Communication-Efficient Parallel Algorithms for Distributed Random-Access Machines

Charles E. Leiserson
Bruce M. Maggs
M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

This paper introduces a model for parallel computation, called the *distributed random-access machine* (DRAM), in which the communication requirements of parallel algorithms can be evaluated. A DRAM is an abstraction of a parallel computer in which memory accesses are implemented by routing messages through a communication network. A DRAM explicitly models the congestion of messages across cuts of the network.

We introduce the notion of a *conservative algorithm* as one whose communication requirements at each step can be bounded by the congestion of pointers of the input data structure across cuts of a DRAM. We give a simple lemma that shows how to "shortcut" pointers in a data structure so that remote processors can communicate without causing undue congestion. We give $O(\lg n)$-step, linear-processor, linear-space, conservative algorithms for a variety of problems on n-node trees, such as computing treewalk numberings, finding the separator of a tree, and evaluating all subexpressions in an expression tree. We give $O(\lg^2 n)$-step, linear-processor, linear-space, conservative algorithms for problems on graphs of size n, including finding a minimum-cost spanning forest, computing biconnected components, and constructing an Eulerian cycle. Most of these algorithms use as a subroutine a generalization of the prefix computation to trees. We show that any such *treefix* computation can be performed in $O(\lg n)$ steps using a conservative variant of Miller and Reif's tree-contraction technique.

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 may be hard pressed to implement networks with the communication bandwidth demanded by this assumption, due in part to packaging constraints. 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.

In a communication network, we can measure the cost of communication in terms of the number of messages that must cross a cut of the network, as in [10] and [18]. Specifically, a *cut* S of a network[1] is a subset of the nodes of the network. The *capacity* $\mathrm{cap}(S)$ is the number of wires connecting processors in S with processors in the rest of the network $\overline{S}$, i.e., the bandwidth of communication between S and $\overline{S}$. For a set M of messages, we define the *load* of M on a cut S to be the number of messages in M whose source is in S and whose destination is in $\overline{S}$ or vice versa. The *load factor* of M on S is

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

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" phenomenon

[1] We assume that in the communication network, each processor has its own local memory, the processors are interconnected as a graph, and routing of messages is performed by the processors. The generalization to the case when processors, memories, and switches are distinct entities is straightforward, but complicates the definitions.

identified by Pfister and Norton [24]. 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 packaging technology that can cause high load factors. For example, the cut of the network that limits communication performance for some set of messages might correspond to the pins on a printed-circuit board or to the cables between two cabinets.

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 [10, 18], as well as the standard universal routing networks, such as the Boolean hypercube [29]. 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 because of packaging limitations. Networks for which the load factor lower bound cannot be approached to within a polylogarithmic factor as an upper bound include linear arrays, meshes, and high-diameter networks in general.

In the PRAM model, the issue of communication bandwidth does not arise even though most parallel computers implement remote memory accesses by routing messages through an underlying network. In the PRAM model, a set of memory accesses is presumed to take unit time, reflecting the assumption that all sets of messages can be routed through the network with comparable ease. In the DRAM model, a set of memory accesses takes time equal to the load factor of the set of messages, which reflects the unequal times required to route sets of messages with different load factors.

This paper gives DRAM algorithms that solve many graph problems with efficient communication. Our algorithms can be executed on any of the popular PRAM models because a PRAM can be viewed as a DRAM in which communication costs are ignored. In fact, our DRAM algorithms can all be performed on an exclusive-read, exclusive-write PRAM, and they are nearly as efficient 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 how a DRAM models the congestion produced by techniques such as "recursive doubling" that are

frequently used in PRAM algorithms. Section 3 defines the notion of a *conservative algorithm* as a concrete realization of a communication-efficient DRAM algorithm, and gives a "Shortcut Lemma" that forms the basis of the conservative algorithms in this paper. Section 4 presents a conservative "recursive pairing" technique that can be used to perform many of the same functions as on lists as recursive doubling. Section 5 presents a linear-space, conservative "tree contraction" algorithm based on the ideas of Miller and Reif [22]. Section 6 presents *treefix computations,* which are generalizations of the parallel prefix computation [3, 7, 23] to trees. We show that treefix computations can be performed using the tree contraction algorithm of Section 5. Section 7 gives short, efficient, parallel algorithms for tree and graph problems, most of which are based on treefix computations. Section 8 discusses the relationship between the DRAM model and more traditional PRAM models, as well as the ramifications of using the DRAM model in practical situations.

2 The DRAM model

This section introduces the abstraction of a distributed random-access machine (DRAM). We show how a parallel data structure can be embedded in a DRAM, and we define the load factor of a data structure. We show how the embedding of a data structure in a network can cause congestion in the underlying network when the pointers of the data structure are accessed in parallel, and we also demonstrate that a parallel algorithm can produce substantial congestion in an underlying network, even when there is little congestion implicit in the input data structure. We illustrate how a DRAM accurately models these two phenomena.

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 that remote memory

accesses are implemented by routing messages through an underlying network. We model the communication limitations imposed by the network by assigning a numerical capacity $\mathrm{cap}(S)$ to each cut (subset of processors) S of the DRAM equal to the number of wires connecting processors in S with processors in the rest of the network. Thus, there are many different DRAM's corresponding to the many possible assignments of capacities to cuts. For a set M of memory accesses, we define $\mathrm{load}(M, S)$ to be the number of accesses in M from a processor in S to a processor in $\overline{S}$ (the rest of the DRAM), or vice versa. The load factor of M on S is $\lambda(M, S) = \mathrm{load}(M, S)/\mathrm{cap}(S)$, and the load factor of M on the DRAM 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 the load factor $\lambda(M)$.* (Local operations take unit time.) This assumption constitutes the principal difference between the DRAM and the network it models. We know that the load factor is a lower bound on the time required in both the network and the DRAM. If the network's message routing algorithm cannot approach this lower bound as an upper bound (for example, if the network has high diameter), then the network is not well modeled by the DRAM. If the network's routing algorithm can nearly achieve the load factor as an upper bound, then the analysis of an algorithm in the DRAM model will reliably predict the actual performance of the algorithm on the network. Section 8 discusses some networks for which the DRAM is a reasonable model, including volume-universal networks such as fat-trees [18].

