COMP 633: Parallel Computing
PRAM Algorithms

Siddhartha Chatterjee
Jan Prins

Fall 2009

Contents

1 The PRAM model of computation

In the first unit of the course, we will study parallel algorithms in the context of a model of parallel computation called the Parallel Random Access Machine (PRAM). As the name suggests, the PRAM model is an extension of the familiar RAM model of sequential computation that is used in algorithm analysis. We will use the *synchronous* PRAM which is defined as follows.

1. There are p processors connected to a single shared memory.

2. Each processor has a unique index $1 \leqslant i \leqslant p$ called the *processor id*.

3. A single program is executed in single-instruction stream, multiple-data stream (SIMD) fashion. Each instruction in the instruction stream is carried out by all processors simultaneously and requires unit time, regardless of the number of processors.

4. Each processor has a flag that controls whether it is *active* in the execution of an instruction. Inactive processors do not participate in the execution of instructions, except for instructions that reset the flag.

The processor id can be used to distinguish processor behavior while executing the common program. For example, each processor can use its processor id to form a distinct address in the shared memory from which to read a value. A sequence of instructions can be conditionally executed by a subset of processors. The condition is evaluated by all processors and is used to set the processor active flags. Only active processors carry out the instructions that follow. At the end of the sequence the flags are reset so that execution is resumed by all processors.

The operation of a synchronous PRAM can result in simultaneous access by multiple processors to the same location in shared memory. There are several variants of our PRAM model, depending on whether such simultaneous access is permitted (concurrent access) or prohibited (exclusive access). As accesses can be *reads* or *writes*, we have the following four possibilities:

1. **Exclusive Read Exclusive Write (EREW)**: This PRAM variant does not allow any kind of simultaneous access to a single memory location. All correct programs for such a PRAM must insure that no two processors access a common memory location in the same time unit.

2. **Concurrent Read Exclusive Write (CREW)**: This PRAM variant allows concurrent reads but not concurrent writes to shared memory locations. All processors concurrently reading a common memory location obtain the same value.

3. **Exclusive Read Concurrent Write (ERCW)**: This PRAM variant allows concurrent writes but not concurrent reads to shared memory locations. This variant is generally not considered independently, but is subsumed within the next variant.

4. **Concurrent Read Concurrent Write (CRCW)**: This PRAM variant allows both concurrent reads and concurrent writes to shared memory locations. There are several sub-variants within this variant, depending on how concurrent writes are resolved.

   (a) **Common CRCW**: This model allows concurrent writes if and only if all the processors are attempting to write the same value (which becomes the value stored).

   (b) **Arbitrary CRCW**: In this model, a value arbitrarily chosen from the values written to the common memory location is stored.

   (c) **Priority CRCW**: In this model, the value written by the processor with the minimum processor id writing to the common memory location is stored.

   (d) **Combining CRCW**: In this model, the value stored is a combination (usually by an associative and commutative operator such as $+$ or **max**) of the values written.

The different models represent different constraints in algorithm design. They differ not in expressive power but in complexity-theoretic terms. We will consider this issue further in Section 5.

We study PRAM algorithms for several reasons.

1. There is a well-developed body of literature on the design of PRAM algorithms and the complexity of such algorithms.

2. The PRAM model focuses exclusively on concurrency issues and explicitly ignores issues of synchronization and communication. It thus serves as a baseline model of concurrency. In other words, if you can't get a good parallel algorithm on the PRAM model, you're not going to get a good parallel algorithm in the real world.

3. The model is explicit: we have to specify the operations performed at each step, and the scheduling of operations on processors.

4. It is a robust design paradigm. Many algorithms for other models (such as the network model) can be derived directly from PRAM algorithms.

Digression 1 In the following, we will use the words *vector* and *matrix* to denote the usual linear-algebraic entities, and the word *sequence* for a linear list. We reserve the word *array* for the familiar concrete data structure that is used to implement all of these other kinds of abstract entities. Arrays can in general be multidimensional. The triplet notation $\ell : h : s$ with $\ell \leqslant h$ and $s > 0$ denotes the set $\{\ell + is \,|\, 0 \leqslant i \leqslant \lfloor \frac{h-\ell}{s} \rfloor\}$. If $s = 1$, we drop it from the notation. Thus, $\ell : h \equiv \ell : h : 1$. □

Example 1 (Vector Sum) As our first example of a PRAM algorithm, let us compute $z = v + w$ where v, w, and z are vectors of length n stored as 1-dimensional arrays in shared memory. We describe a PRAM algorithm by giving the single program executed by all processors. The processor id will generally appear as a program variable i that takes on a different value $1 \leqslant i \leqslant p$ at each processor. So if $n = p$, the vector sum program simply consists of the statement $z[i] \leftarrow v[i] + w[i]$.

To permit the problem size n and the number of processors p to vary independently, we generalize the program as shown in Algorithm 1. Line 4 performs p simultaneous additions and writes p consecutive elements of the result into z. The **for** loop is used to apply this basic parallel step to z in successive sections of size p. The conditional in line 3 ensures that the final parallel step performs the correct number of operations, in case p does not divide n evenly.

Algorithm 1 (Vector sum on a PRAM)
Input: Vectors $v[1..n]$ and $w[1..n]$ in shared memory.
Output: Vector $z[1..n]$ in shared memory.

```
1  local integer h
2  for h = 1 to ⌈n/p⌉ do
3     if (h − 1)p + i ⩽ n then
4        z[(h − 1)p + i] ← v[(h − 1)p + i] + w[(h − 1)p + i]
5     endif
6  enddo
```

To simplify the presentation of PRAM programs, we assume that each processor has some local memory or, equivalently, some unique portion of the shared memory, in which processor-private variables such as h and i may be kept. We will typically assume that parameters such as n and p are are in this memory as well. Under this assumption, all references to shared memory in Algorithm 1 are exclusive, and the algorithm requires only an EREW PRAM. Algorithm 1 requires on the order of $\lceil n/p \rceil$ steps to execute, so the concurrent running time $T_C(n, p) = O(n/p)$. □

2 The Work-Time paradigm

The barebones PRAM model is low-level and cumbersome, and writing anything other than trivial algorithms in this model is a nightmare. We will therefore switch to an equivalent but higher-level abstraction called the Work-Time (WT) paradigm to be independent of these details. After discussing this framework, we will present Brent's Theorem, which will allow us to convert a WT algorithm into a PRAM algorithm.

In the PRAM model, algorithms are presented as a program to be executed by all the processors; in each step an operation is performed simultaneously by all active processors. In the WT model, each step may contain an arbitrary number of operations to be performed simultaneously, and the scheduling of these operations over processors is left implicit. In our algorithmic notation, we will use the **forall** construct to denote such concurrent operations, and we drop explicit mention of the processor id and p, the number of processors. In fact the **forall** construct is the only construct that distinguishes a WT algorithm from a sequential algorithm.

We associate two complexity measures parameterized in the problem size n with the WT description of an algorithm. The *work complexity* of the algorithm, denoted $W(n)$, is the total number of operations the algorithm performs.

In each step, one or more operations are performed simultaneously. The *step complexity* of the algorithm, denoted $S(n)$, is the number of steps that the algorithm executes. If $W_i(n)$ is the number of simultaneous operations at parallel step i, then

$$W(n) = \sum_{i=1}^{S(n)} W_i(n). \tag{1}$$

Armed with this notation and definitions, let us examine our second parallel algorithm. We are given a sequence a of $n = 2^k$ elements of some type T in shared memory, and a binary associative operator $\oplus : T \times T \to T$. Associativity implies that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ for all elements a, b, and c. Examples of such an operator on primitive types include addition, multiplication, maximum, minimum, boolean AND, boolean OR, and string concatenation. More complex operators can be built for structured and recursive data types. We want to compute the quantity $S = \oplus_{i=1}^{n} a_i \equiv a_1 \oplus \cdots \oplus a_n$, again in shared memory. This operation is also called **reduction**.

Algorithm 2 (Sequence reduction, WT description)
Input: Sequence a with $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \to T$.
Output: $S = \oplus_{i=1}^{n} a_i$.

T REDUCE(**sequence**$\langle$T$\rangle$ a, $\oplus : T \times T \to T$)

```
1  T B[1..n]
2  forall i ∈ 1 : n do
3     B[i] ← a_i
4  enddo
5  for h = 1 to k do
6     forall i ∈ 1 : n/2^h do
7        B[i] ← B[2i − 1] ⊕ B[2i]
8     enddo
9  enddo
10 S ← B[1]
11 return S
```

The WT program above is a high-level description of the algorithm—there are no references to processor ids. Also note that it contains both serial and concurrent operations. In particular, the final assignment in line 10 is to be performed by a single processor (since it is not contained in a **forall** construct), and the loop in line 5 is a serial **for**-loop. A couple of subtleties of this algorithm are worth emphasizing. First, in line 7, all instances of the expression on the right hand side must be evaluated before any of the assignments are performed. Second, the additions are performed in a different order than in the sequential program. (Verify this.) Our assumption of the associativity of addition is critical to insure the correctness of the result.

