MIT/LCS/TM-318

COMMUNICATION-EFFICIENT PARALLEL GRAPH ALGORITHMS

CHARLES E. LEISERSON
BRUCE M. MAGGS

DECEMBER 1986

Communication-Efficient Parallel Graph Algorithms

Charles E. Leiserson
Bruce M. Maggs

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract—Communication bandwidth is a resource ignored by most parallel random-access machine (PRAM) models. This paper shows that many graph problems can be solved in parallel, not only with polylogarithmic performance, but with efficient communication at each step of the computation. We measure the communication requirements of an algorithm in a model called the *distributed random-access machine* (DRAM), in which communication cost is measured in terms of the congestion of memory accesses across cuts of an underlying network. The algorithms are based on a communication-efficient variant of the tree contraction technique due to Miller and Reif.

Key Words: communication, fat-trees, graph theory, load factor, parallel algorithms, randomized algorithms, tree contraction, volume-universal networks.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract N00014–80–C–0622. Charles Leiserson is supported in part by an NSF Presidential Young Investigator Award with matching funds provided by AT&T Bell Laboratories and Xerox Corporation. Bruce Maggs is supported in part by an NSF Fellowship.

1. Introduction

Underlying any realization of a parallel random-access machine (PRAM) is a communication network that conveys information between processors and memory banks. Yet in most PRAM models, communication issues are largely ignored. The basic assumption in these models is that in unit time each processor can simultaneously access one memory location. For truly large parallel computers, however, computer engineers will be hard pressed to implement networks with the communication bandwidth demanded by this assumption. The difficulty of building such networks threatens the validity of the PRAM as a predictor of algorithmic performance. This paper introduces a more restricted PRAM model, which we call a *distributed random-access machine* (DRAM), to reflect an assumption of limited communication bandwidth in the underlying network.

We measure the cost of communication in a network in terms of the number of messages that must cross a cut of the network, as in [8] and [11]. Specifically, a *cut* $S = (A, \overline{A})$ of a network[1] is a partition of the network into two sets of processors A and $\overline{A}$. The *capacity* $\text{cap}(S)$ is the number of wires connecting processors in A with processors in $\overline{A}$, i.e., the bandwidth of communication between A and $\overline{A}$. For a set M of messages we define the *load* of M on a cut $S = (A, \overline{A})$ to be the number of messages in M between a processor in A and a processor in $\overline{A}$. The *load factor* of M on S is

$$\lambda(M, S) = \frac{\text{load}(M, S)}{\text{cap}(S)},$$

and the *load factor* of M on the entire network is

$$\lambda(M) = \max_{S} \lambda(M, S).$$

The load factor provides a simple lower bound on the time required to deliver a set of messages. For instance, if there are 10 messages to be sent across a cut of capacity 3, the time required to deliver all 10 messages is at least the load factor 10/3.

There are two commonly occurring types of message congestion that the load factor measures effectively. One is the "hot spot" phemomenon identified by Pfister and Norton [17]. When many processors send messages to a single other processor, large delays can be experienced as messages queue for access to that other processor. In this situation, the load factor on the cut that isolates the single processor is high. The second phenomenon is message congestion due to pinboundedness. In this case, it is the limited bandwidth imposed by the technology that can cause high load factors. For example, the cut of the network that limits communication performance might correspond to the pins on a printed-circuit board or to the cables between two cabinets.

[1] We assume that the communication network is an *interconnection network*, meaning that the processors are interconnected as a graph, and routing of messages is performed by the processors. The generalization to a *routing network*, where routing can be done by switches that are not processors, is straightforward, but complicates the definitions.

The load-factor lower bound can be met to within a polylogarithmic factor as an upper bound on many networks, including volume and area-universal networks, such as fat-trees [8,11] and meshes of trees [12], as well as the standard universal routing networks, such as the Boolean hypercube, the butterfly (a.k.a. FFT, Omega), and the cube-connected cycles. The lower bound is weak on the standard universal routing networks because every cut of these networks is large relative to the number of processors in the smaller side of the cut, but these networks may be more difficult to construct on a large scale than the volume and area-universal networks [11]. Networks for which the load factor lower bound cannot in general be approached to within a polylogarithmic factor as an upper bound include linear arrays, meshes, and high-diameter networks in general.

Whereas communication is essentially free in PRAM models, the cost of communication in a DRAM depends on the locality of memory accesses as measured by the load factor of an underlying network. The DRAM is an attempt to abstract the essential communication characteristics of volume and area-universal networks without relying in detail on any particular network. Much as the PRAM can be viewed as an abstraction of a hypercube, in that algorithms for a PRAM can be implemented on a hypercube with only polylogarithmic performance degradation, the DRAM can be viewed as an abstraction of a volume or area-universal network. Fast, communication-efficient algorithms on a DRAM translate directly to fast, communication-efficient algorithms on, for example, a fat-tree.

This paper shows that many graph problems for a graph $G = (V, E)$ can be efficiently solved with $O(|E|)$ processors in the DRAM model. The algorithms we give apply to all of the popular PRAM models because a PRAM can be viewed as a DRAM in which communication costs are ignored. In fact, the algorithms we give can all be performed on an exclusive-read, exclusive-write DRAM, and when run on a PRAM, they are nearly as efficient in the PRAM model as corresponding concurrent-read, exclusive-write PRAM algorithms in the literature.

The remainder of this paper is organized as follows. Section 2 contains a specification of the DRAM model and the implementation of data structures in the model. The section demonstrates why the "recursive doubling" technique frequently used in parallel algorithms is inefficient in the DRAM model. It also defines the notion of a *conservative algorithm* as a concrete realization of a communication-efficient algorithm, and gives a "Shortcut Lemma" that forms the basis of the conservative algorithms in this paper. Section 3 presents a conservative "recursive pairing" technique that can be used to perform many of the same functions as recursive doubling. Section 4 presents a linear-space, conservative "tree contraction" algorithm based on the ideas of Miller and Reif [16]. Section 5 presents *treefix computations,* which are generalizations of the parallel prefix computation [3,7,18] to trees. We show that treefix computations can be performed using the tree contraction algorithm of Section 4. Section 6 gives short, efficient parallel algorithms for tree and graph problems, most of which are based on treefix computations. Section 7 contains some concluding remarks.

2. The DRAM model

This section presents the DRAM model. We show how a data structure can be embedded in a DRAM, and we define the load factor of a data structure. We demonstrate that many existing PRAM algorithms are not communication efficient in the DRAM model by examining the "recursive doubling" technique [22] used extensively by algorithms in the literature. We introduce the notion of a *conservative* algorithm as one in which the load factor of each set of memory accesses can be bounded above by the load factor of the input data structure. Our conservative algorithms are based on a simple lemma that shows how pointers in a data structure can be "shortcut" without increasing the load factor.

A DRAM consists of a set of n processors. All memory in the DRAM is local to the processors, with each processor holding a small number of $O(\lg n)$-bit registers. A processor can read, write, and perform arithmetic and logical functions on values stored in its local memory. It can also read and write memory in other processors. (A processor can transfer information between two remote memory locations through the use of local temporaries.) Each set of memory accesses is performed in a memory access *step*, and any of the standard PRAM assumptions about simultaneous reads or writes can be made. Our algorithms use only mutually exclusive memory references, however, so these special cases never arise.

