*9.44. Suppose that we had a polynomial time subprogram TSP to solve the traveling salesman decision problem (i.e., given a complete weighted graph and an integer $k$, it determines whether there is a tour of total weight at most $k$).

a) Show how to use the TSP subprogram to determine the weight of an optimal tour in polynomial time.

b) Show how to use the TSP subprogram to find an optimal tour in polynomial time.

*9.45. Show that if there were a polynomial time approximation algorithm for the knapsack problem that was guaranteed to fill the knapsack with objects whose total value differed from the optimal by a constant, then an optimal solution could be found in polynomial time. (In other words, if there is a polynomial time algorithm $A$ and an integer $k$ such that for all inputs $f$, $opt(f) - v_A(f) \leq k$, then an optimal solution could be found in polynomial time.)

*9.46. Suppose that there is a polynomial time algorithm for the CNF-satisfiability problem. Give a polynomial time algorithm that when given a CNF expression, finds a truth assignment for the variables that satisfies the expression, if one exists, or tells that the expression is not satisfiable, if that is the case.

Notes and References

The two papers that began the intensive study of $NP$-complete problems are Cook (1971) and Karp (1972). The latter outlines proofs of polynomial reducibility among many $NP$-complete problems. Both Stephen Cook and Richard Karp have won the ACM Turing Award, and their Turing Award lectures (1983 and 1986, respectively) present interesting overviews of computational complexity and their own views of the context of their work.

A major source for more detail, formalism, applications, implications, approximation algorithms, etc., is Garey and Johnson (1979), an entire book on the subject of $NP$-completeness. The definition of $NP$ given here uses an informal version of the definition of nondeterministic Turing machines given in Garey and Johnson. The latter also contains a proof of Cook's theorem, a list of several hundred $NP$-complete problems, and a long bibliography (thus we will not repeat most of the original references here).

The approximation algorithms in Sections 9.4 through 9.6 are from Sahni (1975) (knapsack); Garey, Graham, and Ullman (1972) and Johnson (1972) (bin packing); and Johnson (1974) and Wigderson (1983) (graph coloring). (The faster approximation scheme mentioned for the knapsack problem is in Ibarra and Kim (1975).) There are more algorithms and references in Garey and Johnson (1979). Empirical studies of the expected behavior of the bin-packing heuristics are in Bentley et al. (1983). Approximation algorithms for scheduling problems are in Sahni (1976). Lawler (1985) is a book on the traveling salesman problem. Garey and Johnson (1979) has more theorems concerning the unlikelihood of obtaining good approximation algorithms for some problems.
10.1 Parallelism, the PRAM, and Other Models

10.1.1 Introduction

In most of this book, our model of computation has been a general-purpose, deterministic, random access computer that carried out one operation at a time. Several times we used specialized models to establish lower bounds for various problems; these were not general-purpose machines, but they too carried out one operation at a time. We will use the term sequential algorithm for the usual one-step-at-a-time algorithms that we have studied up to now. (Such algorithms are sometimes called serial algorithms.) In this chapter we will consider parallel algorithms, algorithms in which several operations may be executed at the same time in parallel, that is, algorithms for machines that have more than one processor working on one problem at the same time.

In recent years, as microprocessors have become cheaper and the technology for interconnecting them has improved, it has become possible and practical to build general-purpose parallel computers containing a very large number of processors. There has been a burst of activity in developing the hardware, the algorithms, and the theoretical models to make use of parallel computers. It seems clear that in the future more and more parallelism will be used. The purpose of this chapter is to introduce some of the concepts, formal models, techniques, and algorithms from the growing area of parallel computation.

Parallel algorithms are natural for many applications. In image processing (for example, in vision systems for robots) different parts of a scene may be processed simultaneously, i.e., in parallel. Parallelism can speed up computation for graphics displays. In search problems (e.g., bibliographic search, scanning news stories, and text editing) different parts of the database or text can be searched in parallel. Simulation programs often do the same computation to update the states of a large number of components in the system being simulated; these can be done in parallel for each simulated time step. Artificial intelligence applications (which include image processing and a lot of searching) can benefit from parallel computing. The fast Fourier transform is implemented on specialized parallel hardware. Algorithms for many combinatorial optimization problems (such as the optimization versions of some of the NP-complete problems described in Chapter 9) involve examining a large number of feasible solutions; some of the work can be done in parallel. Parallel computation can also speed up computation easily and substantially in many other application areas.