Let us determine $S(n)$ and $W(n)$ for of Algorithm 2. Both are determined inductively from the structure of the program. In the following, subscripts refer to lines in the program.

$$\begin{array}{cc}
S_{2\text{–}4}(n) = \Theta(1) & W_{2\text{–}4}(n) = \Theta(n) \\
S_{6\text{–}8}(n) = \Theta(1) & W_{6\text{–}8}(n,h) = \Theta(\frac{n}{2^h}) \\
S_{5\text{–}9}(n) = kS_{6\text{–}8}(n) = \Theta(\lg n) & W_{5\text{–}9}(n) = \sum_{h=1}^{k} W_{6\text{–}8}(n,h) = \Theta(n) \\
S_{10}(n) = \Theta(1) & W_{10}(n) = \Theta(1) \\
S(n) = S_{2\text{–}4}(n) + S_{5\text{–}9}(n) + S_{10}(n) = \Theta(\lg n) & W(n) = W_{2\text{–}4}(n) + W_{5\text{–}9}(n) + W_{10}(n) = \Theta(n)
\end{array}$$

It is reassuring to see that the total amount of work done by the parallel algorithm is (asymptotically) the same as that performed by an optimal sequential algorithm. The benefit of parallelism is the reduction in the number of steps.

We extend the PRAM classification for simultaneous memory references to the WT model. The algorithm above specifies only exclusive read and write operations to the shared memory, and hence requires only an EREW execution model.

2.1 Brent's Theorem

The following theorem, due to Brent, relates the work and time complexities of a parallel algorithm described in the WT formalism to its running time on a p-processor PRAM.

Theorem 1 (Brent 1974) *A WT algorithm with step complexity $S(n)$ and work complexity $W(n)$ can be simulated on a p-processor PRAM in no more than $\lfloor \frac{W(n)}{p} \rfloor + S(n)$ parallel steps.*

Proof: For each time step i, for $1 \leqslant i \leqslant S(n)$, let $W_i(n)$ be the number of operations in that step. We simulate each step of the WT algorithm on a p-processor PRAM in $\lceil \frac{W_i(n)}{p} \rceil$ parallel steps, by scheduling the $W_i(n)$ operations on the p processors in groups of p operations at a time. The last group may not have p operations if p does not divide $W_i(n)$ evenly. In this case, we schedule the remaining operations among the smallest-indexed processors. Given this simulation strategy, the time to simulate step $W_i(n)$ of the WT algorithm will be $\lceil \frac{W_i(n)}{p} \rceil$ and the total time for a p processor PRAM to simulate the algorithm is

$$\sum_{i=1}^{S(n)} \lceil \frac{W_i(n)}{p} \rceil \leqslant \sum_{i=1}^{S(n)} (\lfloor \frac{W_i(n)}{p} \rfloor + 1) \leqslant \lfloor \frac{W(n)}{p} \rfloor + S(n).$$

There are a number of complications that our simple sketch of the simulation strategy does not address. For example, to preserve the semantics of the **forall** construct, we should generally not update any element of the left-hand side of a WT assignment until we have evaluated all the values of the right-hand side expression. This can be accomplished by the introduction of a temporary result that is subsequently copied into the left hand side. □

Let us revisit the sequence reduction example, and try to write the barebones PRAM algorithm for a p-processor PRAM, following the simulation strategy described. Each **forall** construct of Algorithm 2 is simulated using a sequential **for** loop with a body that applies up to p operations of the **forall** body at a time.

Algorithm 3 (Sequence reduction, PRAM description)
Input: Sequence a with $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \to T$, and processor id i.
Output: $S = \oplus_{i=1}^{n} a_i$.

T PRAM-REDUCE(**sequence**⟨T⟩ a, $\oplus : T \times T \to T$)

```
1  T B[1..n]
2  local integer h, j, ℓ
3  for ℓ = 1 to ⌈n/p⌉ do
4     if (ℓ − 1)p + i ⩽ n then
5        B[(ℓ − 1)p + i] ← a_{(ℓ−1)p+i}
6     endif
7  enddo
8  for h = 1 to k do
9     for ℓ = 1 to ⌈n/2^h / p⌉ do
10       j ← (ℓ − 1)p + i
11       if j ⩽ n/2^h then
12          B[j] ← B[2j − 1] ⊕ B[2j]
13       endif
14    enddo
15 enddo
16 if i = 1 then
17    S ← B[1]
18 endif
19 return S
```

The concurrent running time of Algorithm 3 can be analyzed by counting the number of executions of the loop bodies.

$$T_C(n,p) = \lceil \frac{n}{p} \rceil \Theta(1) + \sum_{h=1}^{k} \lceil \frac{n/2^h}{p} \rceil \Theta(1) + \Theta(1) = O(\frac{n}{p} + \lg n)$$

This is the bound provided by Brent's theorem for the simulation of Algorithm 2 with a p processor PRAM. To verify that the bound is tight, consider the summation above in the case that $p > n$ or the case that p is odd. With some minor assumptions, the simulation preserves the shared-memory access model, so that, for example, an EREW algorithm in the WT framework can be simulated using an EREW PRAM.

2.2 Designing good parallel algorithms

PRAM algorithms have a time complexity in which both problem size and the number of processors are parameters. Given a PRAM algorithm with running time $T_C(n,p)$, let $T_S(n)$ be the optimal (or best known) sequential time complexity for the problem. We define the **speedup**

$$\mathcal{SP}(n,p) = \frac{T_S(n)}{T_C(n,p)} \tag{2}$$

as the factor of improvement in the running time due to parallel execution. The best speedup we can hope to achieve (for a deterministic algorithm) is $\Theta(p)$ when using p processors. An asymptotically greater speedup would contradict the assumption that our sequential time complexity was optimal, since a faster sequential algorithm could be constructed by sequential simulation of our PRAM algorithm.

Parallel algorithms in the WT framework are characterized by the single-parameter step and work complexity measures. The work complexity $W(n)$ is the most critical measure. By Brent's Theorem, we can simulate a WT algorithm on a p-processor PRAM in time

$$T_C(n,p) = O(\frac{W(n)}{p} + S(n)). \tag{3}$$

If $W(n)$ asymptotically dominates $T_S(n)$, then we can see that with a fixed number of processors p, increasing problem size decreases the speedup, *i.e.*

$$\lim_{n\to\infty} \mathcal{SP}(n,p) = \lim_{n\to\infty} \frac{T_S(n)}{\lfloor \frac{W(n)}{p} \rfloor + S(n)} = 0$$

Since scaling of p has hard limits in many real settings, we will want to construct parallel WT algorithms for which $W(n) = \Theta(T_S(n))$. Such algorithms are called **work-efficient**.

The second objective is to minimize step complexity $S(n)$. By Brent's Theorem, we can simulate a work-efficient WT algorithm on a p-processor PRAM in time

$$T_C(n,p) = O(\frac{T_S(n)}{p} + S(n)). \tag{4}$$

Thus, the speedup achieved on the p-processor PRAM is

$$\mathcal{SP}(n,p) = \Omega(\frac{T_S(n)}{\frac{T_S(n)}{p} + S(n)}) = \Omega(\frac{pT_S(n)}{T_S(n) + pS(n)}). \tag{5}$$

Thus, $\mathcal{SP}(n,p)$ will be $\Theta(p)$ (asymptotically the best we can achieve) as long as

$$p = O(\frac{T_S(n)}{S(n)}). \tag{6}$$

Thus, among two work-efficient parallel algorithms for a problem, the one with the smaller step complexity is more *scalable* in that it maintains optimal speedup over a larger range of processors.

3 Basic PRAM algorithm design techniques

We now discuss a variety of algorithm design techniques for PRAMs. As you will see, these techniques can deal with many different kinds of data structures, and often have counterparts in design techniques for sequential algorithms.

3.1 Balanced trees

One common PRAM algorithm design technique involves building a balanced binary tree on the input data and sweeping this tree to and from the root. This "tree" is not an actual data structure but rather a concept in our head, often realized as a recursion tree. We have already seen a use of this technique in the array summation example. This technique is widely applicable. It is used to construct work-efficient parallel algorithms for problems such as prefix sum, broadcast, and array compaction. We will look at the prefix sum case.

In the *prefix sum* problem (also called *parallel prefix* or *scan*), we are given an input sequence $x = \langle x_1, \ldots, x_n \rangle$ of elements of some type T, and a binary associative operator $\oplus : T \times T \to T$. As output, we are to produce the sequence $s = \langle s_1, \ldots, s_n \rangle$, where for $1 \leqslant k \leqslant n$, we require that $s_k = \oplus_{i=1}^{k} x_i$.

The sequential time complexity $T_S(n)$ of the problem is clearly $\Theta(n)$: the lower bound follows trivially from the fact that n output elements have to be written, and the upper bound is established by the algorithm that computes s_{i+1} as $s_i \oplus x_{i+1}$. Thus, our goal is to produce a parallel algorithm with work complexity $\Theta(n)$. We will do this using the balanced tree technique. Our WT algorithm will be different from previous ones in that it is recursive. As usual, we will assume that $n = 2^k$ to simplify the presentation.