A natural way to embed a data structure in a DRAM is to put one record of the data structure into each processor, as in the "data parallel" model [11]. The record can contain data, including pointers to records in other processors. We measure the quality of an embedding by treating the data structure as a set of pointers and generalizing the concept of load factor to sets of pointers. The load of a set P of pointers across a cut S, denoted $\mathrm{load}(P, S)$, is the number of pointers in P from a processor in S to a processor in $\overline{S}$, or vice versa. The load factor of P on the entire DRAM is

$$\lambda(P) = \max_S \frac{\mathrm{load}(P, S)}{\mathrm{cap}(S)}.$$

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 8).

There are generally two situations in which message congestion can arise during the execution of an algorithm on a network, both of which are modeled accurately by a DRAM whose cut capacities correspond to the cut capacities of the network. In the first situation, the embedding of a data structure 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. We now illustrate these two situations.

As an example of the first situation, consider an embedding of a simple linear list in which alternate list elements are placed on opposite sides of a narrow cut of a network. If each element fetches a value from the next element in the list, the load factor across the cut is large. In the DRAM model, this congestion is modeled by the increase in time required for the memory accesses across the cut. (Observe that in a PRAM model, the congestion is not modeled since any set of memory accesses is assumed to take unit time.) Of course, a list can typically be embedded in a network so that the number of list pointers crossing any cut is small compared to the capacity of the cut, again a situation that can be modeled by a DRAM.

In the second situation, the congestion is produced by an algorithm. As an example, consider the "recursive doubling" or "pointer jumping" technique [30] used extensively by PRAM 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 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 $d(i)$ of each element i to the end of the list. Initially, each element i sets $d(i) \leftarrow 1$. At each pointer-jumping step, each element i not pointing to the end of the list computes $d(i) \leftarrow d(i) + d(p(i))$. In a PRAM model, the running time on a list of length n is $O(\lg n)$. Variants of this technique are used for path compression, vertex numbering, and parallel prefix computations [22, 25, 27, 30].

We now show that recursive doubling can be expensive even when a data structure has a good embedding in a network. 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 accesses the data in element 9. In the second step, the load is 2 because element 7 accesses element 9 and

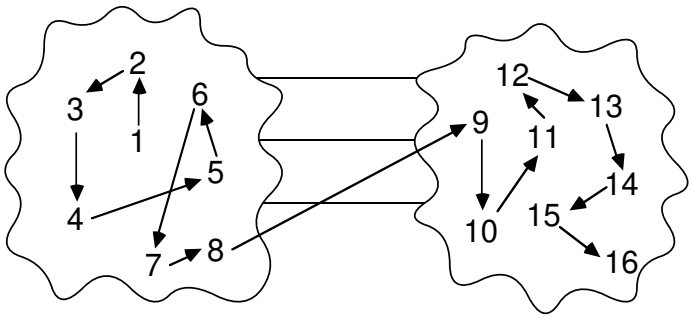

Figure 1: A cut of capacity 3 separating two halves of a linked list. The load of the list on the cut is 1. At the final step of recursive doubling, each element on the left side of the cut accesses an element on the right, which induces a load of 8 on the cut.

element 8 accesses element 10. In the third step, the load is 4, and in the fourth step, each of the first eight elements makes an access across the cut, creating a load of 8. Since the load factor of the cut in the fourth step is 8/3, this set of accesses requires at 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, the cut capacity is insufficient. In a DRAM, this situation is modeled by the increased time to perform the memory accesses in the fourth step compared with those in the first step.

The focus of this paper is avoiding this second cause of congestion. In Section 4, we shall show how a *recursive pairing* strategy can perform many of the same functions as recursive doubling, but in a communication-efficient fashion.

3 Conservative algorithms

This section introduces the notion of a *conservative algorithm*. In the DRAM model, a conservative algorithm is communication efficient in the sense that it never produces more congestion across cuts of the DRAM than is implicit in the input data structure. We give an important lemma that shows how pointers in a data structure can be "shortcut" without introducing congestion.

A *conservative algorithm* is a DRAM algorithm in which the load factor of memory accesses in any step is bounded by the load factor of the input data structure, independent of the cut capacities of the DRAM on which the algorithm is executed. To be precise, we define a set M of memory accesses to be *conservative* with respect to another set M' of memory accesses if for all cuts S of a DRAM, we have $\text{load}(M, S) \leq \text{load}(M', S)$. By implication, whatever the cut capacities of the DRAM, we have $\lambda(M) \leq \lambda(M')$. We make the natural extension of the term conservative to sets of pointers and data structures. A conservative algorithm is thus one all of whose memory accesses are conservative with respect to the input data structure. Thus, if a conservative algorithm runs for T steps on an input data structure with load factor λ, then the total time for the algorithm is at most λT.

If at every step, the memory accesses of an algorithm correspond to a subset of pointers in the input data structure, then the algorithm is certainly conservative since if M is a subset of M', then we have $\text{load}(M) \leq \text{load}(M')$. For example, synchronous distributed algorithms, such as the network flow algorithms of Goldberg and Tarjan [8, 9], are conservative for this reason. We do not wish to restrict our attention to this limited class of conservative algorithms because synchronous distributed algorithms cannot efficiently solve certain problems on graphs with high diameter. For example, the problem considered earlier of determining the distance of each element to the end of the list cannot be solved in less than linear time with a synchronous distributed algorithm. A PRAM algorithm, however, can perform such the computation in logarithmic time, for example, by recursive doubling, but recursive doubling is not conservative.