For the examples of parallel applications just mentioned, and for some of the algorithms studied earlier in this book, there seem to be fairly straightforward ways to break up the computation into parallel subcomputations. Many other well-known and widely used algorithms seem inherently sequential. Thus a lot of work has been done both to find parallel implementations of sequential algorithms where that approach is fruitful, and to develop entirely new techniques for parallel algorithm design.

If the number of processors in parallel computers were small, say two or six, then there would be a practical advantage in using them for some problems in which computation could be speeded up by some small constant factor. But such machines, and the algorithms for them, would not be very interesting in the context of this book, where we often ignore small constants. Parallel algorithms become interesting from a computational complexity point of view when the number of processors is very large. One of the parallel machines introduced in 1986 has 65,536 processors. The number of processors is larger than the input size for many of the actual cases for which a program is used. This is where we can get significant speedups and interesting algorithms.

How much can parallelism do for us? Suppose that a sequential algorithm for a problem does $O(n^3)$ operations in the worst case for input of size $n$, and that we have a machine with $p$ processors. Then the best we can hope for from a parallel implementation of the algorithm is that it runs in $O(n^3/p)$ time, and we cannot necessarily achieve this maximum speedup in every case. Suppose that the problem is putting on socks and shoes, and that a processor is a pair of hands. A common sequential algorithm is: Put on the right sock, put on the right shoe, put on the left sock, put on the left shoe. If we have two processors we can assign one to each foot and accomplish the task in two time units instead of four. However, if we have four processors, we cannot cut the time down to one unit, because the socks must go on before the shoes.

Parallel machines have the potential to solve many problems much faster than before. However, many difficulties, both practical and concerning the theoretical models, remain to be resolved. There are several general-purpose and special-purpose formal models of parallel machines that correspond to various (real or theoretical) hardware designs. We will focus on one major class of models for general purpose parallel computers: the PRAM (pronounced "p ram"), or parallel random access machine. Although the PRAM model has some unrealistic features (which we will mention later), it is now widely used and serves as a good tool for introducing parallel computing.

We will not always give the most efficient algorithm known for a problem; our aim here is to present some techniques and algorithms that can be understood without great difficulty. Since this is a short, introductory chapter, much that is interesting and important in the study of parallel algorithms is left out. The notes and references at the end of the chapter suggest a few additional topics and sources for the reader who wishes to pursue the subject further.

10.1.2 The PRAM

A parallel random access machine (PRAM) consists of $p$ general-purpose processors, $P_1, P_2, \ldots, P_p$, all of which are connected to a large shared, random access memory $M$. (See Figure 10.1.) The processors have a private, or local, memory for their own computation, but all communication among them takes place via the
(Alternatively, the program may reside in a separate control unit that issues instructions to each processor at each step.) Several algorithms use arrays stored in the shared memory. We can assume that these are handled just as arrays in high-level languages are handled. That is, a compiler decides on some fixed arrangement of the arrays in memory following the input, and translates array references to instructions to compute specific memory addresses. For example, if the input occupies \( n \) cells, and \( \alpha \) is the \( k \)-element array, the compiler translates an instruction telling processor \( P \) to read \( \alpha[j] \) into PRAM instructions to compute \( \text{index} := n + 2k + j \), and then to read \( M[\text{index}] \). The address computation takes one PRAM step.