Algorithm 4 (Prefix sum)
Input: Sequence x of $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \to T$.
Output: Sequence s of $n = 2^k$ elements of type T, with $s_k = \oplus_{i=1}^{k} x_i$ for $1 \leqslant k \leqslant n$.

sequence⟨T⟩ SCAN(**sequence**⟨T⟩ x, $\oplus : T \times T \to T$)

```
1  if n = 1 then
2     s_1 ← x_1
3     return s
4  endif
5  forall i ∈ 1 : n/2 do
6     y_i ← x_{2i-1} ⊕ x_{2i}
7  enddo
8  ⟨z_1, ..., z_{n/2}⟩ ← SCAN(⟨y_1, ..., y_{n/2}⟩, ⊕)
9  forall i ∈ 1 : n do
10    if even(i) then
11       s_i ← z_{i/2}
12    elsif i = 1 then
13       s_1 ← x_1
14    else
15       s_i ← z_{(i-1)/2} ⊕ x_i
16    endif
17 enddo
18 return s
```

Figure 1 illustrates the data flow of Algorithm 4.

Theorem 2 *Algorithm 4 correctly computes the prefix sum of the sequence x with step complexity $\Theta(\lg n)$ and work complexity $\Theta(n)$.*

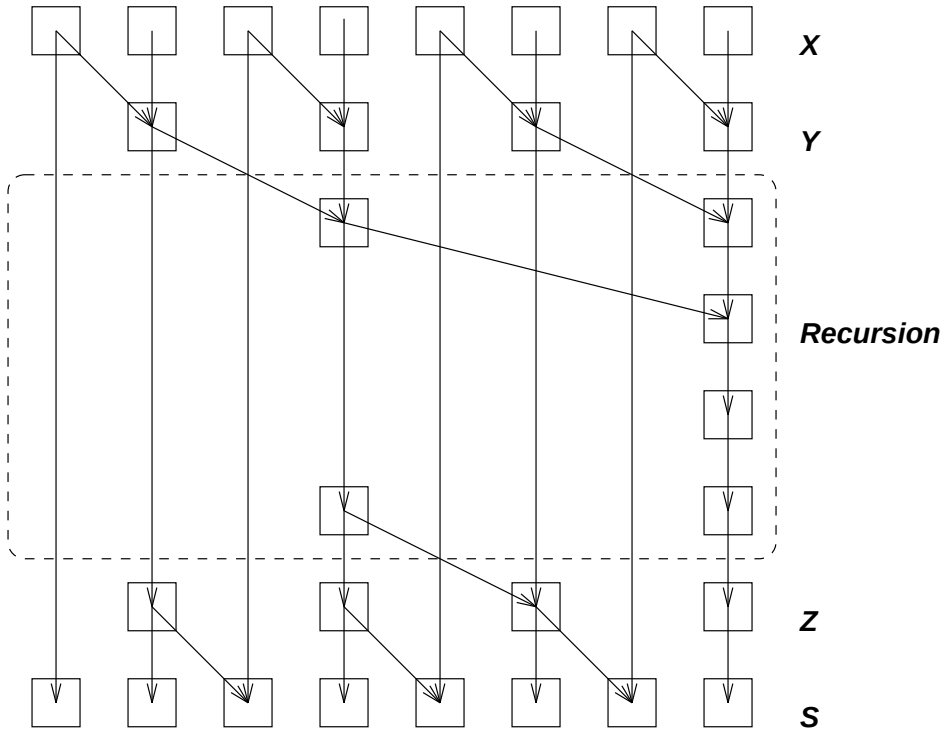


Figure 1: Data flow in Algorithm 4 for an example input of eight elements.

Proof: The correctness of the algorithm is a simple induction on $k = \lg n$. The base case is $k = 0$, which is correct by line 2. Now assume the correctness for inputs of size 2^k, and consider an input of size 2^{k+1}. By the induction hypothesis, z = SCAN(y, $\oplus$) computed in line 8 is correct. Thus, $z_i = y_1 \oplus \cdots \oplus y_i = x_1 \oplus \cdots \oplus x_{2i}$. Now consider the three possibilities for s_i. If $i = 2j$ is even (line 11), then $s_i = z_j = x_1 \oplus \cdots \oplus x_{2j} = x_1 \oplus \cdots \oplus x_i$. If $i = 1$ (line 13), then $s_1 = x_1$. Finally, if $i = 2j + 1$ is odd (line 15), then $s_i = z_j \oplus x_i = x_1 \oplus \cdots \oplus x_{i-1} \oplus x_i$. These three cases are exhaustive, thus establishing the correctness of the algorithm.

To establish the resource bounds, we note that the step and work complexities satisfy the following recurrences.

$$S(n) = S(n/2) + \Theta(1) \tag{7}$$

$$W(n) = W(n/2) + \Theta(n) \tag{8}$$

These are standard recurrences that solve to $S(n) = \Theta(\lg n)$ and $W(n) = \Theta(n)$. □

Thus we have a work-efficient parallel algorithm that can run on an EREW PRAM. It can maintain optimal speedup with $O(\frac{n}{\lg n})$ processors.

Why is the minimal PRAM model EREW when there appears to be a concurrent read of values in $z[1 : n/2 - 1]$ (as suggested by Figure 1)? It is true that each of these values must be read twice, but these reads can be serialized without changing the asymptotic complexity of the algorithm. In fact, since the reads occur on different branches of the conditional (lines 11 and 15), they *will* be serialized in execution under the synchronous PRAM model. In the next section, we will see an example where more than a constant number of processors are trying to read a common value, making the minimal PRAM model CREW.

We can define two variants of the scan operation: *inclusive* (as above) and *exclusive*. For the exclusive scan, we require that the operator $\oplus$ have an identity element I. (This means that $x \oplus I = I \oplus x = x$ for all elements x.) The exclusive scan is then defined as follows: if t is the output sequence, then $t_1 = I$ and $t_k = x_1 \oplus \cdots \oplus x_{k-1}$ for $1 < k \leqslant n$. It is clear that we can obtain the inclusive scan from the exclusive scan by the relation $s_k = t_k \oplus x_k$. Going in the other direction, observe that for $k > 1$, $t_k = s_{k-1}$ and $t_1 = I$.

Finally, what do we do if $n \neq 2^k$? If $2^k < n < 2^{k+1}$, we can simply *pad* the input to size 2^{k+1}, use Algorithm 4, and discard the extra values. Since this does not increase the problem size by more than a factor of two, we maintain the asymptotic complexity.

3.2 Pointer jumping

The technique of *pointer jumping* (or *pointer doubling*) allows fast parallel processing of linked data structures such as lists and trees. We will usually draw trees with edges directed from children to parents (as we do in representing disjoint sets, for example). Recall that a rooted directed tree T is a directed graph with a special root vertex r such that the outdegree of the root is zero, while the outdegree of all other vertices is one, and there exists a directed path from each non-root vertex to the root vertex. Our example problem will be to find all the roots of a forest of directed trees, containing a total of n vertices (and at most $n-1$ edges).

We will represent the forest using an array P (for "Parent") of n integers, such that $P[i] = j$ if and only if (i, j) is an edge in the forest. We will use self-loops to recognize roots, *i.e.*, a vertex i is a root if and only if $P[i] = i$. The desired output is an array S, such that $S[j]$ is the root of the tree containing vertex j, for $1 \leqslant j \leqslant n$. A sequential algorithm using depth-first search gives $T_S(n) = O(n)$.

Algorithm 5 (Roots of a forest)
Input: A forest on n vertices represented by the parent array $P[1..n]$.
Output: An array $S[1..n]$ giving the root of the tree containing each vertex.

```
1  forall i ∈ 1 : n do
2     S[i] ← P[i]
3     while S[i] ≠ S[S[i]] do
4        S[i] ← S[S[i]]
5     endwhile
6  enddo
```

Note again that in line 4 all instances of $S[S[i]]$ are evaluated before any of the assignments to $S[i]$ are performed. The pointer $S[i]$ is the current "successor" of vertex i, and is initially its parent. At each step, the tree distance between vertices i and $S[i]$ doubles as long as $S[S[i]]$ is not a root of the forest. Let h be the maximum height of a tree in the forest. Then the correctness of the algorithm can be established by induction on h. The algorithm runs on a CREW PRAM. All the writes are distinct, but more than a constant number of vertices may read values from a common vertex, as shown in Figure 2. To establish the step and work complexities of the algorithm, we note that the **while**-loop iterates $\Theta(\lg h)$ times, and each iteration performs $\Theta(1)$ steps and $O(n)$ work. Thus, $S(n) = \Theta(\lg h)$, and $W(n) = O(n \lg h)$. These bounds are weak, but we cannot assert anything stronger without assuming more about the input data. The algorithm is not work-efficient unless h is constant. In particular, for a linked list, the algorithm takes $\Theta(\lg n)$ steps and $\Theta(n \lg n)$ work. An interesting exercise is to associate with each vertex i the distance d_i to its successor measured along the path in the tree, and to modify the algorithm to correctly maintain this quantity. On termination, the d_i will be the distance of vertex i from the root of its tree.

