If we "unravel" 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 \( \lceil \log n \rceil \) 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.

\[
P_i \text{ writes } \alpha \text{ (some large key) in } M[n+1];
\]

\[
\text{for } i := 1 \text{ to } \lceil \log n \rceil \text{ do }
\]

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

\[
\left\lfloor \frac{\log n}{i} \right\rfloor \text{ (if list of size } k \text{ beginning at } M[j];
\]

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

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

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

\[
\sum_{i=1}^{\lceil \log n \rceil} (i+1) = \frac{(\lceil \log n \rceil + 1)(\lceil \log n \rceil + 2)}{2} - 1
\]

steps, and there is one initialization step.

Figure 10.7 Assignment of processors for one merge pass. Processors \( P_{2^k}, P_{2^k+1}, \ldots, P_{2^k+n-1} \) merge the \( k \)th pair of sublists of size \( k \).

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 \( O(\max(n, m)) \) 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 \( O(\log n) \) 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 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 \( \text{parent} \), such that \( \text{parent}(v) \) is the parent of vertex \( v \). (The \( \text{parent} \) 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 \( \text{parent} \) of each vertex to make the vertex point directly to its grandparent. That is, the shortcutting operation on vertex \( v \) is

\[
\text{parent}(v) := \text{parent}(\text{parent}(v)).
\]

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 \( \text{parent}(v) = v-1 \) (and \( \text{parent}(1) = 1 \)). Figure 10.9(b) shows the \( \text{parent} \) pointers after the first and second applications of the shortcutting operation. If we started with \( n \) vertices in the chain, after \( \lceil \log n \rceil \) applications of shortcutting, all vertices would have the same parent.
Shortcutting never joins two separate trees. We need another operation, called hooking, to connect trees. Hook($i, j$) means attach $i$'s root to the parent of $j$. Hook($i, j$) is applied only if $i$ is a root or the child of a root; thus the operation is $\text{parent}(\text{parent}(i)) := \text{parent}(j)$.

There are two versions of hooking in the algorithm. The first is

*Conditional hooking*: If the parent of $i$ is a root, $j$ is adjacent to $i$ (in $G$), and $j$'s parent is less than $i$'s parent, then Hook($i, j$).

The requirement that we hook to the smaller of the two parents helps avoid the introduction of cycles. Conditional hooking is illustrated in Fig. 10.10(b). The second hooking operation is

*Star hooking*: If $i$ belongs to a star, $j$ is adjacent to $i$ (in $G$), and $j$ is not in $i$'s star, then hook Hook($i, j$).

Star hooking is illustrated in Fig. 10.10(c).

Figure 10.9 The effect of shortcutting in a simple example.
At any one time, for a particular vertex $i$, there may be several vertices $j$ that satisfy the conditions for conditional hooking or star hooking, but only one value can be stored as the new parent of $i$’s root. In the parallel algorithm, different processors will be trying the different choices for $j$, and several processors may try to write in $\text{parent}[$parent$[j]]$ at the same time. Only one succeeds in writing, but the algorithm will work properly no matter which one succeeds. By running long enough, the algorithm eventually hooks together all trees that are part of one connected component.

### 10.4.2 The Algorithm

The algorithm begins with each vertex of $G$ in a separate tree. It repeatedly does hooking and shortcutting until the desired structure is achieved. We first give a high-level description. Unfortunately, Algorithm 10.6 does not quite work. Rather than make it more complicated, we will solve the problem by slightly modifying the input graph and the initialization step. After presenting the algorithm, we will explain the problem and the solution.

**Algorithm 10.6 Finding Connected Components of a Graph**

**Input:** A graph $G$ (undirected).

**Output:** A forest of directed trees of depth 1, represented by the array $\text{parent}$, indexed by the vertices. Each tree contains the vertices of one connected component.

**Comment:** An instruction specified for a vertex $v$ is performed in parallel for all vertices. The hooking steps are performed in parallel for all edges $ij$ in $G$ (and only for pairs $i$ and $j$ such that $ij$ is an edge). Each edge, say $xy$, is processed twice (in parallel), once with $x$ in the role of $i$, and once with $y$ in the role of $i$.

```plaintext
[ Initialization ]
parent[v] := v;
repeat
[ Conditional hooking ]
if $i$’s parent is a root and parent[j] = parent[i] then
    Hook(i, j)
end if
[ Star hooking ]
if $i$ is in a star and $j$ is not in $i$’s star then
    Hook(i, j)
end if
[ Shortcutting ]
if $v$ is not in a star then
    parent[v] := parent[parent[v]]
end if
until the shortcutting step did not produce any changes.
```

