Deterministic Simulations of PRAMs on Bounded Degree Networks *

Kieran T. Herley
Gianfranco Bilardi

88-951
November 1988

Department of Computer Science
Cornell University
Ithaca, New York 14853

*Research partially supported by NSF grant DCR-86-02307, NSF grant MIP-86-02256, and JSEP contract F49620-87-C-0044

Deterministic Simulations of PRAMs on Bounded Degree Networks [1]

Kieran T. Herley [2] and Gianfranco Bilardi [3]

Abstract

The problem of simulating a PRAM with n processors and memory size $m \geq n$ on an n-node bounded degree network is considered. A scheme is presented which simulates an arbitrary PRAM step in $O((\log n \log m)/\log\log n)$ time in the worst case on an expander-based network. By extending a previously established lower bound, it is shown that the proposed simulation is optimal whenever $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^\alpha})$ for some $\epsilon > 0$ and some $\alpha > 0$.

Keywords: parallel computation, shared memory machines, simulations, networks of processors, expander graphs.

AMS(MOS) subject classification: 68Q05

Abbreviated title: Deterministic Simulations of PRAMs on Networks.

---

[1]A preliminary version of this paper appears in the Proceedings of the 26^{th} Annual Allerton Conference on Communication, Control and Computation, Monticello, Illinois, September 1988.

[2]Computer Science Department, Cornell University, Ithaca, NY14853. The work of this author was supported in part by the National Science Foundation under grant DCR-86-02307 and by the Joint Services Electronics Program under contract F49620-87-C-0044.

[3]Computer Science Department, Cornell University, Ithaca, NY14853. The work of this author was supported in part by the National Science Foundation under grant MIP-86-02256 and by the Joint Services Electronics Program under contract F49620-87-C-0044.

1 Introduction

Parallel Random Access Machines(PRAMs) play a central role in the study of parallel computation and in the development of parallel algorithms[KR88]. A PRAM is essentially a set of synchronous processors connected to a shared memory. The basic feature of the PRAM is that references to distinct memory cells can be made simultaneously by different processors. Simultaneous references to the same memory cell are allowed in various degrees in different variants of the PRAM model [FW78,LPV81,Gol82,SV81, BH85,Sch80]. By abstracting several of the constraints that arise in the physical realization of a multiprocessor, the PRAM model provides an attractive framework for algorithm design. However, performance estimates obtained for the PRAM are not necessarily accurate when referred to a realistic multiprocessor [Sny86].

Two important constraints not captured by PRAMs are the granularity of the memory (the memory is partitioned into banks or modules each of which can respond to only one request per step), and the fact that components (processors and modules) can be connected directly only to a small number of other components, the number being independent of the machine size. Thus, the Bounded Degree Network(BDN) of processors and memory modules provides a more realistic model of parallel computers than PRAMs. The question naturally arises of how well a PRAM can be simulated by a suitably chosen BDN. This question has received considerable attention in the literature, and we briefly review some of the results here.

One approach, proposed in [MV84] and further developed in [Upf84,KU88,Ran87, LPP87a], distributes the address space of the PRAM among the modules of the BDN according to some suitably chosen hash function. The objective is to ensure that, at any node of the network, congestion is low with high probability and hence obtain a good *expected-time* performance. In particular [Ran87] proposes a protocol for an n-node butterfly network that simulates $T \geq m/n$ steps of an n-processor PRAM with m memory cells in $O(T \log n)$ time, with high probability, assuming that m is polynomial in n. The protocol is deterministic, except for the random choice of the hash function that defines the address map.

Another approach, also proposed in [MV84], relies on representing the contents of a PRAM memory cell by using several BDN cells. This approach forms the basis for a number of deterministic simulation schemes [UW87,AHMP87,LPP87b] including those proposed in this paper. The goal of these schemes is to provide good *worst-case* performance.

In [UW87], a scheme is advanced to simulate an arbitrary step of a PRAM with n processors and $m \geq n$ memory cells by an n-node BDN in time $O(\log n \log m \log\log m)$. Nodes of the BDN consist of a RAM processor and a memory module. They are interconnected as in the graph (V, E_{AKSL}), Leighton's modification of the AKS sorting network[AKS83,Lei85]. This time bound was subsequently improved to $O(\log n \log m)$ in [AHMP87], and is further improved to $O((\log n \log m)/\log\log n)$ in the present paper. Our network is also an n-node BDN, $(V, E_{AKSL} \bigcup E_{PU} \bigcup E_{Tree})$ where (V, E_{PU}) is a square-root expander [PU86,LPS86], and (V, E_{Tree}) is a complete binary tree.