### 10.1.3 Other Models

Although the PRAM provides a good framework for developing and analyzing algorithms for parallel machines, the model would be difficult or expensive to provide in actual hardware. The PRAM assumes a complex communication network that allows all processors to access any memory cell at the same time, in one time step, and to write in any cell in one time step. Thus any processor can communicate with any other in two steps: One processor writes some data into a memory location in one step, and the other processor reads that location in the next step. Other parallel computation models do not have a shared memory, thus restricting communication between processors. A model that more closely resembles some actual hardware is the hypercube. A hypercube has \( 2^d \) processors for some \( d \) (the dimension), each connected to its neighbors. Figure 10.2(a) shows a hypercube of dimension 3. Each processor has its own memory and communicates with the other processors by sending messages. At each step each processor may do some computation, then send a message to one of its neighbors. To communicate with a nonneighbor, a processor may send a message that includes routing information indicating the ultimate destination; the message may take as many as \( d \) time steps to reach its destination.

In a hypercube with \( p \) processors, each processor is connected to \( \log p \) other processors. Another class of models, called bounded degree networks, restricts the connections further. In a bounded degree network, each processor is directly connected to at most \( d \) other processors, for some constant \( d \) (the degree). There are different designs for bounded degree networks; a \( 5 \times 5 \) grid is illustrated in Fig. 10.2(b). Hypercubes and bounded degree networks are more realistic models than the PRAM, but algorithms for them can be harder to specify and analyze. The routing of messages among processors, an interesting problem in itself, is eliminated in the PRAM.

Since the PRAM, while not very practical, is conceptually easy to work with when developing algorithms, a lot of effort has gone to finding efficient ways to simulate PRAM computations on other parallel models, particularly models without shared memory. For example, each PRAM step can be simulated in approximately \( O(\log p) \) steps on a bounded degree network. Thus we can develop algorithms for the PRAM and know that these algorithms can be translated to algorithms for actual machines. The translation may even be done automatically by a translator program.

In Chapter 9, we defined the class of problems \( P \) to help distinguish between tractable and intractable problems. \( P \) contains problems that can be solved
(a) A hypercube (dimension = 3). (b) A bounded degree network (degree = 4).

Figure 10.2 Other parallel architectures. (a) A hypercube (dimension = 3). (b) A bounded degree network (degree = 4).

Polynomially bounded time. For parallel computation, too, we classify problems according to their use of resources: time and processors. NC is defined as the class of problems that can be solved by a parallel algorithm with \( p \) (the number of processors) bounded by a polynomial in the input size, and the number of time steps bounded by a polynomial in the log of the input size. More succinctly, if the input size is \( n \), then \( p(n) \in O(n^k) \) for some constant \( k \), and the worst-case time, \( T(n) \) is in \( O((\log n)^m) \) for some constant \( m \). The time bound, sometimes referred to as "poly-log time" because it is a polynomial in the log of \( n \), is quite small, but we expect parallel algorithms to run very fast. The bound on the number of processors is not so small. Even for small \( k > 1 \), it may be impractical to use \( n^k \) processors for moderately large input. The reasons for using a polynomial bound in the definition of NC are similar to the reasons for using a polynomial bound on time to define the class \( P \). First, the class of problems that can be solved in poly-log time using a polynomially bounded number of processors is independent of the specific parallel computation model used (among a large class of models considered "reasonable"). Thus \( NC \) is independent of whether we are using a PRAM or a bounded degree network. Second, if a problem cannot be solved fast with a polynomial number of processors, then that is a strong statement about how hard the problem is. In fact, for most of the algorithms we will look at, \( p \in O(n) \).

10.2 Some PRAM Algorithms and the Handling of Write Conflicts

10.2.1 The Binary Fan-in Technique

Consider the problem of finding the largest key in a list of \( n \) keys. We have seen two algorithms for this problem: Algorithm 1.3 and the tournament method described in Section 3.3.2. In Algorithm 1.3, we proceeded sequentially through the list, comparing \( max \), the largest key found so far, to each remaining key. After each comparison, \( max \) may change; we cannot do the next comparison in parallel because we do not know which value to use. In the tournament method, however, elements are paired off and compared in "rounds." In succeeding rounds, the winners from the preceding round are paired off and compared. (See Fig. 3.1.) The largest key is found in \( \lceil \log n \rceil \) rounds. All of the comparisons in one round can be performed at the same time. Thus the tournament method naturally gives us a parallel algorithm.

