2

List Ranking and Parallel Tree Contraction

Margaret Reid-Miller
Gary L. Miller
Francesmary Modugno

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This chapter discusses parallel algorithms for two problems: list ranking and parallel tree contraction. List ranking is used often as part of a solution to other parallel algorithms. While parallel tree contraction is a technique that has wide application to tree based problems.

2.1 List Ranking

A common problem in computer science is to find the location of an element in a linked list with respect to the first element of the list, called the **head** of the list. This problem, referred to as list ranking, is a fundamental operation for many applications such as computing the prefix sum for any associative operation over a linked list; determining the preorder numbering of nodes on trees; and evaluating expressions on trees. On a RAM the problem can be solved by a straightforward sequential algorithm that traverses the linked list in $O(n)$ time.

In this section we discuss various parallel list ranking algorithms. We start by introducing a simple deterministic parallel algorithm due to Wyllie [Wyl79]. Wyllie's algorithm is not optimal; the total "work" performed by all the processors is greater than the work performed by a single processor using a sequential algorithm. We then introduce a randomized algorithm by Miller and Reif [MR85] that optimizes the work, but still needs as many processors as cells in the linked list. This algorithm was improved by Anderson and Miller [AM90] to use an optimal number of processors and is described next. The final section give another optimal solution by Anderson and Miller [AM88] which is deterministic. The following two chapters introduce deterministic coin tossing and discuss list ranking further.

2.1.1 The problem:

We consider the slightly modified problem of finding the location or *rank* of an element in a linked list with respect to the last element, or **tail** of the list. From solutions to this problem, it is not difficult to obtain solutions to the problem of finding an element's rank with respect to the head of the list.

Formally, the problem is stated as follows:

Given: a linked list of values.

Output: a cell's position with respect to the tail of list.

Assumptions: the linked list is in a continuous block of shared memory and the tail of the list points to a distinguished value, *nil*.

In order to evaluate the algorithms of this section, we examine their efficiency in terms of the number of processors they require and the total amount of work performed by the processors. An algorithm is *optimal* if it is not possible to reduce simultaneously both the runtime and the number of processors. For the algorithms in this chapter, when we say an algorithm is optimal we mean a stronger notion: that the amount of work required is no more than the amount of work required by an optimal sequential algorithm for the same problem. For list ranking this implies that the work done be $O(n)$. The amount of work done can be measured in two ways: it is equal to the number of processors used times the runtime of the algorithm; or it is equal to the sum of the actual amount of work done by each processor. Very often the two measurements are equal. They differ only when many processors are initially used but become idle as time goes on. In this section we see examples of when the two measurements are and are not equal.

Before beginning, we take a moment to briefly explain some notation. Throughout the entire chapter we describe algorithms in a parallel Algol like code for a PRAM with a host. The meaning of most constructs is clear from context. The "**in Parallel do** *body*" construct is intended to mean that the host broadcasts *body* one instruction at a time to the each processor, which then executes each instruction as it arrives. In the "**in Parallel while** *condition* **do** *body*" construction the *condition* is computed by the host and while it is true the host broadcast *body* an instruction at a time to the individual processors. Finally, the special symbol "!" is used to emphasize that the element indexed is local to that processor. Any other index into the array of elements may or may not be local to the processor.

2.1.2 Wyllie's Algorithm for List Ranking

Wyllie presented the simplest algorithm for the list ranking problem. While the algorithm is not optimal, it serves to illustrate a technique that is common to all algorithms for list ranking problem known as *dereferencing* or *pointer jumping*. Let $v = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$ be a link list of cells, not necessarily stored in order. Pointer jumping is the process of reassigning a cell's successor link to point to its successor's successor. That is, if $v_i \rightarrow v_{i+1} \rightarrow v_{i+2}$, then one application of pointer jumping to v_i will yield $v_i \rightarrow v_{i+2}$.

Assume each cell, i, has a pointer, $succ[i]$, that points to its successor in the linked list. The tail of the list points to *nil*. Each cell i maintains a value $rank[i]$ which is the distance the cell is from its current successor $succ[i]$. Initially, $rank[i]$, is 1. The ranks on the edges keep track of how many pointers were jumped and, in the end, tell us how far from the tail of the list a given cell is. Each processor is assigned one cell.

ALGORITHM **2.1**

Wyllie's Algorithm for list ranking

Procedure *Wyllie:*
  In Parallel $rank[!] := 1;$ /* initialize rank */
  In Parallel while $succ[head] \neq nil$ **do**
    if $succ[!] \neq nil$ **do**
      $rank[!] := rank[!] + rank[succ[!]];$ /* update rank */
      $succ[!] := succ[succ[!]];$
    end if
  end in parallel
end *Wyllie*

After each round each linked list is divided into two linked list of half the length so that after $O(\log n)$ rounds all the cells point to nil and the $rank[v]$ is the rank of cell v. Figure 2.1 shows two rounds of the algorithm. For illustrative purposes, cells are arranged in their order in the linked list. The ranks of the cells after each round are given on the links. Observe, that the algorithm is exclusive read and exclusive write and thus works on an EREW PRAM.

Recall that the sequential algorithm takes time $O(n)$. Since this parallel algorithm uses n processors, each taking $O(\log n)$ time, the total work is $O(n \log n)$, which is not optimal. This is because each processor duplicates the work done by another processor. For example, during Round 1 we produce two chains when only one is needed. Processors for b, d, and f need not have done any work because their final rank can easily be computed once the final rank of their successors are known. Rather than all processors simultaneously computing the final rank of their elements, some processors can wait until the rank of their successor is known and then in unit time compute their own rank.

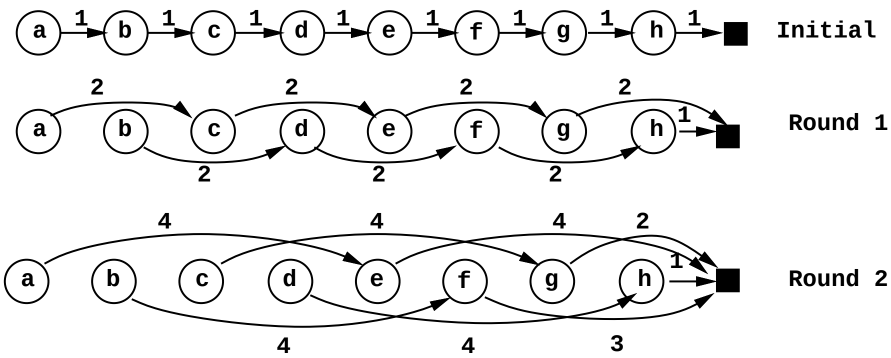

FIGURE 2.1
Two rounds of Wyllie's list ranking algorithm. The black box indicates the value *nil*. The numbers on the links represent the rank of a cell during that round.

There are two major issues that arise when trying to devise optimal list ranking algorithms: resolving contention and identifying elements to "jump over". The first issue is that two adjacent list cells can not be jumped over at the same time; otherwise two chains would form. To resolve contention Reif introduced randomization into the *Random_Mate* algorithm [MR85], whereas Cole and Vishkin [CV86b] developed deterministic coin tossing based on a cell's address, used by the list ranking algorithms discussed in the the next two chapters. The randomized algorithm is given in the next section. The second issue is addressed by Anderson and Miller [AM90, AM88] in the two following algorithms. In order to splice out elements efficiently, it is necessary to have fewer processors than list cells. A processor needs to identify cells on which to work, once its current cell is spliced out. A problem with naive strategies is that as cells are removed from the linked list, it becomes more difficult to find cells still in the list on which to work. A major feature of the Anderson and Miller algorithms is a rather simple scheduling strategy that allows the processors to keep busy with cells remaining in the linked list.

2.1.3 List Ranking using Randomization

In this section we present a very simple randomized algorithm for list ranking. It is optimal in the sense that the total work performed is $O(n)$, although it requires one processor for each cell in the linked list and $O(\log n)$

time. The work is optimal because at each round a constant fraction of the processors are freed up to perform other unrelated work. The algorithm uses randomization to resolve contention. Randomization for the list ranking problem has the advantage of being very simple to implement, while always producing the correct answer. However, there is a very small probability that a particular run of the algorithm may take a long time to complete. On average, though, the randomized algorithms are much faster than many deterministic algorithms, because they have much smaller constants.

Our intuition tells us that in order to eliminate excess work, once a processor's cell has been jumped over, the processor should stop jumping cells. That is, at each round half the processors of the previous round ($\lfloor n/2 \rfloor$) should stop jumping cells. Figure 2.2 shows the rounds of this ideal situation. At the final round we have a tree of depth $\log n$, where the numbers on the arcs show the original distance from the cells to which the arcs point. For example, cell a is distance 8 from *nil*, and cell c is distance 2 from cell e. Following this splicing out phase is a reconstruction phase, in which the linked list is reconstructed and the distance of each cell from the end of the list is computed to obtain its rank. The reconstruction phase is simpler than the first phase since the reconstruction is performed by undoing the work performed in the first phase. Thus, the rank is computed for each cell in reverse order of their removal. For example, first the rank is found for cell e, then cells c and g, and finally for cells b, d, f, and h.

How does a processor know whether it should jump a cell? One approach is use randomization to determine whether a processors should jump a cell or not. Once a processor's cell is jumped over, it stops working on this ranking problem and is free to work on other problems until it is needed for a reconstruction phase.

On each round, a processor determines whether to jump a cell by flipping a coin and assigning a sex, either male or female. If the sex of a processor's cell is female and the next cell in the linked list has sex male then the processor can dereference its cell. Otherwise, the processor waits for the next round. We refer to this process as *random mating*. It is clear that no two adjacent cells will be jumped over in the same round. Once a cell is jumped over its processor becomes inactive until the reconstruction phase. As in Wyllie's algorithm, during the splicing out phase we maintain the cell's distance in the original linked list to its new successor by adding to its distance the distance of its old successor. At the end of the pointer jumping phase, cells either point to *nil* and have their final value for *rank* or are inactive and their rank

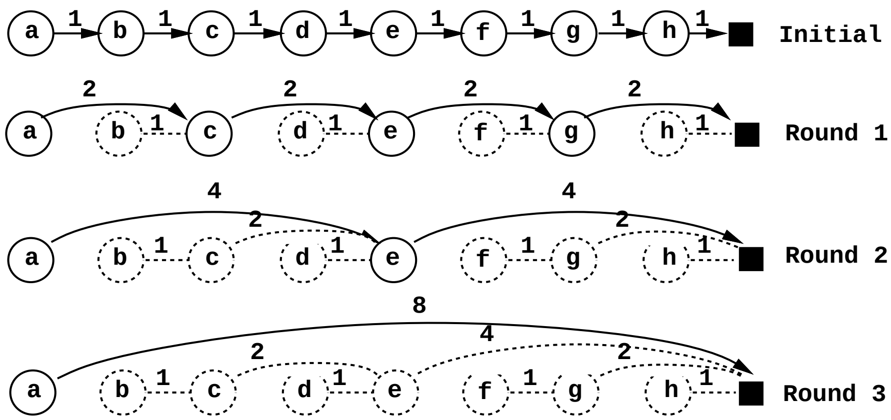

FIGURE 2.2
List ranking with minimal parallel work. The dotted cells indicate that corresponding processors were inactive during the pointer jumping.

is their distance in the original linked list to their current successor.

After the pointer jumping phase is the reconstruction phase, in which processors are reactivated in the reverse order in which they became inactive. The distance of a cell from the tail of the list is equal to the distance to its successor cell plus the distance of its successor cell to the tail of the list. Because we reactivate the processors in the reverse order in which they became inactive, when a processor is reactivated, its successor processor has already been reactivated and has its final rank. Thus, a cell's final rank is simply the rank of itself plus the rank of its successor. In order to determine when to reactivate a cell we use a time stamp, indicating which round a cell was deactivated.

Each processor runs the same program and all the processors start each iteration of the while loops at the same time. Algorithm 2.2 gives the parallel Algol code for the *Random_Mate* algorithm.

Analysis

Using this algorithm, we guarantee that no two adjacent cells in the linked list are jumped over in the same round. The probability a cell is jumped over is $\frac{1}{4}$ because, with probability $\frac{1}{2}$, $sex(i) = F$ and, with probability $\frac{1}{2}$,

ALGORITHM **2.2**

A randomized list ranking algorithm

```
Procedure Random_Mate:
    In Parallel rank[!] := 1; active[!] := true;              /* Initialize */
    host set t := 1;

    In Parallel while succ[head] ≠ nil do           /* Pointer jumping phase */
        if active[!] = true and succ[!] ≠ nil then do
            sex[!] := Random{M, F};
            if sex[!] := F and sex[succ[!]] = M then do
                time[succ[!]] := t;
                active[succ[!]] := false;
                rank[!] := rank[!] + rank[succ[!]];
                succ[!] := succ[succ[!]];
            end then
            t := t + 1;
        end then

    In Parallel while t > 0 do                    /* Reconstruction phase */
        if time[!] = t and succ[!] ≠ nil then
            rank[!] := rank[!] + rank[succ[!]];
        t := t − 1;
    end in parallel
end Random_Mate
```

$sex[succ[i]] = M$. How many rounds of random mate are needed to jump over all cells?

THEOREM 2.1

Simple Random Mate computes the rank of each element of a linked list of length n in $O(\log n)$ time using n processors on a EREW PRAM.

PROOF

Note that the cells are not statistically independent of each other. If a cell i is jumped over then, with probability one, the cell $succ[i]$ is *not* jumped over in the same round. If cell i is not jumped over then the cell $succ[i]$ is jumped over in the same round with probability $1/3$. This is because the probability that $succ[i]$ is jumped over given that i is not jumped over is equal to the probability that both the $succ[i]$ is jumped over and i is not jumped over (1/4) divided by the probability that i is not jumped over (3/4). However, the probability that a cell is not jumped over in one round is independent of whether it is jumped over in the next round. Let P_i be the probability that the i^{th} cell is still not jumped over after k rounds. Then

$$P_i = (3/4)^k, i = 2, \ldots, n$$

If we choose k so that $P_i = 1/n^c, c \geq 2$, then

$$k = c\lceil \log_{\frac{4}{3}} n \rceil$$

The probability that at least one cell has not been jumped over after k rounds is the disjunction of the probabilities that each cell is not jumped over after k rounds. This disjunction is bounded from above by the sum of the probabilities of each cell not having been jumped over after k rounds. That is,

$$\begin{aligned}
&\text{Prob (number of cells not jumped over after } k \text{ rounds} > 0) \\
&= P_1 \vee P_2 \vee \ldots \vee P_n \\
&\leq \sum_{i=1}^{n} P_i \\
&= 1/n^{(c-1)}
\end{aligned}$$

■

Thus, for large n, the probability that the algorithm runs for more than $c \log n$ rounds is small. The amount of work done is $n+\frac{3}{4}n+(\frac{3}{4})^2 n+\ldots = O(n)$, which is optimal, although the product of the runtime and processor count is not.

2.1.4 A Simple, Optimal Randomized Algorithm for List Ranking

The problem with the algorithm of the previous section is that it is not optimal in the sense that the number of processors that are actively working on the list ranking problem decreases geometrically each round. Once a processor's cell is jumped over it is free to do other work, until it is needed during the reconstruction phase. However, scheduling these freed processors with other work introduces overhead and in some Single Instruction Multiple Data (SIMD) architectures these processors must remain idle.

In this section we introduce an approach to keep a fixed set of processors busy most of the time. If we assume that each round takes $O(1)$ time, then in order to obtain an algorithm that takes $O(\log n)$ time and $n/\log n$ processors we must remove $O(n/\log n)$ cells per round.

One approach would be to simulate Random Mate using $n/\log n$ processors, by letting each processor do the work of $\log n$ virtual processors. We assign each processor $\log n$ adjacent memory cells. Note that adjacent cells in memory need not be adjacent cells in the linked list. Each processor assigns the cells in its set a sex, and jumps over all males pointed to by a female. However, a processor may be unlucky and have all its $\log n$ cells assigned female, each of which points to a male. Thus, the unlucky processor jumps over other processors' cells, but none of its own cells are eliminated. If this happens repeatedly, we get an $O((\log n)^2)$ algorithm. However, it is not unrealistic to assume that prefix sum is a unit time operation [Ble90]. In this model it is then possible to rebalance the work among the processors using prefix sum operations, while maintaining a $O(\log n)$ running time.

Another approach is to think of a processor's $\log n$ cells as a queue. A processor looks only at the top of the queue in each round. During each round of the list ranking algorithm, each processor attempts to jump over the successor of the top of its queue. When the top of a queue is jumped over, the processor moves to the next element, etc. However, we need to prevent two processors from jumping over adjacent cells in the linked list to avoid contention. Contention occurs when both i and $succ[i]$ are at the top of queues. When several tops of queues are adjacent cells in the linked list, we call the set of adjacent cells a **chain**. Cells of a chain cannot all be spliced out in the same round. Again, contention can be avoided if, during each round, each processor chooses independently and randomly a sex for the top of its queue. If during the round, two adjacent cells are at the tops of queues, then $succ[i]$ is jumped over only if i is female and $succ[i]$ is male. Since only cells that are at the top of a queue are attempting to jump cells, we can assume that all cells not at the top of a queue have a male sex. A given round is shown in Figure 2.3. A solid arrow head indicates the top of the queue.

Unfortunately, this algorithm can also become imbalanced. If a large number of tops of queues all point to queues of only a few processors, then

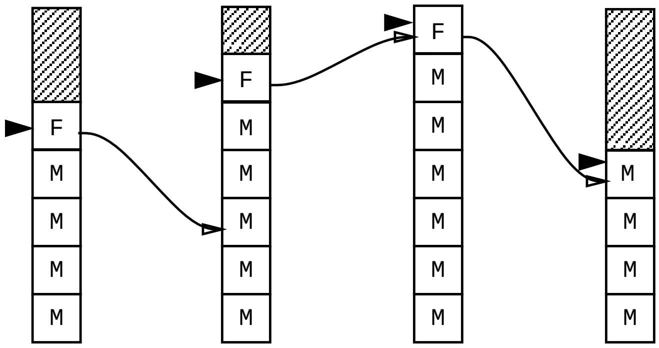

FIGURE 2.3
Random Mate using queues. Solid arrow heads indicate the top of the queues.

these few processors have most of their cells jumped over and have little work to do, while the remaining processors have almost all the work to do. If these same tops of queues again point to queues of another small number of processors, then a few more processors have little work remaining to do. Eventually a small number of processors would have all the remaining work to do and a load balancing step would be required in order to keep the running time down. Figure 2.4 illustrates how cells might be eliminated in this fashion, leading to imbalanced queues. The problem here is that some processors may be lucky and have many of their cells eliminated, either by themselves or by the work of other processors. While other processors are unlucky and work on eliminating other processors' cells and only have a few of their own eliminated.

To avoid this imbalance we let each processor **splice** out the cells assigned to *itself*. We distinguish here between *splicing* out one's *own* cell and *jumping over* a successor cell. A processor can splice its own cell if the linked list is doubly linked, *i.e.*, each cell, i, has a pointer to both its predecessor, $pred[i]$, and its successor, $succ[i]$, and the successor of the tail of the linked list and the predecessor of the head of the list point to nil. If we are splicing out cell i, we set $pred[succ[i]] := pred[i]$ and $succ[pred[i]] := succ[i]$. As with the previous algorithms the head of the list is never spliced out, and when the tail of the list is spliced out the successor of the new tail points to nil.

EXERCISE 2.1

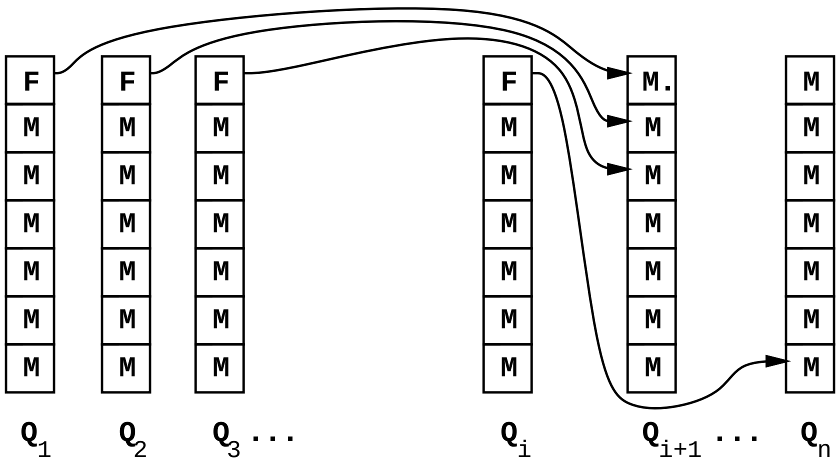

FIGURE 2.4
Imbalance caused by jumping over successor cells.