We would like to know conditions under which processors in a DRAM can communicate directly with distant locations in a data structure without increasing communication requirements as measured by the load factor. The following simple, but important, lemma provides conditions that are sufficient for any DRAM.

Lemma 0.1 (Shortcut Lemma) *Suppose a set P of pointers in a data structure contains pointers $a \to b$ and $b \to c$. Then the set Q of pointers defined by*

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

is conservative with respect to P. Moreover, any set Q of pointers is conservative with respect to another set P of pointers if there exist pointer-disjoint paths in P that connect the endpoints of pointers in Q.

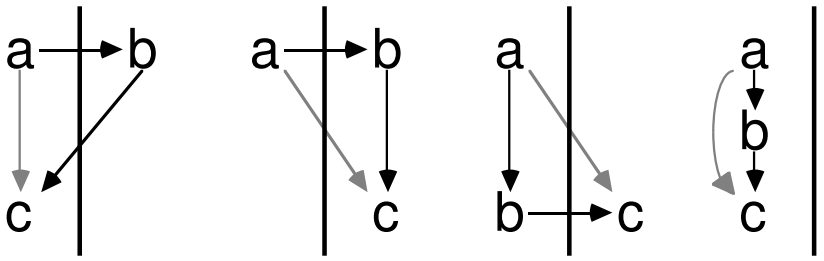

Figure 2: The Shortcut Lemma. In each of the four cases illustrated, the load factor across the cut is either unchanged or diminished by replacing $a \rightarrow b$ and $b \rightarrow c$ with $a \rightarrow c$.

Proof: We show only the first part of the lemma since the second part follows immediately by induction. We shall show that $\text{load}(Q, S) \leq \text{load}(P, S)$ for any cut S of the underlying network. 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 across the cut is either unchanged or diminished when $a \rightarrow b$ and $b \rightarrow c$ are replaced with $a \rightarrow c$, as is shown in Figure 2. □

In summary, this section has introduced the notion of a conservative algorithm. An upper bound on the time required by a conservative algorithm can be determined solely from the embedding of an input data structure on the DRAM. If the number of steps of the conservative algorithm is T and the load factor of the input data structure is λ, then the total time is at most λT. A user of a conservative algorithm therefore need only minimize the congestion of pointers in the input data structure across cuts of the DRAM to minimize the time required by the algorithm. If the embedding of the data structure is good, that is, its load factor is small, then a conservative algorithm that uses a small number of steps runs fast.

4 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 *contract* an input list by repeatedly pairing and

merging adjacent elements of the list until only a single element remains. The merges are recorded as internal nodes of a binary *contraction tree* whose leaves are the elements in the input 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 $O(\lg n)$, where n is the number of elements in the input list. 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 a contraction tree of height $O(\lg n)$.

The recursive pairing strategy is illustrated in Figure 3 for a list (A, B, C, D, E). In the first step, elements B and C pair and merge, as do elements D and E. The merges are shown as contours in the figure. A new *contracted* list (A, BC, DE) is formed from the unpaired element A and the two compound elements BC and DE. After the second step of the algorithm, the contracted list consists of the elements ABC and DE. The third and final step reduces the list to the single element $ABCDE$.

In Algorithm LC, the contours of Figure 3 are represented in a data structure called a *contraction tree*. The leaves of the contraction tree are the list elements, and the internal nodes are the contours. To maintain the contraction-tree data structure, the algorithm requires constant extra space for each element in the input list. Each processor contains two elements: an element in the input 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 [17] for similar reasons, but in a different context.)

We now describe the operation of Algorithm LC, which is illustrated in Figure 4 for the example of Figure 3. (A description in pseudocode can be found in [19].) In the first step, each element of the input list randomly picks either its left or right neighbor. An element at the left or right end of the list always picks its only neighbor. If two elements pick each other, then they merge. The merge is recorded by making the spare of the left element of the pair be the root of a new contraction tree. The spare of the right element becomes the spare for the root, and the elements themselves become the children of the root. The roots of the new contraction trees and the unpaired list elements now form themselves into a new list representing

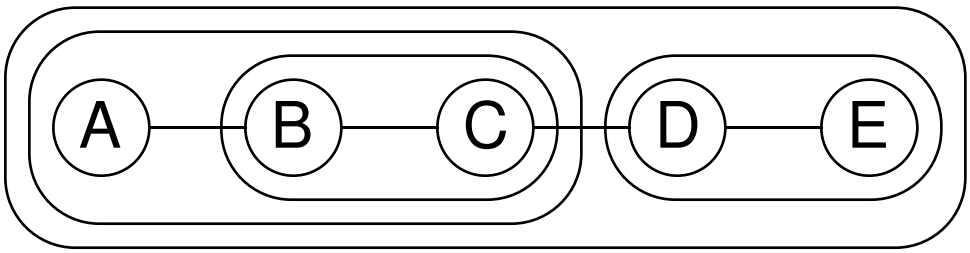

Figure 3: The recursive pairing strategy operating on a list (A, B, C, D, E). Merged nodes are shown as contours, and the nesting of contours gives the structure of the contraction tree.

the contracted list, upon which the algorithm operates recursively.

At each step of the algorithm, any given element of the contracted list is a set of consecutive elements in the input list—a contour in Figure 3. The set is represented by a contraction-tree data structure whose leaves are the elements of the set and whose internal nodes record the merges. When the entire input list has been contracted to a single node, the algorithm terminates and a single contraction tree records all of the merges.

To describe the efficiency of randomized algorithms such as Algorithm LC, we shall sometimes say that an algorithm runs in $O(T(n))$ steps "with high probability," by which we shall mean that for any constant $k > 0$, there are constants $c_1 > 0$ and $c_2 > 0$ such that with probability $1 - c_1/n^k$, the algorithm terminates in at most $c_2 T(n)$ steps.

