JOURNAL OF COMPUTER AND SYSTEM SCIENCES 50, 98–113 (1995)

Efficient Asynchronous Simulation of a Class of Synchronous Parallel Algorithms

NAOMI NISHIMURA*

Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Received August 28, 1992; revised April 27, 1994

Several recent papers have introduced asynchronous shared memory parallel models in an attempt to discover how removing the assumption of synchronization of processor steps may alter the parallel complexities of problems. Preliminary work has resulted in the development and analysis of algorithms for a few specific problems. The best known general technique for transforming synchronous algorithms into asynchronous ones has been to synchronize all processors after each step of a synchronous computation. This results in the time complexity being multiplied by a factor that may be logarithmic in the number of processors, where time is defined to be the expected maximum numbers of steps taken by any processor, with respect to several families of distributions on processor schedules. Here we give a transformation technique that forms an asynchronous algorithm that runs in $O(t + \log p)$ time using p processors from any synchronous algorithm that runs in $O(t)$ time on a p-processor CROW PRAM. This result can be extended to more general classes of synchronous algorithms. The technique yields an $O(\log p)$ time algorithm for p-input sorting, making possible the simulation of algorithms that run on the PRIORITY CRCW PRAM.

1. INTRODUCTION

Asynchronous models of shared memory parallel computation have been the focus of a number of recent papers [3, 5, 6, 9, 10, 12, 17–19, 21, 25–29, 32]. Work in this area has been directed towards extending the notion of parallel complexity to include practical issues that arise in actual parallel machines. In order to be able to understand the impact of asynchrony on complexity theory, it is necessary to be able to compare the synchronous and asynchronous complexities of large numbers of problems.

Preliminary work in the area has yielded complexity results for a few specific problems, such as computing the OR of the input bits and computing the rank of each element in a linked list. In addition, there have been investigations into the complexity of executing a *synchronization barrier*, that is, of each processor waiting until all processors have reached a certain point in the algorithm. A synchronous algorithm running on an ARBITRARY CRCW PRAM will run correctly on an asynchronous model if a barrier is executed after each PRAM timestep. If a barrier could be executed in constant time, then any synchronous algorithm could be executed on an asynchronous machine without an asymptotic increase in the running time.[1] However, the cost of executing a barrier in most of the models is in $\Omega(\log p)$, where p is the number of processors. By this naive transformation technique, a problem that can be solved by an algorithm running in time $O(t)$ on a p-processor ARBITRARY CRCW PRAM can be executed in $O(t \log p)$ time on an asynchronous model using the same number of processors. In this paper, it is shown how a problem that can be solved by an algorithm running in time $O(t)$ on a p-processor CROW PRAM can be solved asynchronously in $O(t + \log p)$ time using the same number of processors.

Section 2 introduces the asynchronous shared memory model considered in this paper. The definition of time as a measure of the average case over distributions of processor schedules is explained and justified, and then compared to measures of time in related models. A few basic upper and lower bounds are mentioned, including a result showing that any synchronous algorithm running on an ARBITRARY CRCW PRAM can be simulated asynchronously by increasing the running time by a factor of $\log p$. In addition, it is shown that synchronization of p processors requires $\Omega(\log p)$ time. It is then clear that certain synchronous algorithms cannot be transformed into asynchronous ones without significantly increasing the resources used; earlier papers [5, 6, 9, 28, 29] demonstrate certain other algo-

* Research supported by the Natural Sciences and Engineering Research Council of Canada. E-mail: nishi@plg.waterloo.edu.

[1] The running time for an asynchronous algorithm is defined to be the expected maximum number of steps taken by any processor, with respect to several families of distributions on processor schedules. This notion will be discussed in further detail in Section 2.

0022-0000/95 $6.00

rithms that can be transformed in a resource-preserving manner.

Section 3 contains a technique for transforming a class of synchronous algorithms into asynchronous ones, without increasing resource requirements by more than a constant factor. This technique depends on a particular representation of the algorithm. It is not difficult to see, as shown in Section 3.1.1, that the running of any parallel algorithm on a fixed input can be represented as a directed acyclic graph. Consequently, an algorithm can be represented as a set of graphs, one for each input. Because of the lower bound in Section 2, it is evident that in general, such a representation cannot be used to find a resource-preserving transformation. However, for the restricted class of representations presented in Section 3.1.2, such a transformation is always possible. This is demonstrated in Section 3.2.

In Section 3.3, we consider the set of synchronous algorithms whose representations fall into the restricted class. This set includes the iterated application of any associative binary function, comparator network algorithms [20] such as sorting, and algorithms computed on a butterfly graph [23]. The sorting result allows us to extend our naive insertion of synchronization barriers to apply to PRIORITY CRCW PRAM algorithms.

More generally, it is possible to transform algorithms running on a specially restricted COMMON CRCW PRAM. This type of PRAM can alternatively be viewed as an extension of the CROW PRAM; consequently, CROW PRAM algorithms have representations in the restricted class. Questions concerning the relations between this class and others are considered, along with other directions for further work, in Section 4.

2. AN ASYNCHRONOUS MODEL

2.1. Definition of the Model

Intuitively, the model used in this paper is a PRAM in which the global clock is replaced by a single local clock for each processor. As in the PRAM, each processor is a random access machine [1] with an instruction set consisting of reads and writes of local and global memory and local operations. At each tick of its clock, a processor can execute one operation. Since we are not assuming that the processors are synchronized, processor actions may be interleaved or overlapped in arbitrary ways. Any algorithm on this model must be correct for any such schedule of processor steps.

The memory cells in the model are atomic read/write cells [22], which by definition satisfy the correctness condition of linearizability [14]. This condition guarantees that no matter how processor actions are interleaved or overlapped, the values returned by the global reads are consistent with some total order of processor actions. To consider all possible schedules of processor steps, it will thus suffice to consider all possible total orders of processor steps. We call such a total order a *linearization*; a linearization for p processors consists of a sequence made up of the IDs of the p processors. It has been shown that the power of a model of computation depends heavily on the atomicity assumptions [11]. Models with atomic read/write cells are at one end of the spectrum; atomic synchronization barriers that cost unit time put the synchronous PRAM model at the other end.

In the synchronous setting, the time complexity of an algorithm on a particular input is equivalent to the maximum number of steps taken by any processor in the course of the execution. The actions performed in the execution of a particular asynchronous algorithm on a particular input might depend on the particular schedule of processor steps that occurs. A worst-case measure can in essence be a determination of how well the algorithm degrades into a sequential algorithm; moreover, it is based on an event that has low probability. Instead, we consider time to be defined as the expected maximum number of steps taken by any processor, where the expectation is calculated given a distribution on all linearizations. The number of steps taken by each processor can be normalized by a weighting factor for relative processor speeds, so that the measure is sensitive to actual processor speeds. (In particular, we can distinguish between the effects of replacing half the processors with faster ones and replacing half the processors with slower ones, by having an absolute rather than a relative notion of speed.) For the distributions considered in this paper, weighting factors are not used. Further details on this and other aspects of the model can be found in related papers [27–29].

We consider two general classes of probability distributions, the *simple delay distributions*, denoted $\mathscr{S}$, and the *constant speed distributions*, denoted $\mathscr{C}$. Simple delay distributions can be represented by sets of two-state Markov chains, one chain per processor. In each Markov chain, one state corresponds to the associated processor taking a step; the other corresponds to idle time. The ID of the processor is the output of the step state. Transitions are made simultaneously on the Markov chains for all the processors, and the resultant outputs are ordered to form a linearization. The outputs are ordered according to a sequence of permutations of processor IDs, one permutation per transition; the outputs resulting from a particular transition are ordered in a manner consistent with the permutation for that transition. Let ρ_i be the probability of transition from the step state to the step state in the chain for processor i, and let $\bar{\rho}_i$ be the probability of transition from the idle state to the step state. We stipulate that all ρ_i's and $\bar{\rho}_i$'s are within a constant factor of each other over all values of i, independent of p, the number of processors, and are in $\Omega(1/p^{O(1)})$. For I an input of size n, $\mathscr{O}$ a sequence of permuta-

tions of IDs of p processors, and l a linearization, the time complexity of an algorithm A is

$$\max_I \max_{\mathcal{C}} \sum_l Pr_{\mathcal{C}}[l] \times \max_j \left\{ \begin{pmatrix} \text{number of steps taken by} \\ \text{processor } j \text{ on input } I \\ \text{using algorithm } A \text{ with linearization } l \end{pmatrix} \right\}.$$

The time complexity of a problem can be defined, as in the synchronous setting, as the minimum over all algorithms solving the problem of the time complexity of the algorithm.

We may wish to model a situation in which the probabilities of the various processors taking steps are interdependent, unlike in the simple delay distributions. We can think of a constant speed distribution as being generated by the repeated throwing of an uneven p-sided die. Whenever side i lands upward, the ID of processor i is the next position in the linearization. The probability of the die landing with side i up is defined to be ρ_i; we consider the situation in which for all i and j, ρ_i/ρ_j is bounded by a constant independent of p. For I an input of size n, δ a distribution on linearizations for p processors, and l a linearization for p processors, the time complexity of an algorithm A is

$$\max_I \sum_l Pr_\delta[l] \times \max_j \left\{ \begin{pmatrix} \text{number of steps taken by} \\ \text{processor } j \text{ on input } I \\ \text{using algorithm } A \text{ with linearization } l \end{pmatrix} \right\}.$$

As above, the time complexity of a problem can be defined analogously.