The essential difference between a DRAM and a PRAM is that the DRAM models communication costs. We presume remote memory accesses are implemented by routing messages through an underlying network. Each cut $S = (A, \overline{A})$ of the network has an assigned capacity $\text{cap}(S)$. For a set M of memory accesses, we define $\text{load}(M, S)$ to be the number of accesses in M between a processor in A and a processor in $\overline{A}$. The load factor of M on S is $\lambda(M, S) = \text{load}(M, S)/\text{cap}(S)$, and the load factor of M on the entire network is $\lambda(M) = \max_S \lambda(M, S)$. The basic assumption in the DRAM model is that *the time required to perform a set M of memory accesses is $\lambda(M)$.*

A natural way to embed a data structure in a DRAM is to put one record of the data structure into each processor. The record can contain pointers to records in other processors, as well as auxilliary local storage. We measure the quality of an embedding by generalizing the concept of load factor to a set of pointers. The load of a set P of pointers across a cut $S = (A, \overline{A})$, denoted $\text{load}(P, S)$, is the number of pointers in P from a processor in A to a processor in $\overline{A}$ or vice versa, and the load factor is $\lambda(P) = \max_S \text{load}(P, S)/\text{cap}(S)$. The load factor of a data structure is the load factor of the set of its pointers. For many problems, good embeddings of data structures can be found in particular networks for which the DRAM is a good abstraction (see Section 7).

The embedding of a data structure can influence the performance of a DRAM algorithm. As an example, consider the embedding of a simple linear list in which alternate list elements are placed on opposite sides of a narrow cut. If each element fetches a value from the next element in the list, the load factor across the cut is large. Thus, a set of memory accesses that theoretically takes unit time in the PRAM model may actually take considerably more time due to network congestion. On the other hand, there are better embeddings for the list in which the number of list pointers crossing any cut is small compared to the capacity of the cut.

There are generally two situtations in which message congestion can arise during the execution of a DRAM algorithm. In the first situation, which we have just seen, the embedding of a data structure in the network causes congestion because many of its pointers cross a relatively small cut of the network. A parallel access of the information across those pointers generates substantial message traffic across the cut. In the second situation, the data structure is embedded with few pointers crossing the cut, but the algorithm itself generates substantial message traffic across the cut. The focus of this paper is this second cause of congestion.

To see how a parallel algorithm can generate congestion, consider the "recursive doubling" or "pointer jumping" technique which is used extensively by algorithms in the literature. The idea is that each element i of a list initially has a pointer $p(i)$ to the next element in the list. At each step, element i computes $p(i) \leftarrow p(p(i))$, doubling the distance $d(i)$ between i and the element it points to (until it points to the end of the list). This technique can be used, among other things, to compute the distance of each element to the end of the list. For each element i, $d(i)$ is initially one. At each pointer jumping step, element i computes $d(i) \leftarrow d(i) + d(p(i))$. In a PRAM model, the running time on a linked list of length n is $O(\lg n)$. Variants of this technique are used for path compression, vertex numbering, and parallel prefix computation [16,19,21,22].

In the DRAM model, recursive doubling can be expensive even when a data structure has a good embedding. Figure 1 shows a cut of capacity 3 separating the two halves of a linked list of 16 elements. In the first step of recursive doubling, the load on the cut is only 1 because the only access across the cut occurs when element 8 copies the pointer of element 9. In the second step the load is 2 because element 7 accesses element 9 and element 8 accesses element 10. In the third step, the load is 4, and in the fourth step, as each of the first eight elements makes an access across the cut, the load is 8. Since the load factor of the cut in the fourth step is 8/3, this set of accesses requires a least 3 time units. Whereas the capacity of the cut is large enough to support the memory accesses across it in the first step, by the fourth step it is insufficient. Unless every cut of the network is sufficiently large to accommodate worst-case communication patterns, the performance of recursive doubling suffers due to message congestion, a phenomenon not predicted by analysis in the PRAM model. In the next section, we shall show how a *recursive pairing* strategy can perform many of the same functions as recursive doubling in a communication-efficient fashion.

All of our algorithms have the property that the load factor of memory accesses in any step is bounded by the load factor of the input data structure. We define a set M of memory accesses to be *conservative* with respect to another set M' of memory accesses if $\lambda(M) \leq \lambda(M')$, and we make the natural generalization of this

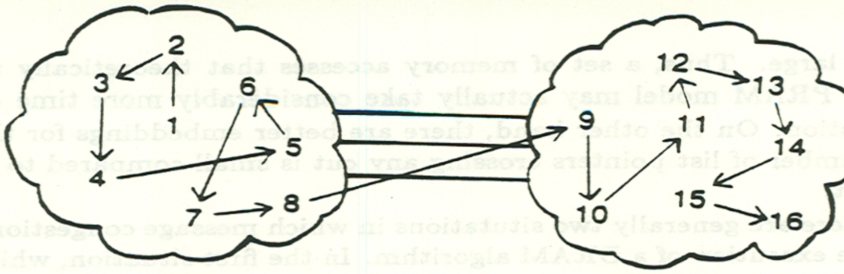

Figure 1: A cut of capacity 3 separating two halves of a linked list. The load of the list on the cut is 1. At the final step of recursive doubling, each element on the left side of the cut accesses an element on the right, which induces a load of 8 on the cut.

definition to pointers and data structures. A *conservative algorithm* is one all of whose memory accesses are conservative with respect to the input data structure.

An algorithm that communicates only across pointers in the input data structure is conservative, but may require time linear in the diameter of the data structure to pass information between two elements. The following simple, but important, lemma shows how to shortcut pointers in the input data structure without increasing communication requirements.

Lemma 1 (Shortcut Lemma) *Suppose a set P of pointers in a data structure contains pointers $a \to b$ and $b \to c$. Then the set P' of pointers defined by*

$$P' = P \cup \{a \to c\} - \{a \to b, b \to c\}$$

is conservative with respect to P.

Proof: We shall show that $\lambda(P', S) \leq \lambda(P, S)$ for any cut S of the underlying network, which implies that $\lambda(P') \leq \lambda(P)$. Consider the eight ways in which a, b, and c can be assigned to sides of the partition induced by a cut S. Half the cases can be eliminated by symmetry if we assume that a is on the left side. In each of the four remaining cases, the load factor across the cut is either unchanged or diminished when $a \to b$ and $b \to c$ are replaced with $a \to c$, as is shown in Figure 2. ∎

We shall typically use a straightforward generalization of the Shortcut Lemma. Specifically, we can shortcut any set of pointer-disjoint paths in a data structure without increasing the load factor.

Because the algorithms presented in this paper are based on the Shortcut Lemma, they are not only conservative, but they are also independent of the cut capacities of the DRAM and of the embedding of an input data structure in the DRAM. Thus, independent of the underlying network, the algorithms are correct, and if the embedding of the input data structure is good, the algorithms run fast. Moreover, for a specific embedding on a specific DRAM, the running time can be analyzed precisely.

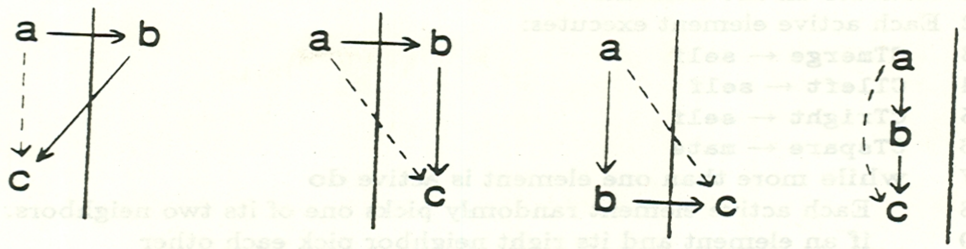

Figure 2: The Shortcut Lemma. In each of the four cases illustrated, the load factor across the cut is either unchanged or diminished by replacing $a \to b$ and $b \to c$ with $a \to c$.