Theorem 0.2 *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. For each token destroyed, the length of the list decreases by one, and the algorithm terminates when no token remains. In any iteration, an existing token has

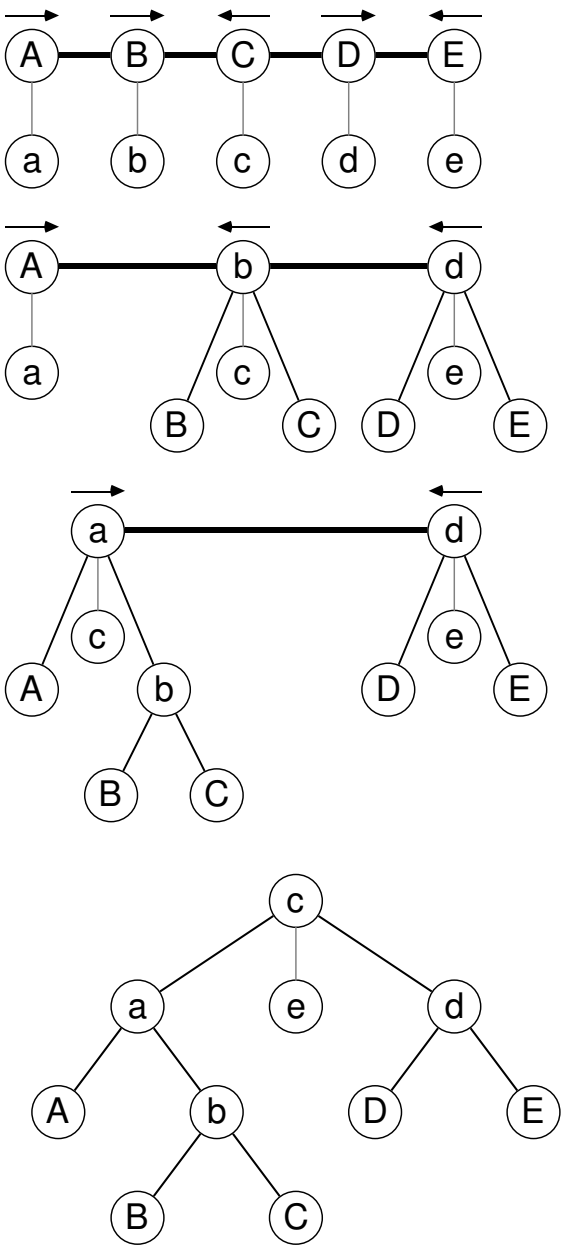

Figure 4: The operation of Algorithm LC on the example of Figure 3. The input list is (A, B, C, D, E), and the corresponding spares are in lower case. When elements B and C pair and merge in the first step of the algorithm, the spare b becomes the root of a contraction tree with leaves B and C to represent the compound node BC. The spare for b is c. At the end of the first step, the list consisting of the elements A, b, and d represents the contracted list (A, BC, DE). After two more contraction steps of Algorithm LC, the input list is contracted to a single element $ABCDE$, which is represented by a contraction tree whose root is c and whose leaves are the elements of the input list (A, B, C, D, E).

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 the probability that any token remains after m iterations is given by

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

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

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

□

Theorem 0.3 *With high probability, a contraction tree constructed by Algorithm LC on a list of n elements has height $O(\lg n)$.*

Proof: The height of the contraction tree is no greater than the number of iterations of Algorithm LC. □

We now prove that Algorithm LC is conservative.

Theorem 0.4 *Algorithm LC is conservative.*

Proof: By convention, let the mate of an element in the input list lie in the order between that element and its right neighbor. The key idea is that the order of the list elements and their spares is preserved by the merging operation, and consequently, after each contraction step, the pointers in the contracted list correspond to disjoint paths in the original list, and the pointers between elements and their spares also correspond to disjoint paths. By the Shortcut Lemma these two sets of pointers are each conservative with respect to the input list, and since each set of memory accesses in a contraction step of the algorithm is a subset of one of these two sets, the algorithm is conservative. □

Although a contraction tree itself is not conservative with respect to an input list, it can be used as a data structure in conservative algorithms. For

example, contraction trees can be used to efficiently broadcast a value to all of the elements of a list and to accumulate values stored in each element of a list.

More generally, contraction trees are useful for performing *prefix computations* in a conservative fashion. Let $\mathcal{D}$ be a domain with a binary associative operation $\cdot$ and an identity ε. A prefix computation [3, 7, 23] on a list with elements $x_1, x_2, \ldots, x_n$ in $\mathcal{D}$ puts the value $y_i = x_1 \cdot x_2 \cdots x_i$ in element i for each $i = 1, 2, \ldots, n$.

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. When an internal node receives a value z_l from its left child and a value z_r from its right child, the node saves the value z_l and passes $z_l \cdot z_r$ to its parent. When the root receives values from its children, the second top-down phase begins. The root passes ε to its left child and its z_l value to its right child. When an internal node receives a value z_p from its parent, it passes z_p to its left child, and passes $z_l \cdot z_p$ to its right child. When a leaf receives z_p it computes $y = z_p \cdot x$.

The number of steps required by the prefix computation is proportional to the height of the tree, which with high probability is $O(\lg n)$. 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 0.5 *Let CT be a contraction tree computed by Algorithm LC on an input list L, and suppose P is a subset of the pointers of CT. If no pointer in P is an ancestor of another pointer in P, 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, if no pointer in P is an ancestor of another pointer in P, 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 at most 1. The overall running time for list contraction is thus $O(\lg n \lg^* m)$.

As a final comment, we observe that with minor modifications, Algorithm LC can be used to contract circular lists with the same complexity bounds as for linear lists.