Various situations can be modeled by other probability distributions. A setting in which processors are subject to fail-stop errors could be modeled by a distribution in which processors cease to be included in the linearization with certain probabilities. The computation by a PRIORITY CRCW PRAM can be seen as a computation on this model in which a single linearization is allowed, the linearization in which the $(ip+1)$th through $(i+1)$ pth steps consist of the p processor IDs in order from lowest priority to highest, for all i. Because the asynchronous models we consider allow this linearization to occur, we can view PRAM time measures as special cases of our measure; the common framework gives us a basis for comparison between synchronous and asynchronous time measures. Accordingly, we will use the term *time* to refer to our measure when comparing our results to results for synchronous algorithms; a simulation is *time-preserving* if the running time of the resulting asynchronous algorithm, according to our measure, is within a constant factor of the running time of the original synchronous one.

2.2. Related Work

In the last few years, several different asynchronous models have been proposed; we restrict our attention to those which communicate through shared memory. These models differ from ours primarily in the definition of complexity measures; the basic model is derived from a PRAM in which the assumption of lock-step behavior is removed and algorithms are correct only if they perform correctly no matter what schedule of processor steps occurs. In all these models it is assumed that inputs are present in shared memory at the start of the computation. Omitted from the discussion are results in the distributed computing literature in which the assumption of lock-step behavior is removed and algorithms are correct only if they perform correctly no matter what schedule of processor steps occurs. In all these evolved. The first stream [5, 6, 9, 10, 27–29], of which this work is part, is concerned with modeling the execution times of algorithms. In this setting, processors do not fail. Each measure is defined to avoid the degenerate sequential case mentioned in Section 2.1; some restrict the legal linearizations and some restrict the algorithms under consideration. The term *asynchronous PRAM* has been used to describe several different specific models; we avoid it here due to the confusion caused by a lack of a standard nomenclature. Further details of these measures are discussed in the remainder of this section.

In the second set of work [3, 16–19, 25, 26, 32] measures are devised to model the amount of work needed to complete a computation, typically the simulation of a single PRAM step. The models allow varying degrees of asynchrony, ranging from synchronous executions in which processors are subject to fail-stop errors to full-blown asynchrony plus fail-stop errors. Some models are deterministic, and some are probabilistic; those that rely entirely on probabilistic algorithms can be analyzed with respect to probabilistic, rather than deterministic, synchronous models. Several of the models assume actions known to be stronger than atomic read/write [11]; these models cannot be directly compared to models in the first stream of research (see the discussion in Section 2.1).

Even for the models most closely related to those that measure time, the demands of optimizing for work and the demands of optimizing for time are often incompatible (in the synchronous setting, consider the use of sequential versus parallel algorithms in this context). For some of the models, time measures are provided; they tend to rely on fairly restrictive assumptions about the degree of asynchrony required. This is not surprising, for as we saw in the discussion in Section 2.1, intuitively it would seem that a computation designed to work well in the worst case would necessarily require a large running time. Although some of the measures assume a probabilistic adversary, for the most part research in the second stream is concerned

with worst-case measures for computations that are *wait-free*; that is, computations in which each processor is guaranteed to finish its computation in a finite number of steps independent of the schedules (including failures) of other processors. Herlihy demonstrates the difficulty of developing wait-free algorithms on models in the first stream [12] (he also considers the use of randomization in his own work, in a decision that is consistent with that made here [13]). This difference, too, makes it difficult to compare the two streams of research.

Since the two streams of work are in a sense incomparable, we focus our attention on models that are designed to determine elapsed time rather than total work. We wish to have a measure that reflects the actual elapsed time achieved on real machines, and, therefore, that appropriately captures the notions of processor speed and load balance. The relationship with processor speeds can be phrased in the following simple-minded fashion. Consider the time complexity, according to some measure, of executing an algorithm with processors operating at certain fixed speeds. If we then consider the complexity when several of the processors are replaced by faster (or slower) ones, the time measure should not increase (respectively, decrease). The issue of load balance can be illustrated by considering an execution in which all processors have the same speeds and the work is balanced among the processors. If we compare the complexity with that arising from a situation in which the work is unbalanced, it would be expected that the complexity should not be any higher in the balanced case. Intuitively, if the machine speeds are close and the parallelism is high, the time measure should be low; if imbalance in speeds results in the inefficient use of processors, the time measure could be high. Finally, we would like an asynchronous time measure to be a generalization of the measure used in the synchronous PRAM model, so that comparisons between synchronous and asynchronous complexities have some meaning.

Gibbons [9] observes that, although any synchronous algorithm can be transformed into a synchronous one by inserting a synchronization barrier after each step, for certain algorithms fewer synchronization barriers are necessary. To keep the analysis simple, Gibbons stipulates that if one processor reads a cell and another processor writes the same cell, there must be a synchronization barrier between these two actions. This results in the inclusion of barriers, and consequently the increase in running time, in algorithms for problems that have no inherent need for synchronization. Moreover, the set of allowable algorithms is constrained by this rule. The time complexity of a sequence of steps between synchronizations, a *phase*, is taken to be the maximum number of steps taken by any processor during the phase. The insertion of a synchronization barrier after each phase ensures that the algorithm is slowed to the speed of the slowest processor for each phase. Although the complexity measure remains the same no matter what interleaving of processor steps occurs, the actual elapsed time is in fact very sensitive to change in the speeds of processors. In contrast, the measure used in this paper imposes no restriction on allowable algorithms and varies with variations in processor speeds.

The model proposed by Kruskal, Rudolph, and Snir [21] and Cole and Zajicek [5] measures time as the maximum number of minimal rounds needed to complete the algorithm, where a *round* is a sequence of steps during which each processor takes at least one step. This has the effect of removing certain linearizations from consideration. In the case of pointer jumping, the maximum occurs in the synchronous case, when each processor takes only a single step each round. In order to consider all possible linearizations, more recently Cole and Zajicek [6] defined a model based on the distributions of interleavings. Their unbounded delay model consists of a class of distributions that is strictly contained in the classes of distributions considered in this work. The paper contains algorithms for a subset of problems considered in independent, simultaneous work [28]. A more thorough discussion of the relationships between the various models can be found in earlier papers [28, 29].

2.3. Basic Results

The following results will be of use in the next section; their proofs depend on elementary probability theory and can be found in earlier work [28, 29]. Let $\sigma^* = \max_i\{\rho_i, \bar{\rho}_i\}$, for distributions in $\mathcal{S}$. For a distribution $\delta \in \mathcal{C} \cap \mathcal{S}$, we define time to be the maximum of the values obtained using the two definitions above, in essence pessimistically stressing the difficulties arising from possible interdependence of processor actions.

Fact 2.1. For a distribution in $\mathcal{S}$, the expected maximum number of steps taken by any of p processors in an expected t transitions is in $O(t\sigma^*)$, for $t \in \Omega((\log p)/\sigma^*)$.

Fact 2.2. For a distribution in $\mathcal{C}$, the expected maximum number of steps taken by any of the p processors in a linearization of expected length t is in $O(t/p)$, provided that $t \in \Omega(p \log p)$.

Fact 2.3. For a distribution in $\mathcal{S}$ or $\mathcal{C}$, if the expected number of steps taken by a processor is at least E, then the actual number of steps taken by that processor is at least $E/2$ with probability at least $1 - e^{-E/8}$.

THEOREM 2.4. *If an asynchronous algorithm A can be decomposed into A' and A" such that the preconditions for A" are a subset of the postconditions for A', then the time needed to execute A is at most the sum of the times needed to execute A' and A".*

THEOREM 2.5. *Synchronization of p processors can be*

performed in $O(\log p)$ time for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cap \mathscr{S}$. Synchronization of p processors requires $\Omega(\log p)$ time for any distribution δ' on linearizations such that $\delta' \in \mathscr{C} \cup \{s \in \mathscr{S} \mid \sigma^* < k/\ln p$ for a constant $k < 1\}$.

As a consequence of Theorem 2.5, we obtain the following relation between synchronous and asynchronous complexity classes by inserting synchronization barriers after steps in PRAM algorithms.

COROLLARY 2.6. *Any problem that can be solved in t time on an ARBITRARY CRCW PRAM using p processors can be solved asynchronously in time $O(t(p/p' + \log p'))$ using p' processors for any distribution δ such that $\delta \in \mathscr{C} \cup \mathscr{S}$, and for $1 \leqslant p' \leqslant p$.*

Since we will be discussing various classes of PRAM algorithms in the course of this paper, we introduce notation to simplify our discussion. The class of problems that can be solved by an ARBITRARY CRCW PRAM in t time using p processors is called ARBITRARY(t, p) with analogous notation used for other PRAM variants.

3. TRANSFORMATIONS OF SYNCHRONOUS INTO ASYNCHRONOUS ALGORITHMS

In order to classify problems into asynchronous complexity classes, it would be helpful to have general methods for determining time bounds for problems. To be able to take advantage of extensive work done on synchronous models, we would like to be able to transform synchronous algorithms into asynchronous ones in a straightforward manner. In the following development, we represent such a transformation which preserves both the number of processors and the asymptotic time bounds for certain classes of synchronous algorithms. Our technique involves two steps: first, we present a synchronous parallel algorithm as a set of directed acyclic graphs (DAGs) and then we develop asynchronous algorithms from the DAGs.