3. List contraction

In this section we present a conservative "recursive pairing" algorithm, Algorithm LC, that can perform many of the same functions on lists as recursive doubling. The idea is to repeatedly "contract" the list until it consists of a single node. To record the contractions, we construct a binary *contraction tree* whose leaves are the elements in the list. After building the contraction tree, operations such as broadcasting from the root or parallel prefix can be performed in a conservative fashion. Algorithm LC is a randomized algorithm, and with high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$. A deterministic variant based on deterministic coin tossing [5] runs in $O(\lg n \lg^* m)$ steps, where m is the number of processors in the DRAM, and produces trees of height $O(\lg n)$.

Algorithm LC requires a constant amount of extra space for each element in the input list. Each processor contains two elements, an element in the list, and a *spare* element that will act as an internal node in the contraction tree. We call the two elements in the same processor *mates*. Each element holds a pointer to an unused internal node, which for each list element initially points to its mate. The use of spare nodes allows the algorithm to distribute the space for the internal nodes of the contraction tree uniformly over the elements in the list. Spare internal nodes are used in [1] and [13] for similar reasons, but in a different context.

Figure 3 shows Algorithm LC. Each element has a `self` register that points to itself, a `mate` register that points to its mate, and two registers `left` and `right` that point to the next and previous elements in the list. The `left` register of the first element of the list and the `right` register of the last element point to the elements themselves. An element also has three pointers, `CTmerge`, `CTleft`, and `CTright`, that represent the element's parent and left and right children in the contraction tree. To simplify boundary conditions for the algorithm, these three registers initially point to the element itself. Finally, each element holds a pointer `CTspare` to an unused internal node, which for each list element initially points to its mate.

```
Activate all list elements.
Each active element executes:
  CTmerge ← self
  CTleft ← self
  CTright ← self
  CTspare ← mate
  while more than one element is active do
    Each active element randomly picks one of its two neighbors.
    if an element and its right neighbor pick each other
      then the element executes:
      CTmerge ← CTspare
      right.CTmerge ← CTmerge
      CTmerge.CTspare ← right.CTspare
      CTmerge.CTmerge ← CTmerge
      CTmerge.CTleft ← self
      CTmerge.CTright ← right
      CTmerge.left ← left
      CTmerge.right ← right.right
      Deactivate the element and its right neighbor, and activate its spare.
    Each active element executes:
      left ← left.CTmerge
      right ← right.CTmerge
```

Figure 3: Algorithm LC. This conservative algorithm constructs a contraction tree for an input list by recursively pairing and merging elements of the list.

Algorithm LC proceeds as follows. In each iteration, each element in the list randomly picks either its left or right neighbor. The leftmost and rightmost elements always pick their single neighbor. If two elements pick each other, they merge, and the left element takes control. A new internal node of the contraction tree is made using the spare of the left element. The spare for the new node is the spare of the right element. The new node's left child is the left element, and its right child is the right element. The new nodes and the unpaired nodes form themselves into a "contracted" list in the terminology of Miller and Reif [16]. The algorithm operates on this contracted list in the next iteration.

In fact, a similar algorithm works for circular lists. Since there are no leftmost or rightmost elements in circular list, each element always randomly picks one of its two neighbors. When a circular list consists of exactly two elements, one is arbitrarily chosen to be the left element in the pair.

To describe the efficiency of randomized algorithms such as Algorithm LC, we shall use the term "with high probability," by which we shall mean "with probability $1 - O(1/n^k)$ for any constant $k > 0$," where n is the size of the input.

Theorem 2 *With high probability, Algorithm LC takes $O(\lg n)$ steps to construct a contraction tree for a list of n elements.*

Proof: We show that the algorithm terminates after $(k+1)\log_{4/3} n$ iterations with probability at least $1-1/n^k$. We use an accounting scheme involving "tokens" to analyze the algorithm. Initially a unique token resides between each pair of elements in the input list. Whenever two list elements pick each other, we destroy the token between them. The key observation is that for each token destroyed, the length of the list decreases by one. Thus the algorithm terminates when no token remains. In any iteration, an existing token has probability at least $1/4$ of being destroyed. Thus after m iterations, a token has probability at most $(3/4)^m$ of remaining in existence. Let T_i be the event that token i exists after m iterations, and let T be the event that any token remains after m iterations. Then by the principle of inclusion and exclusion, the probability that any token remains is given by

$$\begin{aligned}\Pr(T) &= \Pr(T_1 \cup T_2 \cup \ldots \cup T_{n-1}) \\ &\le \Pr(T_1) + \Pr(T_2) + \cdots + \Pr(T_{n-1}) \\ &\le (n-1)\left(\frac{3}{4}\right)^m .\end{aligned}$$

For $m = (k+1)\log_{4/3} n$ iterations we have

$$\begin{aligned}\Pr(T) &\le (n-1)\left(\frac{3}{4}\right)^{(k+1)\log_{\frac{4}{3}} n} \\ &\le \frac{1}{n^k} .\end{aligned}$$

∎

Theorem 3 *With high probability, the height of the contraction tree is $O(\lg n)$.*

Proof: The height of the contraction tree is not greater than the number of iterations of Algorithm LC. ∎

We now prove that Algorithm LC is conservative.

Theorem 4 *Algorithm LC is conservative.*

Proof: The key idea is that the order of the list elements and their spares is preserved by the operation of contraction. By convention, let the mate of an element in the input list lie in the order between that element and its right neighbor. Then in each iteration, an active element's spare lies between the element and its right neighbor in the contracted list. Thus, both the pointers of the contracted list and the pointers between active elements and their spares correspond to disjoint paths in the input list. The memory accesses in a step of the algorithm correspond to a set of pointers between active elements and their left or right neighbors in the contracted list, or to a set of pointers between active elements and their spares. ▌

Once a contraction tree has been constructed, it can be used for broadcasting a value to all of the elements of the list, for accumulating values stored in each element of the list, and more generally, for performing *prefix computations*. Let $\mathcal{D}$ be a domain with a binary associative operation $\oplus$ and an identity ε. A prefix computation [3,7,18] on a list with elements $x_1, x_2, \ldots, x_n$ in $\mathcal{D}$ puts the value $y_i = x_1 \oplus x_2 \oplus \cdots \oplus x_i$ in position i for $i = 1, 2, \ldots, n$.

A prefix computation on a list can be performed by a conservative, two-phase algorithm on the contraction tree. The leaves of the contraction tree from left to right are the elements in the list from x_1 to x_n. The first phase proceeds bottom up on the tree. Each leaf passes its x value to its parent. As the algorithm proceeds, each internal node receives values from its left and right children, call them z_l and z_r. The node saves value z_l, and passes $z_l \oplus z_r$ to its parent. The second phase proceeds top down after the root receives values from its children. The root then passes ε to its left child and its z_l value to its right child. Each child receives a value from its parent, call it z_p, and passes that value to its left child and $z_l \oplus z_p$ to its right child. When a leaf receives z_p it computes $y = z_p \oplus x$.

The algorithm performs the prefix computation in $O(\lg n)$ steps. At each step, the algorithm communicates across a set of pointers in the contraction tree, all of which are the same distance from the leaves in the first phase and the same distance from the root in the second. That this computation is performed in a conservative fashion is a consequence of the following lemma.