Show how, if the cells are singly linked, they can be doubly linked in $\log n$ steps using $n/\log n$ processors on an EREW PRAM.

Again, each processor is assigned $\log n$ consecutive cells in memory and treated as a queue, and the algorithm has two phases. During the first phase, all cells are spliced out of the linked list. When all the processors have completed the first phase, the second phase starts. The cells are put back into the linked list in the reverse order of their removal and the distance of the cells from the tail of the list is computed. Code for the algorithm is given in Algorithm 2.3.

The linked list is represented as a two-dimensional array, indexed by the processor ID and position in the queue. We represent this index pair using the infix operator at (@). For example the index pair 7@23 means the 23^{rd} element of processor 7's queue. If i is an index pair, we use $i.0$ to indicate the first element of the pair, namely the processor ID, and $i.1$ to indicate the second element of the index pair, namely the position of the processor's queue. Again we use ! to represent a processor's ID. The variable *top* is an index pair, where *top*[!] is the top of the queue for the current processor. We process the queue by increasing positions. That is, *top*[!] is initialized to !@1 and proceeds to !@2, and so on.

A cell is spliced out only if it is a male pointed to by a female. Since only males are spliced out, all cells not at the top can be assumed to be female. During the splice out phase, each cell maintains a record of its original distance, $rank[i]$, from the cell to which it points, so that we may reverse the first phase in order to compute the cell's final rank. Initially, $rank[i] :=$ 1 for all cells. If during a round i is spliced out, then we add $rank[i]$ to $rank[pred[i]]$. During the reconstruction phase the rank of a cell is computed in reverse order of removal. When i is added back to the linked list the rank is $rank[i] := rank[i] + rank[succ[i]]$.

Analysis

THEOREM 2.2

Optimal_Random_Mate computes the rank of each element of a linked list of length n in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM.

PROOF

The probability that the top of a queue is removed during any given round is at least $\frac{1}{4}$. To make the analysis easy, we assume that if a cell is not adjacent to another cell at the top of a queue, it only has probability of $\frac{1}{4}$ of being removed. The number of items that are removed from a queue after t rounds can be viewed as a binomial random variable $S_t^p, p = \frac{1}{4}$ (i.e., sum of t independent Bernoulli trials with success probability p). The *expected* time for a queue to become empty is at most $4\log n$. Chernoff [Ch52] shows that S_t^p is substantially less than its expected value, pt, with small probability:

$$\text{Prob}[S_t^p < (1-\beta)pt] < e^{-\beta^2 pt/2}, \text{ for } 0 \leq \beta \leq 1$$

If we take $p = \frac{1}{4}$, $t = 16\log n$, and $\beta = 3/4$, we have:

$$\text{Prob}[S_t^p < \log n] < e^{-\frac{9}{8}\log n} < 1/n$$

Thus, the probability that a particular queue is not empty after $16\log n$ rounds is less than $1/n$. Since there are $n/\log n$ queues, the probability that there is a nonempty queue at time $16\log n$ is less that $1/\log n$. It follows that the expected runtime is $O(\log n)$. ∎

ALGORITHM **2.3**

An optimal randomized list ranking algorithm

```
Procedure splice_out(i)
    rank[pred[i]] := rank[pred[i]] + rank[i];
    succ[pred[i]] := succ[i];
    if succ[i] ≠ nil then pred[succ[i]] := pred[i];
end splice_out


Procedure Optimal_Random_Mate
    In Parallel do                                              /* Initialize */
        for i =!@1 to !@⌈log n⌉ do
            rank[i] := 1; sex[i] := F;
        end for
        top[!] :=!@1;
    end in parallel
    host set sex[nil] := M; t := 1;

    In Parallel while succ[head] ≠ nil do           /* Pointer jumping phase */
        if top[!].1 ≤ ⌈log n⌉ then do
            sex[top[!]] := Random{M, F};
            if (sex[pred[top[!]]] = F and sex[top[!]] := M) then do
                splice_out(top[!]);
                top[!].1 := top[!].1 + 1; splicetime[top[!]] := t;
            end then
        end then
        t := t + 1;
    end in parallel

    In Parallel while t ≥ 0 do                       /* Reconstruction phase */
        top[!].1 := ⌈log n⌉;
        if (splicetime[top[!]] = t and succ[top[!]] ≠ nil) then do
            rank[top[!]] := rank[top[!]] + rank[succ[top[!]]];
            top[!].1 := top[!].1 − 1;
        end then
        t := t − 1;
    end in parallel
end Optimal_Random_Mate
```

It is possible to improve the runtime of this algorithm by a constant factor by splicing out elements from the linked list until there are $n/\log n$ elements left. This takes just a little over $4\log n$ rounds. Then using the $n/\log n$ processors, the elements can be spliced out using Wyllie's algorithm in time $O(\log n)$. Finally, the linked list is reconstructed in the reverse order the cells were splice out in the first phase. Reducing the problem to a smaller linked list that can use Wyllie's algorithm is a common approach to optimal list ranking algorithms. That is, the steps are:

1. Reduce the problem to $O(n/\log n)$ elements.
2. Solve the list ranking problem on the reduced linked list with Wyllie's algorithm.
3. Fill in the ranks for the remaining elements.

2.1.5 An Optimal Deterministic List Ranking Algorithm

In the previous algorithm, when the top cells of two processor queues are adjacent in the linked list, we avoided attempting to splice out both top cells simultaneously by randomly tossing a male/female coin. In this section we use a variant of the deterministic coin tossing technique devised by Cole and Vishkin for breaking symmetry in parallel algorithms. Unlike the optimal CRCW PRAM list ranking algorithms presented in the next two chapter, which use 2-ruling sets, the algorithm presented here only needs to find $\log\log n$-ruling sets to get an optimal $O(\log n)$ time $n/\log n$ processor EREW PRAM algorithm. Finding $\log\log n$-ruling sets is substantially simpler than finding 2-ruling sets, which requires a complicated sorting step, and hence much larger constants. In addition, the scheduling step of the algorithm in this section is simple and has the advantage that cells are reallocated to processors only once. With local memory architectures reallocation can add sizable overhead to an algorithm. In this section we describe the basic deterministic algorithm. In the next section we show how to find $\log\log n$ ruling sets.

The basic idea of the deterministic algorithm is the same as with the *Optimal_Random_Mate* algorithm of the previous section. Here we also assume that we have $n/\log n$ processors, each originally assigned $\log n$ continuous blocks of memory. The main difference between the two algorithms is that here we find a k-ruling set (defined below) to resolve contention instead of randomization.

The algorithm consists of three phases: First, a deterministic list ranking algorithm splices out items from the linked list until $n/\log n$ items remain. At this point in the algorithm the linked list has been reduced to an $n/\log n$ cell *weighted* linked list. Second, we use a prefix sum algorithm to assign one pointer to each of the $n/\log n$ processors and then apply Wyllie's algorithm with one processor per list cell. Finally the rank of the cells are computed by adding them back into the linked list in the reverse of the order that they were removed.

Ruling Set and Graph Colorings

In this subsection we show how to find colors and ruling sets for linked lists. They need not be doubly linked. We begin by defining the notion of a k-ruling set, which is also defined in the next chapter. We treat a linked list L as a directed graph. If $V_1, \ldots, V_n$ are the cells of L then we represent L as the directed graph $G = (\{V_1, \ldots, V_n\}, E)$, where E is the set of edges $\{e_{ij}\}$ such that V_i is linked to V_j in L.

DEFINITION

Let $G = (V, E)$ be the directed graph representation of a linked list L with vertices $V_1, \ldots, V_n$. We define a subset S of V to be a **k-ruling set** *if:*

1. *No two vertices in S are adjacent.*
2. *For each vertex V_i in V there is a directed path from some vertex in S to V_i with length at most k.*

This definition implies that there are at most k vertices between any two vertices in a *k-ruling set.* The **kingdom** of a vertex V_i in S is the list of all vertices in V that lie between V_i and the next vertex in S. The vertices in S are called the **rulers** and the vertices between rulers are called the **subjects**.

$$\underbrace{\overset{ruler}{V_1} \rightarrow \overset{sbjct}{V_2} \rightarrow \overset{sbjct}{V_3} \rightarrow \ldots \rightarrow}_{kingdom} \overset{ruler}{V_i} \rightarrow \ldots \rightarrow \overset{ruler}{V_j} \rightarrow \ldots$$

In the notation of the figure above, the vertices in the *k-ruling set* are marked with *ruler*. Note that a kingdom has at least one subject (not including the vertex in the ruling set) and at most k subjects.

The notion of *k-ruling set* is similar to the concept of a graph coloring. Let $G = (V, E)$ be an undirected graph. A map $C : V \rightarrow \{0, \ldots, k-1\}$ is a **k-coloring** of G if $(x, y) \in E$ implies that $C[x] \neq C[y]$. We first observe that given a k-coloring of the vertices, a $2k$-ruling set can be obtained in unit time using n processors. Then we show how to find a coloring for a linked list when k is small.

Let C be a k coloring of a linked list L. A *2k-ruling set* is defined as follows:

1. The head of the linked list, $head$, is a ruler.
2. Element x is a ruler if $pred[x] \neq head$ and $C[pred[x]] > C[x]$ and $C[succ[x]] > C[x]$.

Observe that the head of the linked list can have the largest possible kingdom. This happens when the the color of the head is greater than zero and its subjects have colors $0, 1, \ldots, k-1, k-2, \ldots, 1$, respectively. Note that we get a slightly simpler construction if we assume that the head has color zero. We have shown that the problem of constructing a $2k$-ruling set can be reduced to finding k-colorings.

EXERCISE 2.2
Show how to construct a k-ruling set from a k-coloring in constant time using n EREW processors.

EXERCISE 2.3
Show how to define a notion of ruling sets for an arbitrary graph. Can you get small kingdoms from a coloring using a small number of colors?

Small Colorings

We say that a coloring C is an m-bit coloring if each color is written as a length m binary string of zeros and ones. Let C be a m-bit coloring of our linked list L. For example, we could use the processor ID as the coloring, where $m = \log P$ and P is the number of processors. We assume that $head$ has color zero, i.e, a bit string of m zeros. Since C is a coloring there must be some bit of $C[x]$ which is not equal to the corresponding bit of $C[pred[x]]$. Let $B'[x]$ be the index of this differing bit and, for simplicity, let $B'[x]$ be the smallest such index. Thus, $B'[x]$ is a number between 0 and $m-1$ for $x \neq head$. We write $B'[x]$ in binary. Let $B[x] = a \cdot B'[x]$, where a is the

$B'[x]^{th}$ bit of $C[x]$ and $\cdot$ is concatenation. $B[x]$ simply describes the first bit of $C[x]$ that differs from the corresponding bit of $C[pred[x]]$. A $\lceil \log m \rceil + 1$-bit coloring C' is obtained as follows:

$$C'[x] = \begin{cases} B[x] & \text{if } x \neq head \\ 0 & \text{if } x = head \end{cases} \tag{2.1.1}$$

EXERCISE 2.4
Show how to obtain a $\log \log n$-bit coloring in constant time.

EXERCISE 2.5
Show that equation 2.1.1 also correctly colors rooted trees where *head* is now the root.

EXERCISE 2.6
Show how to modify equation 2.1.1 to properly color bounded degree graphs.

Deterministic List Ranking

As before, each of $n/\log n$ processors is assigned $\log n$ cells to remove. When a cell is at the top of a queue and neither its successor nor its predecessor are at the top of a queue, we call the cell an **isolated cell**. There is no contention with isolated cells. The difficulty comes when the cell is part of a chain of cells at the top of several queues. Before, we broke the symmetry by flipping a male/female coin and spliced out a cell when it was a male to which a female pointed. Otherwise the processor remained idle until the next round. Here we break symmetry by using deterministic coin tossing to obtain a $\log \log n$ ruling sets among the cells. Since a chain can have at most $n \log n$ cells and we have $n \log n$ processors, we can use the results of the previous section to obtain $\log \log n$ ruling sets in constant time.

Ruling sets divide a chain of cells into sublists where the head is the *ruler* and the remaining cells are the *subjects*. The key idea is that the rulers are assigned the task of splicing out *all* of their subjects. This means that each processor that finds a subject at the top of its queue can assume that its subject will be spliced out by another processor. Therefore, it can skip over its subject by adjusting the top of the queue to the next cell in the queue. In the next round, this processor can then continue working using its new top. In the mean time, each ruler splices out one subject per round, for up to $\log \log n$ rounds.

A processor stops working when, either its queue becomes empty, or it has executed $5\log n$ rounds. As we show later, $5\log n$ rounds is sufficient to reduce the total number of cells remaining to at most $n/\log n$ cells. Once all processors have stopped, then the remaining cells are allocated to the $n/\log n$ processors, one cell each, and the processors proceed with Wyllie's algorithm. Finally, there is the reconstruction phase.

We give a more formal description of splice out phase of the algorithm by giving the code that each processor executes in Algorithm 2.4. As before, the queue for the processor is represented by an array, with *top* pointing to the cell at the top of the queue. The linked list is assumed to be doubly linked. The subprocedure *splice_out* is the same as in the *Optimal_Random_Mate* algorithm.

A cell can have the status $\{A, I, R, S\}$ denoting active, inactive, ruler, or subject, respectively. Initially, the top of each queue has status *active* and all other cells have status *inactive*. At the beginning of each round the top of a queue can either be active or a ruler.

If the top of a queue is active, the first step is to determine whether it is part of a chain. If it is, it calls the subprocedure *Find_Ruling_Sets*, which subdivides chains into short linked lists, linked by *next*, such that the heads of the linked lists are rulers and the remaining cells are subjects. At this point the top of a queue is either active, a subject or a ruler. Active cells do not have adjacent cells that are active and, therefore, can splice themselves out and advance the top of the queue to the next cell in the queue. Subjects advance the queue top to the next cell in the queue since they will be spliced out by their ruler.

Finally rulers splice out one of their subjects. Once a ruler has spliced out all of its subjects it puts itself back into the active status, and in the next round checks whether it is part of a new chain. Figure 2.5 shows an example of the status of the elements of a linked list after one application of *Deterministic_List_Ranking*.

EXERCISE 2.7

Can the three cases for **if** *status*... be run in parallel? If not, show when parallel execution fails.

The correctness of the algorithm follows from the facts that adjacent cells are never removed at the same time, and every cell is eventually looked at by a processor. In the next section we discuss the scheduling which ensures that the running time of the algorithm is $O(\log n)$.

ALGORITHM **2.4**

The splice out phase of an optimal deterministic list ranking algorithm

```
Procedure increment_top
    top[!].1 := top[!].1 + 1;
    if top[!].1 < log n then if pred[top[!]] = nil then
        top[!].1 := top[!].1 + 1;              /* Don't splice out head of list */
    if top[!].1 < log n then status[top[!]] := A;
end increment_top

Procedure Deterministic_List_Ranking
    In Parallel do                                          /* Initialize */
        for i = 1 to ⌈log n⌉ do
            rank[!@i] := 1; status[!@i] := I;
        end for
        top[!] :=!@0; increment_top;
    end in parallel
    host set status[nil] := I;

    In Parallel for t = 1 to 6 log n do
        if top[!].1 < log n then do                     /* Subdivide chain */
            if status[top[!]] = A and
             (status[pred[top[!]]] = A or status[succ[top[!]]] = A) then
                status[top[!]] := Find_Ruling_Sets(top[!]);

            if status[top[!]] = A then do
                splice_out(top[!]);                /* Splice out isolated cell */
                increment_top;
            end then

            if status[top[!]] = S then
                increment_top;                    /* Advance top past subjects */

            if status[top[!]] = R then do
                splice_out(next[top[!]]);              /* Splice out a subject */
                if next[!] = nil then                  /* No more subjects */
                    status[top[!]] := A;
            end then
        end then
    end Parallel for
end Deterministic_List_Ranking
```

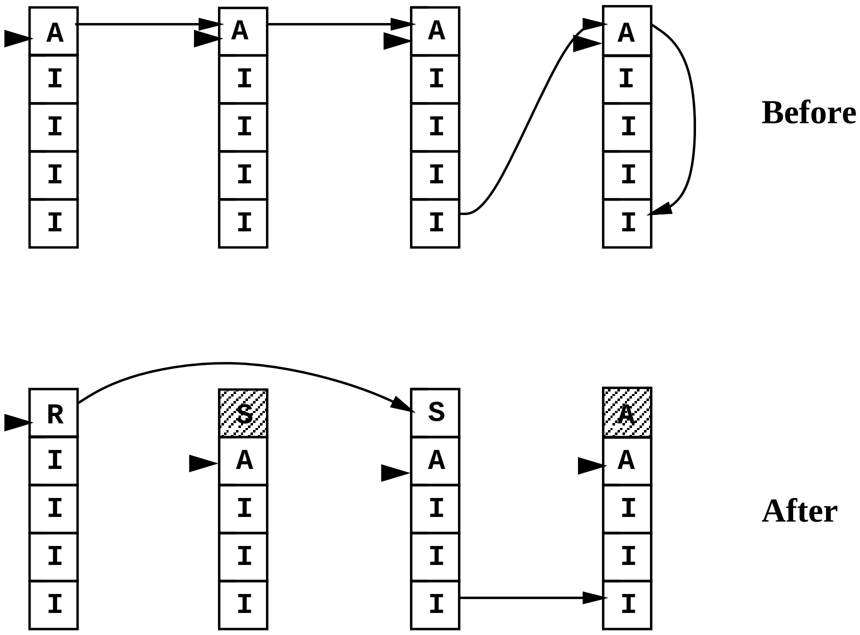

FIGURE 2.5
Status of cells after one application of *Deterministic_List_Ranking*.

Scheduling

In order to keep all the processors busy we need to be sure that the queues become empty at approximately the same time. Queues can become imbalanced because the length of a queue is not reduced while its top is a ruler removing subjects. The solution to this problem has two parts: we insure that chains do not become too long by breaking them into $\log\log n$ ruling sets, and we insure that rulers are assigned to shorter queues as opposed to longer queues.

In section on small colorings we showed how to find $\log\log n$ ruling sets. In this section we consider the second problem, that a long queue could be assigned a ruler. A processor needs to splice out all of its subjects before it can continue working on its own queue, which can lead to an imbalance in queue lengths. We would like the processor with the smallest queue to become a ruler. Therefore, we modify *Find_Ruling_Sets* to subdivide chain into ones that have either monotone increasing or monotone decreasing queue lengths and then find $\log\log n$ ruling sets within the monotone chains. If a ruling set has monotone decreasing queue lengths, we reverse the pointers in constant time so that the chain is traversed backwards instead of forwards to splice out elements. This allows rulers to have minimum height queues and remove cells working uphill. Algorithm 2.5 shows the code for this modified version of *Find_Ruling_Sets* and its subprocedure *Find_Increasing_Chains*. Recall that *Dererministic_List_Ranking* calls *Find_Ruling_Sets* with $top[!]$ as an argument, which is an index pair for two dimensional arrays. The code for *Find_Ruling_Sets* uses the parameter i for this index pair.

Find_Increasing_Chains subdivides each chain into subchains that are monotone with respect to queue lengths; each subchain is linked through the pointers $next$ and the tails point to nil. For those parts of the chain that are monotone increasing or strictly equal in queue length, $next$ is the same as the $succ$ pointer. For parts that are strictly decreasing, $next$ is the same as the $pred$ pointers. In this way, as one traverses these subchains along the $next$ pointers the queue lengths increase monotonically.

We use the boolean variable $inchain$ to mark which cells are part of a chain and which are singletons or inactive. Those cells that do not have $inchain$ set to $true$ are not in a chain. We initialize all $next$ pointers to nil. The boolean $covered$ indicates whether we have determined to which subchain the cell belongs and we initialize it to $false$. First we find subchains from the parts of the chain that have monotone increasing queue lengths. Recall that $top[i.0].1$ is the length of the queue $i.0$. The $next$ pointers are set in the forward direction using the $succ$ pointers. The local maxima are the tails of the these subchains. We mark cells in these subchains, including the tails, as $covered$. The remaining uncovered cells have strictly increasing queue lengths. If a pair of cells are uncovered, the second cell in the pair sets its $next$ pointer to the $pred$ so that the pointers go in the reverse direction. Notice that the first or last cell in the original chain may become a singleton chain.