3.1. Representation of Synchronous Algorithms

3.1.1. *A General Representation of Parallel Algorithms*

The computation of values in the course of a synchronous algorithm on a particular input can be seen as the assignment of values to nodes in a graph, where the graph represents the dependencies between the values. For a fixed algorithm, there may be a different graph for each possible input; moreover, there may be more than one possible method of representing a particular algorithm on a particular input as a graph. In any such graph, there will be one input node for each input variable and one output node for each output variable. In these terms, there are two aspects to determining the output of an algorithm: determining the structure of the graph and determining the values of the nodes in the graph.

To clarify our discussion of these aspects, we refer to the underlying graph as the *DAG structure*; when all values have been determined, the graph is a *completed DAG*. Where unambiguous, the term DAG will be used. More formally, we define a DAG structure as follows.

DEFINITION 3.1. A *DAG structure* is a directed acyclic graph such that each node has a distinct *name*, for the purposes of easy description, and the unique node with name u in a DAG will be called *node u*. Nodes in different DAGs may have the same name. In addition, each node has associated with it a function or a set of functions.

We classify each neighbor of a node u as either an *in-neighbor* or an *out-neighbor*. The former has an edge directed from it towards u; the latter has an edge directed towards it from u.

To form a completed DAG from a DAG structure, it is necessary to assign a value to each node, where the process of determining the value of a node will be called the process of *evaluating* a node. The evaluation of a node u depends on the prior evaluation of a certain set of nodes, including a subset of the in-neighbors of u. It is possible that the prior evaluation of any of several subsets of in-neighbors of u will suffice.

DEFINITION 3.2. Let S be a subset of in-neighbors of u such that the evaluation of each node in S is sufficient for the evaluation of u to proceed. Then S is called a *dependency set*, and f_u is the maximum size of any dependency set of u.

In the underlying DAG structure, a node contains one function for each dependency set, where the function for a particular dependency set has a number of inputs equal in cardinality to the size of the dependency set. Once each node in the dependency set of a node u has been evaluated, then u can be evaluated using its function with the values of the nodes in the dependency set as inputs. Once all the nodes have been evaluated, the completed DAG has been formed.

A set of completed DAGs differs from other representations of algorithms, such as data-flow graphs, in that there may be a different DAG for each possible input and in that there may not be a complete separation between evaluation of nodes in the DAGs and the processors that execute the corresponding tasks in the algorithm they represent. In a DAG that represents a PRAM algorithm, for example, a node may correspond to the contents of a memory cell or to the state of a particular processor. Alternatively, since a processor's program may entail reading different memory cells for different inputs, differences in a processor's internal state may be exhibited in differences in the underlying DAG structures for the same algorithm.

We outline one possible method of representing a t-step PRAM computation as a set of completed DAGs, each of height at most t. In this representation, the values of the nodes correspond to the processor states and the memory cell contents at different steps during the computation, and the edges correspond to the reads and writes performed. For each step T and for each processor P, there exists a node (P, T) which contains the state of processor P at the end of time T. For each step T and for each memory cell M, there exists a node (M, T) which contains the value of memory cell M at the end of time T. Suppose that at time T processor P reads memory cell M. Since P's state at time T is completely determined by its state at time $T-1$ and the contents of memory cell M, there are edges into (P, T) from $(P, T-1)$ and from $(M, T-1)$. The inputs to (M, T) are $(M, T-1)$ and the nodes (P, T) such that P successfully writes to M at time T. For a PRIORITY CRCW PRAM, only the edge from the processor of highest priority will be recorded in the DAG; unsuccessful writes are not represented. Similarly, for an ARBITRARY CRCW PRAM only the single edge from the successful writer would be represented.

It is not difficult to see that algorithms on certain other parallel models can be represented in this manner. For a particular algorithm on a comparator network, the completed DAGs for the various inputs may differ, but the DAG structures are all the same. Here the nodes correspond to the gates, and the edges correspond to wires connecting the gates.

3.1.2. *A Restricted Class of Representations*

We know that we can represent any synchronous PRAM algorithm as a set of DAGs. If we could use any such set of DAGs to create an asynchronous algorithm that uses at most a constant factor more resources than the original PRAM algorithm, we would have a general method for transforming fast synchronous algorithms into fast asynchronous ones. However, since Theorem 2.5 establishes a lower bound of $\Omega(\log p)$ on synchronization problems using p processors that are solvable in $O(1)$ time with synchronous PRAMs, we know that no such general method is possible.

To obtain a possibly resource-inefficient asynchronous algorithm from a supplied DAG representation, each asynchronous processor is assigned the task of creating a part of the completed DAG. When all the processors have completed their programs, the completed DAG will have been created in its entirely. In particular, the value of each output node, and hence the output to the algorithm, will have been determined.

Recall that there may be a different DAG associated with each input. Without knowing the entire input, an asynchronous processor must be able to create part of the correct completed DAG for that input. In essence, the processor must be able to first determine a portion of the DAG structure and then to be able to evaluate nodes in that portion of the structure. In particular, the evaluation of any node requires the prior evaluation of a dependency set of that node. It is the difficulty of identifying and reading the nodes in a dependency set that can slow down the resulting asynchronous algorithm. In the case of the representation given in Section 3.1.1, consider the simultaneous write by many processors into a single memory cell. For a processor to evaluate the node corresponding to the memory cell, it would have to know the values of all nodes in the dependency set. In particular, it would need to know which processor was going to successfully write to the memory cell. In the worst case this could require knowing the states of all p processors. Determining the dependency set could require linear time, resulting in a very slow algorithm.

To guarantee fast asynchronous algorithms from DAG representations, we restrict the set of DAG representations by imposing certain conditions. Although the class under consideration is a restriction on DAGs derived from PRAM algorithms, it is not a restriction *to* DAGs derived from PRAM algorithms, as algorithms on other parallel models may also be represented in this fashion.

In the following definition of a restricted class of DAGs, we specify how nodes of the DAG can be included in directed, not necessarily disjoint paths called *lines*. In addition, we will consider paths from *sources* to *sinks*, or *source–sink paths*, where a *sink* is an output node, and a *source* is an interior node that has no interior nodes as in-neighbors. In particular, input nodes are not sources. There may be more than one path from a particular source to a particular sink. For technical reasons, we will be concerned with the number of different ways a source–sink path can be labeled. The inclusion of nodes in lines indicates which processors are capable of evaluating which nodes; the labeling of a source–sink path identifies a particular execution in which a specific processor is responsible for the evaluation.

DEFINITION 3.3. A *three-way labeling* of a source–sink path π in a DAG is a labeling of the nodes in π from the set of labels $\{D, I, O\}$ (representing the default processor, the processor that evaluates the in-neighbor, and the processor that evaluates the out-neighbor) according to the following rules:

1. Any node can be labeled D; a node contained in a single line can only be labeled D.

2. A node contained in more than one line can be labeled I if it shares a line with its in-neighbor in π, and it can be labeled O if it shares a line with its out-neighbor in π.

Each of several conditions in the following definition is followed by an intuitive explanation relating the condition

to PRAM algorithms. Conditions 1 through 4 below are satisfied by the representations of PRAM algorithms given in Section 3.1.1, for $n = p$. By increasing t by a factor of at most 2 it is possible to transform these representations into representations satisfying condition 5 (where the nature of the transformation is given in the intuitive explanation of the condition below). The inclusion of condition 6 rules out slow evaluation of nodes, such as in the example of the linear-size dependency set given earlier in this section. Condition 7 is a technical condition needed to guarantee fast evaluation.

DEFINITION 3.4. The class RDAG(t, n) is the set of algorithms which have representations as sets of completed DAGs that satisfy the conditions below:

1. For each completed DAG, the names of the leaves are the names of the inputs and the values of the leaves are the values of the inputs.

2. Each DAG contains a sequence of n lines, labeled by the processor IDs $1, 2, \ldots, n$. (Intuitively, each line corresponds to the execution of a single process. These labels are distinct from the names given to nodes as well as from the labels associated with three-way labelings.)

3. For each i, the first node in the ith line is the same for the DAG structures for all inputs of a given size. (The first step of each processes is independent of the input.)

4. The height of each DAG is no greater than t. (In a PRAM, t is the number of time-steps needed to execute the algorithm.)

5. Each interior node in each DAG is contained in at least one line; no leaves are contained in lines. We define lines such that if an interior node is contained in more than one line, then each in-neighbor and each out-neighbor of that node is contained in only a single line. (In order that all nodes be evaluated, each node is in at least one line; since the values of the inputs are assumed to exist in shared memory, leaves need not be evaluated. Given a DAG D and a set of lines that contains all interior nodes, it is not difficult to create a DAG D' such that the height of D' is no more than twice the height of D, D and D' compute the same outputs given the same inputs, and D' satisfies the second part of condition 5. The DAG D' can be formed from D by processing in the following way each interior node u contained in more than one line and having an out-neighbor v that is contained in more than one line: For each line containing both u and v, create and add to the line a new node with the single in-neighbor u, the single out-neighbor v, and the identity function. If no line contains both u and v, create a new node, choose an arbitrary line containing u and connect them together in the following way: the new node is inserted between u and its out-neighbor in the line and given v as another out-neighbor. In total, the new node will have one in-neighbor, namely u, two out-neighbors, namely v and u's former out-neighbor in the line, and the identity function. If this operation was executed previously for u and another out-neighbor, say, creating a new node w because u was contained in more than one line and an out-neighbor v' was contained in more than one line, then instead of creating another node for u and v, we simply add an edge from w to v.)