Theorem 5 *Let CT be a contraction tree computed by Algorithm LC on an input list L, and suppose P is a set of pointers of CT in edge-disjoint subtrees of CT. Then P is conservative with respect to L.*

Proof: An inorder traversal of CT alternately visits list elements (leaves) and their mates (internal nodes) in the same order that the list elements and mates appear in L. Thus, the pointers in P correspond to disjoint paths in L. By the Shortcut Lemma, any set of pointers that correspond to disjoint paths in the list L are conservative with respect to L. ▌

Algorithm LC, which constructs a contraction tree in $O(\lg n)$ steps, is a randomized algorithm. By using the "deterministic coin tossing" technique of Cole and Vishkin [5] the algorithm can be performed nearly as well deterministically. Specifically, the randomized pairing step can be performed deterministically in $O(\lg^* m)$ steps on a DRAM with m processors, where $\lg^* m$ is the number of times the logarithm function must be successively applied to reduce m to a value no greater than 1. The overall running time for list contraction is thus $O(\lg n \lg^* m)$.

4. Tree contraction

This section presents a conservative tree contraction algorithm, Algorithm TC, based on the tree contraction ideas of Miller and Reif [16]. The algorithm uses a

```
1  Activate all tree elements.
2  Each active element executes:
3     CTmerge ← self
4     CTparent ← self
5     CTleft ← self
6     CTright ← self
7     CTspare ← mate
8     CTlevel ← 1
9     call RPT
10    procedure RPT
11       if more than one element is active then
12          call Pair
13          call Contract
14          call RPT recursively
15          call Expand
```

Figure 4: Algorithm TC. This conservative algorithm contracts an input rooted binary tree to a single node recursively pairing and merging elements of the tree. A contraction tree is contructed to record the order of contractions made.

recursive pairing strategy to build a contraction tree for an input rooted binary tree in much the same manner as Algorithm LC does for a list. With high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$. A deterministic variant runs in $O(\lg n \lg^* m)$ steps.

As in Algorithm LC, each processor in Algorithm TC contains two elements, an element in the input tree and its mate, a spare element that will act as an internal node of the contraction tree. Each element has a `self` register that points to itself, and a `mate` register that points to its mate. Registers `left`, `right`, and `parent` point to the element's left and right children, and parent in the input binary tree. One or more of these three registers may point to the element itself. Each element holds a pointer `CTspare` to an unused element, which initially points to its mate. An element also has pointers `CTmerge`, `CTleft`, `CTright`, and `CTparent` that represent its position in the contraction tree and a `CTlevel` register that records the number of the contraction step in which the node merged with another node. To simplify boundary conditions for the algorithm the `CTmerge`, `CTparent`, `CTleft`, and `CTright` registers of each element in the input binary tree (each leaf of the contraction tree) initially point to the element itself.

Algorithm TC is depicted in Figure 4. For clarity, it has been broken into three subroutines, Pair, Contract, and Expand, which we now outline.

In procedure Pair each node of the tree chooses to pair with one of its neighbors. Figure 5 illustrates how the choice is made. A leaf picks its parent with probability 1. A node with exactly one child picks its child or its parent, each with probability 1/2. A node with two children picks each child with probability 1/2. The root,

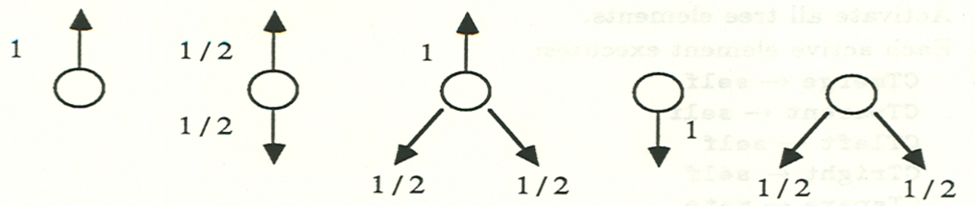

Figure 5: A pairing step. A leaf picks its parent with probability 1. A node with exactly one child picks its child or its parent, each with probability 1/2. A node with two children picks each child with probability 1/2. The root, which has no parent, picks its children with equal probability.

which has no parent, picks its children with equal probability.

The second subroutine, Procedure Contract, shown in Figure 6, performs one contraction step. We now outline a contraction step. If two nodes pick each other they merge and the parent takes over. The merge is recorded by a new internal node in the contraction tree. Space for the new node is provided by the spare, `CTspare`, of the parent in the pair. The spare of the child in the pair becomes the spare for the new node. The `CTmerge` register of both the parent and child of the pair point to the new node. The `CTparent` register points to the parent in the pair, and one of `CTleft` and `CTright` points to the child, depending on whether the node records the pairing of a parent and a left or right child. The other register points to the new node itself. The new nodes and the unpaired nodes form themselves into a new tree, which is guaranteed to be binary by the pairing strategy. The new internal node of the contraction tree inherits its `parent`, `left`, and `right` pointers from the parent and child in the pair. After a contraction step, Procedure RPT is called recursively on the contracted tree (not to be confused with the contraction tree which records the contractions).

Once the input tree has been contracted to a single node, it is "expanded" by undoing the contractions in the reverse of the order in which they occurred. Procedure Expand performs a single expansion step. The procedure, illustrated in Figure 7, undoes all of the merges made in the contracted tree during a single contraction step, but it leaves the contraction tree intact. Each active node holds the number of the contraction step being undone in its `CTlevel` register. An active node expands to a parent–child pair if this number is equal to the number of the contraction step in which the parent and child merged. When the node expands, the tree pointers of the parent and child, which have been undisturbed by the algorithm since the nodes merged, are used to restore the pair to the state it had when the contraction step took place. The expansion requires only constant space at each node. In the next section we will see that tree expansion allows us to describe treefix computations recursively.

In fact, the tree can be expanded by a greedy strategy without consulting the number of the contraction step at which each merge occurred. In each expansion step every active node simply undoes the merge that it records. We have chosen

```
procedure Contract
    Each active element executes:
        if an element and its left child pick each other
            then the element executes:
            CTmerge ← CTspare
            left.CTmerge ← CTspare
            CTmerge.CTspare ← left.CTspare
            CTmerge.CTmerge ← CTmerge
            CTmerge.CTlevel ← CTlevel
            CTmerge.CTparent ← self
            CTmerge.CTleft ← left
            CTmerge.CTright ← CTmerge
            CTmerge.parent ← parent
            CTmerge.right ← right
            if left.left ≠ left then CTmerge.left ← left.left
            else CTmerge.left ← left.right
            Deactivate the element and its left child, and activate its spare.
        elseif an element and its right child pick each other
            then the element executes:
            CTmerge ← CTspare
            right.CTmerge ← CTspare
            CTmerge.CTspare ← right.CTspare
            CTmerge.CTmerge ← CTmerge
            CTmerge.CTlevel ← CTlevel
            CTmerge.CTparent ← self
            CTmerge.CTright ← right
            CTmerge.CTleft ← CTmerge
            CTmerge.parent ← parent
            CTmerge.left ← left
            if right.right ≠ right then CTmerge.right ← right.right
            else CTmerge.right ← right.left
            Deactivate the element and its right child, and activate its spare.
    Each active element executes:
        parent ← parent.CTmerge
        left ← left.CTmerge
        right ← right.CTmerge
        CTlevel ← CTlevel + 1
```