Find_Ruling_Sets takes these monotone chains and subdivides them into $2\log\log n$ rulings sets. First it finds a $\log\log n$ colorings as shown in Section 2. Then it breaks the chains at local minimum colorings, so that chains are at most $2\log\log n$ long. Finally it set the heads of the subchains as rulers and the remaining cells as subjects. Any ruler that does not have a subject is isolated from active cells so it too is set to active.

The effect of these modifications is to have the following two constraints on rulers and their chain of subjects: a ruler never has more that $2\log\log n$

ALGORITHM **2.5**

Algorithm to find chains with monotone increasing queue lengths

```
Procedure Find_Increasing_Chains(i)
    In Parallel do

        inchain[i] := true; next[i] := nil; covered[i] := false;
        if succ[i] ≠ nil then if inchain[succ[i]] then do
            if top[i.0].1 ≤ top[succ[i].0].1 then do
                next[i] := succ[i];          /* Increasing or equal queue lengths */
                covered[i] := covered[next[i]] := true;
            end then
            else                                    /* Decreasing queue lengths */
                if not (covered[i] or covered[pred[i]]) then
                    next[i] := pred[i];                      /* Reverse pointers */
        end then
    end in parallel
end Find_Increasing_Chains


Procedure Find_Ruling_Sets(i)
    In Parallel do
        Find_Increasing_Chains(i);
        Find_Coloring(i);
                                            /* Subdivide chains at local minima */
        if next[i] ≠ nil then if next[next[i]] ≠ nil then
            if (color[i] > color[next[i]] and
                    color[next[i]] < color[next[next[i]]]) then
                next[i] := nil;
        status[i] := R;                                     /* Determine status */
        if next[i] ≠ nil then status[next[i]] := S;
        if next[i] = nil and status[i] = R then status[i]] = A;
    end in parallel
end Find_Ruling_Sets
```

subjects, and the height of a ruler is no greater that the height of its subjects. The performance analysis given in the next section shows that these conditions are sufficient to make the list ranking algorithm an $O(\log n)$ algorithm.

There is one final detail. Because we have reversed the direction that some rulers splice out their subjects, we can have two rulers that splice out adjacent subjects. Therefore, we modify *Deterministic_List_Ranking* so that when rulers splice out their subjects they use the subprocedure *splice_Next* instead of *splice_Out*. The *splice_Next* algorithm, shown in Algorithm 2.6, is written from the point of view of the ruler, who is actually doing the work and lets rulers who have chains running forward to splice first, and then rulers with backward chains to splice next.

ALGORITHM **2.6**

A ruler splicing out its next subject

```
Procedure splice_Next(i);
    In Parallel do
        if next[i] = succ[i] then do                          /* Forward chains */
            rank[i] := rank[i] + rank[succ[i]];
            succ[i] := succ[succ[i]];
            if succ[i] ≠ nil then pred[succ[i]] := i;
        end then
        else do                                               /* Backward chains */
            rank[pred[pred[i]]] := rank[pred[pred[i]]] + rank[pred[i]];
            pred[i] := pred[pred[i]];
            succ[pred[i]] := i;
        end else
        next[i] := next[next[i]];
    end in parallel
end splice_Next
```

Analysis

In order to analyze the running time of these algorithms, we provide an amortization scheme that assigns weights to the items for which a processor is responsible. Then we show that after a given round the weight of the entire system is reduced by a constant factor. In order to determine the efficiency of this algorithm, we use the following accounting scheme. We think of a queue as being stacked vertically, and the height of an element as its distance from the bottom of the queue. We refer to the height of the queue as the number of elements in the queue remaining to be processed. For a given processor queue, each item is assigned a weight relative to its position in the queue. The i-th item from the top of the queue is assigned weight $(1-\alpha)^i$, where $\alpha = 1/\log\log n$.

The weight change to the system is as follows:

1. the removal of an isolated item removes the item's entire weight.
2. the removal of a subject removes half its weight.
3. the identification of a subject removes half its weight.
4. the identification of a ruler removes nothing.

We show that each round reduces the total weight by a factor of at least $1-\frac{\alpha}{4}$. This rate of reduction allows us to bound the number of rounds required to reduce the number of items remaining in the linked list to $n/\log n$.

LEMMA 2.1

A single round of the algorithm reduces the weight by a factor of at least $1-\frac{\alpha}{4}$.

PROOF

To facilitate the argument, we use the following bookkeeping trick: the weight of a processor's queue is the sum of the weights of the remaining items on the queue, plus one half the initial weights of any subjects for which it is responsible. For example, in Figure 2.6, queue Q_i has weight $(1-\alpha)^2 + (1-\alpha)^3 + (1-\alpha)^4 + \frac{1}{2}(1-\alpha)^2 + \frac{1}{2}(1-\alpha)^1$, with the first three terms coming from the items on queue Q_i and the fourth and fifth terms coming from the subjects on queues Q_j and Q_k respectively.

The following facts are useful in the analysis that follows:

1. the number of queues is $n/\log n$.
2. Initially, the weight of each queue is:

$$\sum_{i=0}^{\log n}(1-\alpha)^i \leq \sum_{i=0}^{\infty}(1-\alpha)^i = 1/(1-(1-\alpha)) = 1/\alpha = \log\log n.$$

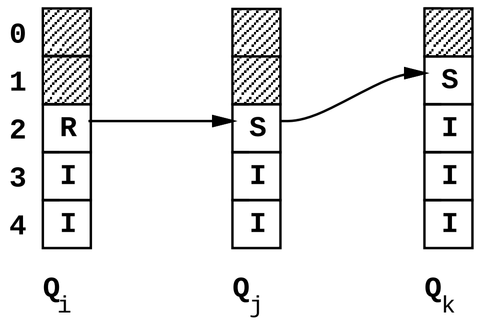

FIGURE 2.6
The weight of Q_i is equal to the weight of the remaining items on the queue plus half the weights of the subjects of the chain.

3. Initially, the total weight of the system is:

$$\leq \frac{n \log\log n}{\log n}.$$

In each round of the algorithm a processor either 1) identifies the top of the queue as an isolated cell, which it splices out, 2) identifies the elements of a chain as either subjects or rulers, or 3) finds the top of its queue is a ruler and splices out one subject. Thus, we divide the total weight into three corresponding categories: 1) isolated active cells and the inactive cells below them, 2) active cells in chains and the inactive cells below them, and 3) rulers and inactive cells below them and subjects, as shown in Figure 2.7. Every cell remaining in the system can be placed in exactly one of these categories. We examine the rate at which the weight is reduced for each case during one round of the algorithm, and show that each case reduces the weight of its category by at least a factor of $1 - \frac{\alpha}{4}$, and hence the total weight by at least a factor of $1 - \frac{\alpha}{4}$.

Case 1. Removal of an isolated cell

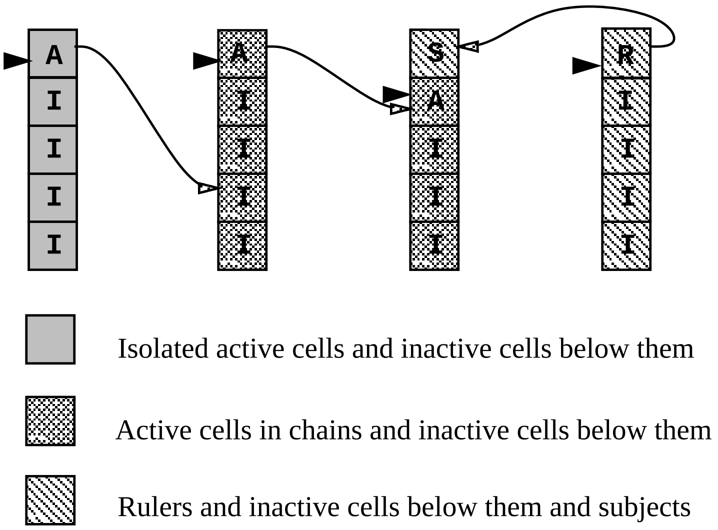

FIGURE 2.7
Categories to which cells belong.

For each isolated cell, assume the weight of the cell is $(1-\alpha)^i$. Initially, the weight of the queue is

$$Before = \sum_{j=i}^{\log n} (1-\alpha)^j.$$

After the cell is removed the weight of the queue becomes

$$After = \sum_{j=i+1}^{\log n} (1-\alpha)^j.$$

$$\begin{aligned}
\frac{After}{Before} &= \frac{Before - (1-\alpha)^i}{Before} \\
&= 1 - \frac{(1-\alpha)^i}{Before} \\
&\leq 1 - \frac{(1-\alpha)^i}{(1-\alpha)^i \sum_{j=0}^{\infty} (1-\alpha)^j}
\end{aligned}$$

$$= \quad 1 - \alpha$$

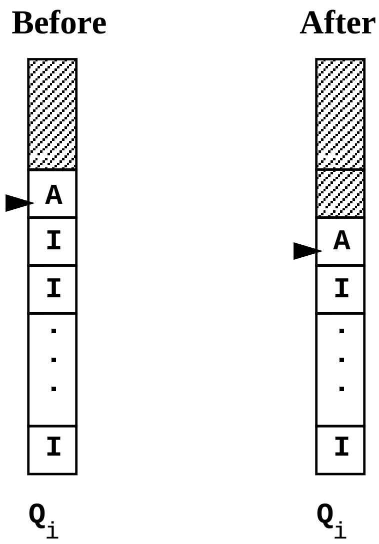

FIGURE 2.8
Removal of an isolated cell.

Case 2. Identification of a Subject

We account for the weight of all of the queues associated with the chain. The queues that have cells that become subjects lose one cell each and the queue that has the ruler picks up the subject cells. Cells that are identified as subjects lose half of their weight. When identifying subjects, the weight of the ruler's queue will reduce the least when the ruler has only one subject and both the ruler and the subject are at the same height, i. Initially, the weight of the two queues is

$$Before = 2 \sum_{j=i}^{\log n} (1-\alpha)^j.$$

After the cell is identified as a subject and half of its weight is removed from system the weight of the two queues becomes

$$After = Before - \frac{1}{2}(1-\alpha)^i$$

Therefore,

$$\begin{aligned}\frac{After}{Before} &= 1 - \frac{\frac{1}{2}(1-\alpha)^i}{Before} \\ &\leq 1 - \frac{\frac{1}{2}(1-\alpha)^i}{2(1-\alpha)^i\frac{1}{\alpha}} \\ &= 1 - \frac{\alpha}{4}\end{aligned}$$

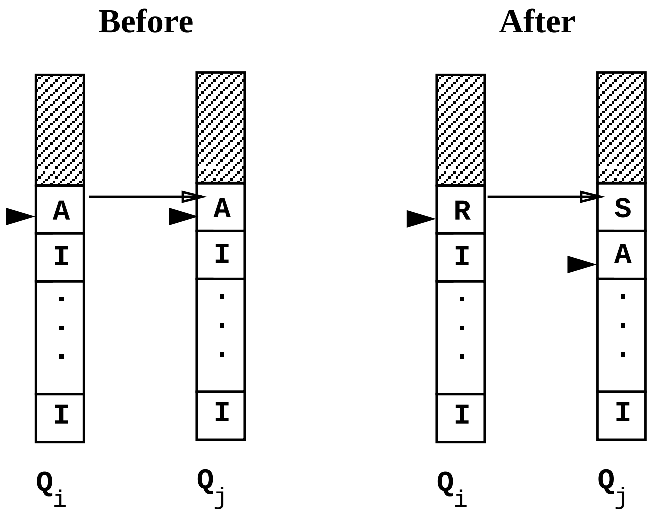

FIGURE 2.9
Identification of a subject.

Case 3. Removal of a Subject

When a subject is removed only the weight of the ruler's queue changes. Recall that in our chains, the ruler has the shortest queue.

For the purpose of the analysis, we can think of a ruler as splicing out the subject with greatest height first. In this way, the weight changes the least when the ruler has the maximum number of subjects, $\log\log n$, and each subject is at the same height as the ruler. Initially, the weight of the ruler is the weight of the ruler's queue plus the weight of the $\log\log n$ subjects:

$$Before = \sum_{j=i}^{\log n} (1-\alpha)^j + \frac{1}{2}\log\log n (1-\alpha)^i.$$

Afterward, half of the initial weight of a subject is removed from system, and the weight of the two queues becomes

$$After = Before - \frac{1}{2}(1-\alpha)^i.$$

Therefore,

$$\begin{aligned} \frac{After}{Before} &= 1 - \frac{\frac{1}{2}(1-\alpha)^i}{\sum_{j=i}^{\log n}(1-\alpha)^j + \frac{1}{2}\log\log n(1-\alpha)^i} \\ &\leq 1 - \frac{\frac{1}{2}(1-\alpha)^i}{(1-\alpha)^i\frac{1}{\alpha} + \frac{1}{2\alpha}(1-\alpha)^i} \\ &= 1 - \frac{\alpha}{3}. \end{aligned}$$

Hence, as long as the queues are not empty every queue is reduced by a factor of at least $(1-\frac{\alpha}{4})$.

■

THEOREM 2.3

The number of list cells remaining after $6\log n$ *applications of Deterministic_List_Ranking is at most* $n/\log n$*.*

PROOF

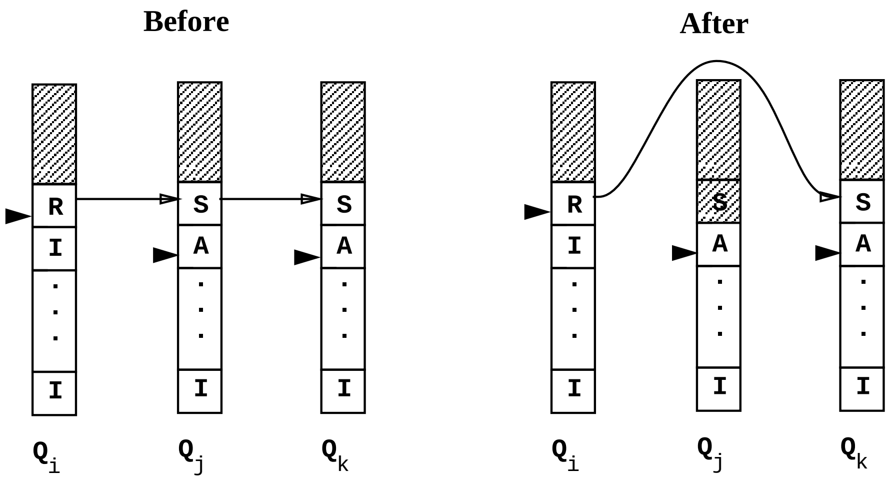

FIGURE 2.10
Removal of a subject.

First we show

$$\alpha > \left(1 - \frac{\alpha}{4}\right)^{\log n}. \tag{2.1.2}$$

For sufficiently large n

$$\begin{aligned} \log n &> 4(\log\log n)^2 \\ &> 4 \log\log n \cdot \log\log\log n \\ &= \frac{4}{\alpha} \log\log\log n. \end{aligned}$$

Thus,

$$\log\log\log n < \frac{\alpha}{4} \log n.$$

Exponentiating both sides:

$$\log\log n < e^{\frac{\alpha}{4}\log n}. \tag{2.1.3}$$

Taking reciprocols and using the facts that $\lim_{\alpha\to 0}(1 - \alpha/4)^{1/\alpha} = e^{-1/4}$ and $(1 - \alpha/4)^{1/\alpha}$ is monotone decreasing:

$$\begin{aligned}
\alpha &> e^{-\frac{1}{4}\alpha\log n} \\
&> \left[\left(1-\frac{\alpha}{4}\right)^{1/\alpha}\right]^{\alpha\log n} \\
&= \left(1-\frac{\alpha}{4}\right)^{\log n}.
\end{aligned}$$

Recall by the facts given at the previous lemma that the total weight of the system is less than $\frac{n}{\log n}\frac{1}{\alpha}$. Since the total weight is reduced by a factor of at least $\left(1-\frac{\alpha}{4}\right)$, the weight after $6\log n$ rounds is at most

$$\begin{aligned}
\frac{n}{\log n}\frac{1}{\alpha}\left(1-\frac{\alpha}{4}\right)^{6\log n} &< \frac{n}{\log n}\left(1-\frac{\alpha}{4}\right)^{5\log n} \\
&< \frac{n}{\log n}\left(1-\alpha-\frac{\alpha}{4}+\frac{5\alpha^2}{8}-\frac{5\alpha^3}{32}\cdots\right)^{\log n} \\
&< \frac{n}{\log n}(1-\alpha)^{\log n}.
\end{aligned}$$

Since there are $n/\log n$ queues and the smallest weight of an item is $(1-\alpha)^{\log n}$, there are at most $n/\log n$ cells remaining after $6\log n$ rounds. ∎

We have shown that by using $n/\log n$ processors we can reduce a linked list of size n to one of size $n/\log n$ in $O(\log n)$ time. Now we can apply Wyllie's algorithm to reduce the linked list to a single cell. Therefore, this algorithm is optimal up to a constant factor.

EXERCISE 2.8
Show that if a chain is *strictly* increasing in queue size, then it is not necessary to find $\log\log n$ ruling sets. Write an algorithm taking advantage of this observation. Note: it is still necessary to find $\log\log n$ ruling sets for chains that have constant size queues.

EXERCISE 2.9
By adjusting how much identifying a subject reduces a queue's weight and refining the analysis in Theorem 2.3, show that only $4\log n$ rounds are necessary to reduce the number of list cells to at most $n/\log n$.

Conclusion

In this section we introduced a number of list ranking algorithms. The simplest and, probably the most practical, is an algorithm due to Wyllie. It is not optimal, with respect to the work done, but has very small constants. The next algorithm, Random Mate, introduced randomization to resolve contention. This algorithm is also very simple and, although optimal with respect to the work done, still requires n processors. The Optimal Random Mate algorithm reduces the number of processors needed to $n/\log n$ by providing a simple scheduling scheme in addition to randomization. Finally, we gave an optimal deterministic list ranking algorithm that is quite simple relative to other deterministic algorithms in the literature. The randomization algorithms tend to have much smaller constants than deterministic algorithms. However, on some architectures getting the random bits to the processors fast enough may limit their practicality.

TABLE 2.1
Time and processor count for the list ranking algorithms discussed in this section.

Problem	**Time**	**Processors**	**Work**
Wyllie's algorithm	$O(\log n)$	n	$O(n \log n)$
random mate	$O(\log n)$	n	$O(n)$
optimal random mate	$O(\log n)$	$n/\log n$	$O(n)$
optimal deterministic algorithm	$O(\log n)$	$n/\log n$	$O(n)$

2.2 Parallel Tree Contraction

2.2.1 Top Down verses Bottom Up Tree Algorithms

Trees play a fundamental role in many computations, both for sequential as well as parallel problems. For sequential algorithms the classic paradigms for trees are depth-first-search and breadth-first-search. However, for processor efficient parallel algorithms, depth-first-search and breadth-first-search have not appeared to be successful in general. Typically, parallel algorithms involving trees use either divide and conquer or parallel tree contraction. Divide and conquer is a "top-down" approach, where first one finds a vertex that separates the tree into two subtrees roughly the same size, and recursively solves the two subproblems. A now classic example is Brent's work of parallel evaluation of arithmetic expressions [Bre74]. In this case one subtree is a proper expression tree, which can fully determine its value, while the other subtree has a "scar", where it has to wait for the value of the first subtree before it can finish computing its own value. The main problem with the divide and conquer approach is finding the separators that separate the tree into components with size not more than $2/3$ of the original size. If we require that we find the separators on-line, that is no preprocessing is allowed, then finding the separators seems to add a factor of $\log n$ to the running time of most algorithms. The second approach is to use parallel tree contraction, which is a technique for constructing parallel algorithms on trees working from the bottom up. That is, all modifications to the tree are done locally. This "bottom-up" approach, which is called CONTRACT has two major advantages over the top-down approach: (1) the control structure is straightforward and easier to implement, facilitating new algorithms using fewer processors and less time; (2) problems for which it was too difficult or too complicated to find polylog parallel algorithms are now easy. It has already been applied to finding small separators for planar graphs in parallel [Mil86] as well as numerous other applications [MR90].