6. If processor P has evaluated the first j nodes in a line for a particular input, then in constant time, P can determine which node u is the $(j+1)$th node in that line, determine a dependency set of u, read each node in the dependency set, and, if the nodes in the dependency set have been evaluated, evaluate u. (Although this condition is concerned with the creation of the completed DAG from the DAG structure, it imposes a restriction on the set of DAG structures for a particular algorithm. Namely, it requires that there be an easily computable relationship between the different DAG structures in the set and the set of lines: given the values of the first j nodes in a line, a process can determine in constant time what node to next evaluate. In the context of a PRAM algorithm, this is roughly equivalent to saying that after executing any number of steps in its program, a processor can determine which step to execute next and can execute that step in constant time. Determining which step to execute next can be done in constant time in any PRAM algorithm given that the previous steps have been executed (in the case of an algorithm like pointer jumping, at the outset a processor will not know which cell to read in the jth step, but after the $(j-1)$th step has been completed, which cell to read will be evident). However, a step that can be executed in constant time on a PRAM does not necessarily translate into a step with a constant-size dependency set, as in the case of reading of a cell written simultaneously by a linear number of processors, a constant time step in a synchronous algorithm which translates into a dependency set of linear size in the DAG representation.

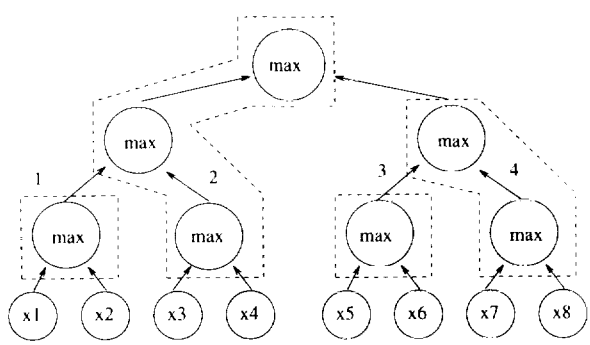

FIG. 1. A DAG representation of a MAX algorithm.

7. For each DAG structure, the total number of *three-way labelings* of all source–sink paths is at most $n^{O(1)}$.

To get an idea of how condition 7 might be satisfied, we consider one possible scenario. If the height of a DAG is in $O(\log n)$ and the fan-in to any node is bounded by a constant, then the number of source–sink paths will be in $n^{O(1)}$. In the three-way labelings of any particular source–sink path, each node can be labeled in at most three ways, yielding a total of $O(n)$ three-way labelings for that path, and hence there are a total of at most $n^{O(1)}$ three-way labelings for all source–sink paths.

Figures 1 and 2 illustrate two possible representations of a maximum finding algorithm as DAGs and lines, labeled by IDs. For each node, the single dependency set is the set of all in-neighbors. Each node labeled "max" can be evaluated by determining the maximum of the values of the nodes in its dependency set. Each unlabeled node can be evaluated by determining the value of its single in-neighbor. Recall that we wish to preserve the property that if an interior node is contained in more than one line, then each in-neighbor and each out-neighbor of that node is contained in only a single line, as stated in condition 5 in the definition of RDAG(t, n). This property is satisfied trivially by Fig. 1, since each node labeled "max" is contained in a single line. In Fig. 2, the existence of the unlabeled nodes ensures that the property holds. Figure 3 illustrates a DAG that is identical to Fig. 1 in which lines are chosen by assigning nodes labeled "max" as in Fig, 2; the DAG in this figure is not in RDAG(t, n), since Condition 5 is violated.

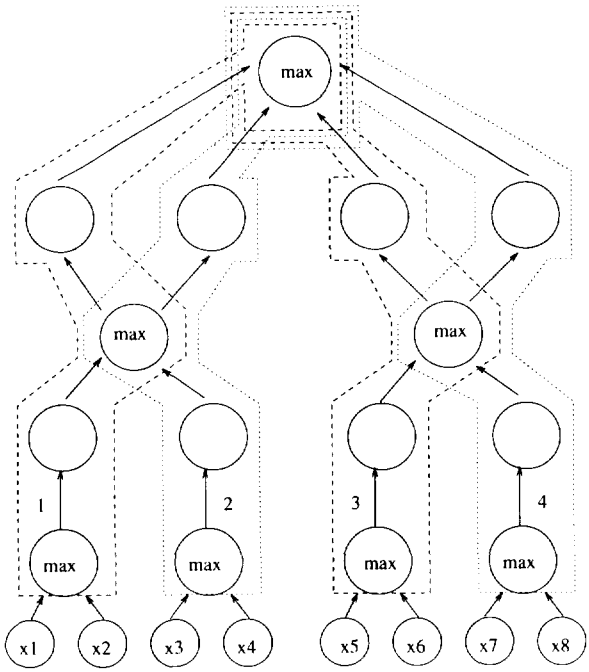

FIG. 2. DAG with "max" nodes in multiple lines.

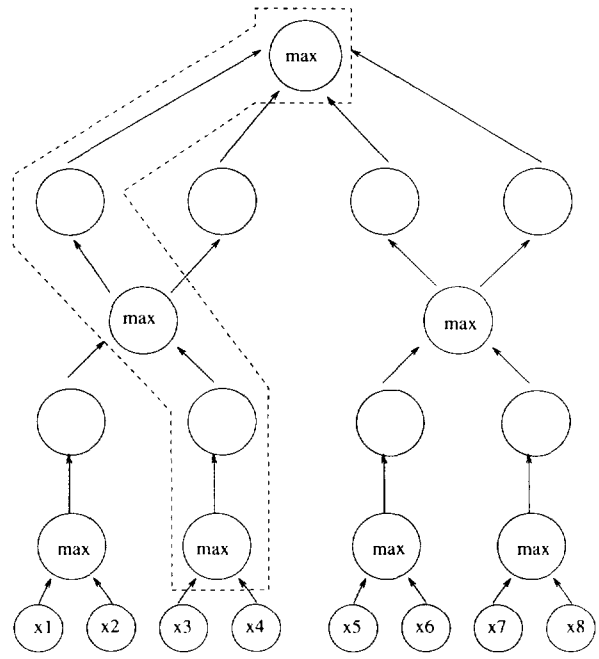

FIG. 3. Condition violated.

In Fig. 2, each source–sink path consists of five nodes; all possible three-way labelings for the path marked in Fig. 4 are DDDDD, DDIDD, DDODD, DDDDI, DDIDI, and DDODI. Figures 2 and 4 together illustrate the fact that not all source–sink paths are lines.

In Section 3.2, we will show that algorithms in RDAG(t, n) can be transformed into asynchronous ones. Then, in Section 3.3, we will show that RDAG(t, n) contains a number of natural classes of synchronous algorithms. The following theorem shows that the class is also contained in a natural synchronous class.

THEOREM 3.5. RDAG(t, n) $\subseteq$ COMMON($O(t), n$).

Proof. To simulate an algorithm in RDAG(t, n) on an n-processor COMMON CRCW PRAM in $O(t)$ time, we make

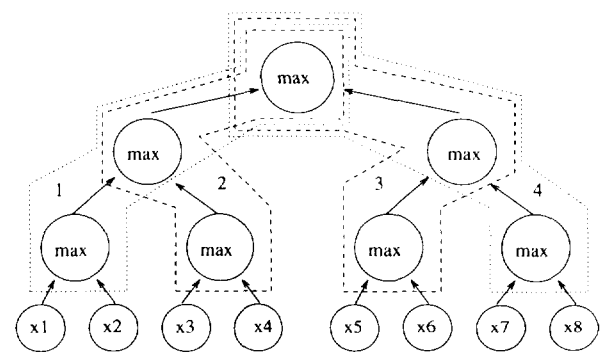

FIG. 4. A marked source–sink path.

use of the DAG representation. We assign one memory cell to each possible node name and one processor to each line. Initially, the *current node* of a processor in a line is the first node in its line. Subsequently, the node u is the current node in a processor P's line if P has not read u and one of the following is true: either P has read u's in-neighbor in the line, or u's in-neighbor in the line has been evaluated by another processor and then read by P. A simulating processor will follow the line in the direction from sources to sinks, reading the current node in the line to see if it has been evaluated, and if not, then repeatedly reading the in-neighbors of the current node in the line until they have been evaluated, evaluating the current node, and then moving to the next node on the line. For each node, the single dependency set is equal to the set of in-neighbors.

Let c be an upper bound on the time needed to read the values of the in-neighbors of a node plus the time needed to evaluate the node. It is easy to show by induction that all nodes of height $h' \leq h$ have been evaluated at the end of ch steps. Since no DAG has height greater than t, all nodes have been evaluated after $O(t)$ steps. ∎

3.2. Asynchronous Simulation Using Representations

In this section we will prove that algorithms in RDAG(t, n) can be simulated asynchronously, where the running time is a function of t and n. The theorem below is adapted from a result by Luby [24]; the relevance of Luby's result to asynchronous parallel computation is introduced in the work of Martel, Park, and Subramonian [25], where it is applied to a different model. Corollaries applying the theorem to particular problems and synchronous models follow.

THEOREM 3.6. *If a problem can be solved by an algorithm in* RDAG(t, n), *then the problem can be solved in time* $O(t + \log n)$ *using* n *processors for any distribution* δ *such that* $\delta \in \mathscr{C} \cup \mathscr{S}$.