One of the facts used in the proof that the algorithm works correctly is that conditional hooking and star hooking do not produce new stars. But unfortunately, on the first pass through the loop, they do. A single vertex hooked to a star yields a star. Then, since stars can be created by conditional hooking, the star hooking step might hook two stars to each other in both directions, thus creating a cycle. These problems are eliminated by attaching a "dummy" vertex $v+n$ to each vertex $v$ in $G$ and initializing $\text{parent}[$parent$[j]]$ to $v$. Thus every tree always has at least two vertices. (Since the dummy vertices are not adjacent to any vertices in $G$, they affect nothing in the algorithm except whether or not a tree is a star.) Figure 10.11 illustrates the action of the algorithm. The proof that the algorithm works correctly includes many details expressed in several lemmas, which follow. The algorithm itself is not very hard to understand if a few examples are worked through, so the reader should examine Fig. 10.11 carefully before proceeding.

Assuming that the dummy vertices have been added and their parents are initialized as described in the previous paragraph, the following theorems show that the algorithm works.

**Theorem 10.4** At any time during execution of Algorithm 10.6, the structure defined by the $\text{parent}$ pointers is a forest. (In which each tree has $\geq 2$ nodes.)

**Theorem 10.5** When Algorithm 10.6 terminates, the forest defined by the $\text{parent}$ pointers consists only of stars, and the vertices in each star are exactly the vertices of a connected component of $G$ (and the "dummy" vertices associated with them).

The proofs of the theorems use several lemmas.

**Lemma 10.6** Conditional hooking and star hooking never create new stars.

**Proof.** When the root of a tree with at least two nodes is attached to another tree, the new tree will have depth at least 2.

**Lemma 10.7** The star hooking step never hooks a star onto another star.

**Proof.** Suppose it does. Then, at the beginning of the star hooking step there were two stars, $S_1$ and $S_2$, containing vertices $i$ and $j$, respectively, such that $ij$ is an edge in $G$. Since conditional hooking does not create stars (Lemma 10.6), $S_1$ and $S_2$ were stars at the beginning of the conditional hooking step. They satisfy the conditions for conditional hooking, so the one with the larger root would have been hooked in the conditional hooking step, and it would no longer be a star.

**Lemma 10.8** Once a vertex becomes a leaf, it will always be a leaf.

**Proof.** It is clear that shortcutting does not change a leaf to a nonleaf. The only way a leaf could become a nonleaf is by having something hooked to it, but the hooking steps always hook to the parent of $j$, for some $j$. The parent of a vertex
The graph

```
1  2  4  3
5  6  8  7
9 10
```

The initial forest. (Each “dummy” vertex \( v + n \) is denoted by \( v' \).)

```
1' 2' 6' 8'
10' 11'
```

After conditional hooking. (The processors that succeeded in writing are \( P_{21}, P_{42}, P_{68}, P_{10}, \) and \( P_{11,10'} \).)

```
1  2  4  3
5  6  8  7
9 10
```

After star hooking. (\( P_{68} \) wrote.)

```
1  3  4  2'
5  6  8  7
11 10' 9'
```

After star hooking. (\( P_{68} \) wrote.)

```
1  3  4  2
5  6  8  7
11 10' 9'
```

After shortcutting.

```
1  3  4  2
5  6  8  7
11 10' 9'
```

There will be no hooking on the next iteration. Shortcutting will turn the first tree into a star.

On the last iteration there will be no changes and the algorithm will stop.

Figure 10.11 Illustration of the connected component algorithm.
Lemma 10.9 After each shortcut step and each conditional hooking step, if \( i < \text{parent}[i] \), then \( i \) is a leaf.

Proof. The condition holds (vacuously) after the initialization because no vertex is less than its parent. Since a leaf never becomes a nonleaf (Lemma 10.8), no step in the algorithm can "undo" the required property for any vertex \( i < \text{parent}[i] \) unless \( i \)'s parent changes. So we must check what happens when a vertex is given a new parent. Conditional hooking hooks \( \text{parent}[i] \) to \( \text{parent}[j] \) only if \( \text{parent}[j] > \text{parent}[i] \), so the lemma is not violated by this change. Star hooking may hook a star with root \( x \) onto a vertex \( y \) where \( x < y \). (So the lemma makes no claim for the star hooking step.) Now, at the beginning of the shortcutting step, consider any chain in some tree of the form \( \ldots \rightarrow x \rightarrow y \rightarrow z \rightarrow \ldots \). If \( x < y \), then either \( x \) is already a leaf or it was the root of a star that just got hooked to \( y \), and it will be a leaf when the shortcutting is done on its children. If \( x > y \), then either \( y \) is the root of a star that was just hooked to \( z \), so \( x \) is a leaf, or \( y > x \), so \( x > z \). In all cases, after shortcutting produces \( x \rightarrow y \), either \( x \) is a leaf or \( x > z \). □