We start by introducing the basic parallel tree contraction paradigm and show that it takes $O(\log n)$ contractions to reduce a tree to its root. Processors assigned to each vertex in the tree work on the tree from the bottom up. During each contraction, they process the leaves in parallel and then removes them from the tree, creating new leaves that are processed at the next round. Removing leaves is called the RAKE operation. Clearly, removing leaves is not sufficient for a fast algorithm; a tree that is a simple list would take a linear number of rounds to reduce the tree to a point. Thus, a second operation is introduced, called COMPRESS that reduces a chain of vertices, each with a single child, to a chain of half the length. Like list ranking, COMPRESS uses pointer jumping. Since RAKE and COMPRESS work on different parts of the tree, they can be run simultaneously. As the algorithm is running, the RAKE operation tends to produce chains that the COMPRESS operation then reduces. Thus, enough processor are kept busy to make the algorithm run quickly. The advantage of this approach to tree based parallel algorithm design is that the processing of leaves can be designed separately from the processing of the internal nodes with single children, simplifying the algorithms.

Initially we restrict ourselves to trees with bounded degree and consider the changes necessary for unbounded degrees at the end. First we describe the abstract parallel tree contraction paradigm and then we give the basic form of the implementation on a CRCW PRAM. As an example of its use, we implement arithmetic expressions evaluation. However, the algorithm is not optimal, in that it uses $O(\log n)$ time and n processors, and uses concurrent reads and writes. As with list ranking, we can reduce the number of processors to $n/\log n$ by using randomization. But unlike list ranking, it seems that a complicated load balancing step is also required. Therefore, we do not show it here.

When the tree is restricted to being binary there is a simple optimal EREW PRAM algorithm, which we present next. Each RAKE operation is immediately followed by a COMPRESS operation so that chains never form. The trick is to work on alternate leaves so that processors do not interfere with each other. Determining which leaves to work on is made easy using prefix-sums.

Next we present an optimal EREW PRAM algorithm for any bounded-degree tree structure. It consists of two stages. The first stage subdivides the tree into subtrees and assigns each subtree to a processor. A processor then reduces its subtree to a single vertex. In this way P processors reduce a tree of size n to one of size P in $O(n/P)$ time. The second stage contracts the tree of size P to its root in $O(\log P)$ time. Since the size of tree has been reduced by the first stage, a processor can be assigned a single vertex; thus the second stage of the algorithm in and of itself need not be optimal. The trick is to ensure that no concurrent reads or writes take place.

Finally, we consider the modifications required by the basic and optimal tree contractions algorithms when the tree has unbounded degree. For the nonoptimal contraction algorithms the concern is that the RAKE operation may not be constant time, but depends on the number of children of a node, which is unbounded. For the optimal contraction algorithm, the assignment of subtrees in the first stage needs to be modified so that no processor gets an unbounded number.

In order to simplify the code, we stop using the ! notation in this section and use the index v to refer to a vertex of the tree. We leave it to the implementor to modify the code to handle the indexing by each processor.

2.2.2 The RAKE and COMPRESS Operations

In this section we introduce two abstract parallel tree contraction operations RAKE and COMPRESS, and note how they reduce the size of a tree. We then show that if both operations are applied $O(\log n)$ times to a tree, the tree reduces to a point. In the next section we give a suboptimal deterministic CRCW PRAM implementation using $O(\log n)$ time and $O(n)$ processors.

Let $T = (V, E)$ be a rooted tree with n vertices and root r. In order to describe the RAKE and COMPRESS operations, we first introduce the definition of a chain.

DEFINITION

Let $v_1, \ldots v_k$ be vertices of a rooted tree. Then $v_1, \ldots, v_k$ is **chain** *of length k if:*

- *v_{i+1}, is only child of v_i $1 \leq i < k$, and*
- *v_k has only 1 child and it is not a leaf.*

A chain is **maximal** *if it is not possible to add more vertices to the chain.*

We now define RAKE and COMPRESS and introduce a new operation, CONTRACT.

RAKE: Let RAKE be the operation that removes all leaves from T. An example of a single RAKE operation is shown in Figure 2.11. It is easy to see that if the tree is highly imbalanced, for example a simple linked list, RAKE would need to be applied a linear number of times in order to reduce T to a single vertex. We can circumvent this problem by adding one more operation.

COMPRESS: In one parallel step, we *compress* all chains by identifying v_i with v_{i+1} for i odd and $1 \leq i < k$, whenever $v_1 \ldots v_k$ is a chain. Thus, the chain $v_1, \ldots, v_k$ is replaced with a chain $v'_1, \ldots, v'_{\lceil k/2 \rceil}$. Let COMPRESS be the operation on T which "compresses" all maximal chains of T in one step. Observe that a maximal chain of length one is not affected by COMPRESS. An example of the COMPRESS operation is shown in Figure 2.12.

CONTRACT: Let CONTRACT be the simultaneous application of RAKE and COMPRESS to the entire tree. We next show that the CONTRACT operation needs only be executed $O(\log n)$ times to reduce T to its root. In particular, we show:

$$|\text{CONTRACT (T)}| \leq \frac{4}{5}|T|$$

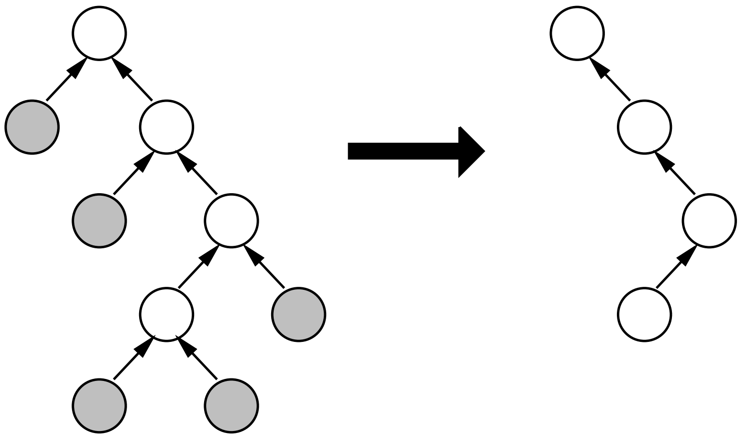

FIGURE 2.11
Result of a single RAKE operation. The shaded nodes of the tree on the left are the nodes that are deleted in order to obtain the tree on the right.

THEOREM 2.4

After $\lceil \log_{5/4} n \rceil$ *applications of* CONTRACT *to a tree with* n *vertices, the tree is reduced to its root.*

PROOF

We partition the vertices of T into two sets Ra and Com such that $|Ra|$ decreases by a factor of $4/5$ after an execution of RAKE and $|Com|$ decreases by a factor of $1/2$ after COMPRESS.

Let

V_0 be the set of leaf nodes of T.
V_1 be the set of nodes with 1 child.
V_2 be the set of nodes with with 2 or more children.

Next, we subdivide V_1 into:

$C_0 = \{v \in V_1 \mid v's \text{ child is in } V_0\}$.
$C_1 = \{v \in V_1 \mid v's \text{ child is in } V_1\}$.
$C_2 = \{v \in V_1 \mid v's \text{ child is in } V_2\}$.

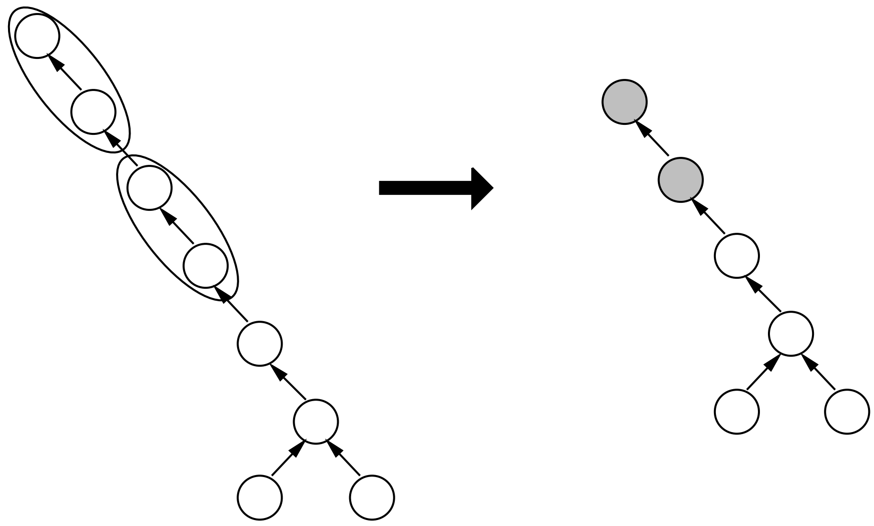

FIGURE 2.12
Result of a single COMPRESS operation. The shaded nodes represent the nodes that replace the pairs of nodes in the tree on the left.

Finally, we consider a subset of C_1:

$$GC_0 = \{v \in C_1 \mid v's \text{ grandchild is in } V_0\}.$$

All vertices in V_1 except those of C_0 belong to a chain, by definition of a chain. In order for Com to decrease by a constant factor after each COMPRESS, we want to exclude chains of length one. Notice, for example, in Figure 2.13 all the chains are of length one and COMPRESS does not remove any vertex. These chains consists of a vertex in either C_2 or in GC_0. Therefore, we exclude C_2 and GC_0 from Com. We put all the remaining vertices in Ra.

Thus, let

$$\begin{aligned} Com &= V_1 - C_0 - C_2 - GC_0 = C_1 - GC_0 \\ Ra &= V - Com = V_0 \cup V_2 \cup C_2 \cup C_0 \cup GC_0 \end{aligned}$$

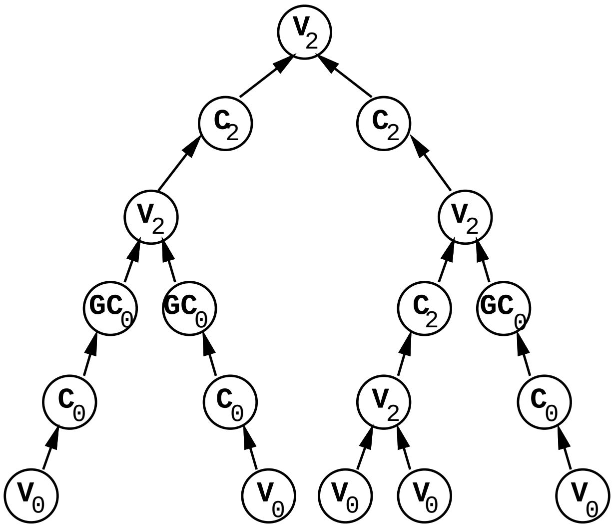

FIGURE 2.13
Vertex classes of a tree.

In this way, every vertex of Com belongs to some maximal chain. Because we have excluded C_2 and GC_0 from Com, if $v_1, \ldots, v_k$ are the vertices of a maximal chain then $v_1, \ldots, v_{k-1}$ are the only elements in the chain belonging to Com. Thus, the number of elements in Com decreases by at least a factor of 1/2 after COMPRESS.

A RAKE operation removes all vertices in V_0. To see that the size of Ra decreases by a factor of 1/5 after each RAKE we show that $|Ra| \leq 5|V_0|$. But this inequality follows by observing the following inequalities: $|C_0| \leq |V_0|$, $|GC_0| \leq |V_0|$, and $|C_2| \leq |V_2| < |V_0|$. ∎

2.2.3 The Basic Tree Contraction Algorithm

In this section we describe in more detail a CRCW PRAM implementation of CONTRACT. This basic algorithm requires $O(n)$ processors to achieve $O(\log n)$ time. For now we assume that the trees are of bounded degree. The analysis of parallel tree contraction on trees of unbounded degree is in

Section 2.2.6. First we describe a particular application, namely expression evaluation in more detail. Next we show that by reversing the contraction algorithm we can expand the tree back to its original structure and in the process compute results for every vertex, not just the root. Thus, we use the same resource bounds to compute over all subtrees as to compute over the tree as a whole. This result is a natural generalization of parallel prefix evaluation [LF80, Fic83, Vis84]. Finally, we give several applications and their implementations.

There are many useful applications of parallel tree contraction and expansion. For each given application, we associate a certain procedure with each RAKE and COMPRESS operation, which we assume can be computed in parallel in constant time. We denote applying these procedures on a vertex v by *rake*(v) and *compress*(v). Typically the vertices of the tree T contain variables storing information relevant to the given application. The *rake* and *compress* procedures modify these variables while the overall CONTRACT procedure modifies the tree structure itself.

Let T be a rooted tree with vertex set $V =$, $|V| = n$, and root $r \in V$. We view each vertex that is not a leaf as a function to be computed. The children of the vertex supply all the information needed to compute its function. Initially, only children that are leaves can supply the necessary information to their parents. We first need to consider how a vertex determines whether it is a leaf vertex, which can evaluate its function, or it is a vertex with only one missing argument and potentially part of a chain. The approach is to have those vertices that have computed their values tell their parents they are done. In particular, these vertices mark a space reserved for them by their parents. The parents need only check whether all or all except one child have marked their spaces in order to determine if they are now leaves or part of a chain.

In order to reserve space at the parent for each child, we need to assume that the tree is ordered so that for each vertex v the children of v are ordered $v_1, \ldots, v_k$ and each child knows its index. That is, let $index[v_i]$ be the index of v_i in this ordering of children, i.e., $index[v_i] = i$. For each vertex v we set aside k locations $label[v, i], i = 1, \ldots, k$ in shared memory that the children can mark. Initially each $label[v, i]$ is empty or *unmarked.* Let $Arg(v)$ compute the number of unmarked labels for v. Thus, initially $Arg(v) = k$, the number of children of v. When the function at v_i can be computed, indicated by $Arg(v_i) = 0$, we apply the *rake* procedure and mark $label[v, i]$, which we denote by $mark(label[v, i])$. Let $P[v]$ be the vertex that is the sole parent of

v. When a vertex v and its parent $P[v]$ have only one unevaluated argument, then v and $P[v]$ are members of a chain. The *compress* procedure is applied to v, and $P[v]$ is jumped over, i.e., $P[v] = P[P[v]]$. Algorithm 2.7 shows the *Basic_Contract* procedure.

ALGORITHM **2.7**

The Basic Contract Phase

Procedure *Basic_Contract*
    In Parallel *Initialize*(v)
    In Parallel while *Arg* $(root) > 0$ **do**
        if $P[v] \neq nil$ **then do**
            Parallel Case *Arg* (v) **equals**
            0) *rake* (v); /* RAKE */
                mark $(label[P[v], index[v]])$;
                $P[v] := nil$;
            1) **If** *Arg* $(P[v]) = 1$ **then** /* COMPRESS */
                    compress (v);
                    $P[v] := P[P[v]]$;
            end case
        end then
    end in parallel
    host set *rake* $(root)$;
end *Basic_Contract*

The algorithm is equivalent to one application of CONTRACT if one notes that case 0 is RAKE and case 1 is COMPRESS.

THEOREM 2.5

After $O(\log_{4/3} n)$ applications to a tree with n vertices, Basic_Contract reduces the tree to its root. If RAKE *and* COMPRESS *take $O(1)$ time then the time to reduce a tree to its root is $O(\log n)$.*

PROOF

Observe that after *Basic_Contract* every maximal chain decomposes into two chains, one *essential chain* corresponding to COMPRESS and an unnecessary chain that is out of phase. The head of this second chain is unevaluated. For the purpose of analysis we can discard the second chain, since it will never be evaluated.

Note that *Basic_Contract* is slightly faster than CONTRACT, since it does not test if the only child of a vertex is a leaf or not. Thus, some pointer jumping occurs in *Basic_Contract* that does not occur in CONTRACT. That is, chains in *Basic_Contract* contain all V_1 vertices, including C_0 vertices. Therefore, *Com* for *Basic_Contract* can also contain GC_0 vertices and still reduce the size of *Com* by at least 1/2 after every phase. Since *Ra* does not contain GC_0 vertices, the number of vertices in of *Ra* reduces by at least a factor of 1/4 after every phase of *Basic_Contract*. Thus, after $O(\log_{4/3} n)$ applications of *Basic_Contract* to a tree with n vertices, *Basic_Contract* reduces the tree to its root. ∎

Expression Tree Evaluation

More intuition can be gained by seeing *Basic_Contract* applied to expression evaluation over $\{+, \times\}$. Let T be a binary expression tree in which the internal nodes hold the operators and the leaves hold the operands. The *value of a leaf* is the constant assigned to it. The *value of an internal node* is defined recursively as the operation at that node applied to the value of its children.

For reasons that will become clear later, we modify the definition of an expression tree so that associated with each edge $(v, P[v])$ is a function, f_v, in one variable. For expression trees over $\{+, \times\}$, these functions would be linear forms, $aX + b$ where X is an indeterminant and a and b would be constants. The *value of leaf* remains the value initially assigned to it. If v is an internal node with left and right children L and R and operation $\odot_v$ then the *value of an internal node* is

$$val(v) = f_L(val(L)) \odot_v f_R(val(R)),$$

where f_L and f_R are the unary functions for edges (L, v) and (R, v), respectively. Initially, the function on every edge is simply the identity function, so that every vertex of this modified expression tree has the same value as the original expression tree. We can think of $f_L(val(L))$ as the *contribution* of L to its parent's value. Let *rake* correspond to computing a vertex's value and *mark* to computing its contribution to its parent.

Next, let us consider the COMPRESS operation. After every application of COMPRESS we want an expression tree in which the value of every vertex in the current tree is the same as the value of the same vertex in the original tree. Suppose $Arg(v) = Arg(L) = 1$. In particular, suppose the value of R has been computed but L is still missing one argument. Figure 2.14 depicts this situation. During the COMPRESS operation L pointer jumps over v and points to $P[v]$, the parent of v. However, we want to be sure that $P[v]$ computes the same value after the COMPRESS operation as before. Let C_v represent the contribution of v to the value of $P[v]$ and C_R represent the contribution of R to the value of v. Since $val(R)$ is known, C_R is also known and is a constant. Let

$$val(v) = f_L(X) \odot_v C_R = f_\odot(X). \tag{2.2.4}$$

Thus, $val(v)$ is a linear form in X, which we denote by $f_\odot(X)$.

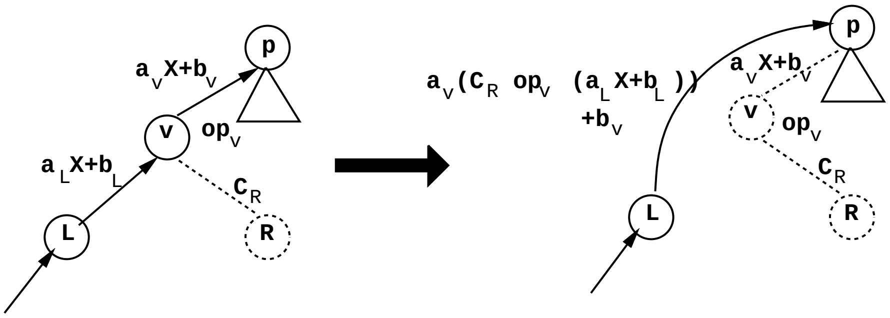

FIGURE 2.14
COMPRESS on an expression tree. The dotted nodes represent nodes that have been removed.

But then C_v is the composition of the two linear forms f_v and $f_\odot$. This composition is also a linear form. Let f'_L denote this linear form. That is,

$$C_v = f_v(val(v)) = f_v(f_\odot(X)) = f'_L(X)$$

Therefore, after pointer jumping, the correct function on the edge from v_L to $P[v]$ is f'_L. Observe that the contribution of L to $P[v]$ after the COMPRESS operation is the same as the contribution of v to $P[v]$ before the COMPRESS operation. This observation motivates the use of the modified expression tree. Figure 2.14 depicts the case in which the function of an edge depends only on the value of a node and its children. In general, however, the function value may be a composition of the operations of all the nodes that have been jumped over on the path between a node and its new parent.