An $\Omega((\log n \log m)/\log\log m)$ lower bound for the time to simulate a step of a PRAM with n processors and m memory cells by and n-node BDN was established independently in [AHMP87] and [KU88]. The lower bound makes the so-called point-to-point assumption, that if different copies of a variable are generated by a given BDN processor, then they must be transmitted by the network as distinct entities. Furthermore it is assumed that, for some $\epsilon > 0$, $\Omega(n^{2+\epsilon}) \leq m \leq 2^{O(\sqrt{n \log n})}$. By refining the proof technique of [AHMP87] and [KU88], we obtain a lower bound of $\Omega((\log(m/n))^2/\log\log(m/n))$ for $\Omega(n) \leq m \leq O(n^{2+\epsilon})$.

It is interesting to observe that the simulation described in this paper complies with the point-to-point assumption, and that its performance matches the cited lower bound as long as $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^\alpha})$, for some $\epsilon > 0$ and some $\alpha > 0$.

In [LPP87b] a PRAM simulation is presented on a BDN consisting of an $n \times n$ mesh-of-trees [Lei81,NMB83], where the roots of the trees are fully fledged RAM processors, while the other $\Theta(n^2)$ nodes are packet-switching elements. The scheme is attractive for the regularity and simplicity of the interconnection. For m polynomial in n, a PRAM step is simulated in time $O(\log^2 n/\log\log n)$, the same as that achieved by our simulation for this range of m. However, it should be noted that the use of $\Theta(n^2)$ nodes, albeit mere switchs, makes the cited lower bound inapplicable to the mesh-of-trees.

In most of the studies referred to above the number n of nodes in the BDN is assumed to be proportional to to the number of processors in the PRAM being simulated, and is independent of of the size m of the PRAM memory. It is certainly of interest to investigate simulations where the the size of the BDN is allowed to depend on both n and m. In this direction it is worth mentioning the scheme of [Vis84], which has running time $O(\log n + \log m)$ on a BDN of $O(n \log n + m \log m)$ nodes.

The remainder of this paper is organized as follows. Section 2 is devoted to the formulation of the problem. Section 3 discusses the memory organization, i.e. the method by which the memory of the PRAM is represented by the memory of the BDN. Introducing the graph-theoretic notion of generalized expander, we investigate the trade-off between the number of copies and memory congestion in organizations of the type proposed in [UW87]. The choice of an appropriate point in this trade-off is critical in order to balance the work needed to extract the relevant information from the memory modules with the work needed to route this information to the intended processors. Section 4 describes the memory access protocol. In several steps of the protocol, the necessity arises to process sets of elements distributed unevenly among the nodes of the BDN. This problem is solved by a careful combination and adaptation of sorting [AKS83,Lei85] and routing [PU86] techniques. An analysis of the protocol yields the bounds stated above. Section 5 is devoted to lower bounds. Section 6 concludes the paper by indicating a number of interesting open problems.

2 Problem Formulation

We consider a PRAM of n processors, $P_1^{PRAM}, P_2^{PRAM}, \ldots, P_n^{PRAM}$ sharing a set U of m memory cells. Our goal is to simulate such a PRAM by a BDN of n nodes (P_1, M_1), $(P_2, M_2), \ldots, (P_n, M_n)$ where P_i is a RAM processor and M_i is a memory module. We are interested in simulation algorithms that are deterministic and on-line, i.e. the simulation of a step is independent of future steps.

We assume that the instruction to be executed by PRAM processor P_i^{PRAM} at a given step is available to the corresponding BDN processor P_i at the beginning of the simulation of this step. Since simulation of arithmetic and logical operations is straightforward, we can regard a generic PRAM step simply as an n-tuple of memory accesses. A memory access can either be of the form read(u), ($u \in U$), which results in the contents of the memory cell u being copied into a distinguished register of the executing processor, or of the form write(u,x), ($u \in U, x$ an integer value), which results in the integer value x being written to memory cell u. For simplicity we assume that all the accesses in a given step are of the same type (read or write), and that no two accesses refer to the same memory cell. Both assumptions can be removed by established techniques [Vis83] without altering the stated time bounds, as already noted in [UW87].