In a tournament, the number of keys under consideration at each round decreases by half, so the number of processors needed at each round also decreases by half. To keep the description of the algorithm short and clear, however, we specify the same instructions for all processors at each time step. The extra work being done may be confusing: it is helpful to look at Fig. 10.3 first. The figure shows the work that actually contributes to the answer. A slanted line represents a read operation. A vertical line represents data that has been saved in a processor's local memory. Each processor always saves the largest key it has seen so far. A twisted line represents a write operation: a processor writes (the largest key it has seen) in the memory cell with the same number as the processor (i.e., \( P_i \) writes in \( M[i] \)). A dot represents a comparison (and some "bookkeeping" computation). If a read line comes into \( P_i \) from \( P_j \), that means \( P_i \) reads from \( M[j] \), since that is where \( P_j \) wrote. Figure 10.4 shows a complete example of the activity of all the processors. The shaded parts correspond to Fig. 10.3 and show the computations that affect the answer.

![Figure 10.3 A parallel tournament.](image)
Algorithm 10.1 A Parallel Tournament for Finding the Largest Key

**Input:** $n$ keys $x_1, x_2, \ldots, x_n$, initially in memory cells $M[1], M[2], \ldots, M[n]$.

**Output:** The largest key will be left in $M[1]$.
Parallel Algorithms

\[ \text{big} := \max[M[i], M[i+2^{-n}]]. \]

We can use the induction hypothesis for the contents of \( M[i] \), but not always for \( M[i+2^{-n}] \). The latter may not be one of the first \( n \) cells; some of the processors access cells with index greater than \( n \) (though never greater than \( 2n \)). The induction hypothesis does not tell us what is in these cells, but a simple observation does. For \( n \leq 2 \), \( M[i] \) always contains \( \pm \infty \) because, after the initialization preceding the loop, no processor ever writes into these cells. Thus it is always true that \( M[i] \) contains the maximum of \( x_j, \ldots, x_{2n} \). So, returning to the computation of \( \text{big} \), we see that

\[ M[i] \text{ contains the maximum of } x_i, \ldots, x_{2^n-1}. \]

and that

\[ M[i+2^{-n}] \text{ contains the maximum of } x_{i+2^n}, \ldots, x_{i+2^{n-1}-1}, \]

that is, the maximum of \( x_i, \ldots, x_{2^n-1} \) (or the maximum of \( x_i, \ldots, x_{2^n} \) if \( i+2^n-1 > 2n \)).

So, at the \( i \)th iteration, \( \text{big} \) is assigned the maximum of \( x_i, \ldots, x_{2^n-1} \), and this value is written in \( M[i] \). Also, at the \( i \)th iteration, \( \text{incr} \) is doubled, so now \( \text{incr} = 2^i \).

This establishes the induction claim for \( i \) and completes the proof. \( \square \)

Note that Algorithm 10.1 overwrites the input data. If this is not desirable, it is a simple matter to copy the input (in one parallel step) to a scratch area in memory and do the computation there.

With only slight modification to Algorithm 10.1, the binary fan-in scheme can be used to find the minimum of \( n \) keys, to compute the Boolean or or Boolean and of \( n \) bits, and to compute the sum of \( n \) keys, each in \( O(\lg n) \) steps, without any write conflicts.

10.2.2 Some Easily Parallelizable Algorithms

Consider the problem of multiplying two \( n \times n \) matrices \( A \) and \( B \). The usual matrix multiplication algorithm has a natural parallel version. Recall the formula for the entries of the product matrix \( C \):

\[ c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj} \quad \text{for } 1 \leq i, j \leq n. \]

Since concurrent reads are allowed, we can simply assign one processor to each element of the product, thus using \( n^2 \) processors. Each processor \( P_{ij} \) can compute its \( c_{ij} \) in \( 2n \) steps. (There are \( n \) terms to add, and each term needs two reads.) With more processors we can do better. Clearly, the binary fan-in scheme used in Algorithm 10.1 can be used to add \( n \) integers in \( \lceil \lg n \rceil + 1 \) steps. The work done would "look" the same as in Fig. 10.3, but the dots would represent additions instead of comparisons. Thus the matrix product can be computed in \( O(\lg n) \) time with \( O(n^2) \) processors. (Of course, for this problem the output does not all go in \( M[1] \); we assume that \( n^2 \) cells of memory are designated for the elements of \( C \).)