The algorithm glosses over one important detail: how do we know when to stop iterating the **while**-loop? The first idea is to use a fixed iteration count, as follows. Since the height of the tallest tree has a trivial upper bound of n, we do not need to repeat the pointer jumping loop more than $\lg n$ times.

```
forall i ∈ 1 : n do
   S[i] ← P[i]
enddo
for k = 1 to lg n do
   forall i ∈ 1 : n do
      S[i] ← S[S[i]]
   enddo
enddo
```

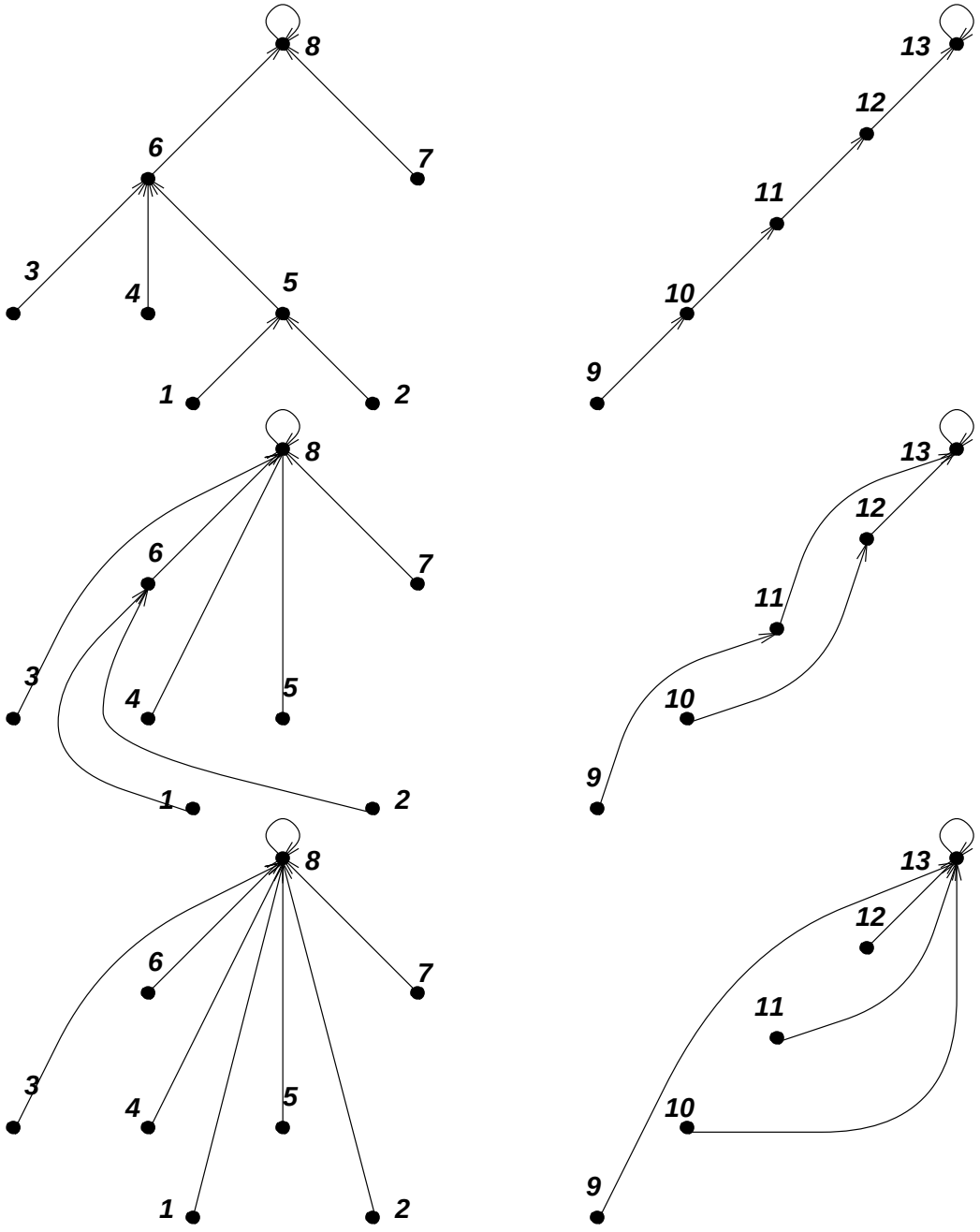


Figure 2: Three iterations of line 4 in Algorithm 5 on a forest with 13 vertices and two trees.

This is correct but inefficient, since our forest might consist of many shallow and bushy trees. Its work complexity is $\Theta(n \lg n)$ instead of $O(n \lg h)$, and its step complexity is $\Theta(\lg n)$ instead of $O(\lg h)$. The second idea is an "honest" termination detection algorithm, as follows.

forall $i \in 1:n$ **do**
  $S[i] \leftarrow P[i]$
enddo
repeat
  forall $i \in 1:n$ **do**
    $S[i] \leftarrow S[S[i]]$
    $M[i] \leftarrow$ **if** $S[i] \neq S[S[i]]$ **then** 1 **else** 0 **endif**
  enddo
  $t \leftarrow$ REDUCE(M, $+$)
until $t = 0$

This approach has the desired work complexity of $O(n \lg h)$, but its step complexity is $O(\lg n \lg h)$, since we perform an $O(\lg n)$ step reduction in each of the $\lg h$ iterations.

In the design of parallel algorithms, minimizing work complexity is most important, hence we would probably favor the use of the honest termination detection in Algorithm 5. However, the basic algorithm, even with this modification, is not fully work efficient. The algorithm can be made work efficient using the techniques presented in the next section; details may be found in JáJá §3.1.

3.3 Algorithm cascading

Parallel algorithms with suboptimal work complexity should not be dismissed summarily. *Algorithm cascading* is the composition of a work-inefficient (but fast) algorithm with an efficient (but slower or sequential) algorithm to improve the work efficiency. We can sometimes convert a work-inefficient algorithm to a work-efficient algorithm using this technique.

Our example problem to illustrate the method is the following. Given a sequence $L[1..n]$ of integers in the range $1:k$ where $k = \lg n$, find how many times each integer in this range occurs in L. That is, compute a histogram $R[1..k]$ such that for all $1 \leqslant i \leqslant k$, $R[i]$ records the number of entries in L that have value i.

An optimal sequential algorithm for this problem with $T_S(n) = \Theta(n)$ is the following.

$R[1:k] \leftarrow 0$
for $i = 1$ **to** n **do**
  $R[L[i]] \leftarrow R[L[i]] + 1$
enddo

To create a parallel algorithm, we might construct $C[1..n, 1..k]$ where

$$C_{i,j} = \begin{cases} 1 & \text{if } L[i] = j \\ 0 & \text{otherwise} \end{cases}$$

in parallel. Now to find the number of occurrences of j in L, we simply sum column j of C, i.e. $C[1:n, j]$. The complete algorithm is

forall $i \in 1:n, j \in 1:k$ **do**
  $C[i,j] \leftarrow 0$
enddo
forall $i \in 1:n$ **do**
  $C[i, L[i]] \leftarrow 1$
enddo
forall $j \in 1:k$ **do**
  $R[j] \leftarrow$ REDUCE($C[1:n, j]$, $+$)
enddo

The step complexity of this algorithm is $\Theta(\lg n)$ as a result of the step complexity of the REDUCE operations. The work complexity of the algorithm is $\Theta(nk) = \Theta(n \lg n)$ as a result of the first and last **forall** constructs. The algorithm is not work efficient because C is too large to initialize and too large to sum up with only $O(n)$ work.

However, a variant of the efficient sequential algorithm given earlier can create and sum k successive rows of C in $O(k) = O(\lg n)$ (sequential) steps and work. Using $m = n/\lg n$ parallel applications of this sequential algorithm we can create $\hat{C}[1..m, 1..k]$ in $O(k) = O(\lg n)$ steps and performing a total of $O(mk) = O(n)$ work. Subsequently we can compute the column sums of $\hat{C}$ with these same complexity bounds.

Algorithm 6 (Work-efficient cascaded algorithm for label counting problem)
Input: Sequence $L[1..n]$ with values in the range $1:k$
Output: Sequence $R[1..k]$ the occurrence counts for the values in L

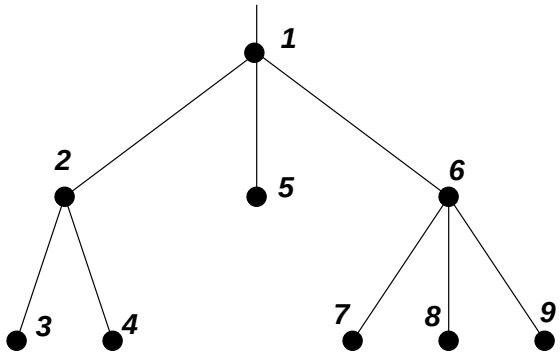
```
1  integer Ĉ[1..m, 1..k]
2  forall i ∈ 1:m, j ∈ 1:k do
3     Ĉ[i, j] ← 0
4  enddo
5  forall i ∈ 1:m do
6     for j = 1 to k do
7        Ĉ[i, L[(i − 1)k + j]] ← Ĉ[i, L[(i − 1)k + j]] + 1
8     enddo
9  enddo
10 forall j ∈ 1:k do
11    R[j] ← REDUCE(Ĉ[1:m, j], +)
12 enddo
```

The cascaded algorithm has $S(n) = O(\lg n)$ and $W(n) = O(n)$ hence has been made work efficient without an asymptotic increase in step complexity. The algorithm runs on the EREW PRAM model.