In any simulation scheme, upon termination of the simulation of a given PRAM step, the state of the PRAM memory must be uniquely reconstructible from the state of the BDN memory. The method chosen for representing the state of the PRAM memory by the state of the BDN memory is referred to as the *memory organization.* The algorithm that probes the state of the BDN memory when simulating a read step, and that modifies it when simulating a write step is called the *memory access protocol.*

A simulation scheme is consistent if after the simulation of an arbitrary sequence of PRAM steps, the simulation of a step where processor P_i^{PRAM} executes read(u) results in loading the distinguished register of P_i with the correct value of u, which is if course the value assigned to u by the the latest instruction of the form write(u,x).

In the next two sections we describe and analyse a new simulation scheme. Section 3 is devoted to the memory organization, and Section 4 to the memory access protocol.

3 Memory Organization

Following [UW87], we consider a memory organization where, for each PRAM variable $u \in U$, the BDN maintains $2c - 1$ copies in the memory modules of a set $\Gamma(u) = \{\gamma_1(u), \gamma_2(u), \ldots, \gamma_{2c-1}(u)\}$ of distinct nodes. Each copy contains a value and a timestamp indicating the time at which that copy was last written. When simulating a step where u is written, at least c copies of u are updated. When simulating a step where u is read, at least c copies of u are retrieved: the one with the most recent timestamp is guaranteed to contain the correct value of u. We will adopt the same read/write discipline.

All the memory organizations considered in this paper will be static in the sense that the location of the copies of a particular variable will not vary with time. This type of memory organization can be modelled conveniently as a bipartite graph $G = (U, V, E)$ where U is the set of PRAM variables, V is the set of the nodes of the BDN, and an edge $(u, \gamma(u))$ represents the fact that the node $\gamma(u)$ contains a copy of the variable u.

Let $S \subseteq U$ and let $F \subseteq E$ have exactly k edges incident upon each $u \in S$. We call F a *k-bundle* for S. We say that the subset of V on which the edges of F are incident is a *k-neighbourhood* of S and denote it by $\Gamma_F(S)$. For any k-bundle F of a set S we say F has *congestion* C, if the degree of each vertex $v \in V$ is at most C in the graph $(S, \Gamma_F(S), F)$.

When the copy scheme described above is adopted, the simulation of a PRAM step accessing the variables of S requires that the BDN access all the copies of some c-bundle, say F. Clearly this takes at least $c|S|/|\Gamma_F(S)|$ time, since at least $c|S|$ copies have to be accessed from $|\Gamma_F(S)|$ modules. It is desirable to have a memory organization with large c-neighbourhoods. We would also like to access a c-bundle with low congestion. The preceding remarks motivate the following definition.

Definition 1 *A bipartite graph $G = (U, V, E)$ with $|U| = m$ and $|V| = n$, and with each node in U having degree d is a (λ, d, c, σ)-generalized expander if, for every $S \subseteq U$ such that $|S| \leq \sigma n$ and for every c-bundle F of S, $|\Gamma_F(S)| \geq \lambda c|S|$.*

Clearly $\lambda \leq 1$ and $\sigma \leq 1/\lambda c$ are necessary conditions for the existence of a (λ, d, c, σ)-generalized expander.

We can establish the existence of certain types of generalized expanders. (The proof is a based on standard types of counting arguments, and can be skipped without loss of continuity.)

Theorem 1 *For every n, m, g and λ, with $m \geq n$, $g \geq 1$ and $\lambda < 1$, there is a β depending only on λ such that for every $c \geq \beta \log(m/n)/\log g$ there exists a $(\lambda, 2c-1, c, 1/gc)$-generalized expander $G = (U, V, E)$ with $|U| = m$ and $|V| = n$.*

Proof Let $B_{m,n,2c-1}$ be the class of all bipartite graphs $G = (U, V, E)$ with *labelled* nodes and *unlabelled* edges such that $|U| = m$, $|V| = n$, and each node of U has degree $2c-1$.

Note that $|B_{m,n,2c-1}| = \binom{n}{2c-1}^m$. Let η be the fraction of graphs in $B_{m,n,2c-1}$ that are *not* $(\lambda, 2c-1, c, 1/gc)$-generalized expanders. We will show that $\eta < 1$.