Proof of Theorem 10.4 The algorithm starts with trees; we have to show that no step introduces a cycle. It is clear that shortcutting cannot introduce a cycle. Since star hooking always hooks a star to a nonstar (Lemma 10.7), it does not introduce a cycle. Suppose that a cycle were formed in conditional hooking. Let \( i \) be the largest vertex in the cycle, and let \( j \) be the vertex in the cycle that points to \( i \). Then \( i < \text{parent}[i] = j \). But by Lemma 10.9, \( i \) must be a leaf, so it cannot be in a cycle. □

Lemma 10.10 Any star that exists at the end of the star hooking step must be an entire connected component.

Proof. By Lemma 10.7, the star was a star at the beginning of the star hooking step. But if any vertex in the star were adjacent (in \( G \)) to a vertex in any other tree, the star hooking step would have hooked the star to another tree, and it would no longer be a star. □

Proof of Theorem 10.5 Since the vertices of \( G \) start out in disjoint trees, and two trees are hooked only if they contain vertices \( i \) and \( j \) that are adjacent, all the vertices in any one tree at any time are in the same connected component. The algorithm stops when shortcutting produces no changes. This can happen only when there are no vertices of distance \( 2 \) from their roots; i.e., when all vertices are in stars at the end of the star hooking step. By Lemma 10.10, each such star is an entire component. □

10.4.3 PRAM Implementation of the Algorithm

Now we consider how the instructions in the algorithm are assigned to processors, as well as how many PRAM steps are needed to carry out each instruction.

Some processors have two "names." When we must do some operation for each vertex (say shortcutting), we will use the first \( 2n \) processors, referring to them as \( P_0 \). When we must do an operation for each edge, we use the first \( 2m \) processors, referring to them by the names \( P_1 \). Since operations on vertices and operations on edges are done in different instructions, each processor will be doing only one thing at a time.

The PRAM algorithm assumes that the input is in the form of a list of edges in the graph \( G \). Each edge appears twice in the list: that is, if \( ij \) is an edge, the pairs \( (i, j) \) and \( (j, i) \) are in the input list. Each of the \( 2m \) processors reads a (distinct) memory cell containing an edge, and from then on considers itself to be the processor for that particular (oriented) edge; i.e., it "knows" that its alternative name is \( P_0 \) if it read \( (i, j) \).

The form of the input is not critical to the speed of the algorithm. If the input were provided as an adjacency matrix, we would have \( n^2 \) processors read the matrix entries in the first step. Those that read a zero would just quit without doing any more work for edges.

We present the algorithm again with more implementation detail. The important observation to make here is that each step of the algorithm can be implemented in a constant number of PRAM steps.

Algorithm 10.7 Finding the Connected Components of a Graph

Input: A list of edges in the graph, each edge listed in both orientations: \( n \), the number of vertices.

Output: A forest of directed trees of depth \( 1 \), represented by the array \( \text{parent} \), indexed by the vertices. Each tree contains the vertices of one connected component.

Comment: A Boolean array \( \text{star} \) is used to tell if a vertex is in a star; \( \text{star}[v] \) is true if and only if \( v \) is in a star. The instructions for computing \( \text{star} \) are shown after the main algorithm. The shared Boolean variable \( \text{noChange} \) tells whether or not the shortcutting step has made any changes at each iteration of the loop.

```plaintext
{ Initialization }
foreach edge processor reads an edge;
each processor reads \( n \); 
\( P_0 \) and \( P_{\infty} \) write \( v \) in \( \text{parent}[v] \) and \( \text{parent}[\infty] \), respectively, where \( 1 \leq v \leq n \); 
repeat
{ Conditional hooking }
\( P_0 \) reads \( \text{parent}[i] \), \( \text{parent}[\text{parent}[i]] \), and \( \text{parent}[j] \); 
if \( \text{parent}[i] = \text{parent}[\text{parent}[i]] \) \( \text{i.e.} \), if \( \text{parent}[i] \) is a root \( \text{and} \text{parent}[i] > \text{parent}[j] \) then 
write \( \text{parent}[j] \) in \( \text{parent}[\text{parent}[i]] \) 
end { if }; 
```