5 Tree contraction

This section presents a conservative tree contraction algorithm, Algorithm TC, based on the tree contraction ideas of Miller and Reif [22]. The algorithm uses a recursive pairing strategy to build a contraction tree for an input 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 $O(\lg n)$, where n is the number of nodes in the input tree. A deterministic variant runs in $O(\lg n \lg^* m)$ steps and produces a contraction tree of height $O(\lg n)$.

The recursive pairing strategy for trees is illustrated in Figure 5 for a tree with nodes A, B, C, D, E, and F. In the first step nodes A and B pair and merge, as do nodes C and D; the merges are shown as contours in the figure. A new *contracted* tree is formed from the unpaired nodes E and F, and the compound nodes AB and CD. In the next step of the algorithm, node E pairs and merges with CD to form a node CDE. After two more steps the 6-node input tree has been contracted to a single node. Notice that each node shown as a contour in the figure is a connected subgraph of the input tree, and that the node has at most two children in the contracted tree.

Algorithm TC represents the contours of Figure 5 in a contraction-tree data structure in the same manner as Algorithm LC represents the contours of Figure 3. Space for the internal nodes of the contraction tree is again provided by spares. Initially, the spare of each node in the input tree is its mate, an unused node stored in the same processor.

We now outline Algorithm TC in more detail. (A description in pseudocode can be found in [19].) In the first step, nodes in the input tree are paired. The pairing strategy has each node pick from among its neighbors according to how many children it has. A leaf picks its parent with proba-

bility 1. A node with exactly one child picks either its child or its parent, each with probability 1/2. A node with two children picks either child, each with probability 1/2. The root, which has no parent, picks its children with equal probability. If two nodes pick each other, then they merge. The merge is recorded by making the spare of the parent in the pair be the root of a new contraction tree. The spare of the child in the pair becomes the spare for the root, and the parent and child themselves become the children of the root. The new nodes and the unpaired nodes form themselves into a new tree that represents the contracted tree, upon which the algorithm operates recursively. The contracted tree is binary because the pairing strategy ensures that no node with two children pairs with its parent.

In the next section, we shall need to *expand* a contracted tree in order to describe treefix computations recursively. Expansion consists of undoing the merges in the reverse of the order in which they occurred. From the time that a parent and child merge to the time that the node representing their merge in the contraction tree expands, the pointers of the pair are undisturbed. Consequently, these pointers can be used to restore the pointers of the neighbors of the pair to the state they had immediately before the pair merged. To ensure that the merges are undone in the exact reverse order, as is assumed in the next section, it is helpful to store in each internal node of the contraction tree the step number in which the merge took place. In fact, the tree can be expanded by a greedy strategy without consulting the number of the contraction step at which each merge occurred.

The proof that with high probability, Algorithm TC takes $O(\lg n)$ steps to contract an input 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-type bound [4] on the tail of a binomial distribution.

Lemma 0.6 *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*

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

Lemma 0.7 *Let $X \leq b$ be a discrete random variable with expected value*

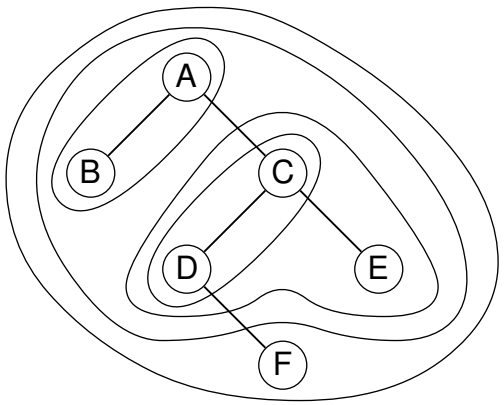

Figure 5: The recursive pairing strategy operating on a tree with nodes A,B,C,D,E, and F. Merged nodes are shown as contours, and the nesting of contours gives the structure of the contraction tree.

μ. *For* $w < b$, *we have*

$$\Pr\{X \geq w\} \quad \geq \quad \frac{\mu - w}{b - w} \ . \square$$

The final lemma presents a bound on the tail 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

$$B(s,t,p) \quad = \quad \sum_{k=0}^{s-1} \left(\begin{array}{c} t \\ k \end{array} \right) p^k (1-p)^{t-k} \ .$$

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

Lemma 0.8 *For* $s < t/2$ *and* $p < 1/2$, *we have*

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

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.

Theorem 0.9 *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 0.6 to show 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 0.7 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 0.8 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 $|V|/4$, provided that $|V| > 1$. A child 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 pairs with its parent with probability at least $1/4$. Let P be the number of nodes pairing with a parent. Then we have

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

and applying Lemma 0.6 yields 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 is at least $1/3$. We call such a step *successful*. At most half of the nodes pair with their parents. Using Lemma 0.7 with $b = |V|/2$, $w = |V|/8$, and $\mu \geq |V|/4$, we have

$$\Pr\left\{ P \geq \frac{|V|}{8} \right\} \geq \frac{\frac{|V|}{4} - \frac{|V|}{8}}{\frac{|V|}{2} - \frac{|V|}{8}} = \frac{1}{3} .$$

Finally, we show that with high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract the input tree to a single node. The size of the tree after a contraction following a successful pairing step is at most $7/8$ the size before the contraction. After $\log_{8/7} n$ successful steps, the tree must

consist of a single node. By Lemma 0.8, the probability that fewer than $\log_{8/7} n$ successful steps occur in $\alpha \log_{8/7} n$ steps is

$$\begin{aligned} B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3) &\leq 2\left((2/3)^{\alpha} \alpha e\right)^{\log_{8/7} n} \\ &= 2n^{\log_{8/7}((2/3)^{\alpha} \alpha e)}. \end{aligned}$$

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

We now prove that Algorithm TC is communication efficient in the DRAM model.

Theorem 0.10 *Algorithm TC is conservative.*

Proof: Each node of a contracted tree is a connected subgraph of the input tree. The root of the contraction tree that represents the subgraph is called the *representative* of the subgraph. The representative and its spare are each either a node of the subgraph or a mate of a node of the subgraph.

