computations with geometric, algebraic and other applications, *International Journal of Parallel Programming*, Vol. 18, 1989, 485–503.

[S96] Stojmenović, I., Constant time BSR solutions to parenthesis matching, tree decoding, and tree reconstruction from its traversals, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, 1996, 218–224.

[S83] Stout, Q.F., Mesh-connected computers with broadcasting, *IEEE Transactions on Computers*, Vol. 32, 1983, 826–830.

[SS91] Sur, S., Srimani, P.K., A fault tolerant routing algorithm in star graphs, *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1991, 267–270.

[TV85] Tarjan, R.E., Vishkin, U., An efficient parallel biconnectivity algorithm, *SIAM Journal of Computing*, Vol. 14, 1985, 862–874.

[TW91] Trew, A., Wilson, G. (eds.), *Past, Present, Parallel*, Springer-Verlag, Berlin, 1991.

[U84] Ullman, J.D.,*Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

[V84] Vishkin, U., A parallel-design distributed implementation (PDDI) general-purpose computer, *Theoretical Computer Science*, Vol. 32, 1984, 157–172.

[V85] Vishkin, U., On efficient parallel strong orientation, *Information Processing Letters*, Vol. 20, 1985, 235–240.

[WH86] Wagner, W., Han, Y., Parallel algorithms for bucket sorting and data dependent prefix problems, *Proceedings of the International Conference on Parallel Processing*, 1986, 924–930.

[W79] Wyllie, J.C., *The Complexity of Parallel Computations*, Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

[XU98] Xiang, L., Ushijima, K., ANSV problem on BSRs, *Information Processing Letters*, Vol. 65, 1998, 135–138.

[XU99] Xiang, L., Ushijima, K., Decoding and drawing on BSR for a binary tree from its $i-p$ sequence, to appear in *Parallel Processing Letters*, 1999.

# Index

All-pairs shortest paths, 31
Array packing, 39
Asymptotic behavior, 6

Binary tree of processors, 4
Broadcasting with selective reduction (BSR), 58
Bus, 53

Combinational circuit, 2, 61
Computational geometry, 37
Convex hull, 42
Cost, 8
Cost-optimal algorithm, 9

Degree of a network, 21
Depth-first traversal, 50
Determinant, 35
Diameter of a network, 11
Distance between processors, 11
Divide and conquer, 37, 42
Dynamic agents, 68

Efficiency, 9
Euler tour, 49

Graph theory, 31

Hamilton cycle, 26
Hypercube network, 20, 28

Inorder traversal, 51
Interconnection network, 3

Level of a vertex, 51
Linear array, 10
List ranking, 54
Lower bound on running time, 8
Lowest common ancestor, 52

Matrix multiplication, 27
Maximum-sum subsequence, 59
Memory access unit (MAU), 2, 61
Mesh of processors, 12, 53
Minimum spanning tree, 33
Multidimensional array, 15, 24

Number of processors, 8

Oblivious algorithm, 11, 14
Optical buses, 55

Parallel algorithm, 1, 4
Parallel model of computation, 2
Parallel Random Access Machine (PRAM), 2
Parallel synergy, 65
Pipeline, 56
Pointer jumping, 46
Prefix sums, 37, 47

Reconfigurable bus, 54
Ring of processors, 27
Row-major order, 12
Running time, 7

Shared-memory model, 2
Size of a problem, 7
Slowdown, 8, 65
Snakelike order by dimension, 16
Sorting, 10, 40, 58
Speedup, 7, 65
Star network, 20
Suffix sums, 51

Time unit, 7
Time-optimal algorithm, 8

Zero-one principle, 11, 14

# Table of Contents

**The Design of Efficient Parallel Algorithms**
Selim G. Akl . . . . . . 0

1. INTRODUCTION . . . . . . 1
2. PARALLEL MODELS AND ALGORITHMS . . . . . . 1
    2.1 What is a parallel model? . . . . . . 2
    2.2 What is a parallel algorithm? . . . . . . 4
    2.3 How do we analyze parallel algorithms? . . . . . . 6
3. SORTING . . . . . . 10
    3.1 Linear array . . . . . . 10
    3.2 Mesh . . . . . . 11
    3.3 Multidimensional array . . . . . . 15
    3.4 Hypercube . . . . . . 20
    3.5 Star . . . . . . 20
4. MATRIX MULTIPLICATION . . . . . . 27
    4.1 Hypercube algorithm . . . . . . 28
    4.2 Graph theoretical problems . . . . . . 31
    4.3 Numerical computations . . . . . . 34
5. COMPUTING THE CONVEX HULL . . . . . . 37
    5.1 Prefix sums on the PRAM . . . . . . 37
    5.2 Array packing . . . . . . 39
    5.3 Sorting on the PRAM . . . . . . 40
    5.4 Divide and conquer . . . . . . 42
6. POINTER-BASED DATA STRUCTURES . . . . . . 46
    6.1 Pointer jumping . . . . . . 46
    6.2 Prefix sums computation . . . . . . 47
    6.3 The Euler tour . . . . . . 49
    6.4 Applications . . . . . . 51
7. CONCLUSION . . . . . . 52
    7.1 Meshes that use buses . . . . . . 53
    7.2 Broadcasting with selective reduction . . . . . . 58
    7.3 Synergy in parallelism . . . . . . 64
8. BIBLIOGRAPHICAL REMARKS . . . . . . 68