Many dynamic programming algorithms can be speeded up easily by doing computation in parallel. Recall that dynamic programming solutions usually involve computing elements of a table. Often the elements in one row (or column or diagonal) depend only on entries in earlier rows (columns or diagonals). Thus with \( n \) processors, all elements in one row of an \( n \times n \) table can be computed in parallel, cutting the running time of the algorithm by a factor of \( n \).

10.2.3 Handling Write Conflicts

PRAM models vary according to how they handle write conflicts. The CREW (Concurrent Read, Exclusive Write) PRAM model requires that only one processor write in a particular cell at any one step: an algorithm that would have more than one processor write to one cell at the same time is an illegal algorithm. In the Common-Write model, it is legal for several processors to write in the same cell at the same time if and only if they all write the same value. In the Priority-Write model, if several processors attempt to write in the same memory cell at the same time, the processor with the smallest index succeeds.\(^1\) The models differ in how fast they can solve various problems. To illustrate the difference, we will consider the problem of computing the Boolean or function on \( n \) bits.

Using the binary fan-in scheme of Algorithm 10.1, each processor performs an or operation on a pair of bits at each round, and the problem is solved in \( O(\lg n) \) time. This method will work on all three of the models mentioned because there are no write conflicts: the processors write the results of their operations in different memory cells. It has been shown that for the CREW model \( O(\lg n) \) time is required to compute the or of \( n \) bits (even if more than \( n \) processors are used). Consider the following algorithm, using \( n \) processors:

**Algorithm 10.2** Computing the or of \( n \) Bits

**Input:** Bits \( x_1, \ldots, x_n \) in \( M[1], \ldots, M[n] \).

**Output:** \( x_1 \lor \cdots \lor x_n \) in \( M[1] \).

1. \( P_i \) reads \( x_i \) from \( M[i] \);
2. If \( x_i = 1 \), then \( P_i \) writes 1 in \( M[1] \).

Since all the processors that write, write the same value, this is a legal program for the Common-Write and Priority-Write models. Thus the or of \( n \) bits can be computed in one step on these models with \( n \) processors.

\(^1\) Warning: The abbreviations used for the various models in research papers are not consistent. CREW and EREW are used consistently for Exclusive Read, Exclusive Write and Concurrent Read, Exclusive Write, respectively, but CREW (Concurrent Read, Concurrent Write) sometimes means Common-Write, sometimes Priority-Write, and sometimes other variants. To avoid ambiguity, we will spell out the rule for write conflicts.
10.2.4 A Fast Algorithm for Finding Max

If we use the Common-Write or Priority-Write PRAMs, we can obtain an algorithm for finding the maximum of \( n \) numbers in less time than the binary fan-in method. This algorithm uses \( n(n-1)/2 \) processors. The strategy is to compare all pairs of keys in parallel, then communicate the results through the shared memory. We use an array \( \text{loser} \) that occupies memory cells \( M[n+1], \ldots, M[2n] \). Initially, all entries in this array are zero. If \( x_i \) "loses" a comparison, then \( \text{loser}[i] \) will be assigned the value 1.

Algorithm 10.3 Finding the Largest of \( n \) Keys

Input: \( n \) keys \( x_1, x_2, \ldots, x_n \), initially in memory cells \( M[1], M[2], \ldots, M[n] \) \((n>2)\).

Output: The largest key will be left in \( M[1] \).

Comment: For clarity, the processors will be numbered \( P_{i,j} \), for \( 1 \leq i < j \leq n \). Figure 10.5 illustrates the algorithm.

Step 1

\( P_{1,i} \) reads \( x_i \) (from \( M[i] \)).

Step 2

\( P_{i,j} \) reads \( x_j \) (from \( M[j] \)).

\( P_{i,j} \) compares \( x_i \) and \( x_j \).

Let \( k \) be the index of the smaller key.

(If the keys are equal, let \( k \) be the smaller index.)