Every set of memory accesses performed by the algorithm is of one of two types. In the first type, each representative of a subgraph communicates with its spare, if at all. In the second type, each representative of a subgraph communicates with the representative of one of its children in the contracted tree. In either of these two cases, the set of memory accesses corresponds to a set of disjoint paths in the input graph, and hence, by the Shortcut Lemma, is conservative with respect to the input graph. □

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 that can pair form chains, and by Lemma 0.6 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.

6 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 that uses $O(n)$ space, where n is the number of nodes in

the input tree. As we shall see in Section 7, 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 $\cdot$ and an identity ε. Let T be a rooted, binary tree in which each vertex $i \in T$ has an assigned input value $x_i \in \mathcal{D}$. The *rootfix* problem is to compute for each vertex $i \in T$ with parent j, the output value $y_i = y_j \cdot x_i$, 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 output value $y_i = x_i \cdot y_j \cdot y_k$, 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. For simplicity, however, we describe a recursive version.

Theorem 0.11 *Let T be a binary tree of n nodes on a DRAM with m processors. A rootfix or leaffix computation can be performed on T 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. Both algorithms use $O(1)$ space per node of the tree.*

Proof: Both treefix computations are performed by executing a single contraction step on the input tree T to produce a contracted tree T'. Each node in T' is assigned an input value, and the treefix computation is executed recursively on T'. The contracted tree T' is then expanded to yield T once again, and the output value of each node in T is computed from the input values of T and the output values of T'.

The algorithm for leaffix is based on each node i maintaining a value s_i which has the form $a_i \sqcup b_i \sqcup c_i$, where $a_i, b_i, c_i \in \mathcal{D}$ are elements of the domain, and the character "$\sqcup$" represents symbolically a slot to be filled in with a value. The number of slots is equal to the number of children of the node, and each slot corresponds to a specific child. When a parent and child pair during the course of the leaffix algorithm, the value of the

child is substituted into the corresponding slot in the value of its parent. For example, suppose node i pairs with its right child j, where the value of i is $s_i = a_i \sqcup b_i \sqcup c_i$ and the value of j is $s_j = a_j \sqcup b_j$. The value s_k of the merged node k is computed from s_i and s_j by substituting s_j into the second slot in s_i, yielding the value $s_k = a_i \sqcup b_i \cdot a_j \sqcup b_j \cdot c_i$. The $\cdot$ operations are carried out immediately so that s_k has the proper form.

The leaffix algorithm initializes each node i by $s_i \leftarrow x_i$, $s_i \leftarrow x_i \sqcup$, or $s_i \leftarrow x_i \sqcup \sqcup$ depending on the number of children of node i. The algorithm then proceeds as follows. At the end of a contraction step, each node k in T' that results from the merging of parent i and child j computes its value s_k by substituting s_j into the appropriate slot of s_i. The leaffix algorithm is then performed recursively on T' using the s values as inputs and yielding y values as output. (The y values contain no slots and are simply elements of the domain $\mathcal{D}$.) During the expansion step, the parent node i sets $y_i \leftarrow y_k$. Each child node j gets its output value y_j by substituting the y values of its children into the slots of s_j.

In the rootfix algorithm, each node i maintains a value s_i, as in the leaffix algorithm, but each s_i now has the general form $s_i = \sqcup a_i$, and the slot of a node corresponds to the node's parent. The rootfix algorithm initializes each node i by $s_i \leftarrow \sqcup x_i$, except for the root r which performs $s_r \leftarrow x_r$. After the pairs have been determined for the contraction step, each node j that is the child in a pair, and which itself has a child, substitutes s_j in the appropriate slot of its child's value. At the end of a contraction step, each node k in T' that resulted from the merging of parent i and child j computes its value by $s_k \leftarrow s_i$. The rootfix algorithm is then performed recursively on T', yielding y values as output. During the expansion step, the parent node i sets $y_i \leftarrow y_k$. Each child node j gets its output value y_j by substituting y_i into the slot of s_j.

The time and space bounds claimed in the theorem are apparent by inspection. Each step of a treefix algorithm adds only a constant amount of work to a corresponding step in the tree contraction and expansion algorithms. The additional space required by the treefix algorithms is the $O(1)$ space per node for the x, y, and s values. □

7 Graph algorithms

This section presents a collection of conservative DRAM algorithms for solving graph problems. The algorithms use two processors per edge of an input graph $G = (V, E)$ and require only constant extra space in each processor.

Most of the algorithms use treefix computations as subroutines.

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 [14, pp. 332–333].

We now present brief descriptions of the algorithms. The performance is given in 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 time required 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 $y_i = x_i \cdot y_1 \cdot y_2 \cdots y_k$.* Each nonprincipal element is assigned the identity 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 descendants, its height, or its depth.* The number of dependents for each node can be computed by a leaffix computation with $\cdot$ as integer addition and $x_i = 1$ for all nodes. The height of a node can also be computed by a leaffix computation where $a \cdot b = \max(a + 1, b + 1)$, the identity is $\varepsilon = -1$, and $x_i = -1$ for all nodes.[2] The depth of a node can be computed by a

[2]Technically, $\varepsilon = -1$ is not an identity for the operation $a \cdot b = \max(a + 1, b + 1)$. Nonetheless, this leaffix computation correctly computes the height of each node in a

rootfix computation with $\cdot$ 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 [27] 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. Use Algorithm LC to 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 toward k.* The algorithm for rooting a tree can be used by picking k as the root instead of the root of the contraction 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 $\cdot$. 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 $x_k \leftarrow 1$. If node k is a right child, set x_k to the size of its sibling subtree plus 1. A rootfix computation with $+$ yields y_k, which is the preorder numbering of node k. The inorder numbering can be computed similarly. If node k is a left child, set $x_k \leftarrow 0$. If k is a right child, set x_k to the size of its sibling subtree plus 1. Compute y_k for each node using a

binary tree. Moreover, this leaffix computation also generalizes to a directed tree with arbitrary vertex degree.

rootfix computation with $+$. The inorder numbering of node k is y_k plus the size of its left subtree plus 1. The postfix numbering can be computed by setting $x_k \leftarrow 0$ 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 y_k plus the sizes of its two subtrees plus 1. *Performance:* $O(\lg n)$.