Proof. We present a proof for $\delta \in \mathscr{C}$; the proof for $\delta \in \mathscr{S}$ is quite similar and, hence, is omitted. For convenience, let $h = t + \log n$. Furthermore, let r be a constant such that $p_i \geq r/n$ for all i. We will achieve the $O(h)$ time bound by showing that the expected total number of steps taken by all processors to evaluate the DAG is in $O(nh)$. We will overcount the steps needed, by assuming that all processors are working actively on the problem until the DAG is completely evaluated. We then apply Fact 2.2 to conclude that the expected maximum number of steps per processor is in $O(h)$, as claimed.

First, we describe a simple way of transforming an algorithm in RDAG(t, n) into an asynchronous algorithm. We allocate one memory cell to each potential node name and assign one simulating processor to each line. When node l is evaluated, we write the value into the memory cell for the name l. In the remainder of this discussion, we will refer to the act of writing a value into the memory cell for l as *writing node* l and to the act of reading a value from the memory cell for l as *reading node* l, where the value may have been written by a processor or may be a special blank cell symbol. A cell that contains the blank cell symbol is said to be *empty*. Note that, although more than one processor may write a cell, the write of only one processor will transform a particular cell from being empty to containing the node's value. To evaluate node u, the processor first reads u to see if it has been evaluated. If u has not been evaluated, the processor repeatedly reads the nodes in a particular dependency set until it detects that all have been evaluated. Once all the nodes in the dependency set have been evaluated, the processor computes u's value and writes u. If the processor has to read any of the nodes in the dependency set more than once, we will say that the processor is *busy-waiting*. Now we can simply describe a simulating processor's algorithm: starting at the first node in its line, it evaluates a node in its line and then moves on to the next node in the line, resorting to busy-waiting wherever needed, until all the nodes in its line have been evaluated.

We will show in Lemma 3.9 that there exists a three-way labeling of a source–sink path such that the time needed to evaluate the DAG is bounded above by the time needed to evaluate that path as if it were independent of the DAG. Since in general we cannot predict which linearization will occur, we cannot identify which labeling of which source–sink path will have this property. However, by showing that it is unlikely that any labeling yields a large evaluation time, we will be able to conclude that the time to evaluate the DAG is in $O(t + \log n)$, as claimed.

In order to prove our lemma, we will make use of a translation of three-way labelings of source–sink paths to sequences of processor steps, or *path sequences*. Intuitively, a path sequence for a source–sink path π is a sequence of steps such that if each node in π can be evaluated without any busy waiting, then when the path sequence is complete, π will have been completely evaluated. A more formal definition of a path sequence follows.

DEFINITION 3.7. A *path sequence* can be formed from a three-way labeling of a source–sink path π by replacing each label u by $f_u + 2$ copies of processor ID P, where P is chosen according to the rules below, and then appending as many copies of the lowest numbered ID as are needed to yield a total of $(f_{\max} + 2)\,h$ labels. Recall that if an interior node is contained in more than one line, then each in-neighbor and each out-neighbor of that node is contained in only a single line.

1. If u is contained in a single line, then P is the ID of the line containing u.
2. If u is contained in more than one line, then

(a) if u is labeled I, P is the ID of the line containing u's in-neighbor in π;

(b) if u is labeled O, P is the ID of the line containing u's out-neighbor in π; and

(c) if u is labeled D, P is the ID of the lowest numbered processor whose line contains u and whose probability of taking a step is no greater than the probability for any other processor whose line contains u.

In Fig. 4, the three-way labelings of the marked path, DDDDD, DDIDD, DDIDI, and DDDDI, are translated into the following path sequences, assuming that processor 2 is the most likely to take the next step, processor 1 the next most likely, processor 3 the next most likely, and processor 4 the least likely. In each path sequence, the parentheses have been added to indicate the grouping of IDs associated with one node; they are not part of the actual path sequences:

(2222)(222)(1111)(111)(444444),

(2222)(222)(2222)(111)(444444),

(2222)(222)(2222)(111)(111111),

(2222)(222)(1111)(111)(111111).

DEFINITION 3.8. A finite prefix of a linearization *covers a path sequence* λ if it contains λ as a subsequence.

In our example above, the path sequence (2222)(222)(2222)(111)(444444) is covered by the linearization prefix

1342241422443232113242112232223112431324132444324323 34.

LEMMA 3.9. *Given an algorithm in the class* RDAG(t, n) *and a fixed input, consider a fixed linearization. Let* α *be the number of steps in the linearization needed to complete the algorithm. For a source-sink path* π *in the* DAG *associated with the input, there is a method for choosing a distinct three-way labeling of* π. *Let* β_π *be the number of steps in the linearization needed to cover the path sequence formed from that three-way labeling of* π. *Let* β *be the maximum of* β_π *taken over all source–sink paths. Then* $\alpha \leqslant \beta$.

Proof. We will show that there is a source–sink path μ such that any prefix of the linearization that is sufficiently long to evaluate μ is also sufficient to evaluate the entire DAG (that is, once all nodes in μ have been evaluated, the DAG will have been evaluated). Then, we give a procedure for selecting a distinct three-way labeling of μ. Finally, we will complete the lemma by showing that the minimal prefix that covers the path sequence formed from the distinct three-way labeling of μ is sufficient to evaluate μ. In this proof, we ignore any copies of the lowest numbered ID that are added after the copies generated by the three-way labeling; these only increase the length of the path sequence and, hence, the prefix of the linearization that covers the path sequence of μ.

We construct a path μ, working backwards from the sinks to the sources to create its reverse, μ'. At the same time we choose a particular three-way labeling of μ. Intuitively, we can think of μ as being the path of maximum delay. We first choose the sink that was evaluated after all the other sinks. In the case where there is a single dependency set per node, it would seem that we could identify the path of maximum delay by selecting as the next node in μ' the in-neighbor of the sink that was evaluated after all the other in-neighbors in the dependency set and by repeating this process until we reach a source. However, there are in fact two possible causes of delay. Suppose that the sink is node z, the processor that evaluates it is P_i, and the in-neighbor of z in P_i's line is the node z'. To evaluate z, P_i must first evaluate z' and then read the values of all other in-neighbors of z. Suppose that P_i has evaluated z' and read all other in-neighbors of z, but has detected that certain in-neighbors have not been evaluated. In this case, the in-neighbor that was evaluated last can be seen as the cause of delay by forcing P_i to busy-wait. Now, suppose, instead, that before z becomes the current node of P_i, all the in-neighbors of z have been evaluated. In particular, z' has been evaluated by a processor other than P_i. In this case, which in-neighbor was evaluated last is unimportant, since P_i does not have to busy-wait: the real cause of delay was the time taken for P_i to evaluate z' and to reach z as its current node.

In order to capture both notions of delay, we select nodes in μ' according to a procedure more complicated than the one initially suggested. To be able to distinguish between the two cases and to help us in the choice of a three-way labeling, we associate with each node one of the processors that evaluates it. The sequence of nodes chosen will form μ' and the sequence of associated processors will be used to choose the three-way labeling of μ. The first node in μ' is the sink that was first evaluated after all other sinks. Its associated processor is the processor that first evaluated that sink. We then choose the next node and the associated processor by repeating the following procedure until we reach a source. Suppose that u is the last node that was chosen, P is its associated processor, and P evaluates u using the dependency set $\mathscr{D}$. One of the following two rules applies.

Rule 1. Suppose that node v is the previous node in P's line and that all other nodes in $\mathscr{D}$ have been evaluated before P first reads them. Then v is added to μ' and P is the processor associated with v.

Rule 2. Suppose that of the nodes in $\mathscr{D}$, w is the last one that P detects to be evaluated and that either node u is the first node in P's line, or node v is the previous node in P's line and $w \neq v$. Then w is added to μ' and the processor that first evaluates w is the processor associated with w.

In either case we have chosen an in-neighbor of u as the next node in μ'. Moreover, because of the way μ' has been created, any prefix of the given linearization that is sufficiently long to evaluate μ is also sufficient to evaluate the entire DAG: once all the nodes in μ have been evaluated, by necessity all the nodes in the DAG will have been evaluated.

We can now use the associated processors to choose a three-way labeling. For convenience, we will use $P(u)$ to denote the processor associated with u, u_i to denote the in-neighbor of u in μ, and u_o to denote the out-neighbor of u in μ. If u is contained in a single line, then u is labeled D. Otherwise, u is labeled I if $P(u) = P(u_i)$, labeled O if $P(u) = P(u_o)$ and differs from $P(u_i)$, and labeled D otherwise. From the definitions of three-way labelings and path sequences, it is not difficult to see that the path sequence derived from this three-way labeling takes at least as long to evaluate as a path sequence formed from concatenating in order the appropriate numbers of IDs of associated processors. In particular, the only case in which the associated processor might differ from the ID chosen in the path sequence is when w is labeled D and contained in more than one line; this can occur when w is added to μ' by Rule 2 and the processor that first evaluates w, $P(w)$, is not the lowest numbered processor whose line contains w and whose probability of taking a step is not greater than the probability for any other processor whose line contains w. In this case, the number of steps of the linearization needed to allot the appropriate number of steps to this lowest numbered processor is an upper bound on the number of steps needed to allot the appropriate number of steps to $P(w)$.