We begin by observing that, if $G \in B_{m,n,2c-1}$ is not a $(\lambda, 2c-1, c, 1/gc)$-generalized expander, then there exists a triple (S, W, F) with the following properties: (i) $S \subseteq U, |S| \leq n/gc$; (ii) $W \subseteq V, |W| = \lceil \lambda c|S| - 1 \rceil$; (iii) $F \subseteq E$ a c-bundle of S such that $\Gamma_F(S) \subseteq W$. We shall call such a triple a *witness.* If we let f_s be the fraction of all graphs in $B_{m,n,2c-1}$ having a witness (S, W, F) with $|S| = s$, then we can bound η as follows:

$$\eta \leq \sum_{1 \leq s \leq n/gc} f_s. \tag{1}$$

We now claim that

$$f_s = \binom{m}{s}\binom{n}{\lambda cs}\binom{2c-1}{c}^s \left\{\frac{\binom{\lambda cs}{c}}{\binom{n}{c}}\right\}^s. \tag{2}$$

Standard combinatorial arguments show that there are $\binom{m}{s}$ ways of choosing S, $\binom{n}{\lambda cs}$ ways of choosing W, $\binom{\lambda cs}{c}^s$ ways of choosing F, and a fraction $\left\{\binom{2c-1}{c}/\binom{n}{c}\right\}^s$ of the graphs in $B_{m,n,2c-1}$ for which triple (S, W, F) is a witness. Then, simple arithmetic yields Equation 2. Use of the inequalities $\binom{m}{s} < (em/s)^s$, $\binom{n}{\lambda cs} < (en/\lambda cs)^{\lambda cs}$, $\binom{2c-1}{c} < 2^c$, and $\binom{\lambda cs}{c}/\binom{n}{c} < (\lambda cs/n)^c$ in Equation 2 yields (after some manipulations)

$$f_s < \left\{(\frac{ecm}{n})\left[A(\lambda)(cs/n)^{1-\lambda-1/c}\right]^c\right\}^s,$$

where $A(\lambda) = 4e^{\lambda}\lambda^{(1-\lambda)}$. Assuming $c > 2/(1-\lambda)$ and $s \leq n/gc$, we have

$$f_s < \left\{(\frac{ecm}{n})\left[A(\lambda)(1/g)^{2/(1-\lambda)}\right]^c\right\}^s.$$

Tedious but straightforward manipulations show that for a suitable $\beta = \Theta(1/(1-\lambda))$, if $c > \beta \log(2m/n)/\log g$, then $f_s < 1/2^s$. The latter relation together with Equation 1 allows us to conclude that $\eta < 1$, as claimed. □

Upfal and Wigderson[UW87] established the existence of a family of $(\lambda, 2c-1, c,$ $1/(2c-1))$-generalized expanders for every $\lambda \leq 1/2$, with $c \geq \beta \log m$. These expanders and similar ones define the memory organizations used in [UW87,AHMP87,LPP87b]. They can essentially be derived from Theorem 1 by choosing $g = 2$.

The above theorem demonstrates that $c = \beta \log(m/n)/\log g$ is sufficient to guarantee the existence of $(\lambda, 2c-1, c, 1/gc)$-generalized expanders. A natural question is whether such a large value of c is really necessary. The answer is positive as argued below.

Consider some $W \subseteq V$. We say that W *contains* $u \in U$ if $\Gamma(u) \subseteq W$. A simple combinatorial argument given in both [MV84] and [UW87] shows that the subsets of V of size w contain an average of $m\binom{n-2c+1}{w-2c+1}/\binom{n}{w} = m[w]_{2c-1}/[n]_{2c-1}$ variables apiece. In particular some set $W \subseteq V$ of size w will contain a set of variables S with at least that many variables. Choose $w = \lambda c(n/gc)$ and let W and S be chosen as above. Then we have $\Gamma(S) \subseteq W$, $|\Gamma(S)| \geq \lambda c|S|$ and $m[|\Gamma(S)|]_{2c-1}/[n]_{2c-1} \geq |S|$. The combination of these facts implies that $c = \Omega(\log(m/n)/\log g)$.

One important facet of Theorem 1 lies in the introduction of the parameter g, which implies a trade-off between the number of copies and the largest set for which expansion can be guaranteed. In fact the size of the smallest set of modules containing a given number of variables depends strongly on the value of c: the smaller this value the smaller the size of the set of modules. Roughly speaking, increasing g reduces the number of copies to be processed, but creates higher congestion at memory modules, thereby making the job of accessing those copies more difficult.

We now explore some of the properties of generalized expanders which will prove useful in the next section. The following lemma is essentially a rephrasing of the definition of generalized expanders.