3.4 Euler tours

The *Euler tour* technique is used in various parallel algorithms operating on tree-structured data. The name comes from Euler circuits of directed graphs. Recall that an **Euler circuit** of a directed graph is a directed cycle that traverses each edge exactly once. If we take a tree $T = (V, E)$ and produce a directed graph $T' = (V, E')$ by replacing each edge $(u, v) \in E$ with two directed edges (u, v) and (v, u) in E', then the graph T' has an Euler circuit. We call a Euler circuit of T' an **Euler tour** of T. Formally, we specify an Euler tour of T by defining a successor function $s : E' \to E'$, such that $s(e)$ is the edge following edge e in the tour. By defining the successor function appropriately, we can create an Euler tour to enumerate the vertices according to an inorder, preorder or postorder traversal.

For the moment, consider rooted trees. There is a strong link between the Euler tour representation of a tree and a representation of the tree structure as a balanced parenthesis sequence. Recall that every sequence of balanced parentheses has an interpretation as a rooted tree. This representation has the key property that the subsequence representing any subtree is also balanced. The following is a consequence of this property: if $\oplus$ is a binary associative operator with an inverse, and we place the element e at each left parenthesis and its inverse element $-e$ at each right parenthesis, then the sum (with respect to $\oplus$) of the elements of any subtree is zero.



$L =$	1	2	3	5	8	10	11	13	15										
$R =$	18	7	4	6	9	17	12	14	16										
$A =$	1	1	1	-1	1	-1	-1	1	-1	1	1	-1	1	-1	1	-1	-1	-1	
$B =$	0	1	2	3	2	3	2	1	2	1	2	3	2	3	2	3	2	1	
$D =$	0	1	2	2	1	1	2	2	2										

Figure 3: Determining the depth of the vertices of a tree.

Algorithm 7 (Depth of tree vertices)
Input: A rooted tree T on n vertices in Euler tour representation using two arrays $L[1..n]$ and $R[1..n]$.
Output: The array $D[1..n]$ containing the depth of each vertex of T.

```
1  integer A[1..2n]
2  forall i ∈ 1 : n do
3     A[L[i]] ← 1
4     A[R[i]] ← −1
5  enddo
6  B ← EXCL-SCAN(A, +)
7  forall i ∈ 1 : n do
8     D[i] ← B[L[i]]
9  enddo
```

Algorithm 7 shows how to use this property to obtain the depth of each vertex in the tree. For this algorithm, the tree T with n vertices is represented as two arrays L and R of length n in shared memory (for left parentheses and right parentheses, respectively). These arrays need to be created from an Euler tour that starts and ends at the root. L_i is the earliest position in the tour in which vertex i is visited. R_i is the latest position in the tour in which vertex i is visited.

Figure 3 illustrates the operation of Algorithm 7. The algorithm runs on an EREW PRAM with step complexity $\Theta(\lg n)$ and work complexity $\Theta(n)$.

We now describe how to construct an Euler tour from a pointer-based representation of a tree T. We assume that the $2n-2$ edges of T' are represented as a set of adjacency lists for each vertex and assume further that the adjacency list L_u for vertex u is circularly linked, as shown in Figure 4. An element v of the list L_u defines the edge (u, v). The symmetric edge (v, u) is found on list L_v. We assume that symmetric edges are linked by pointers in both directions (shown as dashed arrows in Figure 4). Thus there are a total of $2n-2$ edges in the adjacency lists, and we assume that these elements are organized in an array E' of size $2n-2$.

If we consider the neighbors of vertex u the order in which they appear on $L_u = (v_0, \ldots, v_{d-1})$, where d is the degree of vertex u, and we define the successor function as follows: $s((v_i, u)) = (u, v_{(i+1) \bmod d})$, then we have defined a valid Euler tour of T. (To prove this we have to show that we create a single cycle rather than a set of edge-disjoint cycles. We establish this fact by induction on the number of vertices.)

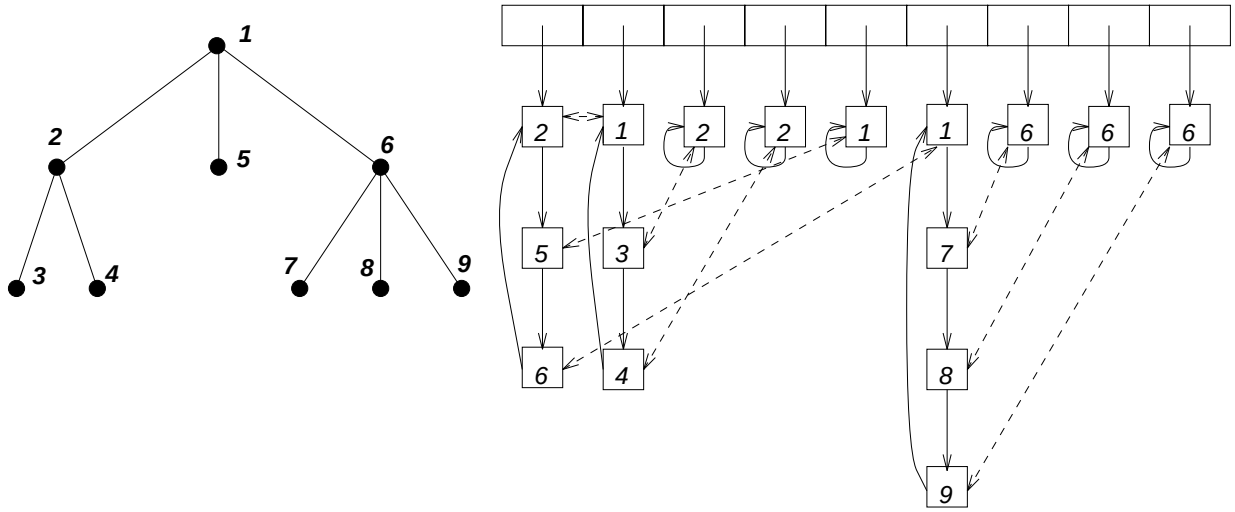


Figure 4: Building the Euler tour representation of a tree from a pointer-based representation.

The successor function can be evaluated in parallel for each edge in E' by following the symmetric (dashed) pointer and then the adjacency list (solid) pointer. This requires $\Theta(1)$ steps with $\Theta(n)$ work complexity, which is work-efficient. Furthermore, since the two pointers followed for each element in E' are unique, we can do this on an EREW PRAM.

Note that what we have accomplished is to link the edges of T' into an Euler tour. To create a representation like the L and R array used in Algorithm 7, we must do further work.

3.5 Divide and conquer

The *divide-and-conquer* strategy is the familiar one from sequential computing. It has three steps: dividing the problem into sub-problems, solving the sub-problems recursively, and combining the sub-solutions to produce the solution. As always, the first and third steps are critical.

To illustrate this strategy in a parallel setting, we consider the planar convex hull problem. We are given a set $S = \{p_1, \ldots, p_n\}$ of points, where each point p_i is an ordered pair of coordinates (x_i, y_i). We further assume that points are sorted by x-coordinate. (If not, this can be done as a preprocessing step with low enough complexity bounds.) We are asked to determine the **convex hull** CH(S), *i.e.*, the smallest convex polygon containing all the points of S, by enumerating the vertices of this polygon in clockwise order. Figure 5 shows an instance of this problem.

The sequential complexity of this problem is $T_S(n) = \Theta(n \lg n)$. Any of several well-known algorithms for this problem establishes the upper bound. A reduction from comparison-based sorting establishes the lower bound. See §35.3 of CLR for details.

We first note that p_1 and p_n belong to CH(S) by virtue of the sortedness of S, and they partition the convex hull into an **upper hull** UH(S) and an **lower hull** LH(S). Without loss of generality, we will show how to compute UH(S).

The division step is simple: we partition S into $S_1 = \{p_1, \ldots, p_{n/2}\}$ and $S_2 = \{p_{n/2+1}, \ldots, p_n\}$. We then recursively obtain UH(S_1) = $\langle p_1 = q_1, \ldots, q_s \rangle$ and UH(S_2) = $\langle r_1, \ldots, r_t = p_n \rangle$. Assume that for $n \leqslant 4$, we solve the problem by brute force. This gives us the termination condition for the recursion.