Figure 6: Procedure Contract. In this procedure each pair of nodes that pick each other are merged. A new internal node of the contraction tree is created to record each merger.

```
procedure Expand
  Each active element executes:
    CTlevel $\leftarrow$ CTlevel $-$ 1
    if CTparent $\neq$ self then
      if CTparent.CTlevel $=$ CTlevel then
        if CTparent.parent $\neq$ CTparent then
          if CTparent.parent.left $=$ self then
            CTparent.parent.left $\leftarrow$ CTparent
          else CTparent.parent.right $\leftarrow$ CTparent
        if CTleft $\neq$ self then
          if CTleft.left $\neq$ CTleft then
            CTleft.left.parent $\leftarrow$ CTleft
          elseif CTleft.right $\neq$ CTleft then
            CTleft.right.parent $\leftarrow$ CTleft
          if CTparent.right $\neq$ CTparent then
            CTparent.right.parent $\leftarrow$ CTparent
          Activate CTleft.
        elseif CTright $\neq$ self then
          if CTright.left $\neq$ CTright then
            CTright.left.parent $\leftarrow$ CTright
          elseif CTright.right $\neq$ CTright then
            CTright.right.parent $\leftarrow$ CTright
          if CTparent.left $\neq$ CTparent then
            CTparent.left.parent $\leftarrow$ CTparent
          Activate CTright.
        Deactivate the element and activate CTparent.
```

Figure 7: Procedure Expand. This procedure "undoes" the most recent contraction performed by Procedure Contract.

to contrain each expansion step to correspond to a contraction step in order to simplify the descriptions of the treefix computations in the next section.

The proof that, with high probability, Algorithm TC takes $O(\lg n)$ steps to contract an input rooted binary tree to a single node requires three technical lemmas. The first lemma shows that in a binary tree, the number of nodes with two children and the number of leaves are nearly equal. The second lemma provides an elementary bound on the expectation of a discrete random variable with a finite upper bound. The last lemma presents a "Chernoff" [4] type bound on the tail end of a binomial distribution.

Lemma 6 *Suppose $T = (V, E)$ is a rooted binary tree, and let V_0, V_1 and V_2 denote the sets of nodes in T (excluding the root), with zero, one, or two children,* *respectively, and let $d(r)$ be the degree of the root. Then we have*

$$|V_2| + d(r) = |V_0| . \blacksquare$$

Lemma 7 *Let $X \leq b$ be a discrete random variable with expected value μ. For $w < b$, we have*

$$\Pr(X \geq w) \geq \frac{\mu - w}{b - w} . \blacksquare$$

The final lemma presents a bound on the tail end of a binomial distribution. Consider a set of t independent Bernoulli trials, each occurring with probability p of success. The probability that fewer than s successful trials occur is

$$B(s, t, p) = \sum_{k=0}^{s-1} \binom{t}{k} p^k (1-p)^{t-k} .$$

The lemma bounds the probability $B(s,t,p)$ that fewer than s successes occur in t trials when $t > 2s$ and $p < 1/2$.

Lemma 8 *For $t > 2s$ and $p < \frac{1}{2}$, we have*

$$B(s, t, p) \leq \left(\frac{1-p}{1-2p}\right)\left((1-p)^t\right)\left(\frac{et}{s}\right)^s . \blacksquare$$

With these lemmas we can now prove that with high probability, Algorithm TC takes $O(\lg n)$ steps to contract a rooted binary tree to a single node. The key observation in the proof is that for each node that pairs with its parent, the number of nodes in the tree decreases by one.

Theorem 9 *With high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract a rooted binary tree of n nodes to a single node.*

Proof: The proof has three parts. First, we use Lemma 6 to show that that if a rooted binary tree has $|V|$ nodes, the expected number of nodes pairing with a parent in a single contraction step is at least $|V|/4$. Next, we use Lemma 7 to show that the probability that at least $|V|/8$ nodes pair with a parent in any step is at least $1/3$. Finally, we use Lemma 8 to show for any constant k that after $\alpha \log_{8/7} n$ steps, for some constant $\alpha > 2$, the probability that the tree has not contracted into a single node is $O(1/n^k)$.

We first show that the expected number of nodes pairing with a parent is at least $|V|/4$. A node is picked by its parent with probability 1 when its parent is a degree 1 root, and $1/2$ otherwise. Thus a leaf pairs with its parent with probability at least 1/2, and a node (other than the root) with one child picks its parent with probability at least 1/4. Let P be the number of nodes pairing with a parent. We apply Lemma 6 to the simple bound on the expected value of P,

$$\mathrm{E}(P) \;\geq\; \frac{|V_0|}{2} + \frac{|V_1|}{4},$$

to yield the desired result:

$$\mathrm{E}(P) \geq \frac{|V_0| + |V_1| + |V_2| + d(r)}{4} \geq \frac{|V|}{4}\,.$$

Now we show that the probability that at least $|V|/8$ nodes pair with a parent in a single contraction step at least 1/3. We call such a step *successful.* At most half of the nodes pair with their parents. Using Lemma 7 with $b = |V|/2$, $w = |V|/8$, and $\mu \geq |V|/4$, we have

$$\Pr(P \geq |V|/8) \geq \frac{\frac{|V|}{4} - \frac{|V|}{8}}{\frac{|V|}{2} - \frac{|V|}{8}} = \frac{1}{3}\,.$$

Finally, we show that with high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract the input tree to a single node. In the contraction following a successful pairing step, the size of the tree decreases by a factor of 7/8 or more. After $\log_{8/7} n$ successful steps, the tree must consist of a single node. By Lemma 8 with $p = 1/3$, the probability that fewer than $s = \log_{8/7} n$ successful steps occur in αs steps is

$$B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3) \;\leq\; 2\left(\left(\frac{2}{3}\right)^{\alpha} \alpha e\right)^{\log_{8/7} n}\,.$$

For any value k, we can choose α so that $B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3)$ is $O(1/n^k)$. In particular, for $k = 1$ a value of $\alpha = 8$ suffices.

We now prove that Algorithm TC is conservative.

Theorem 10 *Algorithm TC is conservative.*

Proof: The key idea is that each active element in the contracted tree is a "representative" of a subgraph of the input tree that has been contracted to a single node. The contracted subgraphs, which are trees, are disjoint in the input tree. The representative and spare of a subgraph are either elements in or mates of elements in the subgraph. The pairing strategy ensures that each subgraph is adjacent by an edge to at most one subgraph which is higher in the input tree, and to at most two subgraphs which are lower. The representative of the subgraph has pointers to the representatives of these subgraphs, and to the spare of the subgraph.

As in the list contraction algorithm, the set of memory accesses in a step of the tree contraction algorithm corresponds to a set of disjoint paths in the input data structure. Since each subgraph is connected, the pointers between representatives and spares correspond to disjoint paths in the input tree. Similarly, any set of pointers between each representative and the representative of at most one of the two adjacent subgraphs lower in the input tree corresponds to a set of disjoint paths in the input tree. The memory accesses in a step correspond to a set of pointers between representatives and spares or to a set of pointers between each representative and the representative of at most one of the two adjacent subgraphs lower in the input tree. ∎

Tree contraction can be performed conservatively and deterministically on a DRAM with m processors in $O(\lg n \lg^* m)$ steps using the deterministic coin tossing algorithm of Cole and Vishkin [5]. The key idea is that in Algorithm TC, the nodes in the tree that can pair form chains, and by Lemma 6 these chains contain at least half the tree edges. The chains can be oriented from child to parent in the tree, and deterministic coin tossing can be used to perform the pairing step in $O(\lg^* m)$ steps.

5. Treefix computations

This section presents a generalization of the parallel prefix computation to binary trees. We present two kinds of *treefix* computations—*rootfix* and *leaffix*—and show how they can be implemented by an $O(\lg n)$-step conservative algorithm in linear space. As we shall see in Section 6, treefix computations can greatly simplify the description of many parallel graph algorithms in the literature, and moreover, treefix computations can be performed by conservative algorithms.