Initial memory contents \((n = 4)\).

<table>
<thead>
<tr>
<th>Input</th>
<th>loser</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 7 3 6</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

\( P_{1,4} \) writes 1 in \( \text{loser}[k] \).

\( P_{1,2} \) writes 1 in \( \text{loser}[i] \).

At this point, every key other than the largest has lost a comparison.

Step 3

\( P_{1,i} \) reads \( \text{loser}[i] \) (and \( P_{1,n} \) reads \( \text{loser}[n] \)).

Any processor that reads a 0 writes \( x_i \) in \( M[1] \). \((P_{1,n} \) would write \( x_n \)).

\( P_{1,i} \) already has \( k \) in its local memory; \( P_{1,n} \) has \( x_n \).}

This algorithm does only three steps. However, the number of processors is in \( \Theta(n^2) \). If the number of processors is limited to \( n \), the largest key can be found in \( O(n \log n) \) time by an algorithm that uses Algorithm 10.3 on small groups of keys repeatedly. (See the notes and references at the end of the chapter.)

This algorithm shows that, if common writes are allowed, the binary fan-in scheme is not the fastest way to find the maximum key. In the matrix multiplication example in Section 10.2.2, we suggested using binary fan-in to add \( n \) integers in \( O(n \log n) \) time. The reader may now wonder if addition can also be done in constant time on Common-Write PRAMs. In Section 10.5.2 we will show that it cannot. Thus adding \( n \) integers is a harder problem than finding the maximum of \( n \) integers.

10.3 Merging and Sorting

10.3.1 Introduction

It is not difficult to find ways to speed up some of the sorting algorithms studied in Chapter 2 by doing some of the operations in parallel. The reader should be able to find \( \Theta(n) \)-time parallel implementations of, for example, Insertion Sort and Mergesort (to sort \( n \) keys). In this section we will present a parallel sorting algorithm based on Mergesort that does roughly \( (\log n)^2 / 2 \) PRAM steps using \( n \) processors.

The algorithm presented here gives a dramatic improvement over \( \Theta(n \log n) \) sequential sorting. For example, a list of 1000 keys can be sorted in 25 parallel steps; a sequential algorithm does about 10,000 comparisons. It is not the fastest parallel sorting algorithm known; parallel sorting can be done in \( O(n \log n) \) time. However, the algorithm we describe is very easy to understand. It uses few \( n \) processors, and the number of steps is a small multiple of \( (\log n)^2 \).

As usual, we will assume that we wish to sort into nondecreasing order.

10.3.2 Parallel Merging

As shown in Section 2.3.6, we can merge two sorted lists of \( n/2 \) keys each by doing at most \( n-1 \) comparisons. The merging algorithm we used there (Algorithm 2.6) seems essentially sequential; the two keys compared at one step depend on the result of the previous comparison. Here we use a different approach to merge in \(\log n \) parallel steps. Since we intend to use the merge algorithm in a Mergesort-style sorting algorithm in which we will always merge two lists of equal length, we will
write the merge algorithm for lists of equal length. It is an easy exercise to generalize the algorithm and its analysis to lists of different sizes.

Suppose that the two sorted lists are in the first \( n \) memory cells \( M[1], \ldots, M[n/2] \) and \( M[n/2+1], \ldots, M[n] \). For clarity, we will refer to the first list as \( X = (x_1, x_2, \ldots, x_{n/2}) \) and to the second as \( Y = (y_1, y_2, \ldots, y_{n/2}) \). Each of the \( n \) processors is assigned to one key and has the task of determining where that key belongs in the merged list. Assume for the moment that all the keys are distinct. A processor assigned to a key in \( X \), say \( x_i \), does a binary search in \( Y \) to determine how many keys of \( Y \) are smaller than \( x_i \). Specifically, the processor finds the smallest \( j \) such that \( x_i < y_j \). Then, since \( x_i \) is greater than exactly \( i-1 \) keys in \( X \) and \( j-1 \) keys in \( Y \), its proper position in the merged list is in \( M[i+j-1] \). (If the processor finds that \( x_i \) is greater than all the keys in \( Y \), then clearly, \( x_i \) belongs in position \( n/2+i \).) Similarly, processors assigned to keys in \( Y \) do a binary search in \( X \) to find the correct position for their key. After the binary searches are completed, each processor writes its key in the correct position. (See Fig. 10.6.)