The combination step is nontrivial. Let the **upper common tangent** (UCT) be the common tangent to UH(S_1) and UH(S_2) such that both UH(S_1) and UH(S_2) are below it. Thus, this tangent consists of two points, one each from UH(S_1) and UH(S_2). Let UCT(S_1, S_2) = (q_i, r_j). Assume the existence of an $O(\lg n)$ sequential algorithm for determining UCT(S_1, S_2) (See Preparata and Shamos, Lemma 3.1). Then UH(S) = $\langle q_1, \ldots, q_i, r_j, \ldots, r_t \rangle$, and contains $(i + t - j + 1)$ points. Given s, t, i, and j, we can obtain UH(S) in $\Theta(1)$ steps and $O(n)$ work as follows.

forall $k \in 1 : i + t - j + 1$ **do**
  UH(S)$_k$ $\leftarrow$ **if** $k \leqslant i$ **then** q_k **else** $r_{k+j-i-1}$ **endif**

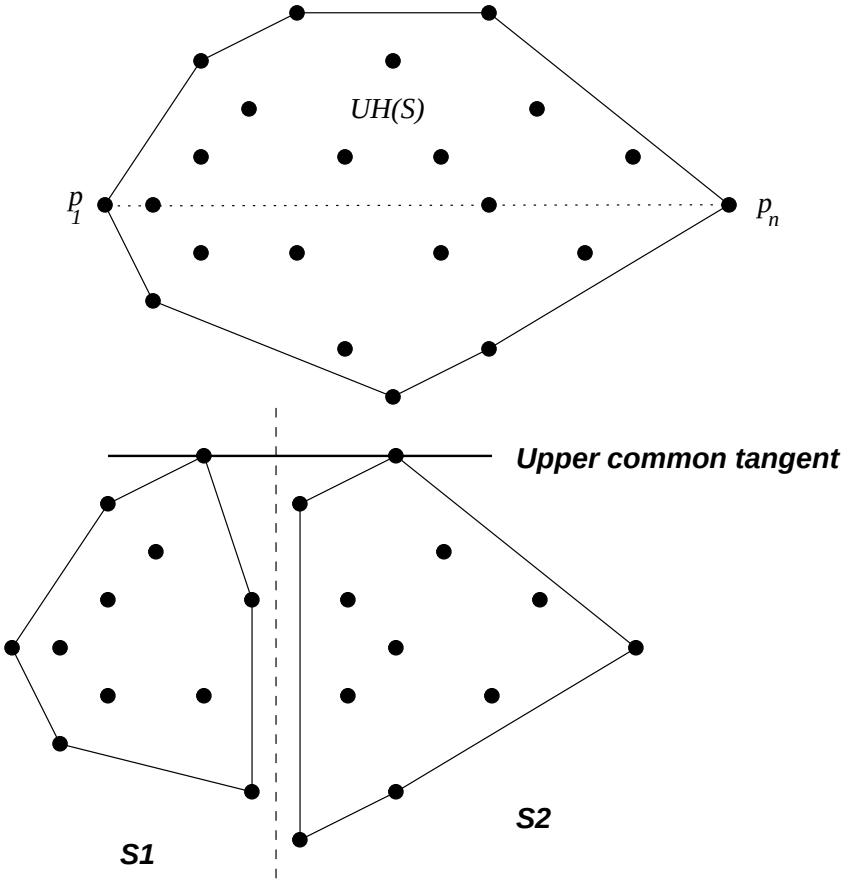


Figure 5: Determining the convex hull of a set of points.

enddo

This algorithm requires a minimal model of a CREW PRAM. To analyze its complexity, we note that

$$S(n) = S(n/2) + O(\lg n) \tag{9}$$
$$W(n) = 2W(n/2) + O(n) \tag{10}$$

giving us $S(n) = O(\lg^2 n)$ and $W(n) = O(n \lg n)$.

3.6 Symmetry breaking

The technique of *symmetry breaking* is used in PRAM algorithms to distinguish between identical-looking elements. This can be deterministic or probabilistic. We will study a randomized algorithm (known as the **random mate** algorithm) to determine the connected components of an undirected graph as an illustration of this technique. See §30.5 of CLR for an example of a deterministic symmetry breaking algorithm.

Let $G = (V, E)$ be an undirected graph. We say that edge (u, v) *hits* vertices u and v. The *degree* of a vertex v is the number of edges that hit v. A *path* from v_1 to v_k (denoted $v_1 \rightsquigarrow v_k$) is a sequence of vertices $(v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for $1 \leqslant i < k$. A *connected subgraph* is a subset U of V such that for all $u, v \in U$ we have $u \rightsquigarrow v$. A *connected component* is a maximal connected subgraph. A *supervertex* is a directed rooted tree data structure used to represent a connected subgraph. We use the standard disjoint-set conventions of edges directed from children to parents and self-loops for roots to represent supervertices.

We can find connected components optimally in a sequential model using depth-first search. Thus, $T_S(V, E) = O(V + E)$. Our parallel algorithm will actually be similar to the algorithm in §22.1 of CLR. The idea behind the algorithm is to merge supervertices to get bigger supervertices. In the sequential case, we examine the edges in a predetermined order. For our parallel algorithm, we would like to examine multiple edges at each time step. We break symmetry by arbitrarily choosing the next supervertex to merge, by randomly assigning genders to supervertices. We call a graph edge (u, v) *live* if u and v belong to different supervertices, and we call a supervertex *live* if at least one live edge hits some vertex of the supervertex. While we still have live edges, we will merge supervertices of opposite gender connected by a live edge. This merging includes a path compression step. When we run out of live edges, we have the connected components.

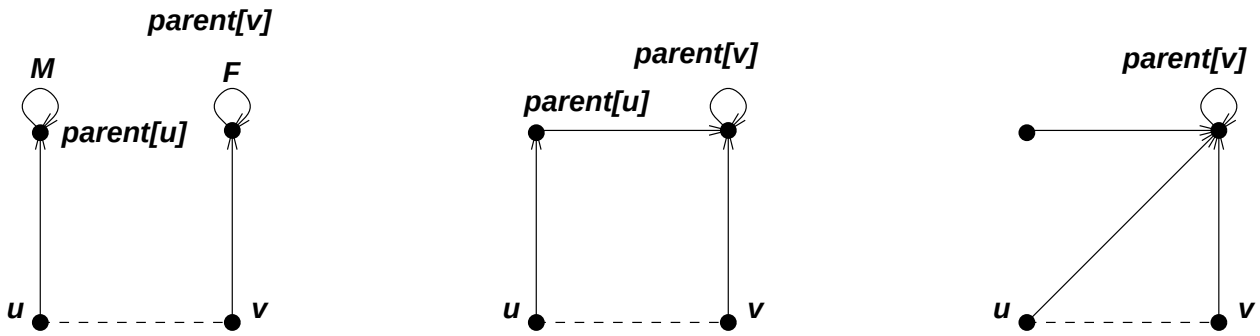


Figure 6: Details of the merging step of Algorithm 8. Graph edges are undirected and shown as dashed lines. Supervertex edges are directed and are shown as solid lines.

Algorithm 8 (Random-mate algorithm for connected components)
Input: An undirected graph $G = (V, E)$.
Output: The connected components of G, numbered in the array $P[1..|V|]$.

```
1  forall v ∈ V do
2     parent[v] ← v
3  enddo
4  while there are live edges in G do
5     forall v ∈ V do
6        gender[v] = rand({M, F})
7     enddo
8     forall (u, v) ∈ E | live(u, v) do
9        if gender[parent[u]] = M and gender[parent[v]] = F then
10          parent[parent[u]] ← parent[v]
11       endif
12       if gender[parent[v]] = M and gender[parent[u]] = F then
13          parent[parent[v]] ← parent[u]
14       endif
15    enddo
16    forall v ∈ V do
17       parent[v] ← parent[parent[v]]
18    enddo
19 endwhile
```

Figure 6 shows the details of the merging step of Algorithm 8. We establish the complexity of this algorithm by proving a succession of lemmas about its behavior.

Lemma 1 *After each iteration of the outer **while**-loop, each supervertex is a star (a tree of height zero or one).*

Proof: The proof is by induction on the number of iterations executed. Before any iterations of the loop have been executed, each vertex is a supervertex with height zero by the initialization in line 2. Now assume that the claim holds after k iterations, and consider what happens in the $(k + 1)$st iteration. Refer to Figure 6. After the **forall** loop in line 8, the height of a supervertex can increase by one, so it is at most two. After the compression step in line 16, the height goes back to one from two. □

Lemma 2 *Each iteration of the **while**-loop takes $\Theta(1)$ steps and $O(V + E)$ work.*

Proof: This is easy. The only nonobvious part is determining live edges, which can be done in $\Theta(1)$ steps and $O(E)$ work. Since each vertex is a star by Lemma 1, edge (u, v) is live if and only if parent[u] $\neq$ parent[v]. □

Lemma 3 *The probability that at a given iteration a live supervertex is joined to another supervertex* $\geqslant 1/4$.

Proof: A live supervertex has at least one live edge. The supervertex will get a new root if and only if its gender is M and it has a live edge to a supervertex whose gender is F. The probability of this is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. The probability is at least this, since the supervertex may have more than one live edge. □

Lemma 4 *The probability that a vertex is a live root after* $5 \lg |V|$ *iterations of the* ***while****-loop is* $\leqslant 1/|V|^2$.

Proof: By Lemma 3, a live supervertex at iteration i has probability $\leqslant \frac{3}{4}$ to remain live after iteration $i+1$. Therefore the probability that it is lives after $5 \lg |V|$ iterations is $\leqslant (\frac{3}{4})^{5 \lg |V|}$. The inequality follows, since $\lg(\frac{3}{4})^{5 \lg |V|} = -2.075 \lg |V|$ and $\lg \frac{1}{|V|^2} = -2 \lg |V|$. □

Lemma 5 *The expected number of live supervertices after* $5 \lg |V|$ *iterations* $\leqslant 1/|V|$.