We begin with a definition of treefix computation.

Definition. Let $\mathcal{D}$ be a domain with a binary associative operation $\oplus$ and an identity ε. Let T be a rooted, binary tree in which each vertex $i \in T$ has an assigned value $x_i \in \mathcal{D}$. The *rootfix* problem is to compute for each vertex $i \in T$ with parent j, the value $y_i = y_j \oplus x_i$, where $y_j = \varepsilon$ if i is the root. The *leaffix* problem is to compute for each vertex $i \in T$ with left child j and right child k, the value $y_i = x_i \oplus y_j \oplus y_k$, where $y_j = \varepsilon$ if i has no left child and $y_k = \varepsilon$ if i has no right child.

Simple examples of treefix problems are computing the depth of each vertex in a rooted binary tree and computing the size of each subtree. These and other examples appear in the next section.

Like the prefix computation on lists, treefix computations can be performed directly on the contraction tree. To simplify the description here, however, we describe a recursive version. We execute one contraction step and then recursively perform a treefix computation on the new tree. The treefix values for the input tree can be computed immediately from the treefix values for the new tree.

Theorem 11 *A rootfix or leaffix computation can be performed by a conservative randomized algorithm which, with high probability, takes* $O(\lg n)$ *steps, or by a* *conservative deterministic algorithm which takes $O(\lg n \lg^* m)$ steps, where m is the number of processors in the DRAM.*

Proof: We first describe the computation for rootfix. First the input binary tree T is transformed to a new tree T' by one contraction step. Each new node u in T' resulting from the pairing of parent p and child c in T passes input value x_c to the child in T' that u inherited from c. Node u passes ε to its other child. Each unpaired node v in T' passes ε to each of its children. Each node in T' receives a value from its parent, call it z. Each new node u computes $x'_u \leftarrow z \oplus x_p$. Each unpaired node v computes $x'_v \leftarrow z \oplus x_v$. A rootfix computation is performed recursively on T' using the x' values as input and yielding y' values as output. The contraction step from T to T' is then undone. Each new node u passes y'_u to p and c. Node p computes $y_p \leftarrow y'_u$ and c computes $y_c \leftarrow y'_u \oplus x_c$. Each unpaired node v computes $y_v \leftarrow y'_v$.

We now describe a computation of which leaffix is a special case. Each node i in T is assigned input values x_i, l_i, and r_i. Node i with left child j and right child k computes output value $y_i = x_i \oplus y_j \oplus l_i \oplus y_k \oplus r_i$, where y_j and l_i are ε if i has no left child and y_k and r_i are ε if i has no right child. For the special case of leafix, l_i and r_i are both ε. First, a contraction step transforms T to T'. Consider each new node u in T' resulting from the pairing of parent p and left child c in T with input values x_p, l_p, r_p and x_c, l_c respectively. (The cases where c is a right child, or where c has a right child or is a leaf, are similar.) Node u computes $x'_u \leftarrow x_p \oplus x_c$, $l'_u \leftarrow l_c \oplus l_p$, and $r'_u \leftarrow r_p$. Each unpaired node v computes $x'_v \leftarrow x_v$, $l'_v \leftarrow l_v$, and $r'_v \leftarrow r_v$. The computation is performed recursively on T' using the x' values as input and yielding y' values as output. Each node passes its y' value to its parent in T'. Each node receives values from its left and right children, call these values z_l and z_r. Each new node u passes z_l to c and both z_l and z_r to p. The contraction step from T to T' is undone. Node c computes $y_c \leftarrow x_c \oplus z_l \oplus l_c$. Node p computes $y_p \leftarrow x_p \oplus x_c \oplus z_l \oplus l_c \oplus l_p \oplus z_r \oplus r_p$. Each unpaired node v computes $y_v \leftarrow y'_v$. ∎

6. Conservative algorithms

This section presents a collection of conservative DRAM algorithms, all of which use treefix computations. The algorithms use two processors per edge of an input graph $G = (V, E)$ and require constant extra space in each processor. Since the algorithms are based on shortcutting pointers in the input data structure, they are independent of the underlying DRAM or embedding of the data structure.

We represent each vertex in an undirected graph $G = (V, E)$ by a doubly linked *incidence ring* of processors, one for each edge. Each element of the incidence ring contains pointers to the next and previous elements in the ring, and one pointer for a graph edge. For each edge $(u, v) \in E$ the element in the incidence ring for u contains a pointer to an edge element in the incidence ring for v, and vice versa. A directed graph is represented in the same doubly linked fashion, but the graph edges are labeled with their directions.

We represent trees with arbitrary vertex degrees by an incidence ring structure as well. If the tree is directed, each ring has a unique *principal element* that points toward the root. Breaking the incidence ring before the principal element yields the standard binary tree representation of the tree [9, pp. 332–333].

We now present brief descriptions of the algorithms. The performance is given terms of the number of steps on a DRAM when the input representation has size n. We assume the implicit tree contractions in the algorithms are performed by the randomized Algorithm TC. Deterministic bounds can be obtained by multiplying the number of steps by $O(\lg^* m)$, where m is the number of processors. An upper bound on the actual performance in the DRAM model can be obtained by multiplying the number of steps by the load factor of the input.

Generalized treefix. *Perform a treefix operation on a directed tree with arbitrary vertex degree. The input values* $\{x_i\}$ *are stored in the principal elements of the tree, which is where the output values* $\{y_i\}$ *are to be placed. The leaffix value at a node* i *whose children have values* $y_1, y_2, \ldots, y_k$ *is* $y_i = x_i \oplus y_1 \oplus y_2 \oplus \cdots \oplus y_k$. Each element that is not principal stores the identity element ε for its value. A binary treefix computation performed on the binary tree representation underlying the tree computes the desired values. *Performance:* $O(\lg n)$.

Tree functions. *Given a directed tree, compute for each node the number of descendents, its height, or its depth.* The number of decendents for each node can be computed by a leaffix computation with $\oplus$ as integer addition and $x_i = 1$ for all nodes. The height of a node can also be computed by a leaffix computation where $a \oplus b = \max(a+1, b+1)$, the identity is $\varepsilon = -1$, and $x_i = -1$ for all nodes.[2] The depth of a node can be computed by a rootfix computation with $\oplus$ as addition and $x_i = 1$ for all nodes except the root which has value 0. *Performance:* $O(\lg n)$.

Rooting an undirected tree. *Pick a root of a tree with undirected graph pointers and orient the graph pointers toward the root.* Form an "Eulerian tour" of the pointers of the representation [21] by directing each element of the tree to link its incoming ring pointer with its graph edge directed outward and its graph edge directed inward with its outgoing ring pointer. Each graph edge is used twice in the tour, once in each direction, but each ring pointer is used only once. Using the variant of Algorithm LC which works for circular lists, form a contraction tree of the tour. Choose the root of the contraction tree to be the root of the tree, and break the tour so that it begins with the root. Use parallel prefix to number each node according to its first occurrence in the tour. Use contraction trees to distribute the smallest value in each incidence ring to the elements of the ring. Orient each graph edge from the larger value to the smaller. *Performance:* $O(\lg n)$.