Lemma 1 *Let $G = (U, V, E)$ be a $(\lambda, 2c-1, c, 1/gc)$-generalized expander. If $S \subseteq U$ and $F \subseteq E$ defines a c-neighbourhood $\Gamma_F(S)$ of size smaller than $\lambda n/g$, then $|S| \leq |\Gamma_F(S)|/\lambda c$.*

Proof Suppose, by way of contradiction, that $|S| > |\Gamma_F(S)|/\lambda c$, that is, $|\Gamma_F(S)| < \lambda c|S|$. By Definition 1, it follows that $|S| > n/gc$. Let S' be a subset of S of size n/gc and let $\Gamma_{F'}(S')$ be the c-neighbourhood of S' contained in $\Gamma_F(S)$. Again by Definition 1 we have that $|\Gamma_F(S)| \geq |\Gamma_{F'}(S')| \geq \lambda n/g$, which is a contradiction. □

Consider now a memory organization represented by a bipartite graph $G = (U, V, E)$ and let $S \subseteq U$ be a set of PRAM variables to be accessed. The copies of the variables in S will be distributed among the BDN nodes with node $v \in V$ containing $s_v \triangleq |\{(u, v) : u \in S, (u, v) \in E\}|$ copies. We are interested in analysing the effect of accessing $\min(s_v, q)$ copies from each node $v \in V$, as a function of the integer q. This operation, which we denote by *Decimate(S,q)*, will in fact form the basis of the memory access protocol to be described in the next section.

The variables of S can be partitioned into two sets: those for which at least c copies have been accessed, which we will call the *dead* variables, and the remaining ones which we call the *live* variables. Dead variables play no further role in the simulation since enough copies have been accessed to effect either a read or write. For the live variables, more copies need to be accessed. Therefore, we are interested in bounding the size of the set of variables left alive after *Decimate(S,q)*.

Formally, the set of copies accessed by *Decimate(S,q)* is represented by a set of edges $E' \subseteq E$ such that if $(u, v) \in E'$ then $u \in S$, and for each $v \in V$, there are exactly $\min(s_v, q)$ edges in E' incident upon v. Then, L is the subset of the nodes of S with degree at least c in the graph $(U, V, E - E')$.

Lemma 2 *With the above notation, if the memory organization is a $(\lambda, 2c-1, c, 1/gc)$-generalized expander $G = (U, V, E)$ and if $q > 2\lceil (cg/n)|S|\rceil/\lambda$ then the set L of variables remaining alive after* Decimate(S,q) *satisfies the relation*

$$|L| \leq \left[\frac{2(1-\lambda)}{q - 2\lceil cg|S|/n\rceil + 1}\right]|S|. \tag{3}$$

Proof Let $k = \lceil cg|S|/n\rceil$. Then S can be partitioned into k subsets $S_1, S_2, \ldots, S_k$, each of size no larger than n/cg. For $i = 1, 2, \ldots, k$, let $F_i \subseteq E$ be the set of edges incident upon nodes of S_i. Then F_i can be partitioned into $F_{i,1}$ and $F_{i,2}$ so that $\Gamma_{i,1} \triangleq \Gamma_{F_{i,1}}(S)$ is a c-neighbourhood of S and $\Gamma_{i,2} \triangleq \Gamma_{F_{i,2}}(S)$ is a $(c-1)$-neighbourhood of S. From the expansion property, we have that $|\Gamma_{i,1}| \geq \lambda c|S_i|$ and $|\Gamma_{i,2}| \geq (\lambda c - 1)|S_i|$. We now form, for each $i = 1, 2, \ldots, k$ and $j = 1, 2$, the set $F'_{i,j}$ by (arbitrarily) selecting for each node of $\Gamma_{i,j}$ one edge of $F_{i,j}$ incident upon it. Let $F = \bigcup_{i=1}^{k} \bigcup_{j=1}^{2} F'_{i,j}$. By construction, no more than $2k$ edges of F are incident upon any $v \in V$. Hence, the number of copies accessed by *Decimate(S,2k)* is at least

$$\begin{aligned}
|F| &= \sum_{i=1}^{k}\sum_{j=1}^{2}|F'_{ij}| \\
&= \sum_{i=1}^{k}\sum_{j=1}^{2}|\Gamma_{ij}| \\
&\geq \sum_{i=1}^{k}(\lambda c|S_i| + (\lambda c - 1)|S_i|) \\
&= (2\lambda c - 1)|S|.
\end{aligned}$$

Now let $H = \{v : s_v > q\}$. Each node in H contains at least $q - 2k + 1$ copies of variables in S beside those accessed by *Decimate(S,q)*. Considering that the total number of copies of variables in S is $(2c-1)|S|$ we obtain that

$$(q - 2k + 1)|H| + (2\lambda c - 1)|S| \leq (2c - 1)|S|,$$

or equivalently

$$|H| \leq [2(1-\lambda)c/(q - 2k + 1)]\,|S|. \tag{4}$$

Given that, by assumption, $q > 2k/\lambda$ we have that $|H| < \lambda n/g$.