Prefix and postfix numberings of a directed tree. *Number the edges of an arbitrary directed tree according to the order they are visited in a 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 farthest leaf from the root. Reroot the tree at this leaf. The distance from the new root to the farthest leaf is the diameter. (This algorithm is 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* [20] *is a partition of the vertices of an n-vertex tree into three sets A, B, and C, with $|A| \leq \frac{2}{3}n$, $|B| = 1$, and $|C| \leq \frac{2}{3}n$, 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 $\frac{2}{3}n$ 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 $\{+, -, \cdot, \div\}$, com-*

pute for each internal node the subexpression rooted at that node. A single leaffix-like computation suffices using the ideas of Brent [2] and Miller and Reif [22]. *Performance:* $O(\lg n)$.

Minimum-cost spanning forest. *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 [26, 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 $e \in E$ connecting an edge element $u \in T_i$ with an edge element $v \in T_j$, where $i \neq j$, with smallest weight. 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* $l(v) = l(v')$ *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 [27]. 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 $\mathrm{high}(v)$. The value $\mathrm{nd}(v)$ is the number of descendants of v, and $\mathrm{low}(v)$ ($\mathrm{high}(v)$) is the lowest (highest) numbered vertex (with respect to the preorder numbering of T) that is 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-cost-spanning-forest algorithm. *Performance:* $O(\lg^2 n)$.

8 Conclusion

This section offers a perspective on the DRAM model. We explore the analogy between PRAM's and universal networks on the one hand, and DRAM's and volume-universal networks on the other. We then discuss the issue of how data structures can be efficiently embedded in DRAM's—a problem not faced in the PRAM model. We also suggest how one might define the load factor for data structures other than graphs, such as matrices. Finally, we offer some comments on how some of our definitions and techniques might be extended or generalized.

The literature contains a large body of results concerning *universal* networks, such as the Boolean hypercube [29]. Universal networks are capable of simulating any PRAM program with at most polylogarithmic degradation in time (see, for example, the simulation [12] of an EREW-PRAM on a butterfly network). In light of this work, one might wonder why the DRAM

model should be studied at all.

A potential problem with universal networks is that they may be difficult to physically construct on a large scale. The number of external connections (pins) on a packaging unit (chip, board, rack, cabinet) of an electronic system is typically much smaller than the number of components that the packaging unit contains, and can be made larger only with great cost. When a network is physically constructed, each packaging unit contains a subset of the processors of the network, and thus determines a cut of the network. For a universal network, the capacity of every cut must be nearly as large as the number of processors on the smaller side of the cut; otherwise, the load-factor lower bound would make it impossible to perform certain memory accesses in polylogarithmic time. Thus, when a universal network is physically constructed, the number of pins on a packaging unit must be nearly as large as the number of processors in the unit. Consequently, if all the pin constraints are met, a packaging unit cannot contain as many processors as might otherwise fit. Alternatively, if each packaging unit contains its full complement of processors, then pin limitations preclude the universal network from being assembled.

The impact of pin constraints can be modeled theoretically in the three-dimensional VLSI model [16, 18] where hardware cost is measured by volume and the pinboundedness of a region is measured by its surface area. In this model, the largest universal network that can fit in a given volume V has only about $V^{2/3}$ nodes. In the two-dimensional VLSI model [28], where pinboundedness is measured by perimeter, the bound is even worse.

Since the density of processors in a physical implementation of a universal network is low, it is natural to wonder whether there are other networks that make more efficient use of hardware. Recently, it has been shown that *fat-trees* [10, 18] are such a class of "volume-universal" networks. A fat-tree of volume V can simulate any other network of comparable volume with only polylogarithmic degradation in time. (Figure 6 shows an area-universal fat-tree.) Thus, a fat-tree of volume V can efficiently simulate not only the universal networks with the same volume, but also some networks with almost V nodes. A key component in the proof that fat-trees are volume-universal is an algorithm for routing a set of messages on a fat-tree in time that is at most a polylogarithmic factor larger than the load factor.

With a suitable assignment of capacities to cuts, the DRAM can 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