Rerooting a directed tree. *Given a directed tree and another distinguished vertex* k, *reorient the graph edges of the tree to point to* k. The algorithm for rooting a tree can be used by picking k as the root instead of the root of the contraction

[2]Technically, $\varepsilon = -1$ is not an identity for the operation $a \oplus b = \max(a+1, b+1)$. Nonetheless, this leaffix computation correctly computes the height of each node in a binary tree. Moreover, this leaffix computation also generalizes to a directed tree with arbitrary vertex degree.

tree, but a single treefix computation suffices. Perform a leaffix computation with $x_k = 1$ and $x_i = 0$ if $i \neq k$, and use Boolean OR for $\oplus$. Each principal element whose leaffix value is 1 lies on the path from x_k to the root. Reverse the direction of the graph pointers of these elements. (Note: rerooting a tree changes the principal elements.) *Performance:* $O(\lg n)$.

Tree-walk numberings of a binary tree. *Number the nodes of a binary tree according to the order they would be visited in a preorder/inorder/postorder tree walk.* For each of the walks, we will compute y_k, the number of nodes visited before the left subtree of k. Use a leaffix computation to compute the number $size_k$ of the subtree rooted at k. We first compute the preorder numbering. (For the purposes of these numbering algorithms, we consider the root to be a left child.) If node k is a left child, set x_k to 1. If node k is a right child, set x_k to 1 plus the size of its sibling subtree. A rootfix computation with $+$ yields y_k, which is the preoder numbering of node k. The inorder numbering can be computed similarly. If node k is a left child, set x_k to 0. If k is a right child, set x_k to 1 plus the size of its sibling subtree. Compute y_k for each node using a rootfix computation with $+$. The inorder numbering of node k is 1 plus y_k plus the size of its left subtree. The postfix numbering can be computed by setting x_k to 0 if node k is a left child, and by setting x_k to the size of its sibling subtree if k is a right child. After computing y_k using a rootfix computation with $+$, the postfix numbering of node k is 1 plus y_k plus the sizes of its two subtrees. *Performance:* $O(\lg n)$.

Prefix/postfix numbering of a directed tree. *Number the edges of an arbitrary directed tree according to the order they are visited in preorder/postorder tree walk.* The problem reduces to prefix/postfix numbering on the underlying binary tree representation. *Performance:* $O(\lg n)$.

Diameter and center of a tree. *The diameter is the length of the longest path in the tree. A center is a vertex v such that the longest path from v to a leaf is minimal over all vertices in the tree.* The diameter can be determined by rooting the tree and using rootfix to find the furthest leaf from the root. Reroot the tree at this leaf. The distance from the new root to the furthest leaf is the diameter. (Based on an analog algorithm attributed to J. Wennmacker [6].) A center of the tree can be determined by finding a median element of the path that realizes the diameter. *Performance:* $O(\lg n)$.

Centroid of a tree. *A centroid is a vertex v such that the largest subtree with v as a leaf is minimal over all vertices in the tree.* A centroid can be determined by rooting the tree and computing the size of each subtree. By broadcasting the size m of the tree from the root, each graph edge in each incidence ring can determine the number of elements on the other side of the edge. For each incidence ring, compute the maximum of these values. A vertex with the minimum of these maximum values is a centroid. *Performance:* $O(\lg n)$.

Separator of a tree. *A separator* [14] *is a partition of the vertices of an m-vertex tree into three sets A, B, and C, with $|A| \leq \frac{2}{3}m$, $|B| = 1$, and $|C| \leq \frac{2}{3}m$, such that no edge of the tree goes between a vertex in A and a vertex in C.* Determine a centroid of the tree. This vertex is the separator vertex in B. It remains to partition the remaining vertices between A and C. For each graph edge in the incidence ring, count the number of vertices in the subtree on the other side of the edge. Put the largest subtree in A. Use parallel prefix on the incidence ring to compute a running sum of the sizes of the other subtrees. Put all subtrees whose prefix value is at most $\frac{2}{3}m$ in C, and put the remainder in A. *Performance:* $O(\lg n)$.

Subexpression evaluation. *Given a directed tree in which each leaf has a value and each internal node has an operator from* $\{+, -, \cdot, \div\}$, *compute for each internal node the subexpression rooted at that node.* A single leaffix computation suffices using the ideas of Brent [2] and Miller and Reif [16]. *Performance:* $O(\lg n)$.

Minimum cost spanning tree. *Given an undirected input graph* $G = (V, E)$ *and a cost function* $w : E \to \mathbf{R}$, *determine a set* $F \subseteq E$ *of edges such that each vertex in* V *is incident on an edge of* F, *and the sum of the weights of the edges in* F *is minimal.* We give a conservative DRAM implementation of Boruvka's algorithm, also attributed to Sollin [20, pp. 71–83]. We assume without loss of generality that the edge weights are distinct—otherwise, we can assign the weight of a graph edge e between two incidence ring elements with addresses a and b to be $(w(e), \max(a,b), \min(a,b))$ and then compare weights lexicographically. We determine F by marking edges in G. Initially, no edges are marked. At each step of the algorithm, the currently marked graph edges form a subforest of F. Break each incidence ring by removing a single ring pointer and direct the resulting linear list. At each step of the algorithm, the marked graph edges and the ring pointers form a set $\{T_i\}$ of rooted trees, where the index i of the tree is the address of the root. The algorithm proceeds as follows. For each tree T_i, use a rootfix computation to broadcast i to all of the elements in T_i. Use a leaffix computation on T_i to determine an edge $e \in E$ with the smallest weight $w(e)$ connecting an edge element $u \in T_i$ with an edge element $v \in T_j$, where $i \neq j$. If no such edge exists, the algorithm terminates. If T_j picks the same edge as T_i, the tree with smaller index does nothing. Otherwise, mark e as a member of F, directing it into T_j, and reroot T_i with u as the new root. Repeat this procedure until the algorithm terminates. *Performance:* $O(\lg^2 n)$.

Connected components. *Given an undirected input graph* $G = (V, E)$, *determine a labeling* $l : V \to \mathbf{Z}$ *such that such that* $l(v) = l(v')$ *if and only if* v *and* v' *are in the same connected component of* G. The algorithm is the same as the minimum spanning tree algorithm, choosing the weight of a graph edge e between incidence ring elements with addresses a and b to be $\max(a,b), \min(a,b)$. The label of a vertex is the index of its tree. *Performance:* $O(\lg^2 n)$.

Biconnected components. *Two edges of an undirected graph* $G = (V, E)$ *are in the same biconnected component if they lie on a common simple cycle. Determine a labeling* $l : E \to \mathbf{Z}$ *such that* $l(e) = l(e')$ *if and only if* e *and* e' *are in the same biconnected component of* G. We give a conservative DRAM implementation of the biconnectivity algorithm of Tarjan and Vishkin [21]. We assume that the reader has some familiarity with that algorithm. Find a (directed) minimum spanning tree $T = (V, F)$ of G. Number the vertices in the minimum spanning tree in preorder. Use leaffix computations to compute for each vertex v three values: $\mathrm{nd}(v)$, $\mathrm{low}(v)$, and $\mathrm{high}(v)$. Here $\mathrm{nd}(v)$ is the number of descendants of v, while $\mathrm{low}(v)$ and $\mathrm{high}(v)$ are the lowest and highest vertices (with respect to the preorder numbering of T) that are either a descendant of v or adjacent to a descendant of v by an edge of $E - F$. Build a new graph G' where the edges of F are the vertices of G'. Let e be an edge from u to $p(u)$, where $p(u)$ is the parent of u in F. The adjacency ring for u in G acts as the adjacency ring for e in G'. Add two kinds of edges to G'. For each edge $\{w, v\}$ in $E - F$ such that $v + \mathrm{nd}(v) \leq w$, add an edge $\{\{v, p(v)\}, \{w, p(w)\}\}$ to G'. For each edge $(v, p(v))$ of F such that $v \neq 1$ and $p(v) \neq 1$, and $\mathrm{low}(v) < \mathrm{low}(p(v))$ or $\mathrm{high}(v) \geq p(v) + \mathrm{nd}(p(v))$, add an edge $\{\{v, p(v)\}, \{p(v), p(p(v))\}\}$ to G'. It can be verified that the representation of G' is conservative with respect to the representation of G. Find the connected components of G'. Two edges of F are in the same block if as vertices in G' they are in the same connected component. Finally, for each edge $e = \{w, v\}$ in $E - F$, let $l(e) = l(\{w, p(w)\})$. *Performance:* $O(\lg^2 n)$.