{ Star hooking }

\( P_v \) computes and writes \( \text{star}[v] \) as described below;

\( P_v \) reads \( \text{parent}[i], \text{parent}[j], \) and \( \text{star}[i]; \)

if \( \text{star}[i] \) and \( \text{parent}[i] \neq \text{parent}[j] \) [i.e., \( j \) is not in \( i \)'s star] then

write \( \text{parent}[j] \) in \( \text{parent}[	ext{parent}[i]] \)

end \{ if \};

{ Shortcasing }

\( P_v \) writes \( \text{true} \) in \( \text{noChange} \);

\( P_v \) reads \( \text{parent}[v] \) and \( \text{parent}[\text{parent}[v]] \);

if \( \text{parent}[\text{parent}[v]] \neq \text{parent}[v] \) then

\( P_v \) writes \( \text{parent}[\text{parent}[v]] \) in \( \text{parent}[v] \);

\( P_v \) writes \( \text{false} \) in \( \text{noChange} \)

end \{ if \}

until \( \text{noChange} \) { All processors read \( \text{noChange} \) to determine whether

they should stop. }

A vertex is not in a star if and only if one of the following conditions holds:

1. Its parent is not its grandparent,
2. It is the grandparent, but not the parent, of some other vertex, or
3. Its parent has a nontrivial grandchild.

Figure 10.12 illustrates all three cases and the computation of \( \text{star} \). The computation is described in the following algorithm.

**Algorithm 10.8** Determining if a Vertex is in a Star

*Comment:* These steps are carried out by \( P_v \) (for \( 1 \leq v \leq 2n \)).

write \( \text{true} \) in \( \text{star}[v] \);

read \( \text{parent}[v] \) and \( \text{parent}[	ext{parent}[v]] \);

if \( \text{parent}[v] \neq \text{parent}[	ext{parent}[v]] \) then

write \( \text{false} \) in \( \text{star}[v] \);

write \( \text{false} \) in \( \text{star}[	ext{parent}[	ext{parent}[v]]] \)

end \{ if \};

read \( \text{star}[	ext{parent}[v]] \);

if not \( \text{star}[	ext{parent}[v]] \) then

write \( \text{false} \) in \( \text{star}[v] \).

end \{ if \}

The computation of \( \text{star}[v] \) and each of the steps in Algorithm 10.7's main loop can be carried out in constant time by a PRAM, so the number of iterations of the loop determines the order of the running time. All that remains is to show that the number of iterations is in \( O(\log n) \).

---

(a) How to tell each vertex is not in a star.

(b) The computation.
Lemma 10.11 Let \( d \) be the depth of a nonstar tree before the shortcutting step. After shortcutting, its depth is at most \( 2d/3 \).

Proof. Actually, the depth is cut roughly in half after each shortcutting step. It is easy to show by induction that if \( d \) is even, the depth after shortcutting will be exactly \( d/2 \), and that if \( d \) is odd, the depth after shortcutting will be \( (d+1)/2 \). The worst case is when \( d = 3 \). The new depth will be 2, which is \( 2d/3 \).

For any connected component \( C \), let \( d_C(t) \) be the sum of the depths of all the trees in \( C \) at the end of the \( t \)th iteration of the loop (for \( t \geq 0 \)).

Lemma 10.12 For any connected component whose vertices do not form a star at the beginning of the \( t \)th iteration of the loop (for \( t \geq 1 \)), \( d_C(t) \leq 2d_C(t-1)/3 \).

Proof. Consider what happens to the trees of \( C \) during the \( t \)th iteration. Since a tree is never hooked to a leaf, the depth of a tree that results from hooking is at most the sum of the depths of the two trees that were hooked. After shortcutting, each tree is at most two-thirds as deep as it was before, so the sum of the depths of \( C \) is at most two-thirds what it was before.

Theorem 10.13 Algorithm 10.6 runs in \( O(\lg n) \) time in the worst case.