To complete the proof, we need to show that the time needed to evaluate μ in the DAG is at most the time needed to cover the chosen path sequence of μ. It will suffice to prove the following claim. Suppose that x and y are any two consecutive nodes in μ, where y is an in-neighbor of x, and consider the point in the linearization at which y has been evaluated by the processor labeling y. Then x will have been evaluated when the processor labeling x has taken at most $f_x + 2$ further steps. We consider two different cases, corresponding to the two rules.

Suppose that y was selected for μ by Rule 1. In this case, the processor labeling y is the same processor as the processor labeling x; we call it P. When P has evaluated y, it reads x to see if it has been evaluated; then it reads the rest of the nodes in a particular dependency set and, when they have all been evaluated, it evaluates x. By Rule 1 we know that each of the nodes in the dependency set need only be read once, and, hence, in at most $f_x + 2$ steps P will have evaluated x.

Now suppose, instead, that y was selected for μ by Rule 2. Again, let P be the processor labeling x. In this case, we know that y was the last node in the dependency set for x that P detected to have been evaluated. If node x is the first node in P's line, then once y has been evaluated, in $f_x + 2$ steps P can read y, read each of the nodes in the dependency set, and then write x. If, on the other hand, P contains in its line a node $z \neq y$, we know by the definition of y that P evaluates z before it detects y to be evaluated. Thus, from the point in the linearization at which y has been evaluated, x is the current node of P and all the nodes in the dependency set have been evaluated; hence, in $f_x + 2$ steps x can be evaluated. In this case we are not concerned with the identity of the processor labeling y. It is either the processor labeling the in-neighbor of y in μ, if the in-neighbor was chosen by Rule 1, or it is labeled by the lowest numbered processor whose line contains y and whose probability of taking a step is no greater than this probability for any other processor whose line contains y. In either case, the number of steps required for the processor to evaluate y will be at least as great as the number required for the first evaluator of y to evaluate y. This completes the proof of the lemma. ∎

The proof above depends heavily on the fact that lines are directed paths. To see this, suppose instead that a line contains node u, node v, and then an out-neighbor of u, in that order; further suppose that u and u's out-neighbor are both on the path of maximum delay. Consider the allotment of steps after u has been written. Since v must be written before u's out-neighbor can be written, the counting of steps as in the lemma is no longer valid.

Our lemma gives us a bound when a linearization is fixed; we now use this lemma to determine the expected total number of steps taken to complete the algorithm. Let a be the smallest integer such that the number of three-way labelings of source–sink paths is at most n^{a-1}. The number of three-way labelings is an upper bound on the number of possible path sequences; by Lemma 3.9, we know that for any linearization, the path of maximum delay is represented by one of the path sequences.

We will consider the steps in blocks of size $6B$, where $B = f_{\max} han$, where $f_{\max}$ is the maximum over all the nodes u of f_u and h is the height of the DAG.

We use $\mathcal{T}$ to denote the total number of steps to complete the entire algorithm, and $\mathcal{T}_\lambda$ to denote the total number of steps to cover a path sequence λ. Since a linearization which covers all path sequences in particular covers the path sequence for the path of maximum delay, $\mathcal{T} \leqslant \max_\lambda \mathcal{T}_\lambda$,

$$E[\mathcal{T}] \leqslant \sum_{i=0}^{\infty} Pr[\mathcal{T} > i]$$

$$\leqslant \sum_{j=0}^{\infty} 6B \cdot Pr[\mathcal{T} > 6Bj]. \qquad (3.6.1)$$

Since all the path sequences are of the same length, we can obtain a simple upper bound by determining the probability that an arbitrary "worst-case" path sequence $\hat{\lambda}$ takes a long

time to cover, and multiplying it by the number of path sequences. For a worst case path sequence, we assume for each node in the sequence that the probability of a step being allotted to the processor evaluating the node is r/n,

$$Pr[\mathscr{T} > 6Bj] \leqslant Pr[\max_{\lambda} \mathscr{T}_{\lambda} > 6Bj]$$

$$\leqslant n^{a-1} Pr[\mathscr{T}_{\hat{\lambda}} > 6Bj]. \qquad (3.6.2)$$

Let $L_{\hat{\lambda}}(t)$ be the length of the longest prefix of $\hat{\lambda}$ covered in the first t steps. Since the length of $\hat{\lambda}$ is $(f_{\max}+2)h$, the probability in Eq. (3.6.2) can be rewritten as

$$Pr[\mathscr{T}_{\hat{\lambda}} > 6Bj] = Pr[L_{\hat{\lambda}}(6Bj) < (f_{\max}+2)h]. \qquad (3.6.3)$$

By combining Eqs. (3.6.1) through (3.6.3) and performing some elementary manipulations, we obtain

$$E[\mathscr{T}] \leqslant 6B\left(1 + n^{a-1} \sum_{j=1}^{\infty} Pr[L_{\hat{\lambda}}(6Bj) < (f_{\max}+2)h]\right). \qquad (3.6.4)$$

In order to be able to prove that the expected number of steps is in $O(B) = O(nh)$, it will suffice to show that the summation in Eq. (3.6.4) has a value of at most n^{-a+1}. In order to do this, we make use of Raghavan's variant of the Chernoff bound [30], which states that if Y is the number of successes in a sequence of Bernoulli trials, E is the expected number of successes, and $0 \leqslant \beta \leqslant 1$, then $Pr[Y \leqslant (1-\beta)E] \leqslant e^{-\beta^2 E/2}$. We view each allotment of a step as a Bernoulli trial. A trial succeeds if the length of the prefix of $\hat{\lambda}$ covered is increased by one as a result of the trial; the probability of success is r/n. In our situation $E = E[L_{\hat{\lambda}}(6Bj)] \leqslant 6Bjr/n$, and $\beta = 1/2$. From this point in the proof we consider only constant speed distributions; the proof is identical in the other case. By applying Raghavan's variant of the Chernoff bound, we obtain

$$Pr[L_{\hat{\lambda}}(6Bj) < 3Bjr/n] < e^{-6Bjr/8n}. \qquad (3.6.5)$$

Note that $(f_{\max}+2)h \leqslant 3Bjr/n$ and, hence,

$$Pr[L_{\hat{\lambda}}(6Bj) < (f_{\max}+2)h] < e^{-6Bjr/8n}. \qquad (3.6.6)$$

We can now combine Eqs. (3.6.4) and (3.6.6) as

$$E[\mathscr{T}] \leqslant 6B\left(1 + n^{a-1} \sum_{j=1}^{\infty} e^{-6Bjr/8n}\right). \qquad (3.6.7)$$

To complete the proof, we show that $\sum_{j=1}^{\infty} e^{-6Bjr/8n} = \sum_{j=1}^{\infty} e^{-6f_{\max}hajr/8}$ is at most n^{-a+1}. Since $h \geqslant \log n$,

$$\sum_{j=1}^{\infty} e^{-6f_{\max}hajr/8} \leqslant \frac{n^{-6f_{\max}ar\log e/8}}{1 - n^{-6f_{\max}ar\log e/8}}. \qquad (3.6.8)$$

We observe that the numerator is less than n^{-a} and that the denominator is less than $1/2$ for all values of $n \geqslant 2$. Combining this equation with Eq. (3.6.7), we obtain an upper bound of $18B \in O(nh)$ on the expected total number of steps, as claimed. ∎

3.3. Easy-to-Transform Algorithms

In this section, we consider the transformations of various classes of algorithms into DAG representations, enabling us to apply Theorem 3.6 to obtain asynchronous algorithms. First, we generalize the result illustrated through figures of the maximum finding algorithm on the PRAM. Then, we consider a few specific algorithms designed for other models of parallel computation, such as the comparator network and the butterfly graph. Finally, we consider classes of PRAM algorithms for which such transformations are possible.

We first consider the DAG representation of the iterated application of any associative binary function to n inputs, of which one example is maximum finding. As in Figs. 1 through 4, we fix a balanced binary tree of height $\log n$, with special unlabeled nodes added as needed. The DAG structure for any input will be a variant on such a tree. The values of the n leaves will be the values of the n inputs; the value of an interior node will be the value of the function on the subset of inputs that are descendants of the node. The value of a node at height i can be computed from the values of its two children.

It is not difficult to see that such an algorithm is in RDAG($O(\log n)$, n). For each covering of the nodes by lines we generate a different asynchronous algorithm; Figure 1 illustrates a covering (for a maximum-finding algorithm of the synchronization paradigm) in which each node appears in exactly one line. It follows from Theorem 3.6 that any asynchronous algorithm generated in this way can be executed in $O(\log n)$ time using n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$.