Proof: We compute the expected number of live supervertices by summing up the probability of each vertex to be a live root. By Lemma 4, this is $\leqslant |V| \cdot \frac{1}{|V|^2} = \frac{1}{|V|}$. □

Theorem 3 *With probability at most* $1/|V|$*, the algorithm will not have terminated after* $5 \lg |V|$ *iterations.*

Proof: Let p_k be the probability of having k live supervertices after $5 \lg |V|$ iterations. By the definition of expectation, the expected number of live supervertices after $5 \lg |V|$ iterations is $\sum_{k=0}^{|V|} k \cdot p_k$, and by Lemma 5, this is $\leqslant \frac{1}{|V|}$. Since k and p_k are all positive, $\sum_{k=1}^{|V|} p_k \leqslant \sum_{k=0}^{|V|} k \cdot p_k \leqslant \frac{1}{|V|}$. Now, the algorithm terminates when the number of live supervertices is zero. Therefore, $\sum_{k=1}^{|V|} p_k$ is the probability of still having to work after $5 \lg |V|$ steps. □

The random mate algorithm requires a CRCW PRAM model. Concurrent writes occur in the merging step (line 8), since different vertices can have a common parent.

The step complexity of the algorithm is $O(\lg V)$ with high probability, as a consequence of Theorem 3. The work complexity is $O((V + E) \lg V)$ by Theorem 3 and Lemma 2. Thus the random mate algorithm is not work-optimal.

A key factor in this algorithm is that paths in supervertices are short (in fact, $\Theta(1)$). This allows the supervertices after merging to be converted back to stars in a single iteration of path compression in line 16. If we used some deterministic algorithm to break symmetry, we would not be able to guarantee short paths. We would have multiple supervertices and long paths within supervertices, and the step complexity of such an algorithm would be $O(\lg^2 V)$. There is a deterministic algorithm due to Shiloach and Vishkin that avoids this problem by not doing complete path compression at each step. Instead, it maintains a complicated set of invariants that insure that the supervertices left when the algorithm terminates truly represent the connected components.

4 A tour of data-parallel algorithms

In this section, we present a medley of data-parallel algorithms for some common problems.

4.1 Basic scan-based algorithms

A number of useful building blocks can be constructed using the scan operation as a primitive. In the following examples, we will use both zero-based and one-based indexing of arrays. In each case, be sure to calculate step and work complexities and the minimum PRAM model required.

Enumerate The **enumerate** operation takes a Boolean vector and numbers its **true** elements, as follows.

sequence⟨**integer** ⟩ ENUMERATE(**sequence**⟨**boolean** ⟩ Flag)

```
1  forall i ∈ 1 : #Flag do
2     V[i] ← if Flag[i] then 1 else 0 endif
3  enddo
4  return SCAN(V, +)
```

Flag =	**true**	**true**	**false**	**true**	**false**	**false**	**true**
$V =$	1	1	0	1	0	0	1
Result =	1	2	2	3	3	3	4

Copy The **copy** (or **distribute**) operation copies an integer value across an array.

sequence⟨**integer** ⟩ COPY(**integer** v, **integer** n)

```
1  forall i ∈ 1 : n do
2     V[i] ← if i = 1 then v else 0 endif
3  enddo
4  return SCAN(V, +)
```

$v =$	5						
$n =$	7						
$V =$	5	0	0	0	0	0	0
Result =	5	5	5	5	5	5	5

Pack The **pack** operation takes a vector A of values and a Boolean vector F of flags, and returns a vector containing only those elements of A whose corresponding flags are **true**.

sequence⟨T⟩ PACK(**sequence**⟨T⟩ A, **sequence**⟨**boolean** ⟩ F)

```
1  P ← ENUMERATE(F)
2  forall i ∈ 1 : #F do
3     if F[i] then
4        R[P[i]] ← A[i]
5     endif
6  enddo
7  return R[1 : P[#F]]
```

$A =$	♣	♢	♡	♠	▲	■	♦
$F =$	**true**	**true**	**false**	**true**	**false**	**false**	**true**
$P =$	1	2	2	3	3	3	4
$R =$	♣	♢	♠	♦			

Split The **split** operation takes a vector A of values and a Boolean vector F of flags, and returns a vector with the elements with **false** flags moved to the bottom and the elements with **true** flags moved to the top.

sequence⟨T⟩ SPLIT(**sequence**⟨T⟩ A, **sequence**⟨**boolean** ⟩ F)

```
1  Down ← ENUMERATE(not (F))
2  P ← ENUMERATE(F)
3  forall i ∈ 1 : #F do
4     Index[i] ← if F[i] then P[i] + Down[#F] else Down[i] endif
5  enddo
6  forall i ∈ 1 : #F do
7     R[Index[i]] ← A[i]
8  enddo
9  return R
```

$A =$	♣	♢	♡	♠	▲	■	♦
$F =$	**true**	**true**	**false**	**true**	**false**	**false**	**true**
Down =	0	0	1	1	2	3	3
Index =	4	5	1	6	2	3	7
$R =$	♡	▲	■	♣	♢	♠	♦

This parallel **split** can be used as the core routine in a parallel radix sort. Note that **split** is stable: this is critical.

4.2 Parallel lexical analysis

Our final example is a parallel lexical analyzer that we implement using scans. The problem is that of breaking a string over some alphabet into tokens corresponding to the lexical structure specified by a regular language and recognized by a deterministic finite automaton (DFA). Recall that, formally, a DFA M is the 5-tuple $M = (S, \Sigma, \delta, q_0, F)$, where S is a set of states, Σ is the input alphabet, $\delta : S \times \Sigma \to S$ is the transition function, $q_0 \in S$ is the initial state, and $F \subseteq S$ is the set of final states. The use of scans in this example is interesting in that the binary associative operator used (function composition) is noncommutative.

Consider the family of functions $\{f_i : S \to S \mid i \in \Sigma\}$, where $f_i(s) = \delta(s, i)$. That is, the function f_i describes the action of the DFA M on input symbol i; or, put another way, each input symbol is encoded as a function. Observe that the family of functions $\{f_i\}$ form a partition of δ. In fact, they are precisely the columns of the transition table of the DFA. We can represent each f_i as a one-dimensional array of states indexed by states. Since function composition is a binary associative operator, we can define a scan based on it, and this is the key to this algorithm. The composition of functions represented as arrays can be accomplished by replacing every entry of one array with the result of using that entry to index into the other array.

Algorithm 9 (Parallel lexical analysis)
Input: A DFA $M = (S, \Sigma, \delta, q_0, F)$, input sequence $\rho \in \Sigma^*$.
Output: Tokenization of ρ.

1. Replace each symbol i of the input sequence ρ with the array representation of f_i. Call this sequence ϕ.
2. Perform a scan of ϕ using function composition as the operator. This scan replaces symbol ρ_i of the original input sequence with a function g_i that represents the state-to-state transition function for the prefix $\rho_1 \ldots \rho_i$ of the sequence. Note that we need a CREW PRAM to execute this scan (why?).
3. Create the sequence ψ where $\psi_i = g_i(q_0)$. That is, use the initial state of the DFA to index each of these arrays. Now we have replaced each symbol by the state the DFA would be in after consuming that symbol. The states that are in the set of final states demarcate token boundaries.

A sequential algorithm for lexical analysis on an input sequence of n symbols has complexity $T_S(n) = \Theta(n)$. The parallel algorithm has step complexity $\Theta(\lg n)$ and work complexity $O(n|S|)$. Thus the parallel algorithm is faster but not work-efficient.

5 The relative power of PRAM models

We say that a PRAM model P is *more powerful* than another PRAM model Q if an algorithm with step complexity $S(n)$ on model Q has step complexity $O(S(n))$ on model P. We show that the ability to perform concurrent reads and writes allows us to solve certain problems faster, and then we quantify just how much more powerful the CREW and CRCW models are compared to the EREW model.

5.1 The power of concurrent reads

How long does it take for p processors of an EREW PRAM to read a value in shared memory? There are two possible ways for the p processors to read the shared value. First, they can serially read the value in round robin fashion in $O(p)$ time. Second, processors can replicate the value as they read it, so that a larger number of processors can read it in the next round. If each processor makes one copy of the value as it reads it, the number of copies doubles at each round, and $O(\lg p)$ time suffices to make the value available to the p processors.

The argument above is, however, only an upper bound proof. We now give a lower bound proof, *i.e.*, an argument that $\Omega(\lg p)$ steps are *necessary*. Suppose that as each processor reads the value, it sequentially makes k copies, where k can be a function of p. Then the number of copies grows by a factor of $k(p)$ at each round, but each round takes $k(p)$ time. The number of rounds needed to replicate the value p-fold is $\lg p / \lg k$, and the total time taken is $\lg p \cdot \frac{k}{\lg k}$. This is asymptotically greater than $\lg p$ unless $\frac{k}{\lg k} = \Theta(1)$, *i.e.*, k is a constant. This means that it is no good trying to make more copies sequentially at each round. The best we can do is to make a constant number of copies, and that gives us the desired $\Omega(\lg p)$ bound. A replication factor of $k = 2$ gives us the smallest constant, but any constant value of k will suffice for the lower bound argument.