Proof. From Lemma 10.12, for any connected component \( C \), we have

\[
d_C(t-1) \geq \frac{3}{2} d_C(t).
\]

Iterating this recurrence gives

\[
d_C(0) \geq \left(\frac{3}{2}\right)^T d_C(T).
\]

At the end of the 0th iteration (i.e., after the initialization), there are \( n \) trees, each of depth 1, so \( d_C(0) \leq n \). Let \( T \) be the number of the first iteration after which the vertices of \( C \) are in one star. Then \( d_C(T) = 1 \). So

\[
n \geq d_C(0) \geq \left(\frac{3}{2}\right)^T d_C(T) = \left(\frac{3}{2}\right)^T.
\]

i.e.,

\[
n \geq \left(\frac{3}{2}\right)^T.
\]

So

\[
T \leq \lg_{3/2} n.
\]

Since \( T \) is an integer, we conclude that after \( \lfloor \lg_{3/2} n \rfloor \) iterations, each component is a star. The algorithm does just one more iteration, in which nothing changes, so the total number of iterations, and the running time of the algorithm, is in \( O(\lg n) \).

10.5 Lower Bounds

In this section we present a lower-bound argument for parallel computation. Most of this section is about Boolean functions, that is, computations in which the input consists of \( n \) bits. However, we will also get a lower bound for addition of integers. Before we attack the lower bound, let us first consider how to compute Boolean functions quickly.

10.5.1 Computing Boolean Functions

Let \( f \) be a function from \( \{0,1\}^n \) into \( \{0,1\} \). The Boolean or and and functions are examples. We will show that any such function can be computed in \( O(\lg n) \) time (on any of the versions of the PRAM defined in Section 10.2.3). The key idea is to use the binary fan-in scheme described in Section 10.2.1 to encode the bits into one integer. Then any processor can read the integer and compute the function value. (If it seems strange that all the actual computation of the function will be done by one processor in one step, and all the rest of the time is used to communicate the \( n \) input bits to that processor, recall that models for parallel computation tend to focus on the communication between processors.)

We define the function \( \text{Encode} \) on \( n \) bits \( b_1, b_2, \ldots, b_n \) as follows:

\[
\text{Encode}(b_1, b_2, \ldots, b_n) = b_n b_{n-1} \cdots b_2 b_1 = \sum_{i=1}^{n} b_i 2^{i-1}
\]

\( \text{Encode} \) is computed by the following PRAM algorithm.

Algorithm 10.9 Encoding \( n \) Bits as an Integer

Input: Bits \( b_1, b_2, \ldots, b_n \), initially in the first \( n \) memory cells.
Output: The integer \( b_n b_{n-1} \cdots b_2 b_1 \) in \( M[1] \).
Comment: Each processor uses local variables \( x \) (to accumulate part of the integer), \( \text{temp} \), and \( \text{incr} \) (for determining which processor's output to read next and how much the value read needs to be "shifted"). As usual, \( i \) is the processor number. (Some processors will read from memory cells with indexes larger than \( n \); as usual, we assume that they contain zeros.)

\[
\text{Read } M[i] \text{ into } x;
\]
\[
\text{incr } := 1;
\]
\[
\text{for } \text{step } := 1 \text{ to } \lfloor \lg n \rfloor \text{ do}
\]
\[
\text{read } M[i+\text{incr}] \text{ into } \text{temp};
\]
\[
x := x + 2^{\text{temp}} \cdot \text{temp} \cdot \text{temp};
\]
\[
\text{incr } := 2 \cdot \text{incr};
\]
\[
\text{write } x \text{ into } M[i];
\]
end (for i);
This algorithm clearly takes \( [\lg n] + 1 \) time steps. The proof that the algorithm actually works, i.e., that after the loop terminates, the contents of \( M[1] \) will be the integer \( b_n b_{n-1} \cdots b_2 b_1 \) is similar to the proof of correctness for Algorithm 10.1. We leave the details as an exercise.

**Algorithm 10.10** Computing a Boolean Function on \( n \) Bits

*Input:* Bits \( b_1, b_2, \ldots, b_n \), initially in the first \( n \) memory cells.

*Output:* \( f(b_1, b_2, \ldots, b_n) \) in \( M[1] \), where \( f \) is some specific Boolean function.

\[
	ext{Encode (using Algorithm 10.9):}
\]

1. \( P_1 \) reads \( M[1] \).
2. \( P_1 \) computes \( f(b_1, b_2, \ldots, b_n) \)
3. (by decoding the integer from \( M[1] \) into its component bits if necessary)
4. \( P_1 \) writes the result in \( M[1] \).

We summarize with the following theorem.