Given this information, we now consider an implementation for expression evaluation over $\{+, \times\}$. Algorithm 2.8 gives the implementation, and

Figure 2.15 shows the initialization and two applications of this implementation on an expression tree. Initially, the operands of the expression tree at the leaves are stored in *val* and the operators at internal vertices are stored in *op*. Every vertex keeps a pointer to its parent, an index *side* indicating whether it is a left or right child of its parent, and an index *sib* of its sibling. Since the functions on an edge $(v, P[v])$ is a linear form $aX + b$, we associate with each vertex v a pair of numbers (a, b) that represent the linear function $a[v]X + b[v]$. The linear form at the root is irrelevant. Initially the linear function is simply X. We also set aside storage for the value of the vertex, $val[v]$, and the contributions of its left and right children, $label[v, L]$ and $label[v, R]$. The children supply the values for *label*. We use the function *eval* to evaluate the value of a vertex, given its operator and the contributions of its left and right children. We use the function *simplify* to find the new linear form for v, given (a, b) of itself and its parent, the operator of its parent, and the contribution of its sibling. The contribution of the sibling of v is stored in $label[P[v], sib[v]]$. That is, *simplify* finds the new linear form $f'_v(X)$.

We used the more conservative definition of a chain in CONTRACT since, for some applications, a vertex with a leaf as a child can use the time at this stage to incorporate the value of the child in its own value rather than pointer jumping. That is, in some implementations, it may be preferable to have the RAKE operation not only compute the value of a leaf vertex, but also do some computation at the parent given that the value of a child is now known. For example, in expression tree evaluation, evaluating a vertex is a simple, fast process. Once the value of the vertex is known the parent can partially evaluate its vertex, i.e., to find $f_{\odot}(X)$ in equation 2.2.4. Thus, the *simplify* function is divided into two parts: one part is the partial evaluation of a vertex done by the processors performing the *rake* procedure, and the other part that composes two linear forms and returns a linear form, done by the processors performing the *contract* procedure. In this way, the *rake* and *compress* procedures may be better balanced in terms of the time it takes to complete the two subprocedures.

All Subexpression Evaluation

Note that many vertices are not evaluated. That is, for many vertices v the value $Arg(v)$ is never set to 0 during any stage of *Basic_Contract*. We define a new procedure *Basic_Expand* that allows the evaluation of all vertices, i.e., each vertex eventually has all its arguments after completion of the procedure. We modify *Basic_Contract* so that each vertex keeps a push-down

ALGORITHM **2.8**

Expression Evaluation Contraction Phase

```
Procedure Expression_Contraction:
   In Parallel do                                              /* Initialize */
      if Arg(v) = 0 then do                                    /* Leaves */
         label[P[v], side[v]] := val[v];
         P[v] := nil;
      end then
      else (a, b)[v] := (1, 0);                                /* Internal vertices */
   end in parallel

   In Parallel while Arg (root) > 0 do
      if P[v] ≠ nil then do

         Parallel Case Arg (v) equals
         0)  val[v] := eval (op[v], label[v, L], label[v, R])        /* Rake */
             label[P[v], side[v]] := a[v] * val[v] + b[v];            /* Mark */
             P[v] := nil;
         1)  if Arg(P[v]) = 1 then                                  /* Compress */
                (a, b)[v] := simplify ((a, b)[v], (a, b)[P[v]], op[P[v]], label[P[v], sib[v]]);
                P[v] := P[P[v]];
         end case

      end then
   end in parallel

   host set val[root] := eval (op[root], label[root, L], label[root, R]);
end Expression_Contraction
```

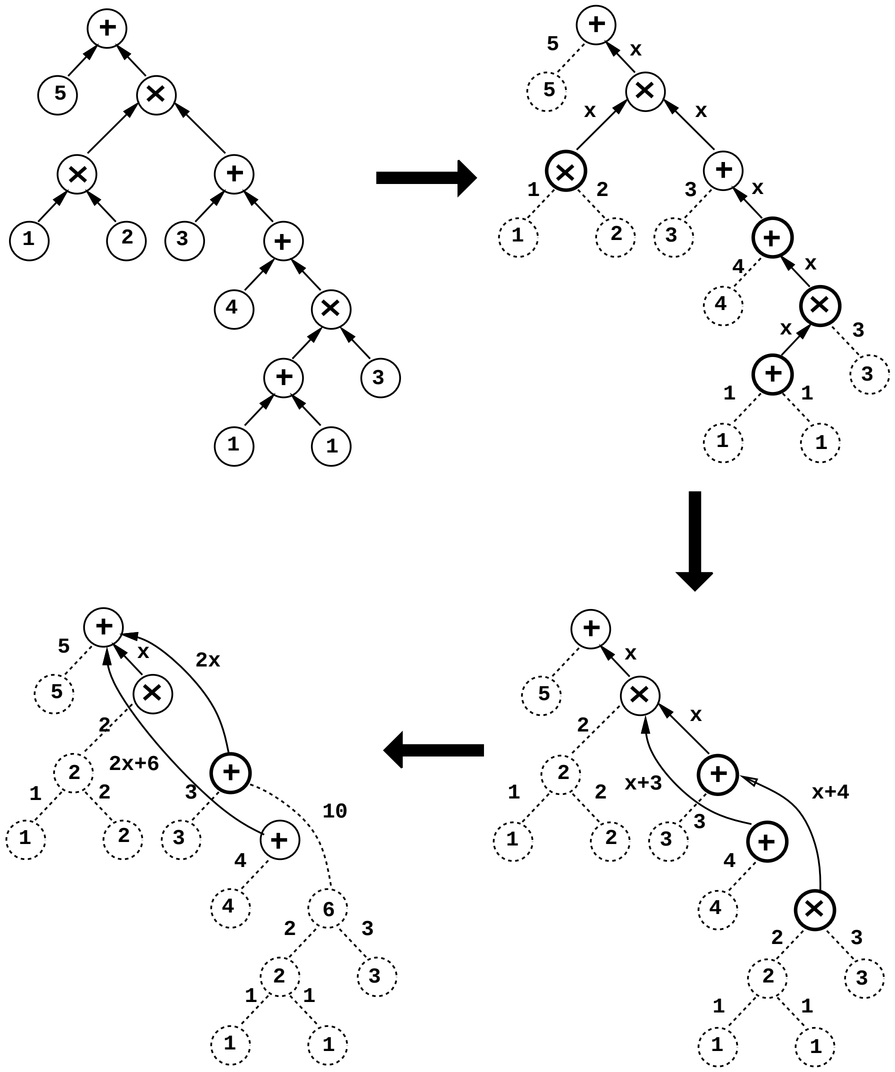

FIGURE 2.15

Initialization and two applications of *Expression_Contraction*. Vertices with heavy lines are working to generate the next version of the tree. Dotted vertices have been raked. The solid arcs show the linear forms for the child vertices. The dotted arcs show the labels sent to the parent vertices.

store $parentStore_v$ of all the previous values of $P[v]$ and add a line before the start of the **Parallel Case** statement of *Basic_Contract*:

Push $P[v]$ **onto** $parentStore[v]$;

We also include a counter t, which counts the number of iterations required to contract the tree to its root.

We now apply *Basic_Contract*, which computes the value of the root r, followed by *Basic_Expand*, given in Algorithm 2.9, which computes the value of all vertices. At each iteration, we reintroduce vertices that were either raked or were jumped over in the corresponding iteration of *Basic_Contract*. We expand the tree for t iterations to reconstruct the whole tree.

THEOREM 2.6

At the completion of Basic_Expand all vertices have their arguments.

PROOF

As in the proof of Theorem 2.5 we can discard those chains where the leaves are unevaluated and consider only essential chains. The proof is by induction on the trees with only essential chains, starting from the trivial tree consisting of a singleton vertex r and finishing with the original tree T. Let $\{r\} = T_0, \ldots, T_t = T$. The structure of these trees correspond one to one with the trees defined during the tree contraction, but in reverse order. Assume that at end of the i^{th} application of *Basic_Expand* tree T_i has all its vertices evaluated. The vertices added to T_{i+1} are either leaves that were evaluated by a *rake*, or vertices that were jumped over by *compress* during the corresponding contraction phase. Thus, every new vertex in T_{i+1} is either a leaf, in which case we know its value, or is missing one argument, the value of a vertex in T_i, which is also known. In the latter case the value of the reintroduced vertex can then be computed. ∎

THEOREM 2.7

At most $\lceil \log_{5/4} n \rceil$ *applications of basic tree contraction and* $\lceil \log_{5/4} n \rceil$ *applications of basic tree expansion are needed to evaluate all the vertices.*

In the implementation, we need to be able to distinguish between vertices on essential chains that can compute their value, and vertices not on essential chains. We use the boolean value *done*, which indicates the vertices in T_i when we are currently generating T_{i+1}. Initially only the root is in T_0. By popping $P[v]$ from the stack at each round, we get the structure of the tree during the corresponding contraction phase. We use the difference in the structure of the new tree from the old tree to note vertices reintroduced. Whenever the parent of a vertex changes either the vertex was a leaf that was raked or it spliced out its parent in the corresponding contraction phase. In the former case, leaves introduced are vertices that previously had been removed from the tree and now are connected to their parents. These leaves are "unraked" and set as done. In the latter case, those vertices that are already done are on an essential chain and can mark their new parents, which are the nodes being reintroduced this round. These parents are nodes that are not done and have zero missing arguments. Because these parents have just been marked, they can evaluate themselves using *uncompress* and set themselves done.

Consider expression evaluation over $\{+, \times\}$ as an example. When a vertex is raked it knows its final value, so *unrake* does nothing. In order to mark a parent, we need to send the contribution of the vertex to its parent. To compute the contribution, we need to know the linear form of the vertex prior to it splicing out its parent during the corresponding contraction phase. We therefore need to modify *compress* in *Expression_Contract* to save its linear form on a stack. We call this stack *envirStore*. We get the old value of the linear form by popping the stack and then computing the contribution to its parent. Finally, vertices that now have all their arguments can compute their own values, using the contributions of their children. Algorithm 2.10 shows the code for the expansion phase of expression evaluation.

EXERCISE 2.10
Show the expansion phase for the expression tree in Figure 2.15.

Applications of Tree Contraction

Expression Evaluation Let T be a tree with vertex set V and root r. We assume each leaf is initially assigned a constant from the domain $\mathcal{D}$, and each internal vertex v, with children $u_1, \ldots, u_k$, has an operator on $\mathcal{D}$ of the form $\odot(u_1, \ldots, u_k)$. To apply parallel tree contraction to an expression problem seems to require finding a general form for implementing and storing the composition of unary functions. A function of the form $f : \mathcal{D} \rightarrow \mathcal{D}$ is a **unary function** over the domain $\mathcal{D}$. The following two closure properties of unary function classes are important to using parallel tree contraction[MT87].

ALGORITHM **2.9**

The Basic Expansion Phase

```
Procedure Basic_Expand:
    In Parallel done[v] := false;                          /* Initialize */
    Host set done[root] := true;

    In Parallel for i := 1 to t do
        if not empty(parentStore[v]) then do              /* Get new parent */
            oldP[v] := P[v];
            P[v] := Pop(parentStore[v]);

            if P[v] ≠ oldP[v] then
                if oldP[v] = nil then do                  /* Leaf reintroduced */
                    unrake (v);
                    done[v] := true;
                end then
                else if done[v] then                      /* Child of spliced node */
                    mark(label[P[v], index[v]);

            if not (done[v] and
                    Arg(v) = 0) then do                   /* Spliced node reintroduced */
                uncompress (v);
                done[v] := true;
            end then

        end then
    end in parallel
end Basic_Expand
```

ALGORITHM **2.10**

The expansion phase for expression evaluation

```
Procedure Expression_Expand:
    In Parallel done[v] := false;
    Host set done[root] := true;

    In Parallel for i := 1 to t do
        if not empty(parentStore[v1]) then do              /* Get new parent */
            oldP[v] := P[v];
            P[v] := Pop(parentStore[v]);

            if P[v] ≠ oldP[v] then do
                if oldP[v] = nil then
                    done[v] := true;                        /* leaf reintroduced */
                else if done[v] then do           /* child of node reintroduced */
                    (a, b)[v] := Pop(envirStore[v]);
                    label[P[v], side[v]] := a[v] * val[v] + b[v];       /* mark */
                end then
            end then

            if not (done[v] and
                    Arg (v) = 0) then do          /* spliced node reintroduced */
                val[v] := eval (op[v], label[v, L], label[v, R]);   /* uncompress */
                done[v] := true;
            end then

        end then
    end in parallel
end Expression_Expand
```

DEFINITION

(Composition) *A unary function class* $\mathcal{F}$ *is* **closed under composition** *if, for all* $f_1, f_2 \in \mathcal{F}$, $f_2 \circ f_1 \in \mathcal{F}$.

DEFINITION

(Projection) *A unary function class* $\mathcal{F}$ *is* **closed under projection** *if for all operators* $\odot$, *for all* $a_1, \ldots, a_k \in \mathcal{D}$, *and for all* $i, 1 \leq i \leq k$:

$$\odot(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_k) \in \mathcal{F}.$$

Consider, for example, arithmetic expression trees over the reals with operators $\odot \in \{+, -, \times, \div\}$. The operations $\{+, -, \times, \div\}$ have their usual interpretations e.g., $a/b + c/d = (ad + bc)/bd$. We assume that the number of arguments at a vertex is at most 2. If not, we assume that in $O(\log n)$ time we can convert it into such a tree. In order to perform *compress* we need a representation for unary functions that is closed under projection and is closed under composition. Consider $\mathcal{F}$, the ratio of a pair of linear functions of the form $(ax + b)/(cx + d)$. The values stored or manipulated are sums, products, and differences of the initial leaf values $val(v)$. The function is the ratio of these elements. $\mathcal{F}$ is closed under composition, because for all $f(x) = (ax + b)/(cx + d)$ and $f'(x) = (a'x + b')/(c'x + d')$

$$f' \circ f = \frac{a'(ax + b)/(cx + d) + b'}{c'(ax + b)/(cx + d) + d'} = \frac{a''x + b''}{c''x + d''}$$

$\mathcal{F}$ is closed under projection, because $+(a, x) = +(x, a) = x + a$, $-(a, x) = -x + a$, $-(x, a) = x - a$, $\times(a, x) = \times(x, a) = ax$, $\div(a, x) = a/x$, and $\div(x, a) = x/a$.

EXERCISE 2.11

Find the maximum independent set of a tree. (*Hint:* Show that a greedy algorithm is sufficient.) Show how this problem is equivalent to evaluating an expression tree.

EXERCISE 2.12

Find the minimum number of registers needed to evaluate an expression tree. (*Hint:* Find the equivalent expression tree and find a function class closed under projection and composition.)

EXERCISE 2.13
Compute the *heigh;t* of each vertex in a tree, where height(v) is the length of the longest path from v to a leaf of the tree.

Ancestors' Maximum Value Given a tree with values at each vertex, for each vertex find the maximum value of the ancestors of that vertex.

This problem is slightly different form the ones we considered before because the information needed to compute the maximums comes from the ancestors, not the descendants, of the vertex. That is, the information needs to flow down the tree not up. Thus, *rake* cannot contribute to the solution. But *unrake* does, by using the information filtered down to its parent.

Solution: We use parallel tree contraction with the following operations:
Initialize: $max[v] := value[P[v]]$
Rake: /* null operation */
Compress: $max[v] := maximum(max[v], max[P[v]])$
Unrake: $max[v] := maximum(max[v], max[P[v]])$
Uncompress: $max[v] := maximum(max[v], max[P[v]])$

During the contract phase *max* is the maximum of the chain of values between a vertex and its current parent. Thus, the maximum value of the ancestors of a vertex v is the maximum of $max[v]$ and the maximum value of the ancestors of $P[v]$. During the expansion, for a vertex that is done, *max* is the maximum of all the ancestors of the vertex. Note that when a vertex is unraked its parent is done. When a vertex in a chain is reintroduced, not only was its child in an essential chain, so was its parent. Hence, it, too, is done.

EXERCISE 2.14
Given a tree with connections between pairs of vertices, for each connection give the vertex that is the lowest common ancestor of the vertices at the ends of the connection.

EXERCISE 2.15
Given T and B, the tree and back edges produced by a depth-first search of a connected, undirected graph $G = (V, E)$, find the low point number,

Low, for the each vertex v in V. That is, assume that the vertices are labeled by their depth-first numbers. Then

$$Low[v] = \min\left(\{v\} \cup \left\{ w \mid \begin{array}{l} \text{there exists a back edge } (x,w) \in \\ B \text{ such that } x \text{ is a descendant of} \\ v, \text{ and } w \text{ an ancestor of } v \text{ in the} \\ \text{depth first spanning forest } (V,T) \end{array} \right\}\right).$$

For a depth-first search sequential algorithm, see [AHU74].

2.2.4 Optimal Parallel Tree Contraction for Binary Trees

In this section we show a new operation, called SHUNT, to contract an ordered binary tree to its root ([ADKP87], [KD88]). If the tree is not binary it sometimes can be interpreted as a binary tree by introducing a new vertex for every child vertex (see [ADKP87]). Otherwise one of the more general algorithms must be used. Parallel tree contraction using shunting can reduce a tree to its root in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM. Note that the *Basic_Contract* algorithm is not optimal because the time and processor product is greater than the time for the optimal sequential algorithm. One of the inefficiencies is that by using Wyllie's pointer jumping for chains we get two chains where only one is essential.

Another problem with *Basic_Contract* is that it does not work on the EREW model. The problem arises at the parent of a chain (a vertex in V_2 with a child in V_1 is called the **parent** of a chain). If v is a parent of a chain, then by using the pointer-jumping algorithm of Wyllie, we encounter the problem that over time many vertices in the chain may eventually point to v. Now, if v later becomes a vertex in V_1, then all these vertices will want to jump over v and, therefore, must read $P[v]$, which requires a concurrent read.

We can avoid using Wyllie's algorithm if we prevent chains from forming. A chain is produced when the tree contains a binary subtree where each internal vertex has one child that is a leaf and one that is not. After a RAKE operation, the subtree becomes a chain. If we apply RAKE to a leaf followed immediately by a COMPRESS on its sibling we prevent chains from forming. We call this operation pair **shunt**. That is, to apply SHUNT to a leaf vertex, v, we delete v and $P[v]$ and set $P[v']$ to $P[P[v]]$, where v' is sibling of v. Figure 2.16 shows an application of SHUNT to a leaf. Note that shunt applied to a leaf v does not produce any new leaves. The only effect on leaves is that it removes v, so that each application of shunt to a leaf reduces the number of leaves by one. Also note that shunt is not defined for a child of the root vertex, because we can not apply COMPRESS to the root.

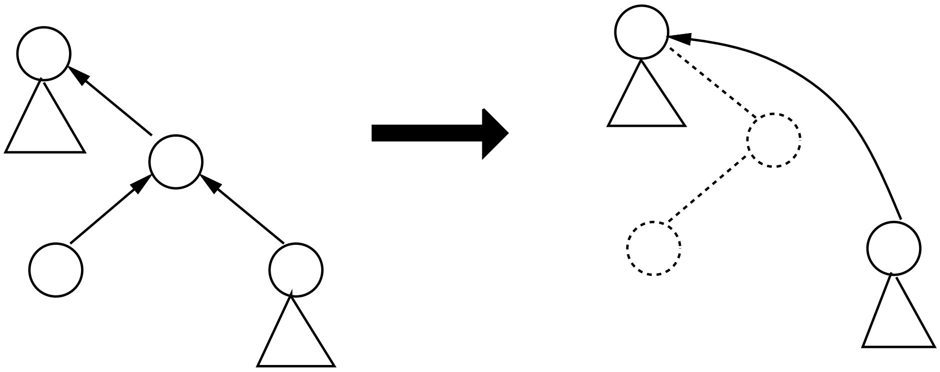

FIGURE 2.16
SHUNT applied to a left leaf. The dotted vertices have been removed from the tree.

However, we can not apply shunt to all leaves simultaneously. To prevent concurrent reads we cannot apply SHUNT to two leaves with the same parent. Otherwise the two leaves would attempt to reconnect their siblings to their grandparent by jumping over their common parent at the same time. But we also have to be careful not to apply SHUNT to leaves with different parents that are also consecutive in a left to right ordering of the leaves. If we do apply SHUNT to two such leaves we end up with two disconnected subtrees, as shown in Figure 2.17. Therefore, we apply SHUNT to the odd numbered leaves only. However, we still can get disconnected subtrees. Figure 2.18 shows such a situation.

EXERCISE 2.16
Number the leaves of an ordered binary tree in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM. *Hint*: Use the Euler Tour of the tree.