If there are duplicate keys, the algorithm as described will not quite work. It will work if we simply require that a key from \( X \) that is equal to a key from \( Y \) be treated as if it were smaller than the key from \( Y \).

**Algorithm 10.4 Parallel Merging**

- **Input:** Two sorted lists of \( n/2 \) keys each, in the first \( n \) cells of memory.
- **Output:** The merged list, in the first \( n \) cells of memory.

**Comment:** Each processor \( P_i \) has a local variable \( x \) (if \( i \leq n/2 \)) or \( y \) (if \( i > n/2 \)) and other local variables for conducting its binary search. Each processor has a local variable \( \text{position} \) that will indicate where to write its key.

**Initialization:**

- \( P_i \) reads \( M[i] \) into \( x \) (if \( i \leq n/2 \)) or into \( y \) (if \( i > n/2 \)).
- \( P_i \) does initialization for its binary search.

**Binary search steps:**

processors \( P_i \), for \( 1 \leq i \leq n/2 \), do binary search in \( M[n/2+1], \ldots, M[n] \) to find the smallest \( j \) such that \( x < M[n/2+j] \), and assign \( i+j-1 \) to \( \text{position} \). If there is no such \( j \), \( P_i \) assigns \( n/2+i \) to \( \text{position} \).

- processors \( P_{n/2+i} \), for \( 1 \leq i \leq n/2 \), do binary search in \( M[1], \ldots, M[n/2] \) to find the smallest \( j \) such that \( y < M[j] \), and assign \( i+j-1 \) to \( \text{position} \). If there is no such \( j \), \( P_i \) assigns \( n/2+i \) to \( \text{position} \).

**Output step:**

Each \( P_i \) (for \( 1 \leq i \leq n \)) writes its key (\( x \) or \( y \)) in \( M[\text{position}] \).

**Theorem 10.2** The parallel merge algorithm does \( \lceil \log_2 n \rceil + 1 \) steps to merge two sorted lists, with \( n/2 \) keys each, using \( n \) processors.

**Proof.** The initialization is one PRAM step. The binary searches are all done in lists of \( n/2 \) keys, so they take \( \lceil \log_2 n \rceil + 1 = \lceil \log_2 n \rceil \) read/computation steps. Since the binary searches do not involve any writing to the shared memory, the output can be done in the last binary search step. Thus the total is \( \lceil \log_2 n \rceil + 1 \).

Note that since there are no write conflicts, the merge algorithm works on all the variations of the PRAM we have described.

### 10.3.3 Sorting

Suppose that we have a list of \( n \) keys to be sorted. Recall the strategy of Mergesort:

- Break the list into two halves.
- Sort the two halves (recursively).
- Merge the two sorted halves.

![Figure 10.6 Parallel merging.](image-url)
If we "unroll" the recursion, we see that the algorithm merges small sorted lists (one key each) first, then merges slightly larger lists (two keys each), then larger lists, and so on until finally it merges two lists of size (roughly) \( n/2 \). The recursive algorithm merges some larger lists before doing all the small lists (since it completely sorts the first half of the keys before even beginning on the second half). To write a systematic, iterative parallel algorithm, we merge all the pairs of lists of size 1 in the first pass (in parallel), then merge all the pairs of lists of size 2 in the next pass, and so on. Clearly we use \( \lceil \lg n \rceil \) merge passes. The assignment of processors to their tasks is very easy (though the indexing in the algorithm obscures it a little). Whenever two sublists occupying, say, \( M[i], \ldots, M[j] \) are being merged, processors \( P_i, \ldots, P_j \) do the merge using Algorithm 10.4. Figure 10.7 illustrates one pass.