5.2 The power of concurrent writes

To show the power of concurrent writes, we reconsider the problem of finding the maximum elements of the sequence $X = \langle x_1, \ldots, x_n \rangle$. We have seen one solution to this problem before using the binary tree technique. That resulted in a work-efficient algorithm for an EREW PRAM, with work complexity $\Theta(n)$ and step complexity $\Theta(\lg n)$. Can we produce a CRCW algorithm with lower step complexity? The answer is yes, as shown by the following algorithm.

Algorithm 10 (Common-CRCW or Arbitrary-CRCW algorithm for maximum finding)
Input: A sequence X of n elements.
Output: A maximum element of X.

```
1  integer M[1..n], B[1..n, 1..n]
2  forall i ∈ 1 : n do
3     M[i] ← 1
4  enddo
5  forall i ∈ 1 : n do
6     forall j ∈ 1 : n do
7        B[i, j] ← if X_i ⩾ X_j then 1 else 0 endif
8     enddo
9  enddo
10 forall i ∈ 1 : n do
11    forall j ∈ 1 : n do
12       if not B[i, j] then
13          M[i] ← 0
14       endif
15    enddo
16 enddo
```

It is easy to verify that $M[i] = 1$ at the end of this computation if and only if X_i is a maximum element. Analysis of this algorithm reveals that $S(n) = \Theta(1)$ but $W(n) = \Theta(n^2)$. Thus the algorithm is very fast but far indeed from being work efficient. However, we may cascade this algorithm with the sequential maximum reduction algorithm or the EREW PRAM maximum reduction algorithm to obtain a work efficient CRCW PRAM algorithm with $S(n) = \Theta(\lg \lg n)$ step complexity. This is optimal for the Common and Arbitrary CRCW model. Note that a trivial work-efficient algorithm with $S(n) = \Theta(1)$ exists for maximum value problem in the Combining-CRCW model, which demonstrates the additional power of this model.

The $\Theta(1)$ step complexity for maximum in CRCW models is suspicious, and points to the lack of realism in the CRCW PRAM model. Nevertheless, a bit-serial maximum reduction algorithm based on ideas like the above but employing only single-bit concurrent writes (i.e. a wired "or" tree), has proved to be extremely fast and practical in a number of SIMD machines. The CRCW PRAM model can easily be used to construct completely unrealistic parallel algorithms, but it remains important because it has also led to some very practical algorithms.

5.3 Quantifying the power of concurrent memory accesses

The CRCW PRAM model assumes that any number of simultaneous memory accesses to the same location can be served in a single timestep. A real parallel machine will clearly never be able to do that. The CRCW model may be a powerful model for algorithm design, but it is architecturally implausible. We will try to see how long it takes an EREW PRAM to simulate the concurrent memory accesses of a CRCW PRAM. Specifically, we will deal with concurrent writes under a priority strategy. We assume the following lemma without proof.

Lemma 6 (Cole 1988) *A p-processor EREW PRAM can sort p elements in $\Theta(\lg p)$ steps.* □

Based on this lemma, we can prove the following theorem.

Theorem 4 *A p-processor EREW PRAM can simulate a p-processor Priority CRCW PRAM with $\Theta(\lg p)$ slowdown.*

Proof: In fact, all we will show is a simulation that guarantees $O(\lg p)$ slowdown. The $\Omega(\lg p)$ lower bound does hold, but establishing it is beyond the scope of this course. See Chapter 10 of JáJá if you are interested.

Assume that the CRCW PRAM has p processors P_1 through P_p and m memory locations M_1 through M_m, and that the EREW PRAM has the same number of processors but $O(p)$ extra memory locations. We will show how to simulate on the EREW PRAM a CR or CW step in which processor P_i accesses memory location M_j. In our simulation, we will use an array T to store the pairs (j, i), and an array S to record the processor that finally got the right to access a memory location.

The following code describes the simulation.

Processor $=$	1	2	3	4	5
Memory location accessed $=$	3	3	7	3	7
Value written $=$	6	2	9	3	7
$T =$	$(3,1)$	$(3,2)$	$(7,3)$	$(3,4)$	$(7,5)$
Sorted $T =$	$(3,1)$	$(3,2)$	$(3,4)$	$(7,3)$	$(7,5)$
$S =$	1	0	1	0	0

Figure 7: Illustration of the simulation of a concurrent write by a five-processor EREW PRAM. Processor P_1 succeeds in writing the value 6 into memory location M_3, and processor P_3 succeeds in writing the value 9 into memory location M_7.

Algorithm 11 (Simulation of Priority CRCW PRAM by EREW PRAM)

Input: A CRCW memory access step, in which processor P_i reads/writes shared memory location M_j, with $i \in 1:p$ and $j \in 1:m$. For a read step, P_i gets the value stored at M_j. For a write step, the value written to M_j is the value being written by processor P_k, where $k = \min\{i \mid P_i \text{writes } M_j\}$.

Output: A simulation of the CRCW step on an EREW PRAM with p processors and $m + p$ shared memory locations.

1 **forall** $i \in 1:p$ **do**
2 Processor P_i writes the pair (j, i) into $T[i]$
3 **enddo**
4 Sort T first on j's and then on i's.
5 **forall** $k \in 1:p$ **do**
6 Processor P_1 reads $T[1] = (j_1, i_1)$ and sets $S[i_1]$ to 1. For $k > 1$, processor P_k reads $T[k] = (j_k, i_k)$ and $T_{k-1} = [j_{k-1}, i_{k-1}]$ and sets $S[i_k]$ to 1 if $j_k \neq j_{k-1}$ and to 0 otherwise.
7 **enddo**
8 **forall** $i \in 1:p$ **do**
9 For CW, processor P_i writes its value to M_j if $S[i] = 1$.
10 For CR, processor P_i reads the value in M_j if $S[i] = 1$ and duplicates the value for the other processors in $O(\lg p)$ time.
11 **enddo**

Line 1 takes $\Theta(1)$ steps. By Lemma 6, line 4 takes $\Theta(\lg p)$ steps. Line 5 again takes $\Theta(1)$ steps, and line 8 takes $\Theta(1)$ steps for a concurrent write access and $O(\lg p)$ steps for a concurrent read access. Thus the simulation of the CRCW step runs in $O(\lg p)$ EREW steps.

Figure 7 shows an example of this simulation. □

5.4 Relating the PRAM model to practical parallel computation

The PRAM model is often criticized because, in practice, it can not be implemented to scale in the number of processors p, as the model suggests. The principal problem is the PRAM assumption of constant latency for parallel memory references, independent of p.

A simple argument shows us why this is ultimately impossible. Each processor and memory location must occupy some physical space in an actual implementation. No matter how we pack $\Omega(p)$ processors and $\Omega(p)$ memory locations into a sphere, at least some pairs of memory locations and processors will be separated by a distance of $\Omega(\sqrt[3]{p})$. Speed-of-light considerations then set a lower bound for communication between such pairs, and hence latency can not be independent of p.

In fact a similar argument can be used to show that as we increase memory size, we also cannot implement the constant-time memory access assumed in the basic sequential RAM model. This very real limitation has motivated the

use of cache memories and the memory hierarchy found in modern processors. And indeed with modern processors, the RAM model is an increasingly inadequate cost model for sequential algorithms as well.

A PRAM implementation suffers from these physical constraints much more than a RAM implementation because of the additional components required to implement a PRAM. For example, a p processor PRAM must deliver $\Omega(p)$ bandwidth between the memory system and the processors through an interconnection network that is significantly larger than the processors themselves.

Another limitation on latency in the PRAM results from the implementation of concurrent read and write operations. Current memory components permit at most a constant number of simultaneous reads and writes. As we have seen in the previous two sections, this means there is an $\Omega(\lg p)$ latency involved in the servicing of concurrent reads and writes using these memory components.

Nevertheless, a PRAM algorithm can be a valuable start for a practical parallel implementation. Any algorithm that runs efficiently on a p processor PRAM model can be translated into an algorithm that sacrifices a factor of L in parallelism to run efficiently on a p/L-processor machine with a latency $O(L)$ memory system, a much more realistic machine than the PRAM. In the translated algorithm, each of the p/L processors simulates L PRAM processors. The memory latency is "hidden" because a processor has L units of useful and independent work to perform while waiting for each memory access to complete.

A good example of latency hiding can be found in a classical vector processor supercomputer with a high-bandwidth memory system: the interconnect and memory system are pipelined to deliver an *amortized* unit latency for a stream of L independent memory references (a vector read or write). Latency hiding is also implemented by the increasingly advanced superscalar and multithreading capabilities incorporated in commodity processors such as implementations of the Intel IA-32 architecture and the Sun UltraSparc architecture, although generally not anywhere near the scale that can fully amortize the latency in all cases. The memory subsystems are the rate-limiting component in systems constructed around such processors.

Instead of running a PRAM algorithm on an expensive latency-hiding supercomputer, a PRAM algorithm may sometimes be restructured so its shared memory access requirements are better matched to shared memory multiprocessors based on conventional processors with caches. This is the topic of the next unit in this course.