To prevent the tree from becoming disconnected, we apply SHUNT first to all the left children that are odd numbered and then to all the right children that are odd numbered. We exclude the children of the root since SHUNT

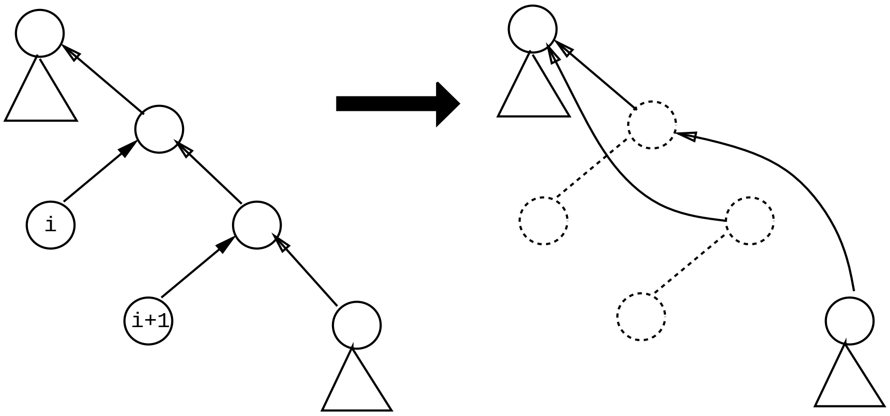

FIGURE 2.17
SHUNT applied to two consecutive leaves. The vertex labels give the leaf numbering.

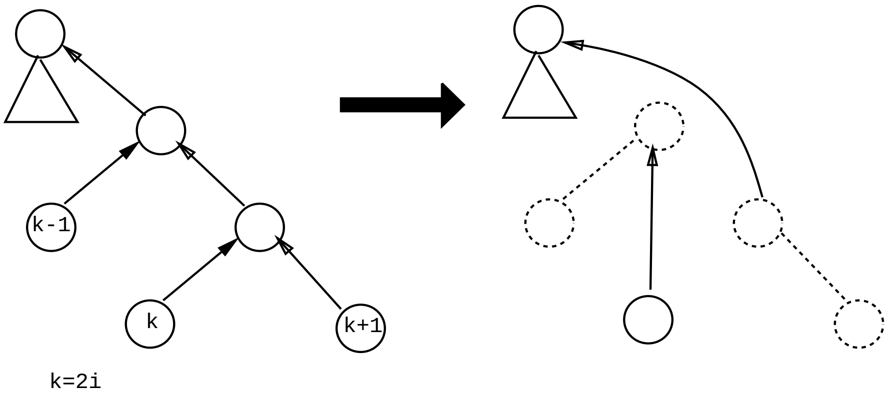

FIGURE 2.18
SHUNT applied to odd numbered leaves. The vertex labels give the leaf numbering.

is undefined for these vertices. Notice that the relative order of the leaves remaining stays unchanged. Therefore, we can renumber the leaves by simply dividing the leaf numbers by 2. We repeat the algorithm until all leaves are removed except for the children of the root at which point we evaluate the root. The basic parallel tree contraction algorithm using SHUNT, called *Shunt_Contract*, is shown in Algorithm 2.11. We use *number* as the index of a leaf vertex in a left to right numbering of leaves, starting at 0. By starting the numbering at 0, shunt is never applied to the left most leaf and this leaf eventually becomes a child of the root. Nonleaf vertices have an index of 0. The boolean *side* indicates whether a vertex is the left or right child of its parent. The procedure $shunt(v)$ applies *rake* to v and *compress* to $sib[v]$ and sets $P[sib[v]]$ to $P[P[v]]$, where *sib* is the sibling of v. Initially only leaves are active. The result of a *shunt* operation on a leaf of an $\{+, \times\}$ expression tree is shown in Figure 2.19.

Figure 2.20 shows the contraction of an expression tree to its root, using *Shunt_Contract*. The linear form saved at a vertex is shown at the arc to the parent. The linear forms in parentheses will be explained in the next section.

LEMMA 2.2

Shunt_Contract runs correctly on an EREW PRAM.

PROOF

Let v_1 and v_2 be nonconsecutive left (right) leaves. Then $P[v_1]$ and $P[v_2]$ are not identical since v_1 and v_2 are both left (right) leaves, nor are $P[v_1]$ and $P[v_2]$ a parent of the other, since v_1 and v_2 are not consecutive leaves. Therefore, v_1 and v_2 can read and write all the information associated with themselves, their parents, and their siblings without conflict. ∎

THEOREM 2.8

After $O(\log n)$ applications of Shunt_Contract on a EREW PRAM with $n/\log n$ processors a tree with n vertices is reduced to its root. If the RAKE *and* COMPRESS *operations run in $O(1)$ time then the overall running time is $O(\log n)$.*

PROOF

ALGORITHM **2.11**

The basic parallel tree contraction algorithm using shunting

```
Procedure shunt(v)
    rake (v); active[v] := false;
    active[P[v]] := false;
    compress (sib[v]);
    P[sib[v]] := P[P[v]];
end shunt

Procedure Shunt_Contract
    In parallel do                                         /* Initialize */
        if isLeaf[v] then
            active[v] :=true;
        else
            active[v] := false;
        number[v] := numberLeaves (v);
    end in parallel

    In parallel for i = 1 to ⌈log n⌉ do                    /* Contraction */
        if v ≠ root and active[v] then
            if isOdd (number[v]) and P[v] ≠ root then do
                if side[v] = left then shunt (v);
                if side[v] = right then shunt (v);
            end then
            else
                number[v] := number[v]/2;
    end in parallel
end Shunt_Contract
```

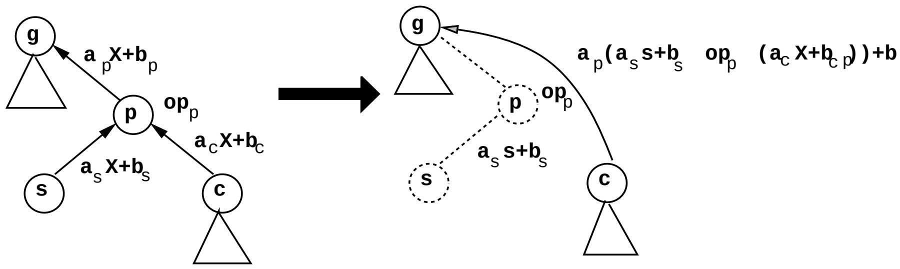

FIGURE 2.19
The *shunt* operation applied to a leaf of a $\{+, \times\}$ expression tree.

> Assign each processor $\log((n+1)/2)$ consecutive leaves. During the i^{th} application of *Shunt_Contract* each processor eliminates 1/2 of its $\log((n+1)/2^i)$ vertices. Hence, after $\log n$ applications, each processor has eliminated all of its leaves. If Rake and Compress run in $O(1)$ time then the overall running time is $O(\log n)$ time. ▪

Note that the *rake* and *compress* operations cannot be performed in parallel as they could in *Basic_Contract* because both operations can simultaneously be operating on the same node. Also note that each round of *Shunt_Contract* works on left leaves followed by right leaves. Therefore one round of *Shunt_Contract* takes at least twice as long as one round of *Basic_Contract*.

All Subexpression Evaluation using Shunt

As before, we can easily modify *Shunt_Contract* to enable us to evaluate all subexpressions of the tree by applying an expansion phase following the contraction phase. Note that when we applied Shunt to a leaf v, $P[v]$ never receives its argument, $sib[v]$, since $P[v]$ is deleted from the tree. However, if we expand the tree by running the contraction phase in reverse, when v and $P[v]$ are reintroduced the value of $sib[v]$ has already been computed. As before, we can prove this fact by using induction on the list of trees formed during the expansion phase and noting that after every expansion all subexpressions of the current tree have been computed. Thus, during the expansion phase, we reintroduce the vertices eliminated during the corresponding application

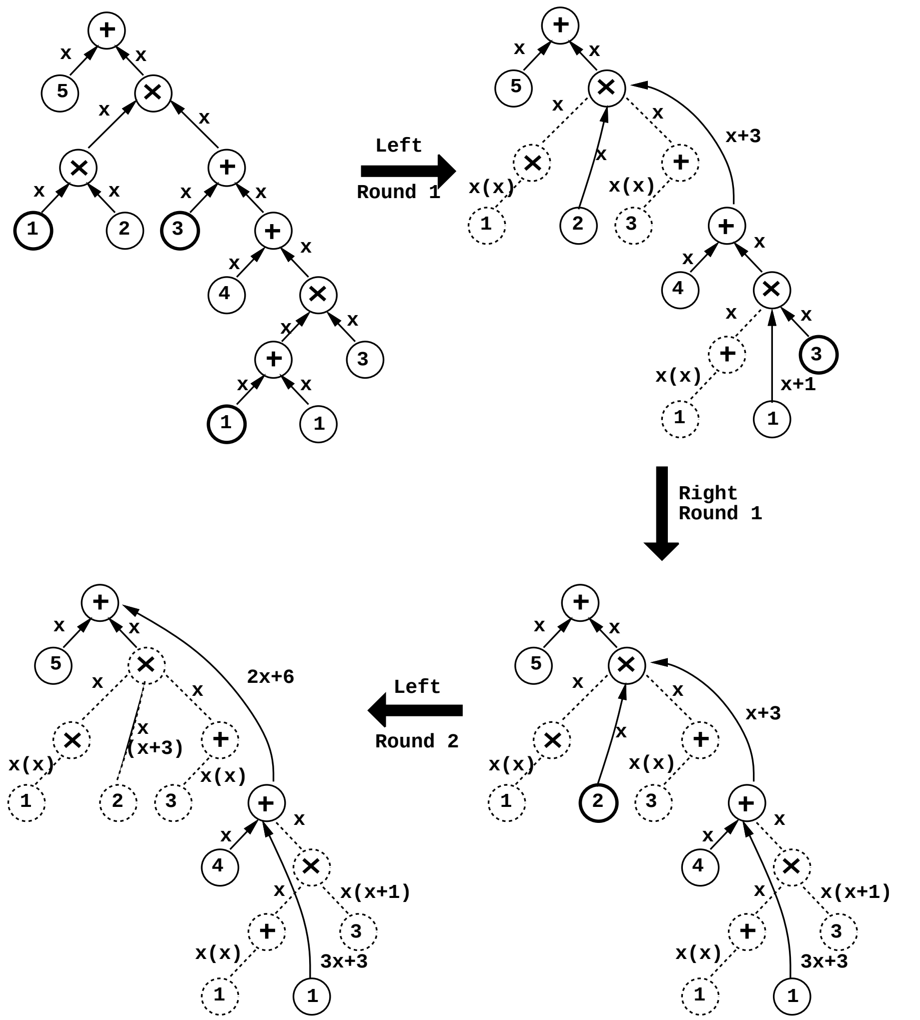

FIGURE 2.20
Parallel tree contraction using *Shunt_Contract*. At each round SHUNT is applied to the highlighted vertices. The expressions in parenthesis are saved for use during the expansion phase.

of the contraction phase. The state of the vertices are restored so that the value of the reintroduced internal nodes can be computed.

Recall that for arithmetic subexpressions, *Compress* changes the linear form saved at the vertex. Therefore in *Basic_Contract* we saved the current state of a vertex on a stack each time we applied *compress*. Then, at the corresponding point during the expansion phase, that state is popped off the stack. Similarly for *Shunt_Contract*, we need to save the current state of $sib[v]$ before we apply the *compress* procedure. As it turns out, when we use *shunt*, we can save the current state of $sib[v]$ at vertex v before we apply *compress* to $sib[v]$. We can eliminate the stack because every time we apply *compress* to a vertex, we know we have just applied *rake* to its sibling and the sibling is different every time. Thus, during the expansion phase when v and $P[v]$ are reintroduced, v has the correct state information for $sib[v]$. Recall that $sib[v]$ has already been computed so we can compute its contribution to $P[v]$, given this state information. Then we can apply *unrake* to v and *uncompress* to $P[v]$. The saved linear forms of the siblings appear in parentheses in Figure 2.20.

EXERCISE 2.17
Show the expansion phase for the example in Figure 2.20.

2.2.5 Optimal EREW PRAM Parallel Tree Contraction Algorithm

In this section we exhibit an optimal deterministic EREW PRAM parallel tree contraction algorithm using $O(n/P)$ time and P ($P \leq n/\log n$) processors [GMT88]. The algorithm has two stages. The first stage uses a new reduction technique called M-CONTRACTION. The basic idea is to dynamically divide the tree into subtrees, each of which has at most one unknown leaf, and then assign the subtrees to P processors, which *partially* evaluate the subtrees and succinctly compress the information to single vertices. In this way a tree of size n is reduced to one of size P in $O(n/P)$ time, using P processors on an EREW PRAM. The second stage uses a technique called ISOLATION to contract a tree of size P to its root in $O(\log P)$ time, using P processors on an EREW PRAM. Isolation eliminates the concurrent reads needed by the Wyllie approach used in the *Basic_Contract* procedure.

This section consists of three subsections. The first subsection contains the basic graph theoretic results and definitions that we need in the following subsection. In that subsection we show how to reduce the problem of size n to one of size P, where P is the number of processors. In the last subsection we show the isolation technique used to implement parallel tree contraction on a deterministic EREW PRAM in $O(\log n)$ time using n processors. We then discuss some implementation techniques and the expansion phase for all subexpression evaluation.

Basic Graph Theoretic Results

In this section we give some graph theoretic results that can be used to find a set of vertices that subdivide a tree into independent subtrees of approximately equal size. From these subtrees we can define the m-contraction of a tree to reduce the size of the tree.

First we consider the decomposition of a tree T into subtrees by finding vertices that partition the edges of T in a natural way. The vertices we consider are called m-critical vertices. Subtrees (subgraphs) are then formed out of each partition of edges by reintroducing the vertices at the end points of the edges. These subgraphs are known as **bridges**. We give the formal definitions needed to define m-critical vertices and bridges next.

Let $T = (V, E)$, be a directed graph in which every vertex, except the root, points to its unique parent. The **weight** of a vertex v in T is the number of vertices in the subtree rooted at v, denoted by $W(v)$. If n equals the number of vertices in T, the weight of the root r is n.

Let m be any integer such that $1 < m \leq n$. In the next subsection we let $m = 2n/P$, where P is the number of processors. A vertex, v, is m-**critical** if

1. v is not a leaf, and
2. $\lceil \frac{W(v)}{m} \rceil > \lceil \frac{W(v')}{m} \rceil$ for all $v' \in$ children (v).

LEMMA 2.3

If v_1 and v_2 are m-critical, then their least common ancestor is m-critical.

PROOF

If either v_1 or v_2 is an ancestor of the other, then the lemma is trivially true. We therefore consider case when neither is an ancestor of the other. Let v be an m-critical vertex and let w be a child of v. Since $\lceil \frac{W(v)}{m} \rceil > \lceil \frac{W(w)}{m} \rceil \geq 1$, the weight of v must be greater than m. Therefore, if v_1 and v_2 are m-critical then both their weights are greater the m. Then

u, the least common ancestor of v_1 and v_2, must have weight greater than $2m$ since it has two descendants with weight greater than m. Each child of u can not have weight greater than $W(u) - m$ because at least one of v_1 and v_2 is not among the child's descendants, implying that u is m-critical. ∎

Let $G = (V, E)$ be a graph and let $C \subset V$. Two edges, e and e' of G, are C-**equivalent** if there exists a path from e to e' that avoids the vertices C. Also, let $E' \subset E$. E' induces a subgraph, $G' = (E', V')$, with $V' = \{v \in V \mid v$ is an endpoint in $E'\}$. That is, the endpoints of E' are included in G'. The graphs induced by the equivalence classes of the C-equivalent edges, are called the **bridges** of C. A bridge is **trivial** if it consists of a single edge. The **attachments** of a bridge B are those vertices of B that are also in C. An example of the C-equivalent classes and their induced bridges of a graph are shown in Figures 2.21 and 2.22.

The m-**bridges** of a tree T are bridges of C, where C is the set of m-critical vertices of T. Note that the attachments of an m-bridge B are either the root of B and/or one of its leaves. In Figure 2.23 we give a tree and its decomposition into its 5-bridges. The vertices represented by boxes are the 5-critical vertices, and the numbers next to these vertices are their weights.

LEMMA 2.4

If B is an m-bridge of a tree T, then B can have at most one leaf attachment.

PROOF

The proof is by contradiction. We assume that B is an m-bridge of a tree T, and v_1 and v_2 are two leaves of B that are also m-critical. We prove that this is impossible. Let w be the lowest common ancestor of v_1 and v_2 in T. Since B is connected, w must be a vertex of B and there must be a path from v_1 to w and from w to v_2. Therefore, by definition of an m-bridge w cannot be m-critical. On the other hand, w is m-critical by the above lemma. ∎

From Lemma 2.4 one can see that there are three types of m-bridges: (1) a **leaf bridge** which is attached by its root; (2) an **edge bridge** which is attached by its root and one leaf; and (3) a **top bridge**, containing the root of T, which exists only when the root is not m-critical. Except for the

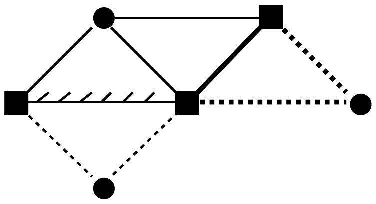

FIGURE 2.21
Square vertices represent members of C. Each line type represents another C-equivalent class.

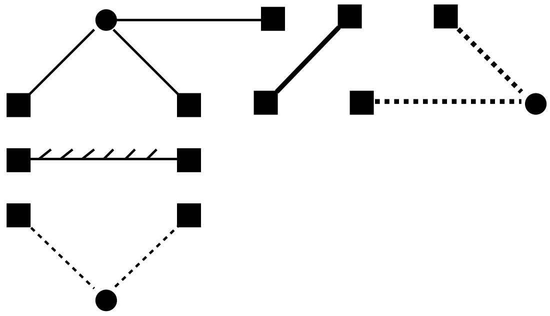

FIGURE 2.22
The bridges of C. Square vertices are the attachments.

top bridge, the root of each m-bridge has a unique child. The edge from this child to its root is called the **leading edge** of the bridge.

LEMMA 2.5

The number of vertices of an m-bridge is at most $m + 1$.

PROOF

Consider the three types of m-bridges: leaf, edge, and top. Suppose B is a leaf bridge with root r'. Then r' is the only m-critical vertex in B and has weight $\geq m$. Because $m > 1$, r' must have a child, w, and this child is unique in B. We claim that w has weight $< m$. Suppose it has weight $\geq m$. Then we claim there is at least one m-critical vertex in the subtree rooted at w. Because all paths from w to a leaf vertex have

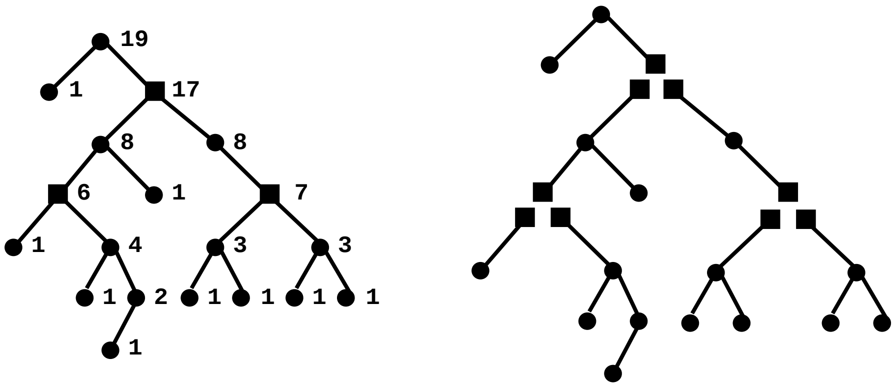

FIGURE 2.23
The decomposition of a tree into its 5-Bridges.

strictly decreasing weight and the leaf has weight 1, there exist some vertex with weight $\geq m$ and all of its children have weight $< m$. By definition this vertex is m-critical, which contradicts that r' is the only m-critical vertex in B. Thus, w must have weight $< m$ and the number of vertices of B is at most m. If B is an edge bridge with m-critical root r' and m-critical leaf u, then r' will have a unique child w in B. Since u is not a leaf of T and all the vertices in the subtree rooted in u are not in B, the number of vertices in B is $W(w) - W(u) + 2$ (we add 2 to include vertices r' and u). Since w is not m-critical, $W(w) - W(u) < m$ by arguments similar to above. Thus, the number of vertices in B is $\leq m + 1$. The case for a top bridge follows by similar arguments. ∎

To devise a parallel algorithm, it would be convenient to have few m-bridges [i.e., $O(n/m)$]. However, that is not always the case. For example, consider an unbounded degree tree of height 1, where $m < n$ and every edge is an m-bridge. The following lemma shows, however, that the number of m-critical vertices is not large.