**Theorem 10.14** A PRAM can compute any Boolean function in \( [\lg n] + 2 \) steps using \( n \) processors.

### 10.5.2 A Lower Bound for Computing a Function on \( n \) Bits

The *Encode* function defined in Section 10.5.1 is a hardest-to-compute function on \( n \) bits. That is, no other function can require time of higher order because, once all the input bits have been encoded into one integer, any other function can be computed by one processor in one step after reading the encoding. In this section we will derive a lower bound on the number of steps required to compute *Encode*.

Several of our earlier lower-bound arguments used decision trees. The idea underlying those arguments was that there had to be enough branching in the tree, enough decisions, to distinguish inputs that should generate different outputs. A similar idea is used here. A PRAM for *Encode* must do enough steps to distinguish between all possible \( 2^n \) inputs because each input is encoded as a distinct integer; i.e., each input generates a distinct output. Since the output is written in \( M[1] \), a PRAM must do enough steps so that any of \( 2^n \) different values could be written in \( M[1] \). Of course, for one particular input, a PRAM always writes exactly one specific value in \( M[1] \) at any step. Here we consider the space of all inputs; we count all the different values a PRAM could write for all possible inputs.

The value in a memory cell depends on what the processors write (or do not write). What a processor writes depends on the "state" of the processor at the beginning of a step and what it reads from memory on that step. Think of the state of a processor as everything internal to the processor that affects its action (e.g., the values of all the variables in its local memory and its own index). The proof of the lower bound counts how many different states processors can be in, and how many values could be written in memory cells, after each step.

**Theorem 10.15** (P. Beame) Any Priority-Write PRAM that computes *Encode* must do at least \( \lg n + 1 - \lg \lg (4p) \) steps (where \( p \) is the number of processors).

**Proof.** We want to find answers to the following two questions:

1. How many different values can be in any particular memory cell \( M[i] \) after \( t \) steps?
2. How many different states can any particular processor \( P_i \) be in after \( t \) steps?

We define two sequences of numbers,

\[
\begin{align*}
r_0 &= 1 \\
r_t &+ 1 = 2r_t \\
s_0 &= 2 \\
s_t &= pr_{t+1} + s_t 
\end{align*}
\]

(10.1)

where \( p \) is the number of processors. Here are the first few values in the sequences.

<table>
<thead>
<tr>
<th>( t )</th>
<th>( r_t )</th>
<th>( s_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2p + 2</td>
</tr>
<tr>
<td>2</td>
<td>2(2p + 2)</td>
<td>2p(2p + 2) + 2p + 2</td>
</tr>
</tbody>
</table>

**Lemma 10.16** The number of distinct states a processor may be in after \( t \) steps (considering all possible inputs) is at most \( r_t \). The number of distinct values that could be in a memory cell after \( t \) steps (considering all possible inputs) is at most \( s_t \).

**Proof.** We prove the lemma by induction on \( t \). For \( t = 0 \) (that is, before the PRAM has executed any instructions), each processor can be in only one state, its initial state. Each memory cell contains one of two possible values: 0 or 1. Since \( r_0 = 1 \) and \( s_0 = 2 \), the basis for the induction is established.

Now, for \( t \geq 0 \), assume that after \( t \) steps a processor can be in any one of at most \( r_t \) steps, and that a memory cell can have one of at most \( s_t \) values. The new state of a processor after step \( t+1 \) depends on the old state (the state after step \( t \)) and the value read from memory by that processor on step \( t+1 \). Thus the number of possible states after step \( t+1 \) is at most \( r_t s_t \), which is \( r_{t+1} \). On step \( t+1 \) any processor can write in a particular memory cell, and a processor can write a different value for each state it could be in. Thus \( pr_{t+1} \) possible values, but it is also possible that np processor writes in the cell on this step, so any of the \( s_t \) values that could have been there before may still be in the cell after step \( t+1 \). Thus the total number of possible values in a memory cell at the end of step \( t+1 \) is \( pr_{t+1} + s_t \), which is \( s_{t+1} \).
Lemma 10.17  For \( i \geq 1 \), \( s_{i+1} \leq s_i^2 \).

Proof. Using Eq. 10.1,
\[
s_{i+1} = pr_i + s_i = pr_i \cdot s_i + s_i = pr_i (pr_i + 1) \leq s_i (pr_i + g \cdot s_i - 1) = s_i^2.
\]

Lemma 10.18  For \( i \geq 1 \), \( s_i \leq (4p)^{2^{i-1}} \).