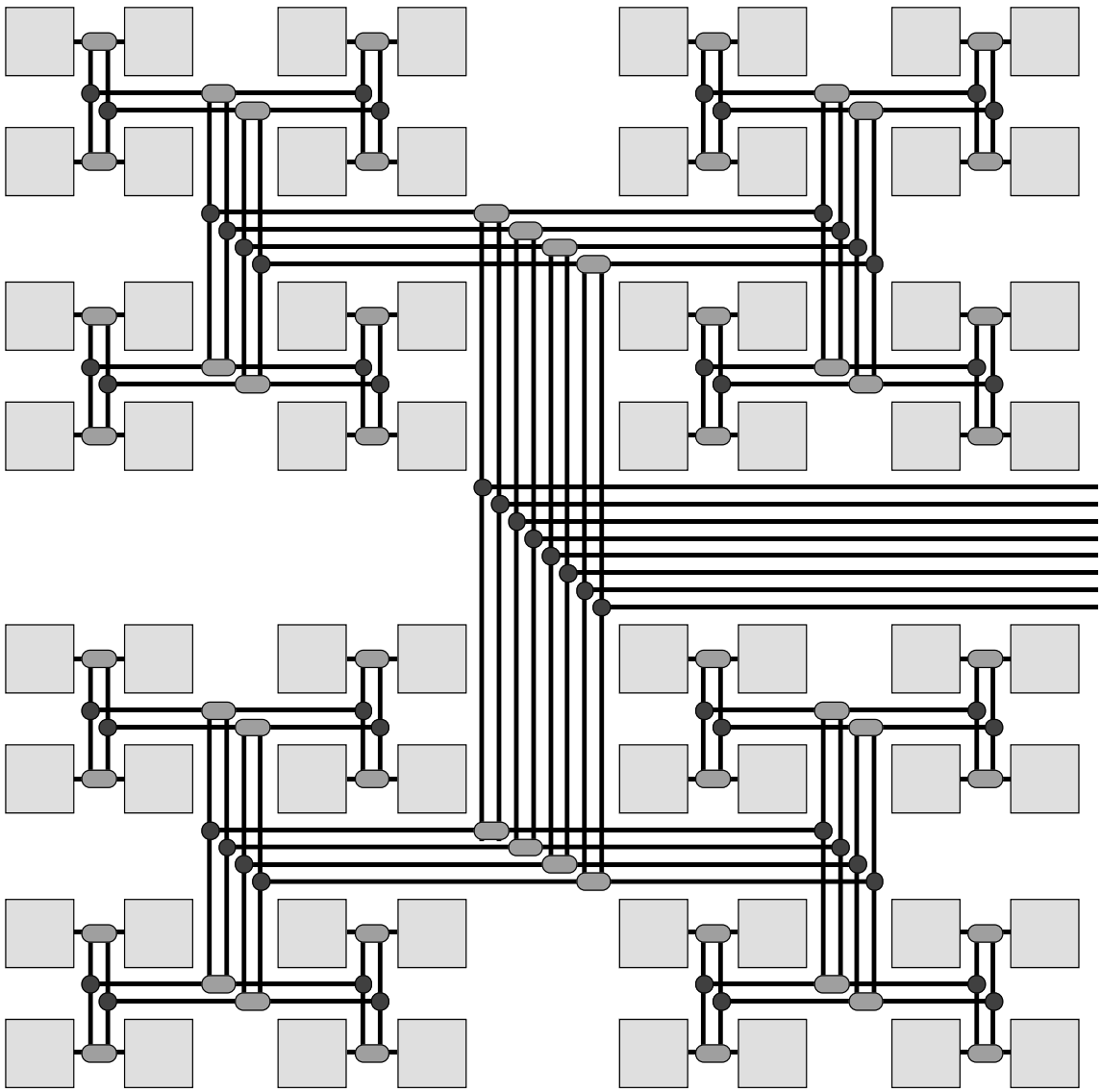

Figure 6: A fat-tree network. An area-universal fat-tree, like the one shown, is capable of efficiently simulating any other network of comparable area. Fat-trees are well modeled by distributed random-access machines.

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 with the appropriate cut capacities translate directly to fast, communication-efficient algorithms on, for example, a fat-tree.

We now turn to the problem of embedding data structures in DRAM's, a problem that must be faced by users of conservative algorithms if the algorithms are to run quickly. In general, the problem of determining the best embedding for an arbitrary data structure 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.

To illustrate how the embedding problem can be solved in certain practical situations, consider the class of DRAM's whose cut capacities correspond to area-universal fat-trees. For this class of DRAM's, the recursive structure of the underlying fat-tree network suggests that a divide-and-conquer approach be taken. 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 would yield small load factors for every cut. Thus, our conservative connected-components algorithm would never cause undue congestion in communicating messages in the underlying network, and the algorithm would run 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* [20] 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 [15, 17] and image processing. Planar graphs and multigrids arise from the solution of sparse linear systems of equations based on the finite-element method. Consequently, conservative DRAM algorithms operating on good embeddings of these graphs would run fast on a fat-tree.

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 [15, 17], and the load factor of the matrix can be defined as the load factor of the mesh.

For some algorithms, the running time may be better characterized as 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 a 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. Such a sorting algorithm is known for fat-trees [13], but whether such an algorithm exists for general DRAM's 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 fat-trees is used in [21] to show that an optimal reordering of a linear list in a fat-tree can be determined efficiently by a conservative algorithm on the fat-tree.

As a final comment, we note that the notion of a conservative algorithm may well be too conservative. 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 not much greater than the input load factor. For example, 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. Algorithms with this looser bound are somewhat easier to code because of the relaxed constraint, and they should perform comparably.

References

[1] S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, April 1984.

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

[3] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C–31(3):260–264, March 1982.

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

[5] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 206–219, May 1986.

[6] A. K. Dewdney. Computer recreations. *Scientific American*, 252(6):18–29, June 1985.

[7] M. J. Fischer and R. E. Ladner. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[8] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, May 1986.

[9] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 7–18, May 1987.

[10] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.* Volume 5 of *Advances in Computing Research*, pages 345–374. JAI Press, Greenwich, CT, 1989.

[11] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[12] A. R. Karlin and E. Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 160–168, May 1986.

[13] J. Kilian, July 1986. Personal communication.

[14] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison–Wesley, Reading, MA, second edition, 1973.

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

[16] F. T. Leighton and A. L. Rosenberg. Three-dimensional circuit layouts. *SIAM Journal on Computing*, 15(3):793–813, August 1986.

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

[18] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[19] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms. Technical Memo MIT/LCS/TM-318, MIT Laboratory for Computer Science, Cambridge, MA, December 1986.

[20] R. J. Lipton and R. E. Tarjan. A planar separator theorem. *SIAM Journal of Applied Mathematics*, 36(2):177–189, April 1979.

[21] B. M. Maggs. *Locality in Parallel Computation.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1989.

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

[23] Yu. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics – Doklady*, 7(7):589–591, 1963. English translation.

[24] G. F. Pfister and V. A. Norton. 'Hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C–34(10):943–948, October 1985.

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

[26] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

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

[28] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[29] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[30] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, Ithaca, NY, August 1979.