LEMMA 2.6

The number of m-critical vertices in a tree of size n is at most $2n/m - 1$ for $n \geq m$.

PROOF

Let n_k be the number of vertices in a minimum size tree with k m-critical vertices. The lemma is equivalent to the statement:

$$n_k \geq (\frac{k+1}{2})m, \quad \text{for} \quad k \geq 1. \tag{2.2.5}$$

We prove inequality 2.2.5 by induction on k. If v is m-critical, then its weight must be at least m. This proves 2.2.5 for $k = 1$. Suppose that 2.2.5 is true for $k \geq 1$ and all smaller values of k. We prove 2.2.5 for $k+1$. Suppose that T is a minimum size tree with $k + 1$ m-critical vertices. The root r of T must be m-critical for it to be of minimal size, because we can discard all of the tree above the first m-critical vertex (the root bridge) without affecting the number of critical vertices. Assuming r is m-critical, there are two possible cases for the children of root r: (1) r has two or more children, $u_1, \ldots, u_t$, and each of their subtrees contains an m-critical vertex; or (2) r has exactly one child u whose subtree contains an m-critical vertex.

We first consider Case 1. Let n_i be the number of vertices, and k_i the number of m-critical vertices in the subtree of u_i, for $1 \leq i \leq t$. Since T is of minimum size, $u_1, \ldots, u_t$ must be the only children of r. Since T has $k + 1$ m-critical vertices, one of which is the root, $k = \sum_{i=1}^{t} k_i$ and $\sum_{i=1}^{t} n_i \leq n_{k+1}$. Using these two inequalities and the inductive hypothesis we get the following chain of inequalities:

$$\begin{aligned} n_{k+1} \quad & \geq \textstyle\sum_{i=1}^{t} n_i \geq \sum_{i=1}^{t} \left(\frac{k_i+1}{2}\right) m \geq \left(\frac{(\sum_{i=1}^{t} k_i)+t}{2}\right) m, \\ & = \left(\frac{k+t}{2}\right) m \geq \left(\frac{k+2}{2}\right) m = \left(\frac{(k+1)+1}{2}\right) m. \end{aligned}$$

This proves Case 1.

In Case 2 the subtree rooted at u contains a unique maximal vertex w which is m-critical, and the subtree of w contains k m-critical vertices. Thus, the induction hypothesis shows that $W(u) \geq (\frac{k+1}{2})m$. We consider two case, when k is odd and even. If k is odd then $(\frac{k+1}{2})m$ is an integral multiple of m. In order for $W(r)$ to be an integral multiple of m greater than $W(u)$,

$$W(r) \geq (\frac{k+1}{2})m + m \geq (\frac{k+2}{2})m.$$

If k is even then $\frac{k}{2}m$ is an integral multiple of m. Again in order for $W(r)$ to be an integral multiple of m greater than $W(u)$,

$$W(r) \geq (\frac{k+1}{2})m + \frac{m}{2} = (\frac{k+2}{2})m$$

■

The m-**contraction** of a tree T with root r is a tree $T_m = (V', E')$, such that the vertices V' are the m-critical vertices of T union r. Two vertices v_1 and v_2 in V' are connected by an edge in T_m if there is an m-bridge in T which contains both v_1 and v_2. Note that every edge in T_m corresponds to a unique m-bridge in T which is either an edge bridge or the top bridge. Thus by Lemma 2.6, T_m is a tree with at most $2n/m$ vertices. In the next section we show how to reduce a tree to its m-contraction, where $m = 2n/P$, in $O(m + \log n)$ time on a EREW PRAM.

Reduction From Size n to Size n/m

In this section we show how to contract a tree of size n to one of size $2n/m$ in $O(m)$ time using n/m processors, for $m \geq \log n$. If we set $m = \lceil 2n/P \rceil$, then this gives us a reduction of a problem of size n to one of size P. In the next section we show how to contract a tree of size P to a point.

From the previous section, we learned that there are at most $2n/m - 1$, m-critical vertices, but possibly many more m-bridges. Since we have no bound on the number of m-bridges in a tree, we cannot simply assign an m-bridge to each processor. However, since we are assuming that the tree is of bounded degree d, there can be at most d m-bridges common to and below an m-critical vertex of T. Therefore, to perform the reduction, we need only find the m-critical vertices and efficiently assign them to processors. A processor is assigned to each m-critical vertex and computes the value (function) of the (at most d) m-bridges below it. A processor is also assigned to the top bridge if the root is not m-critical, and computes the function for the top bridge. Since each m-bridge has at most $m + 1$ vertices, a processor can sequentially compute the value or function of its $O(1)$ m-bridges in $O(m)$ time. The sketch of the algorithm is in Algorithm 2.12.

The following example illustrates this procedure. Consider again an expression tree. The evaluation of a leaf bridge is the value of all the vertices of the subtree. An edge or top bridge can be considered as a unary function, with the leaf attachment as the indeterminate. That is, the evaluation of

ALGORITHM **2.12**

Sketch of m-Contract

Procedure *m-Contract* (T)

1. $m := \lceil 2n/P \rceil$
2. Compute $W[v]$ and $\lceil W[v]/m \rceil$ for all vertices v in T
3. Determine the m-critical vertices in T
4. Assign a processor to each m-critical vertex and one to the root
5. Each processor computes the value of the leaf bridges or the unary function of the edge or top bridges below the m-critical vertex or root assigned to it
6. Return the m-contraction of T

the edge or top bridge, with leaf attachment vertex l_s and root r_s, is a unary function f_s such that $value[r_s] = f_s(value[l_s])$. Figure 2.24 shows a expression tree over $\{+, *, -\}$, its division into 5-bridges, and its 5-contraction.

Assume that a tree is given as a set of pointers from each child to its parent and that the tree is ordered. That is, the children of a vertex are ordered from left to right and each child knows its position (index) in that ordering. Furthermore, assume that each parent has a consecutive block of memory cells, one for each child, so that each child can write its value, when known, into its location using its index. This last assumption permits us to compute the maximum value of each set of siblings needed to determine the m-critical vertices.

The Bounded Degree Case In this subsection we consider each step in more detail for trees of bounded degree. In order to implement many of the steps in the *m-Contract* procedure we use the Euler tour of a tree, which is a list of both forward and backward edges of the tree in depth first order. Therefore, we add finding the Euler tour of the tree T to Step 1.1. For a more detailed discussion on Euler tours and their construction and use see the following chapter.

Step 1.1
Compute m and find the Euler tour of T. Next we use list ranking to

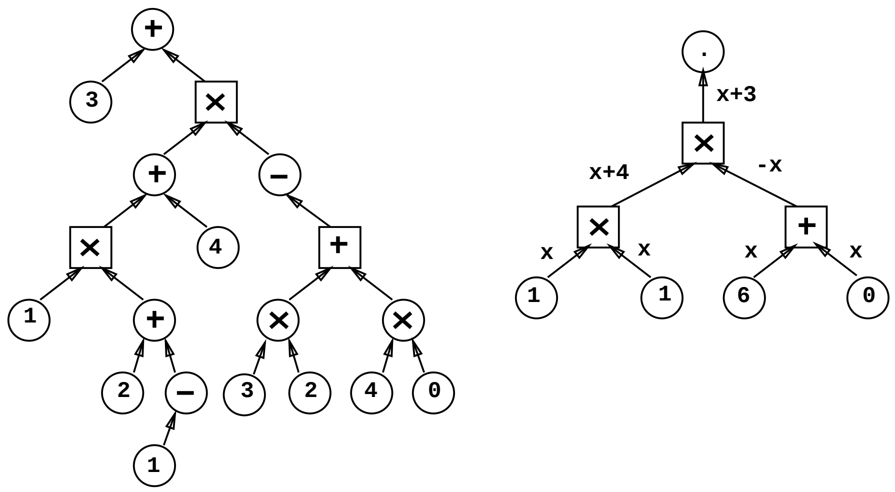

FIGURE 2.24
The 5-contraction of an expression tree.

order the edges of T, which we then use to map the i^{th} edge of the tree to the i^{th} element of an array. By ordering the edges in an array, rather than in a linked list, we can perform the All Prefix Sum operations on the array without traversing the linked list each time.

Implementation note: If we are working on a distributed memory parallel computer, we need to send to each processor the information for the edges corresponding to a consecutive block of the Euler tour array. Karlin and Upfal [KU86] show that once the Euler tour numbering is known the information corresponding to an edge can be moved to its correct location in $O(\log n)$ time using a randomized algorithm. Ranade [Ran87] gives an algorithm for moving the data on a Butterfly network.

Implementation note: Although the running time of All Prefix Sum calculations on a linked list is the same as on an array for a PRAM, on a fixed connection machine, the array representation can result in an $\log n$ improvement in running time. In particular, it can be shown that the All Prefix Sums can be computed in $6 \log n$ time on a binary N-cube parallel computer, where $N = n/\log n$, whereas list ranking on the same size N-cube computer takes $O(\log^2 n)$ time. Therefore, the *m-Contract* algorithm does a single list ranking, converts the linked list to an array, and then performs All Prefix Sum operations on the array.

Step 1.2
We need to compute the weight of each vertex, that is the number of vertices in the subtree root at that vertex. Similar to the number of descendants computed in the chapter on Euler tours, the weights can be computed by numbering every forward edge with 1 and backward edge with 0, finding the All Prefix Sums using addition, and computing the weight of a vertex as one plus the difference between the prefix sum of backward edge leaving the vertex and the prefix sum of forward edge entering the vertex.

Step 1.3
We need to revert back to the original representation of the tree in order to determine which vertices are m-critical. This is easy to do if we save the location of the vertex in the original representation with the forward edge of the Euler tour representation. A processor is responsible for $\lceil n/P \rceil$ vertices. Each vertex writes its weight to the memory locations reserved by its parent, denoted by wt. That is, vertex v writes $weight[v]$ to $wt[P[v], index[v]]$. In this way, each vertex has the weights of all its children in a subarray. We can then perform a Segmented All Prefix Sum operation using the max operator so that the All Prefix Sum starts afresh each time it reaches a new subarray of the wt array. Let $maxwt$ be the result of the Segmented All Prefix Sum. A vertex is m-critical if $\lceil weight[v]/m \rceil > \lceil maxwt[v]/m \rceil$.

Step 1.4
We enumerate the m-critical vertices, by assigning a 1 to m-critical vertices and 0 to all others, and then compute the All Prefix Sum over the addition operator. Processor i is responsible for the i^{th} m-critical vertex and Processor 0 is responsible for the root if the root is not m-critical.

Step 1.5
Each processor evaluates the m-bridges below the m-critical vertex assigned to it. The number of m-bridges below an m-critical vertex is equal to the number of children of that vertex and is bounded by d, the

degree of the tree. If there is a depth-first-search sequential algorithm to evaluate an m-bridge, then a processor can evaluate the m-bridges by simply scanning the Euler tour of the tree starting at the forward edge of the first child of the m-critical vertex. Traversing the Euler tour is equivalent to a depth first traversal. Note that all the edges of a leaf bridge are in consecutive edges of the Euler tour. Thus, if the m-bridge is a leaf bridge, then when the processor returns back to its m-critical vertex it will have completely evaluated the subtree. The next forward edge on the Euler tour is the first edge of the next m-bridge for which the processor is responsible. On the other hand, the edges of an edge or top bridge are in two separate consecutive edges of the Euler tour, the intervening edges being part of the subtree routed at the leaf attachment. Therefore, if the m-branch is an edge or top bridge, the processor will reach an m-critical vertex that is not its own. This vertex represents the indeterminate of the edge or top bridge function. To continue its tour of the m-bridge the processor needs to jump to the corresponding backwards edge. Again, when it reaches its own m-critical vertex it has completed the evaluation of the m-bridge and is ready to proceed with the next m-bridge, if there is one. There is another m-bridge if the next edge is a forward edge. Otherwise, there are no more m-bridges.

Since each m-bridge has at most $m+1$ vertices, if the sequential running time of evaluating an tree of m vertices is $T(m)$, then a processor can evaluate all its m-bridges in $O(T(m))$. For example, the sequential evaluation of expression trees over the operators $\{+, \times\}$ is linear. Therefore, evaluating the m-bridges take $O(m)$ time.

Implementation note: If the *m-Contract* is implemented on a distributed memory parallel machine then each processor maintains the data corresponding to n/P vertices and $2n/P = m$ consecutive edges of the Euler tour. The m-bridges for which a processor is responsible are located in at most $d+1$ separate portions of the Euler tour. Each separate portion can contain at most 2m edges and, therefore, can only be located in the memory of at most 3 processors. Thus, a processor needs to access the memory of at most $3d+3$ processors.

Step 1.6

We return the Euler tour of m-contraction of the tree. The Euler tour is easily computed using the All Prefix Sums of the m-critical vertices.

The Unbounded Degree Case In this subsection we show how to compute the m-contraction of an unbounded degree tree. In the unbounded case the number of m-bridges immediately below an m-critical vertex may be large and in particular the number of leaf m-bridges may be much larger than the number of processors. Therefore, we cannot load balance by simply assigning a processor to all the m-bridges immediately below an m-vertex. On the other hand, the total number of bridges that are either a top bridge or an edge bridge is bounded by the number of m-critical vertices which, in turn, is bounded by $2n/m$. To handle the unbounded degree case, we change Step 5 of *m-Contract* (Algorithm 2.12) into 3 substeps, as shown in Algorithm 2.13.

Each processor is assigned one edge or top bridge and some number of leaf bridges, depending on their size. To evenly divide the leaf bridges among the processors we compute the All Prefix Sums of their weights. That is, we assign the leading edge of a leaf bridge a value equal to the weight of the child vertex of the leading edge. To all other edges, assign a value of zero. Let $S[e]$ be the sum up to edge e. We would like to assign each processor an equal number of vertices. We approximate this by assigning processor i all leaf bridges with leading edge e such that $\lceil (i-1)\cdot n/P \rceil < S[e] \leq \lceil i \cdot n/P \rceil$. A processor need only know the first leaf bridge in its interval for which it is responsible. That is, let $proc[e]$ be the processor number responsible for edge e. Then $proc[e] = \lceil S[e]P/n \rceil$ and processor i is responsible for all leaf bridges with edges e such that $proc[e] = i$. The first leaf bridge in the interval has leading edge e' such that $proc[e'] > proc[e'-1]$.

In the unbounded degree case, a processor may be required to evaluate many small leaf bridges, since there may be a large number of them.

ALGORITHM **2.13**

Step 5 of m-Contract for unbounded degree trees

5. Assign m-bridges to processors:
   (a) Assign to each leading edge of a leaf bridge a value equal to the weight of its bridge. To all other edges, assign a value of zero. Compute All-Prefix-Sums of value; let $S(e)$ be the sum up to e and $proc[e] = \lceil S[e]P/n \rceil$.
   (b) **if** $proc[e] > proc[e-1]$ **then** $firstLeafBridge[proc[e]] = e$
   (c) Using the All-Prefix-Sums procedure, assign a new processor to each edge or root m-bridge.

Isolation and EREW Parallel Tree Contraction

In the previous section we showed that if we find an EREW parallel tree contraction algorithm, which takes $O(\log n)$ time and uses n processors, then we get an $O(\log n)$ time, $n/\log n$ processor EREW PRAM algorithm for parallel tree contraction, by first applying *m-Contract*. Thus, we may restrict our attention to $O(n)$ processor algorithms. In this section we present a technique called ISOLATION and use it to implement parallel tree contraction on an EREW PRAM without increasing the time and processor count of *Basic_Contract*.

Recall that the *Basic_Contract* algorithm seems to require concurrent reads because the pointer jumping technique of Wyllie causes tails of essential and nonessential chains to have the same parent. A concurrent read occurs when this parent eventually has only one unevaluted argument and extends these chains; all the vertices that were the tails of the chains want to jump over the parent. We call a chain an **isolated chain** if no chain can join it and it cannot join another chain in any round of contraction until the chain is compressed to a single vertex. One way to avoid concurrent reads is to compress an isolated chain to a single vertex before allowing it to join another chain. When a vertex has only a single unevaluated argument and it is not part of an isolated chain, we say the vertex is **free**, because it is free to form a new isolated chain. Once an isolated chain is compressed to a single vertex, that vertex is made free. Algorithm 2.14 displays a high level description of *Isolate_Contract*, a deterministic algorithm for parallel tree contraction. We use the subprocedure *isolate* to prevent vertices not in the chain to become part of the chain and the variable *inChain* to indicate whether a vertex is part of an isolated chain or not. We use the subprocedure *isSingleton* to test whether a chain has been reduced to a single vertex.

The difference between the contraction phase used in this algorithm and the contraction phase of *Basic_Contract* is that the COMPRESS is replaced by two operations: ISOLATE and LOCAL COMPRESS. ISOLATE marks each chain so that it cannot become part of another chain. Each LOCAL COMPRESS applies one conventional COMPRESS operation to an isolated chain during each contraction phase.

Implementation Techniques We present one method of implementing the generic contraction phase on an EREW model in $O(\log n)$ time, using n processors. Recall that when we compress a chain using Wyllie's algorithm, two chains are created, one essential and one not useful. We modify

ALGORITHM **2.14**

ISOLATE *and* CONTRACT *for Deterministic Parallel Tree Contraction*

```
Procedure Isolate_Contract
   In Parallel inChain[v] := false;                         /* Initialize */
   end in parallel

   In Parallel while Arg(v) = 0 do
      if P[v] ≠ nil then do

         Parallel Case Arg(v) equals
         0) rake(v);                                           /* RAKE */
            mark(label[P[v], index[v]]);
            P[v] := nil;

         1) if not inChain[v] then                          /* ISOLATE */
               isolate(v);
               inChain[v] := true;
            else                                       /* LOCAL COMPRESS */
               compress(v);
               if isSingleton(v) then inChain[v] := false;
         end case

      end then
   end in parallel
end Isolate_Contract
```

Wyllie's pointer-jumping algorithm so that processors that pointer jump over nonessential chains eventually stop before a concurrent read take place. Any chain found in one round is isolated so that it cannot join any other chain found in succeeding rounds. By isolating any chain of length two or more and compressing it until it becomes a single vertex, we can ensure that all jumping over nonessential chains stops before this single vertex is free to become part of another chain.

One way to isolate a chain is to mark the vertices of the chain as being either the *head*, a *middle* vertex, or the *tail* of the chain. The head of the chain is the first vertex in the chain and its child is either a leaf or has more than one child. The tail of the chain is the last vertex in the chain and its parent has more than one child. A middle vertex lays somewhere between the head and the tail of the chain.

An essential chain contains the head of the original chain which will eventually be evaluated during the contraction phase. After several rounds of pointer jumping, all the vertices of the chain will point to v, the parent of the chain. To avoid having all vertices of the chain trying to find the parent of v, we only allow the head of the chain to jump over v. That is, only the head of the chain can be free to form part of a new chain. All other vertices of the chain stop pointer jumping.

Figure 2.25 shows the isolation and compression of a chain and the tagging of vertices as described below. To isolate a chain we **tag** the vertices of the chain with one of three possibilities: R, M, or T, depending on whether it is the *tail*, *middle*, or *head* of a chain. If a vertex is not part of an isolated chain it is tagged with $\emptyset$. All vertices are initially tagged with $\emptyset$. During the COMPRESS phase, the tail of a chain does no pointer jumping, because it is isolated from any newly isolated chains in front of it. Whenever a middle vertex of a chain jumps over the tail, that middle vertex becomes a tail of a new chain. That is, every round of pointer jumping creates one new chain with a new tail. Eventually, every middle vertex becomes a tail of some nonessential chain and stops pointer jumping. At the same time or at the next round, the head of the chain jumps over the tail; the essential chain is a single vertex, the head of the chain. It is at this point the head becomes "free" to join a new isolated chain. A vertex v is *free* if $Arg(v) = 1$ and $Tag = \emptyset$; otherwise v is *not free*. Thus, the tag of the head is set to $\emptyset$ to indicate that it is now not part of an isolated chain.