Since we wish to find DAG representations of algorithms, a natural starting point is a set of algorithms designed on a model that itself can be viewed as a DAG. For example, the underlying graph of a comparator network [20] can be viewed as a DAG. An n-input comparator network can be drawn as a set of n horizontal data lines, where a comparator is drawn as a vertical connection between the data lines holding the values being compared, the larger number appearing in the lower line after passing the comparator. Such a representation is given in Fig. 5; here we use the ter-

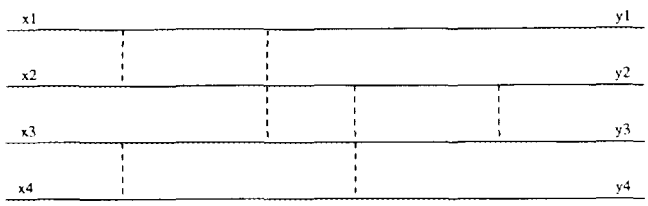

FIG. 5. A comparator network.

minology in Knuth's work [20], where the representation we use appears as Fig. 44. Each comparator in the network can be represented as a node with two inputs and two ordered outputs. The value of a node is an ordered pair $(\max\{x, y\}, \min\{x, y\})$, where x and y are the values of the in-neighbors of the node. We can define the n lines to consist of the nodes in the n horizontal data lines. Figure 6 illustrates the DAG representation of the sorting network illustrated in Fig. 5; the comparator nodes are marked C, the larger values travel on dashed lines, and the smaller values travel on solid lines. The line corresponding to the second horizontal data line is marked by a dotted box. Thus, any problem that can be solved by a comparator network of delay $O(t)$ can be solved in $O(t + \log n)$ time on n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$. In particular, this is the case for the Ajtai, Komlós, and Szemerédi sorting network algorithm [2].

THEOREM 3.10. *Sorting of n inputs can be completed in $O(\log n)$ time using n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$ (with constants a function of the constants used in the AKS algorithm [2]).*

We will later show that these constants can be improved, in Theorem 3.18.

Another model that can be viewed as a DAG is the butterfly [23]. It is not difficult to see that any problem that can be solved on a butterfly with n rows and $\log n$ columns can be solved in $O(\log n)$ time using n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$. There are solutions for the problems of fast Fourier transform and bitonic merge on such graphs [33].

THEOREM 3.11. *The fast Fourier transform can be solved in $O(y \cdot \log n)$ time using n/y processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$ and for $1 \leqslant y \leqslant n$.*

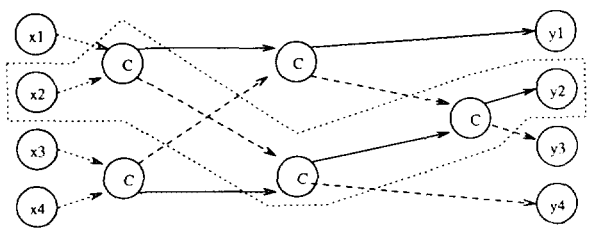

FIG. 6. The DAG representation of a comparator network.

THEOREM 3.12. *Bitonic merge is in $O(y \cdot \log n)$ time using n/y processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$ and for $1 \leqslant y \leqslant n$.*

Of perhaps greater interest is a method for transforming PRAM algorithms into DAGs. Although the iterated application of an associative binary function is easily visualized as a binary tree and, hence, as a set of DAGs, it is not clear how an arbitrary PRAM algorithm could be transformed. In fact, as we observed in Section 3.1.2, we cannot represent every synchronous PRAM algorithm as a set of DAGs. We would like to know for what set of PRAM algorithms this representation is possible. To this end, we introduce the following definitions:

DEFINITION 3.13. A *CRCW PRAM* algorithm is said to be *B-write-bounded* if, for each input, for each step, and for each memory cell, there are at most B processors writing to the cell in that step.

DEFINITION 3.14. A *CRCW PRAM* algorithm is said to be *B-read-bounded* if, for each input, for each step, and for each memory cell, there are at most B reads of the value written to the cell at that step. The reads may occur at any time in the computation.

DEFINITION 3.15. A *PRAM* algorithm is said to be *write-determined* if, for every processor P, every memory cell M, and every step T, if at time T processor P is in a state in which it is to read cell M, it can determine from its local information (before taking the step at time T) the largest $T' \leqslant T$ such that some processor wrote to M in step T'. The computation ends only when the result has been written into a specified answer cell or cells. Moreover, a processor that writes the answer cell must read it, so that all values used are read.

To illustrate the property of write-determination, we consider the constant-time algorithm for determining the OR of n bits on a CRCW PRAM. In the first step, each processor with a 1 writes a 1 into a collection cell. In the second step, all processors read the collection cell, writing a 0 in the answer cell if the collection cell is empty, and writing a 1 otherwise. This algorithm is not write-determined, because the processors reading the collection cell at time 2 do not know the largest $T' \leqslant 2$ such that some processor wrote to the collection cell in step T'. In fact, if a processor knew the time at which the collection cell was last written, it would then know the value of the computation without ever reading the cell.

A special case of write-determination is *obliviousness*, where an algorithm is oblivious if the pattern of accesses of all the processors to global memory is fixed for all inputs. Any problem that can be solved on a p-processor CRCW

PRAM in time t by an oblivious algorithm can be solved on a p-processor write-determined CRCW PRAM with the same form of write-conflict resolution in time t. To see that this is true, we create a write-determined algorithm from the oblivious one. Since the pattern of accesses of all the processors is fixed for all inputs, whenever a memory cell is read, the time at which the cell was last written is unchanging. This information can be written into the algorithm, so that the write-determined restriction is satisfied.

The notions of write-determination and $O(1)$-write-boundedness are sufficient to guarantee transformation into asynchronous algorithms via DAG representations.

THEOREM 3.16. *The following classes of algorithms can be solved in $O(t+\log n)$ time using n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$:*

1. *write-determined $O(1)$-write-bounded* COMMON $(O(t), n)$

2. *write-determined $O(1)$-read-bounded* COMMON $(O(t), n)$.

Proof. We begin by choosing the minimum $r \geqslant 1$ such that $t \leqslant r \cdot \log n$. We now divide the original synchronous algorithm into r subalgorithms each of length at most $\log n$ and insert a synchronization barrier after each one. To conclude that the entire algorithm can be solved in $O(t+\log n)$ time using n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$ by Theorems 3.6 and 2.4, it will suffice to show that each subalgorithm is in RDAG($O(\log n), n$). We now present a representation which, although similar to the representation used in Section 3.1.1, reduces the number of lines and satisfies all the conditions needed for membership in RDAG($O(\log n), n$).

Consider an arbitrary subalgorithm. Let m be the number of cells and let p be the number of processors used by the subalgorithm on a particular input. The DAG consists of at most $O((m+p)\log n)$ nodes. Certain nodes correspond to memory cells; a node of this type is named by an ordered pair (M, T), where M is a memory cell and T is a PRAM time-step at which M is written. The value of (M, T) is exactly the contents of memory cell M at time T. Other nodes correspond to processors; a node of this type is named by a pair (P_i, T), where P_i is a processor and T a PRAM time-step. The value of (P_i, T) is 1; rather than explicitly writing down the processor's state, the cell is used to mark that a particular time-step has been reached.

The ith line consists of one or two nodes for each time step, depending on whether or not P_i writes at a particular time-step, in order from the beginning to the end of the computation. If P_i writes cell M at time T, the ith line will contain (M, T) as the first node for step T and (P_i, T) as the last node for step T, with an edge directed from (M, T) to (P_i, T); otherwise, the ith line will contain (P_i, T) as the first and last nodes for step T. For each time T and each i, we direct an edge from the last node for time T in the ith line to the first node for time T in the ith line.

The Tth node in the ith line represents P_i's state at time T. The state of P_i at time T is a function of P_i's state at time $T-1$, plus any information that P_i reads during time T. Suppose that P_i reads the cell M' during time-step T in the subalgorithm. Since the subalgorithm is write-determined, P_i can determine the time-step at which M' was last written, say time T'. The DAG contains an edge from (M', T') to (P_i, T).

Note that condition 5 is satisfied, since each node of the form (M, T) has as in-neighbors and out-neighbors nodes of the form (P_i, T), each of which appears in exactly one line. The restriction of the algorithm to be $O(1)$-write-bounded ensures constant fan-in, and the restriction of the algorithm to be $O(1)$-read-bounded ensures constant fan-out. In either case, since the height is in $O(\log n)$, the number of source–sink paths is bounded by a polynomial, and hence, each such DAG is in RDAG($O(\log n), n$), as needed. ∎

Any CROW PRAM algorithm [7] in which each processor has a single cell can be made into a write-determined algorithm by having each processor write to its cell at each step. Since a CROW PRAM algorithm is by definition $O(1)$-write-bounded, we obtain the immediate corollary.

COROLLARY 3.17. *Any problem that can be solved in t time on an n-processor CROW PRAM in which there is a single cell per processor can be solved in time $O(t+\log n)$ by n processors for any distribution δ on linearizations such that $\delta \in \mathscr{C} \cup \mathscr{S}$.*

By making certain assumptions, it is possible to simulate an algorithm that runs in t time on a p-processor CROW PRAM in which there is a single cell per processor in $t+c$ time on a p-processor CROW PRAM in which there are c cells per processor. Each processor in the simulating CROW PRAM first reads the values stored in its c cells and then writes the c values, encoded, into a single cell. In each of the subsequent steps, all c values are updated in a single write. A time complexity of t is possible if the values are not initially stored in global memory, since the initial reading stage can be avoided. The assumptions made for the simulation include increasing the number of bits held in a cell by a factor of c, increasing the number of bits written or read in one step by a factor of c, allowing constant time extraction of a particular value from the encoded group of c, and guaranteeing that each processor in the original algorithm knows the owner of each cell it reads.

We note that Cole's parallel mergesort algorithm [4] is nearly oblivious in the way that it proceeds. By making minor modifications to further exploit the regularity of the computation, it is possible to make the algorithm write-determined, and hence, we obtain the following result.

THEOREM 3.18. *Sorting of n inputs can be solved in* $O(\log n)$ *time using n processors for any distribution* δ *on linearizations such that* $\delta \in \mathscr{C} \cup \mathscr{S}$ *(with constants a function of the constants used in Cole's algorithm* [4]).