**Algorithm 10.5 Sorting by Merging**

**Input**: A list of \( n \) keys in \( M[1], \ldots, M[n] \).

**Output**: The \( n \) keys sorted in nondecreasing order in \( M[1], \ldots, M[n] \).

**Comment**: The indexing in the algorithm is easier if the number of keys is a power of 2, so the first step will "pad" the input with large keys at the end. We still use only \( n \) processors.

\[
P_i \text{ writes } \left( \text{some large key} \right) \text{ in } M[n+i];
\]

\[
\text{for } i := 1 \text{ to } \lfloor \lg n \rfloor \text{ do}
\]

\[
\text{k := } 2^{i-1}; \text{ the size of the lists being merged}
\]

\[
P_{i} \ldots, P_{i+2^{i-1}-1} \text{ merge the two sorted lists of size } k \text{ beginning at } M[i];
\]

\[
\text{end } \{ \text{for } \}
\]

**Theorem 10.3** Algorithm 10.5 sorts \( n \) keys in \((\lceil \lg n \rceil + 1)(\lceil \lg n \rceil + 2)/2\) steps. Hence parallel sorting can be done in \( O((\lg n)^2) \) time with \( n \) processors.

**Proof**: At the \( r \)th merge pass, each pair of sublists being merged has a total of \( 2^r \) keys, so, by Theorem 10.2, the \( r \)th merge pass does \( r+1 \) steps. In total, all the merge passes do

\[
\sum_{r=1}^{\lfloor \lg n \rfloor} (r+1) = \frac{(\lceil \lg n \rceil + 1)(\lceil \lg n \rceil + 2)}{2} - 1
\]

steps, and there is one initialization step.

---

**10.4 A Parallel Connected Component Algorithm**

**10.4.1 Strategy and Techniques**

Let \( G = (V, E) \) be a graph (undirected) with \( |V| = n \) and \( |E| = m \). To keep the notation simple, let \( V = \{1, 2, \ldots, n\} \). In Chapter 4 we studied a sequential algorithm to find the connected components of \( G \). It used depth-first search and ran in \( \Theta(\max(n, m)) \) time. Although depth-first search may seem inherently sequential, there has been recent progress in developing fast parallel algorithms to construct depth first search trees. However, it is not necessary to do depth first search to find connected components. In this section we present a parallel algorithm that finds connected components in \( O(\lg n) \) time using max\( \{2n, 2m\} \) processors. The algorithm will have write conflicts, so, among the variations of the PRAM we have described, the Priority-Write PRAM is the only one that must be used here. However, a weaker model will do. In the Arbitrary-Write model, when several processors try to write in the same memory cell at the same time, an arbitrary one of them succeeds. An algorithm for this model must work correctly no matter which processor "wins" the write conflict. The connected component algorithm will work on the Arbitrary-Write model.

The connected component algorithm is more complicated than any of the other parallel algorithms we have looked at so far. We will give a high-level description of the algorithm, then show how its various parts can be implemented on a PRAM.

The algorithm begins with each vertex in a separate tree, then repeatedly combines trees that are in the same connected component and shortens the trees. Ultimately each connected component is converted to a (directed) tree of depth 1, with all the vertices pointing to the root. Such a tree is called a star. See Fig. 10.8 for an illustration. The trees are represented by an array parent, such that parent[v] is the parent of vertex v. (The parent of a root is the root itself.) Once the connected components have been converted to stars, we can determine whether two vertices are in the same component in constant time by comparing their parents.

To shorten trees, the algorithm uses a technique called shortcutting, which is also useful in other parallel graph algorithms. Shortcutting simply changes the parent of each vertex to make the vertex point directly to its grandparent. That is, the shortcutting operation on vertex \( v \) is

\[
\text{parent}[v] := \text{parent}[	ext{parent}[v]].
\]

Shortcutting is applied in parallel to all vertices. To see the speed with which this operation can cut down long paths, consider a simple chain of vertices as in Fig. 10.9(a), where parent[\( v \)] = \( v-1 \) (and parent[1] = 1). Figure 10.9(b) shows the parent pointers after the first and second applications of the shortcutting operation. If we started with \( n \) vertices in the chain, after \( \lceil \lg n \rceil \) applications of shortcutting, all vertices would have the same parent.