When a vertex v is free to become part of a new isolated chain, we need to determine its new tag. A vertex is the tail of a chain if its parent is not free and its child is free; it is the head of a chain if its parent is free and its child is not free; and it is a middle of a chain if both its parent and child are free. In this way, isolated chains contain at least two vertices, a tail and a head. To determine whether a child is free, each child vertex maintains a boolean variable, *free*, that its parent can read to determine whether it is free or not. No concurrent reads can take place because each vertex has only one parent. But, because a vertex can be the parent of several chains, we must be sure that the tails of new isolated chains do not attempt a concurrent read. Therefore, each parent vertex v also maintains an array *pFree* that indicates whether v is free or not multiple times and is indexed by the index of its children (i.e., each child reads one array entry exclusively). Initially all the

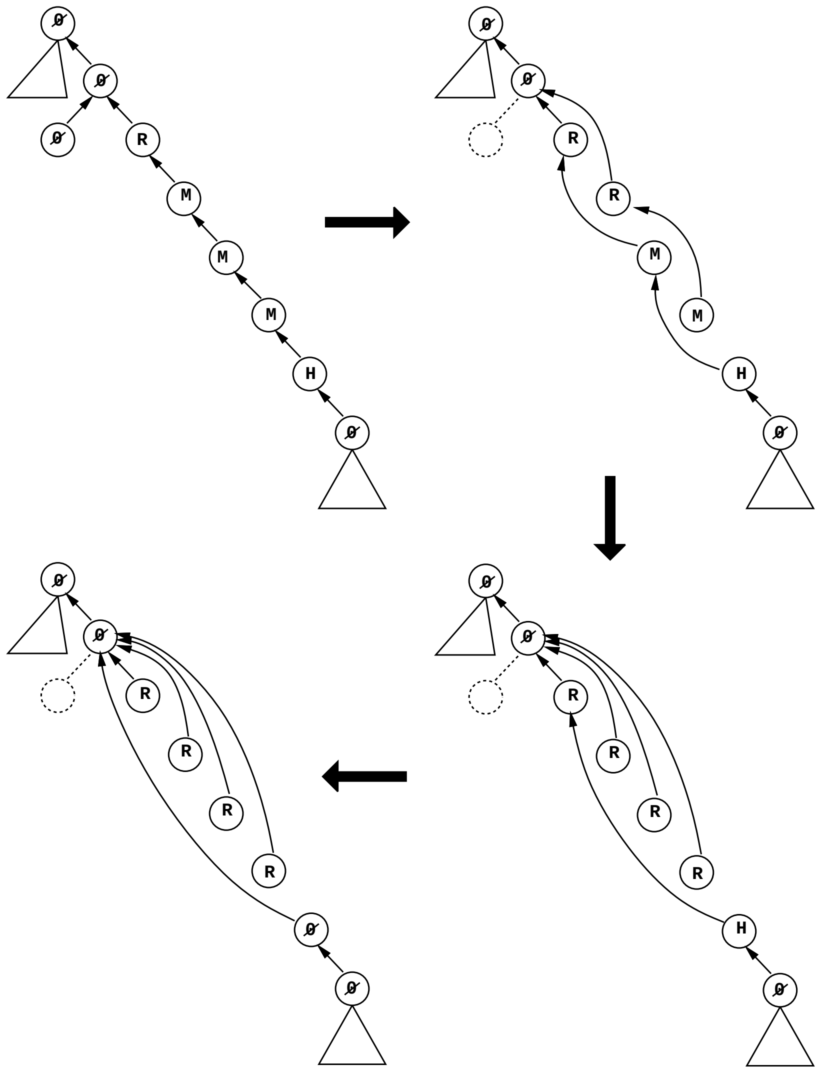

FIGURE 2.25
Three applications of *Isolate_Contract*. Root, middle and head nodes of an isolated chain are labeled R, M and T respectively. Nodes not part of an isolated chain are tagged with $\emptyset$.

elements of the array indicate that a vertex is not free. When a parent vertex becomes free it has only one unevaluated child remaining. We assume that the index of this child is set into the variable *child* by the procedure *Arg*. Thus, the free parent need only update the array entry indexed *child*, since there are no other children to read the other entries. Algorithm 2.15 shows an implementation of the *Isolate_Contract* procedure.

LEMMA 2.7

After each application of Isolate_Contract $|Com|$ decreases by a factor of at least 1/4.

PROOF

By the way the ISOLATE operation is implemented, each isolated chain has a length of at least 2. Moreover, every pair of free vertices is part of an isolated chain, i.e., no two consecutive vertices are singleton vertices. Thus, at every phase of the algorithm, the whole length of a chain consists of a sequence of isolated chains interspersed with singleton vertices. In the worst case, a complete chain consists of an alternating sequence of a singleton and an isolated chain of length three, possibly followed by a singleton. Recall that *Com* does not include the head of a chain so that the final singleton can be ignored. In such a case, after a single COMPRESS only one vertex in each isolated chain is eliminated and all the other vertices in the chain remain; that is only one in four vertices is eliminated. ■

Since step RAKE removes $1/5$ of *Ra* and steps ISOLATE and LOCAL COMPRESS remove $1/4$ of *Com*, together they must remove $1/5$ of the vertices. This gives the following theorem.

THEOREM 2.9

A tree of n vertices is reduced to its root after one applies Isolate_Contract $\lceil \log_{5/4} n \rceil$ times.

THEOREM 2.10

Tree contraction can be performed, deterministically, in $O(n/P)$ time using P processors on an EREW PRAM for all $P \leq n/\log n$.

ALGORITHM **2.15**

An implementation of Isolate_Contract

```
Procedure Isolate_Contract_by_Tagging
   In Parallel Tag[v] := ∅;

   In Parallel while Arg (v) = 0 do
      if P[v] ≠ nil then do

         Parallel Case Arg (v) equals
         0)  rake (v);                                          /* RAKE */
             mark (label[v]);
             P[v] := nil;

         1)  Parallel Case Tag[v] equals
             ∅)  free[v] = true;                                /* ISOLATE */
                 pFree[v, child[v]] = true;
                 if pFree[P[v], index[v]] then
                      if free[child[v]] then
                           Tag[v] := M;
                      else
                           Tag[v] := H;
                 else if free[child[v]] then
                      Tag[v] := R;

             M)  if Tag[P[v]] = R then Tag[v] := R;
                 compress (v); P[v] := P[P[v]];          /* LOCAL COMPRESS */

             H)  if Tag[P[v]] = R then Tag[v] := ∅;           /* isSingleton */
                 compress (v); P[v]; = P[P[v]] ;          /* LOCAL COMPRESS */
             end case
         end case

      end then
   end in parallel
end Isolate_Contract_by_Tagging
```

The Expansion Phase If we use procedure *Isolate_Contract* to evaluate an arithmetic expression it will not return the value of all subexpressions. To compute the value of all subexpressions we run the contraction phase "backwards" in a parallel tree expansion phase. Because *Basic_Expand* in Section 2.2.3 only expands along essential chains *Basic_Expand* uses only exclusive reads and writes. Therefore, we can follow *Isolate_Contract* with *Basic_Expand* to obtain the value of all subexpressions.

If we used *Isolate_Contract* in conjunction with *m-Contract* we need to compute the value of all subexpressions for edge and top bridges. Again, processors can simply re-evaluate the bridges for which they are responsible, as they would leaf bridges, using the now known values of their leaf attachments.

2.2.6 Parallel Tree Contraction for Trees of Unbounded Degree

The discussion on *Basic_Contract* in Section 2.2.3 assumed that the tree was of bounded degree. When the tree has unbounded degree, two problems can arise with *Basic_Contract*. One is that we need a way to test whether $Arg(v)$ equals 0 or 1 in constant time. The other is that the time to perform a Rake may depend on the number of children of a vertex, and hence is not constant time. For this latter problem we show that by simply running Rake and Compress asynchronously we continue to get $O(\log n)$ running time.

If the number of arguments per vertex is bounded we can test whether $Arg(v)$ equals 0 or 1 in constant time by counting the number of unmarked labels for v using the processor assigned to it. But when the number of arguments is unbounded we need to use the processors of the children vertices to compute $Arg(v)$. We start by setting aside a memory location $argIndex[v] = null$. Each processor that does not know the value of its vertex writes its index into the memory location of its parent. That is, all children that do not know their values do a concurrent write to their parent $argIndex[v]$ as shown in Algorithm 2.16. Assume that one of these children succeeds in writing its index. If $argIndex[v] = null$ then all the children know their value and $Arg(v) = 0$. To test whether $Arg(v) = 1$, each processor that doesn't know its value reads $argIndex$ of its parent and if the value is not the same as its own index it rewrites its index to $argIndex$ of its parent. If the value of $argIndex$ does not change then only one child attempted to write to it initially, implying that $Arg(v) = 1$. Otherwise, $Arg(v) \geq 2$.

Up until now we have assumed that the *rake* operation could be performed in constant time. For many applications this is not the case. Miller

ALGORITHM **2.16**

Determining the number of unknown arguments of a vertex of a tree with unbounded degree on a CRCW PRAM

```
Function Arg(v)
    argIndex[v] := null;
    if val[v] = null then                       /* concurrent write to parent */
        argIndex[P[v]] := index[v];
    if argIndex[v] = null then                  /* all args known */
        return 0;
    oldArg[v] := argIndex[v];
    if val[v] = null then
        if argIndex[P[v]] ≠ index[v] then       /* concurrent read */
            argIndex[P[v]] := index[v];         /* concurrent rewrite */
    if oldArg[v] = argIndex[v] then             /* no rewrites*/
        return 1;
    else                                        /* 1 or more rewrites */
        return 2;
end Arg(v)
```

and Reif [MR90] show that finding canonical labels for trees has a *rake* operation that is considerably more complicated than just deletion. It requires sorting the labels assigned to the children of a vertex, which requires $O(\log n)$ time. Thus, the parallel time of raking the leaves of a vertex with k children is $O(\log k)$. If we require one application of CONTRACT to finish completely before we start the next application of CONTRACT then total cost to reduce a tree to its root is the cost for *rake* times the logarithm of the size of the tree. Thus, the naive analysis for canonical labels would be that it runs for $O(\log^2 n)$ time. We improve the running time by a factor of $\log n$ below.

We modify parallel tree contraction so that for those parts of the tree where CONTRACT has already finished we implement a new round of CONTRACT, i.e., each processor executes CONTRACT asynchronously. We shall assume that the time used to remove the leaves of a given vertex is only a function of the number of leaves at that vertex. We should point out that the synchronous and asynchronous versions of CONTRACT may return very different answers. For example, when computing canonical forms for trees by sorting leaves both the synchronous and asynchronous algorithms are correct.

However, the two algorithms produce different sets of canonical labels. In addition, the asynchronous version is faster.

Asynchronous_Contract can be described graph theoretically by viewing it as operating on trees with special leaves which we call *phantom leaves*. The algorithm runs in stages. Initially the tree T has no phantom leaves. We apply the procedure CONTRACT to T to obtain the tree T'. If a given vertex $v \in T$ has $k \geq 2$ children that are nonphantom leaves then we replace them with a new phantom leaf $w \in T'$. Furthermore, if the time required for *Asynchronous_Contract* to process these k children of v is t then the phantom vertex w persists for t stages, at which time it simply disappears. Every time a new block of children of v become leaves a new phantom child replaces them. In this way a vertex may have several phantom children. Until all leaves, including phantom leaves, of a vertex are removed the vertex is not a leaf. The time to execute *Asynchronous_Contract* is the number of stages it takes to reduce the tree to its root and for all phantom leaves to disappear.

THEOREM 2.11

If the cost to rake a vertex with k children is bounded by $O(\log k)$ then Asynchronous_Contract requires only $O(\log n)$ time.

PROOF

Suppose the time to *rake* k children of a vertex is bounded by $c \log k$ for $k \geq 2$ and *rake* for a single child can be performed in unit time. We analyze the time used by *Asynchronous_Contract* using an amortization argument. We assign weights to the vertices of the tree such that, at any stage of the algorithm, the weight of the tree reflects the progress made so far. We show that the weight for the tree as a whole decreases by a constant proportion at each stage.

The problem is that *Asynchronous_Contract* can go through several stages removing no leaves, and then remove many leaves. We want to be able to say that at every stage a constant proportion of the work is done. By introducing phantom leaves for accounting purposes, we can take some credit at each stage for the work performed by *rake*, while making sure that the overall credit for raking k leaves remains k. Therefore, we introduce the notion of a weighted tree. A **weighted tree** is a tree with weights assigned to the vertices. The weight of a tree is the sum of the weights of the vertices in the tree. In this application all vertices have weight 1, except phantom leaves which may have arbitrary real weights greater than or equal to 1. Initially, the weight of the tree is the size of the tree.

First we describe in more detail how weights are assigned to phantom leaves. Suppose the time required to rake the k non-phantom leaves of a vertex v is $f(k)$. There is a subtlety here; if the time to rake k leaves of a vertex varies from vertex to vertex, the way the tree contracts may vary dramatically. Our analysis only depends on an upper estimate for the time to rake the children of a vertex. We define β to be a function of k, such that $\beta^{f(k)-1}(k) = 1$ for $f(k) > 0$. Hence $\beta(k) < 1$ for all $k \geq 2$. The constant $\beta(k)$ is the rate at which the phantom leaf decays. We replace the k leaves of vertex v with a phantom leaf w, which we give weight k. After each successive stage we decrease the weight on w by a factor of $\beta(k)$ until the weight equals one. In the next stage we simply delete the phantom leaf w. Thus, the phantom leaf w exists for $f(k)$ stages, at which time it is deleted. Note that the weight of a phantom vertex is always at least 1.

As in the proof of Theorem 2.4 we partition the vertices of T into two sets, Ra and Com. We claim that the weight of Com decreases by a factor of $1/2$ at each stage while the weight of Ra decreases by a factor of at least $(4+\beta)/5$ at each stage, where $\beta = \max\{\beta(k) | 1 \leq k \leq n\}$. Note that different phantom leaves decay at different rates. We have picked β to be the slowest such rate. The fact that Com decreases by $1/2$ follows by noting that the vertices in Com are processed the same way as in CONTRACT and their weights are all one. Next we consider the case of Ra. Recall that $Ra = V_0 \cup V_2 \cup C_0 \cup C_2 \cup GC_2$, where V_0 is the set of leaves and phantom leaves. Since the weight on any vertex in V_0 is at least one and the weight of any vertex not in V_0 is 1 we see that that weight of V_0 is at least $1/5$ of the weight of Ra. On the other hand the weight of V_0 decreases by at least β at each stage. Thus, the weight of Ra decreases by at least a factor of $4/5 + \beta/5$ at each stage.

This shows that the number of stages is bounded by $\log n$ base $5/(4+\beta)$. For a particular case of interest, when $f(k) \leq c \log k$ for some constant c and $k \geq 2$, we see that β is bounded away from 1 for all n. This proves the Theorem. ∎

2.2.7 Conclusions

In this section we have shown the paradigm of parallel tree contraction, which can be used to perform computations over trees efficiently and is quite general. This paradigm often supplies computationally superior parallel algorithms than algorithms that use divide-and-conquer. In addition, the algorithms are usually easier to devise and understand.

We introduced several concepts that were important in the design of parallel tree algorithms. First we introduced RAKE and COMPRESS which allowed us to separate the processing of leaf vertices from internal vertices. For many applications, finding the *rake* subprocedure is obvious, especially when the natural flow of information is from the bottom up. COMPRESS requires finding a way to combine the information for a pair of vertices in a chain succinctly. For example, for expression trees defining the *compress* subprocedure simply requires finding an efficient representation for the unary function that represents an expression tree with one unknown leaf, and then applying function projection and composition.

When an application either requires results for all subtrees or the natural flow of information is from the top down, an addition expansion phase is required. This expansion phase is simply the contraction phase run in reverse using *unrake* and *uncompress* subprocedures. Often they are similar to their counter parts. However, when information flows from the bottom up and the top down, *unrake* can be quite different from *rake*.

The next major concept we introduced was SHUNT for binary trees. It is possible to use shunt because there is an efficient algorithm to number leaves. SHUNT was applied only to odd numbered leaves, and combined RAKE and COMPRESS into a single operation. However, in designing a *shunt* subprocedure, it is usually easier to think of it as a rake followed by a compress on the sibling vertex.

The next major concept was the notion of dividing a tree into blocks of subtrees on which processors could work. The processors reduce the subtrees to single edges forming a smaller tree. The number of vertices in this tree equals the number of processors so that, at this point, a nonoptimal algorithm can be used to reduce the tree to the root.

To obtain an EREW algorithm we introduced the concept of isolating a chain, so that new vertices cannot join the chain, and reducing the chain to a point before allowing it to become part of a new chain. By isolating chains we prevented concurrent reads and writes.

The basic tree contraction algorithm is not optimal and uses concurrent reads and writes. It runs in $O(\log n)$ time using n processors, as long as *rake* and *compress* take $O(1)$ time. If *rake* takes as much as $O(\log n)$ time and we can rake and compress different parts of the tree asynchronously, then we still only use $O(\log n)$ time overall. The remaining algorithms are optimal and use only exclusive reads and writes. *Shunt* has somewhat greater constants and can only be used on binary trees, which limits its applicability, whereas the constants for the m-contract/isolation algorithms are greater than the other tree contraction algorithms, making them less practical.

TABLE 2.2
Time and processor count for the parallel tree contraction algorithms discussed in this section.

Problem	**Time**	**Processors**	**Work**
basic tree contraction	$O(\log n)$	$O(n)$	$O(n \log n)$
tree contraction for binary trees	$O(\log n)$	$n/\log n$	$O(n)$
deterministic tree contraction	$O(n/P)$	P, $P \leq n/\log n$	$O(n)$
tree contraction, unbounded degree	$O(\log n)$	$n/\log n$	$O(n)$

Bibliography

[ADKP87] K. Abahamson, N. Dadoun, D. K. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm (preliminary version). In *Proceedings of the 25th Annual Allerton Conference on Communication, Control, and Cumputing*, pages 624–633, Monticello, Illinois, Sept/Oct 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AM88] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88*, pages 81–90, N.Y., June/July 1988. Springer-Verlag. Lecture Notes in Computer Science, Vol. 319.

[AM90] Richard J. Anderson and Gary L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, January 1990.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[BOV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree form. *ACM Transactions on Programming Languages and Systems*, 7(2):348–357, April 1985.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal Assoc. Computing Machinery*, 21(2):201–208, April 1974.

[Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23, 1952.

[CV86a] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *27th Annual Symposium on Foundations of Computer Science*, pages 478–491, Toronto, Oct 1986. IEEE.

[CV86b] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal list ranking. *Information and Control*, 70(1):32–53, 1986.

[Fic83] Faith E. Fich. New bounds for parallel prefix circuits. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 100–109, Boston,MA, April 1983. ACM.

[GMT88] H. Gazit, G. L. Miller, and S-H Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, New York, 1988. Plenum Press. Princeton Workshop on Algorithms, Architecture and Technology Issues for Models of Concurrent Computation.

[KD88] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computation by ranking (extended abstract). In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88*, pages 101–110. Springer-Verlag, N.Y., June/July 1988. Lecture Notes in Computer Science, Vol. 319.

[KU86] Anna Karlin and Eli Upfal. Parallel hashing–an efficient implementation of shared memory. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 160–168, Berkeley, May 1986. ACM.

[LF80] Richard E. Ladner and Michael J. Fisher. Parallel prefix computation. *Journal Assoc. Computing Machinery*, 27(4):831–838, October 1980.

[Mil86] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, June 1986. invited publication.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, 1989. Vol. 5.

[MR90] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: Further applications. *SIAM J. Comput.*, 1990. to appear.

[MT87] Gary L. Miller and Shang-Hua Teng. Systematic methods for tree based parallel algorithm development. In *Second International Conference on Supercomputing*, pages 392–403, Santa Clara, May 1987.

[Ran87] A. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, Los Angeles, Oct 1987. IEEE.

[Vis84] U. Vishkin. Randomized speed-ups in parallel computation. In *Proc. of the 16th Annual ACM Symp. on Theory of Computing*, pages 230–239, Washington D.C., April 1984. ACM.

[Wyl79] J. C. Wyllie. The complexity of parallel computation. Technical Report TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.