Eulerian cycle. *An Eulerian cycle of an undirected graph $G = (V, E)$ is a cycle containing each edge in E exactly once.* If any vertex has odd degree, then no Eulerian cycle exists. Form a set of disjoint cycles of the pointers of the representation of G as in the algorithm for directing a tree. The cycles can be merged using an algorithm similar to the minimum spanning tree algorithm. *Performance:* $O(\lg^2 n)$.

7. Conclusion

This paper has addressed the problem of embedding data structures, the use of load factor to evaluate embeddings and algorithms, and the notion of a conservative algorithm as one that is communication efficient. This section gives some examples of graphs that can be embedded efficiently in DRAM networks. We discuss load factors for data structures other than graphs, and we consider relaxing the requirement that an algorithm be conservative in order to be communication efficient.

The efficiency of a DRAM algorithm depends on how well its input is embedded in the DRAM and this embedding problem must be faced by algorithm designers in any bandwidth-limited distributed network. In general, the problem of determining the best embedding is NP-complete, but for many common situations, good embeddings can be found. Moreover, there are many situations in which the input graph structure is simple and known *a priori*, and a good embedding may be easy to construct for a given DRAM network.

To illustrate how the embedding problem can be solved in many practical situations, consider fat-trees as the DRAM network. Because of the recursive structure of fat-trees, the divide-and-conquer heuristic works well for many input graphs. For example, a subproblem in switch-level simulation of a VLSI circuit is the finding of electrically equivalent portions of the circuit. A naive divide-and-conquer embedding of the circuit on the fat-tree yields small load factors for every cut. Thus, our conservative connected components algorithm will never cause undue congestion in communicating messages in the underlying network, and the algorithm will run effectively as fast as on an expensive, high-bandwidth network.

For some graphs, it can be proved that divide-and-conquer yields near optimal embeddings on a fat-tree. Specifically, graphs for which a good *separator theorem* [14] exists can be embedded well. Examples include meshes, trees, planar graphs, and multigrids. Situations in which a mesh might be used include systolic array computation and image processing. Planar graphs and multigrids arise from the solution of sparse linear systems of equations based on the finite-element method.

Many other classes of algorithms can be implemented in a conservative fashion on a DRAM. Any algorithm that communicates only across pointers in an input data structure is conservative. Passing a single datum between two processors, however, can require time linear in the diameter of the data structure, whereas our algorithms all run in a polylogarithmic number of steps. As another example, systolic array algorithms for matrix problems [10,13] can be implemented efficiently if the matrices are properly embedded. In general, any fixed-connection network algorithm will run well on a DRAM if the communication required by the network can be supported by the underlying DRAM network.

Although the algorithms presented in this paper operate primarily on graphs, for which there is a natural definition of load factor, it is also possible to define the load factor of a data structure that contains no explicit pointers. For example, it is natural to superimpose a mesh on the matrix, as is suitable for systolic array computation, and the load factor of the matrix can be defined as the load factor of the superimposed mesh.

For some problems, the running time may be more a function of the load factor of the output than the load factor of the input. As an example, consider the problem of sorting a linear list of elements. A natural question to ask is whether the list can be sorted in a polylogarithmic number of steps where at each step, the load factor is bounded by the load factor induced by the linear list together with the permutation determined by the sorted output. Whether such a sorting algorithm exists is an open question.

Whereas the Shortcut Lemma presented in this paper holds for any network, for particular networks, other shortcut lemmas may hold. For example, another shortcut lemma for tree structures such as fat-trees is used in [15] to show that a certain parallel algorithm for finding the optimal embedding of a list on a fat-tree is conservative.

As a final comment, it may well be that the notion of a conservative algorithm is too conservative. A contraction tree is not conservative with respect to its input tree (though the levels of the contraction tree are), but the load factor of the contraction tree is at most $O(\lg n)$ times the input load factor. As a practical matter, it is probably not worth worrying whether every set of memory accesses is conservative with respect to the input, as long as the load factor of memory accesses is polylogarithmically bounded. Algorithms with this looser bound are somewhat easier to code because of the relaxed constraint, and they should perform comparably.

Acknowledgments

Thanks to Baruch Awerbuch, Ravi Boppana, Benny Chor, Tom Cormen, Johan Hastad, Alex Ishii, Joe Kilian, Tom Leighton, and Ron Rivest of MIT for numerous helpful discussions. Thanks to Ginny Loop for producing the figures.

References

[1] S. N. Bhatt and C. E. Leiserson, "How to assemble tree machines," *Advances in Computing Research*, Vol. 2, *VLSI Theory*, F. P. Preparata, ed., JAI Press, Greenwich, Conn., 1984, pp. 95–114.

[2] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *JACM*, Vol. 21, No. 2, April 1974, pp. 201–208.

[3] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, Vol. C–31, No. 3, March 1982, pp. 260–264.

[4] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 493–507.

[5] R. Cole and U. Vishkin, "Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms," *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing*, May 1986, pp. 206–219.

[6] A. K. Dewdney, "Computer Recreations," *Scientific American*, Vol. 252, No. 6, June 1985, pp. 18–29.

[7] M. J. Fischer and R. E. Ladner, "Parallel prefix computation," *JACM*, Vol. 27, No. 4, October 1980, pp. 831–838.

[8] R. I. Greenberg and C. E. Leiserson, "Randomized routing on fat-trees," *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE, October 1985, pp. 241–249.

[9] D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

[10] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," *SIAM Sparse Matrix Proceedings*, I. S. Duff and G. W. Stewart, ed., 1978, pp. 256–282.

[11] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, Vol. C–34, No. 10, October 1985, pp. 892–901.

[12] F. T. Leighton, *Complexity Issues in VLSI*, MIT Press, Cambridge, Massachusetts, 1983.

[13] C. E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts, 1983.

[14] R. J. Lipton and R. E. Tarjan, "A planar separator theorem," *Siam J. of Applied Math*, Vol. 36, No. 2, 1979, pp. 177–189.

[15] B. M. Maggs, *Communication-Efficient Parallel Graph Algorithms*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1986.

[16] G. Miller and J. Reif, "Parallel tree contraction and its application," *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE, October 1985, pp. 478–489.

[17] G. F. Pfister and V. A. Norton, "'Hot spot' contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, Vol. C–34, No. 10, October 1985, pp. 943–948.

[18] Yu. Ofman, "On the algorithmic complexity of discrete functions," English translation in *Soviet Physics – Doklady*, Vol. 7, No. 7, 1963, pp. 589–591.

[19] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, Vol. 3, 1982, pp. 57–67.

[20] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[21] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, IEEE, October 1984, pp. 12–20.

[22] J. C. Wyllie, *The Complexity of Parallel Computations*, Ph.D. thesis, Cornell University, Ithaca, N. Y., August 1979.