The sorting result enables us to simulate PRIORITY CRCW PRAM algorithms, as shown below.

THEOREM 3.19. PRIORITY(t, p) *is contained in the class of problems that can be solved in* $O(tp)$ *time using one processor for any distribution* δ *on linearizations.*

THEOREM 3.20. PRIORITY(t, p) *is contained in the class of problems that can be solved in* $\min\{O(tp), O(ty \log p)\}$ *time using* p/y *processors for distribution* δ *on linearizations such that* $\delta \in \mathscr{C} \cup \mathscr{S}$ *and for* $1 \leqslant y \leqslant p$.

Proof. It will suffice to consider the simulation of a single write phase. We adapt a technique used to simulate a PRIORITY CRCW PRAM by an EREW PRAM [8, 34]. We present the algorithm for the case $y = 1$; for other values of y, each processor simulates sequentially in round-robin ordering the steps of a set of processors. For all i, processor P_i writes to an auxiliary structure the triple (m_i, i, v_i), where m_i is the cell that P_i wishes to access and v_i is the value that it wishes to write. The triples can be sorted lexicographically in $O(\log p)$ time, using Theorem 3.18. Next, each processor reads a consecutive pair of triples and writes the value in the second triple into the cell in the second triple only if the cells in the two triples differ. Two processors are assigned to the first pair of triples; one of the processors behaves as outlined above, and the other writes the value in the first triple into the cell in the first triple only if the cells in the two triples differ. Finally, the processors synchronize; by Theorems 2.5 and 2.4, the entire process takes $O(\log p)$ time. ∎

4. CONCLUSIONS AND OPEN QUESTIONS

We have shown how the directed acyclic graph representation of a synchronous algorithm of a certain type can be used to obtain an asynchronous algorithm that has resource requirements within a constant factor of those of the original synchronous algorithm. For the resource bounds to be preserved, the original algorithm must have a running time in $\Omega(\log p)$; the resource-preserving transformation of faster algorithms remains an open question.

The class RDAG(t, p) defines a set of problems that can be solved asynchronously through transformation of synchronous algorithms. The exact relationship of the class RDAG(t, p) to synchronous complexity classes remains unknown. We know that the class RDAG$(O(\log p), p)$ may equal COMMON$(O(\log p), p)$, it may equal WD-COMMON$(O(\log p), p)$, or it may be strictly stronger. In this regard, determining the complexity of connectivity is of particular interest. The best known running times for connectivity algorithms are $O(\log^{3/2} n)$ time on a CREW PRAM [15] and $O(\log n)$ on a PRIORITY CRCW PRAM [31]. Although Cole and Zajicek [5] have developed an asynchronous $O(\log n)$ time connectivity algorithm for their deterministic round complexity model (recently simplified by Grove [10]), it is not clear what the running time of the algorithm might be with a probabilistic complexity measure.

The results in this paper show that inclusion in the class RDAG(t, p) guarantees an efficient asynchronous algorithm; however, they do not preclude the existence of efficient simulations of algorithms outside of the class. It is possible that a more general simulation technique might exist, or general techniques for directly designing asynchronous algorithms that have no synchronous counterparts.

Just as there remain many open questions concerning the relationships between synchronous models, there remain questions concerning the relationships among asynchronous models and between synchronous and asynchronous models. Of particular interest is the placement of the class of problems that can be solved asynchronously in $O(\log p)$ time using p processors. It is known to contain WD-COMMON$(O(\log p), p)$, but it is not clear whether it is strictly contained in COMMON$(O(\log p), p)$, or if it and COMMON$(O(\log p), p)$ are incomparable.

ACKNOWLEDGMENTS

This paper benefited from the comments of anonymous referees, whom I wish to thank for the time and effort involved.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Data Structures and Algorithms," Addision-Wesley, Reading, MA, 1983.
2. M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, *in* "Proceedings, 15th Annual ACM Symposium on the Theory of Computing, 1983," pp. 1–9.
3. J. F. Buss, P. C. Kanellakis, P. L. Ragde, and A. A. Shvartsman, Efficient parallel algorithms on restartable fail-stop PRAMs and strongly asynchronous PRAMs, *J. Algorithms*, to appear.
4. R. Cole, Parallel merge sort, *in* "Proceedings, 27th Annual IEEE Symposium on the Foundations of Computer Science, 1986," pp. 511–516.
5. R. Cole and O. Zajicek, The APRAM: Incorporating asynchrony into the PRAM model, *in* "Proceedings, 1st Annual Symposium on Parallel Algorithms and Architectures, 1989," pp. 169–178.
6. R. Cole and O. Zajicek, The expected advantage of asynchrony, *in* "Proceedings, 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, 1990," pp. 85–94.
7. P. Dymond and W. L. Ruzzo, Parallel RAMs with owned global memory and deterministic context-free language recognition, *in* "Proceedings, 13th International Colloquium on Automata, Languages, and Programming, 1986," pp. 95–104.

8. D. M. Eckstein, "Simultaneous Memory Access," Technical Report No. TR-79-6, Computer Science Department, Iowa State University, 1979.
9. P. B. Gibbons, A more practical PRAM model, *in* "Proceedings, 1st Annual ACM Symposium on Parallel Algorithms and Architectures, 1989," pp. 158–168.
10. E. Grove, Connected components and the interval graph, *in* "Proceedings, 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992," pp. 382–391.
11. M. P. Herlihy, Impossibility and universality results for wait-free synchronization, *in* "Proceedings, Seventh Annual ACM Symposium on Principles of Distributed Computing, 1988," pp. 276–290.
12. M. P. Herlihy, Impossibility results for asynchronous PRAM, *in* "Proceedings, 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, 1991," pp. 327–336.
13. M. P. Herlihy, Randomized wait-free concurrent objects, *in* "Proceedings, Tenth Annual ACM Symposium on the Principles of Distributed Computing, 1991."
14. M. P. Herlihy and J. M. Wing, Axioms for concurrent objects, *in* "Proceedings, 14th ACM Symposium on the Principles of Programming Languages, 1987," pp. 13–26.
15. D. B. Johnson and P. Metaxas, Connected components in $O(\lg^{3/2}|V|)$ parallel time for the CREW PRAM, *in* "Proceedings, 32nd Annual IEEE Symposium on the Foundations of Computer Science, 1991," pp. 688–697.
16. P. C. Kanellakis and A. A. Shvartman, "Efficient Parallel Algorithms Can Be Made Robust," Technical Report CS-89-35, Department of Computer Science, Brown University, October 1989; *in* "Proceedings, Eighth Annual ACM Symposium on Principles of Distributed Computing, 1989," pp. 211–221.
17. Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan, Efficient program transformations for resilient parallel computation via randomization, *in* "Proceedings, 24th Annual ACM Symposium on the Theory of Computing, 1992," pp. 306–317.
18. Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, Combining tentative and definite executions for dependable parallel computing, *in* "Proceedings, 23rd Annual ACM Symposium on the Theory of Computing, 1991," pp. 381–390.
19. Z. M. Kedem, K. V. Palem, and P. Spirakis, Efficient robust parallel computations, *in* "Proceedings, 22nd Annual ACM Symposium on the Theory of Computing, 1990," pp. 138–148.
20. D. E. Knuth, "The Art of Computer Programming," Vol. 3, Addison-Wesley, Reading, MA, 1973.
21. C. P. Kruskal, L. Rudolph, and M. Snir, Efficient synchronization of multiprocessors with shared memory, *in* "Proceedings, Fifth Annual ACM Symposium on the Principles of Distributed Computing, 1986," pp. 218–228.
22. L. Lamport, On interprocess communication I: Basic formalism, *Distribut. Comput.* **1** (1986), 77–85.
23. F. T. Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes," Morgan Kaufmann, San Mateo, CA, 1992.
24. M. Luby, "On the parallel complexity of symmetric connection networks," Technical Report 214/88, University of Toronto, 1988.
25. C. Martel, A. Park, and R. Subramonian, "Optimal Asynchronous Algorithms for Shared Memory Parallel Computers," Report CSE-89-8, Division of Computer Science, University of California, Davis, July 1989.
26. C. Martel, R. Subramonian, and A. Park, Asynchronous PRAMs are (almost) as good as synchronous PRAMs, *in* "Proceedings, 31st Annual IEEE Symposium on the Foundations of Computer Science, 1990," pp. 590–599.
27. N. Nishimura, A model for asynchronous shared memory parallel computation, *SIAM J. Computing*, to appear.
28. N. Nishimura, Asynchronous shared memory parallel computation, *in* "Proceedings, 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, 1990," pp. 76–84.
29. N. Nishimura, "Asynchrony in Shared Memory Parallel Computation," Ph.D. thesis, University of Toronto, 1991; Technical Report 253/91, University of Toronto.
30. P. Raghavan, Probabilistic construction of deterministic algorithms: Approximating packing integer programs, *J. Comput. System Sci.* **37** (1988), 130–143.
31. Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3** (1982), 57–67.
32. R. Subramonian, Designing synchronous algorithms for asynchronous processors, *in* "Proceedings, 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992," pp. 189–198.
33. J. D. Ullman, "Computational Aspects of VLSI," Comput. Sci., Rockville, MD, 1984.
34. U. Vishkin, Implementation of simultaneous memory address access in models that forbid it, *J. Algorithms* **4** (1983), 45–50.