Proof. For \( i = 1 \), \( s_1 = pr_0 \cdot s_0 + s_0 = p \times 1 \times 2 + 2 = 2p + 2 \leq 4p \) since \( p \geq 1 \). For \( i \geq 1 \),
\[
s_{i+1} \leq s_i^2 \leq (4p)^{2^{i-1}}.
\]

Now, continuing with the proof of the theorem, we observe that if any PRAM algorithm computes \( \text{Encode} \) in \( T \) steps, then \( s_T \geq 2^n \), because \( 2^n \) distinct outputs could appear in \( M[i] \) when the algorithm terminates. So
\[
2^n \leq s_T \leq (4p)^{2^{T-1}}.
\]
and, taking logs,
\[
n \leq 2^{T-1} \log(4p).
\]

Taking logs again,
\[
\log(n) \leq T - 1 + \log(4p).
\]
Therefore
\[
T \geq \log(n) + 1 - \log(4p).
\]

Corollary 10.19  Any CREW PRAM or Common-Write PRAM that computes \( \text{Encode} \) must do at least \( \log(n) + 1 - \log(4p) \) steps. Hence if \( p \) is bounded by any polynomial in \( n \), the parallel complexity of \( \text{Encode} \) is in \( \Theta(\log(n)) \).

Proof. Any program for either of these models is also a valid program for the Priority-Write model, so the lower bound in the theorem applies.

If \( p \) is bounded by a polynomial in \( n \), then \( \log(4p) \) is in \( \Theta(\log(n)) \), and \( \log(n) - 1 - \log(4p) \) is in \( \Theta(\log(n)) \).

And, finally, here is our lower bound on parallel addition of integers.

Corollary 10.20  Any PRAM algorithm (CREW, Common-Write, or Priority-Write) to compute the sum of \( n \) integers with at most \( n \) bits does at least \( \log(n) - \log(4p) \) steps. (Hence if \( p \) is bounded by any polynomial in \( n \), the parallel complexity of addition of \( n \) integers is in \( \Theta(\log(n)) \).)

Proof. Let \( \text{Add} \) be any PRAM algorithm to compute the sum of \( n \) integers. With one more step, we can compute \( \text{Encode} \) as follows:

Read \( b_i \) from \( M[i] \);
Compute \( b_i / 2^{i-1} \);
Write \( b_i / 2^{i-1} \) into \( M[i] \);
Add.

If \( \text{Add} \) were faster than the corollary claims, we would have a faster way to compute \( \text{Encode} \), contradicting Theorem 10.15.

Exercises

Section 10.2: Some PRAM Algorithms and the Handling of Write Conflicts

10.1. For Algorithm 10.1, what does \( P_i \) compute in the first three iterations of the loop?

10.2. Write a CREW PRAM algorithm to compute the sum of \( n \) integers in \( O(\log(n)) \) time.

10.3. Show that the Boolean and of \( n \) bits can be computed in constant time by a Common-Write or Priority-Write PRAM.

10.4. Show that the Boolean matrix product of two \( n \times n \) Boolean matrices can be computed in constant time by a Priority-Write or Common-Write PRAM. (The number of processors should be bounded by a polynomial in \( n \).)

*10.5. Using the lower bound given in the text for computing the or of \( n \) bits on a CREW PRAM, show that computing the maximum of \( n \) integers requires \( \Omega(\log(n)) \) time on a CREW PRAM.

10.6. Using the lower bound given in the text for computing the or of \( n \) bits on a CREW PRAM, show that Boolean matrix multiplication requires \( \Omega(\log(n)) \) time on a CREW PRAM.

10.7. Modify Algorithm 10.1 so that it outputs an index of the largest key instead of the largest key itself. (The modified algorithm should not have write conflicts, and it should still do \( \Theta(\log(n)) \) steps.)

10.8. Would Algorithm 10.3 work correctly if we did not specify how \( k \) should be chosen when a processor compares two equal keys? Justify your answer with an argument or a counterexample.

10.9. Modify Algorithm 10.3 so that it outputs an index of the largest key instead of the largest key itself. (The modified algorithm should do only a constant number of steps.)

Section 10.3: Merging and Sorting

10.10. Give a PRAM implementation of Insertion Sort for \( n \) keys that runs in \( O(n) \) time steps. (You may use any PRAM variation, but specify which one.)

10.11. Modify the parallel merge algorithm (Algorithm 10.4) to merge two sorted lists of \( n \) and \( m \) keys, respectively. How many steps does the revised algorithm do?

*10.12. Describe an algorithm to sort \( n \) keys in \( O(\log(n)) \) steps on a CREW PRAM. The number of processors may be greater than \( n \), but it should be bounded by a polynomial in \( n \).
(Hint: Exercise 10.2 implies that you can compute the sum of $n$ bits in $O(\lg n)$ time. Use that along with the trick used in the constant-time max algorithm (Algorithm 10.3).

*10.13. Give an algorithm to merge two sorted lists of $n$ keys each in constant time on a CREW PRAM. The number of processors may be greater than $n$, but it should be bounded by a polynomial in $n$.

Section 10.4: A Parallel Connected Component Algorithm

10.14. The connected component algorithm does not tell us the number of connected components in the input graph. Show that the number of connected components can be determined in $O(\lg n)$ time.

10.15. In the example in Fig. 10.11, when more than one processor tried to write in one memory cell at the same time, we made an arbitrary choice as to which one succeeded. Redo the example making a different valid choice at each step at which there was a write conflict.

Section 10.5: Lower Bounds

10.16. Prove that Algorithm 10.9 computes $Encode$. Hint: Show by induction that after the $r$th iteration, $M[j]$ contains

$$b_r + 2b_{r-1} + \ldots + 2^{r-1}b_1 + \ldots + 1.$$

Additional Problems

10.17. The $n$-bit unary representation of an integer $k$ is a sequence of $k$ ones followed by $n-k$ zeroes. For each of the following problems you should use at most $n$ processors.

a) Show that a PRAM (CREW, Common-Write, or Priority-Write) can read an integer $k$ between 0 and $n$ from $M[1]$ and convert it to its unary representation in one step. (The output is to go in cells $M[1], \ldots, M[n]$.)

b) Show that a Priority-Write PRAM with $n$ processors can read the unary representation of an integer $k$ from $M[1], \ldots, M[n]$ and write $k$ in $M[1]$ in one step.

c) Show that a CREW PRAM can solve the problem in (b) in two steps.

d) Show that, in a constant number of steps, a PRAM (any variation) can convert the $n$-bit unary representation of an integer (given in cells $M[1], \ldots, M[n]$) to the $\lfloor \lg n \rfloor + 1$ bit binary representation of the same integer.

*10.18. Suppose that you have a sorted list of $n$ keys in memory and $p$ processors, where $p$ is small compared to $n$. Give a CREW PRAM algorithm for searching the list for a key $x$. How many steps does your algorithm do? (Hint: use a generalization of binary search. You may find part (c) of the previous exercise helpful. Your search algorithm should do $O((\lg n)(\lg p + 1))$ steps in the worst case.)

10.19. Go through the earlier chapters of this book and pick out any algorithm that has a natural parallel version. Write the parallel algorithm and tell how many processors and time steps it uses. (Choose an algorithm for which the running time of the parallel version is of lower order than the sequential version.)

10.20. Make a list of the problems covered in this chapter that are in the class $NC$ (defined in Section 10.1). Are any algorithms in this chapter not $NC$ algorithms?

Notes and References

The PRAM model was presented (in slightly different form) in Fortune and Wyllie (1978) and Goldschlager (1978). There are now many dozens of papers using this model.

Section 10.3 is from Shiloach and Vishkin (1981). Their paper gives algorithms for sorting (and several other problems) in which the number of processors is smaller than the number of keys. It also contains the $O(\lg n)$ algorithm for finding the largest of $n$ keys mentioned in Section 10.2.4 and a solution to Exercise 10.17(c).

Section 10.4 is based on Awerbuch and Shiloach (1983) and Shiloach and Vishkin (1982). Awerbuch and Shiloach also contains a parallel algorithm for finding minimum spanning forests. Section 10.5 is from Beame (1986).

For those who wish to read further, the Bibliography includes a sampling of other papers: Cook, Dwork and Reischuk (1986), on upper and lower bounds for several problems considered in Sections 10.2 and 10.3; Chandra, Stockmeyer and Vishkin (1984), on a number of interesting problems and the relations between their parallel complexity; Kruskal (1983) and Snir (1985), on parallel searching (including the solution to Exercise 10.13); Batcher (1969), on sorting networks; Landau and Vishkin (1986), on approximate string matching; and Tarjan and Vishkin (1985), on biconnected components of graphs. Akl (1985) is a book on parallel sorting (using various models of parallel computation); Richards (1986), a bibliography on parallel sorting, contains nearly 400 entries. Quinn and Deo (1984) is a survey of parallel graph algorithms. Quinn (1987) is a text on parallel algorithms.