Since the copies of the live variables L not accessed by *Decimate(S,q)* are all in H, H must contain a c-bundle D of L. Then since $|\Gamma_D(L)| \leq |H| < \lambda n/g$, we have from Lemma 1 that $|L| \leq |H|/\lambda c$. Finally use of bound 4 yields Equation 3. □

We shall call the quantity $|L|/|S|$ the *survival factor* of the operation *Decimate(S,q)*. *Decimate* is the building block of the memory access protocol described in the next section. In contrast, the protocols of [UW87] and [AHMP87] are built around a *halving* procedure which, although implemented differently, has the same effect as a decimation with survival factor of at most $1/2$.

Intuitively, the advantage of decimation over halving can be understood as follows. From standard considerations on the diameter of an n-node BDN it is clear that any implementation of halving will require $\Omega(\log n)$ time in the worst case. However it will be shown in the next section that $O(\log n)$ time is essentially sufficient for the execution of a *Decimate(S,*$O(\log n)$*)*, as long as $|S| \leq n/(2c-1)$. From Lemma 2 we can see that, for $|S| \leq n/(2c-1)$, choosing $g = \log n$ and q a suitable multiple of g yields a survival factor of $O(1/\log n)$. Thus, decimations can kill more variables than halving in the same amount of time. A similar observation is also employed in the scheme of [LPP87b].

4 The Memory Access Protocol

In this section we describe a memory access protocol for the simulation of one step of a PRAM with n processors and m memory cells. We assume that the memory organization is a $(\lambda, 2c-1, c, 1/gc)$-generalized expander with $c = \beta \log(m/n)/\log g$. If is further assumed that each node of the BDN stores in its local memory a table giving for each PRAM memory cell u the nodes $\gamma_1(u), \gamma_2(u), \ldots, \gamma_{2c-1}(u)$ containing copies of u.

Let X be the set of PRAM variables to be accessed in a given PRAM step. The goal of the simulation is to access a majority of the copies of each variable in X. We shall describe the protocol for simulating a read step, the protocol for a write step being very similar. A high level description of the protocol follows.

I For each processor P_i, generate u_i, the name of the variable it wishes to read.

II For each u_i and each $j \in \{1, 2, \ldots, 2c-1\}$ generate a *request packet* containing an origin P_i, a destination $\gamma_j(u_i)$, the label u_i itself, and a field to store the value and the timestamp of the copy of u_i contained in node $\gamma_j(u_i)$.

III Deliver at least c request packets from each origin to their destinations, and return these packets to their origins with the values and the timestamps.

IV At P_i, select the value from the returned packet(s) with the most recent timestamp as the result of read(u_i).

Steps I, II, and IV are straightforward and are accomplished in time $O(1)$, $O(c)$, and $O(c)$ respectively. We focus our attention on Step III. The latter will be implemented as a sequence of decimations, the first one applied to the set X, and the remaining decimations each applied to the set of variables left alive by the previous decimation.

More specifically, we shall define a procedure *Decimate(S,q)* with the following properties: (i) S is a variable parameter and represents a subset of U of size at most n distributed among the nodes of the BDN, with at most one element per node; (ii) the second argument q is a value parameter of type integer; (iii) the execution of *Decimate(S,q)* will cause $\min(s_v, q)$ packets destined for node v to reach that node and to be returned to their respective origins, with the appropriate values and timestamps (recall from Section 3 that s_v denotes the number of copies of variables in S contained in node v); (iv) the value of parameter S is modified by the execution of *Decimate(S,q)* so that, upon termination, S represents the set of live variables, that is those whose origins have received fewer than c packets.

In terms of procedure *Decimate*, Step III of the memory access protocol can be written as follows.

$$
\begin{aligned}
& S \leftarrow X; \\
& \textbf{for } h \leftarrow 1 \textbf{ to } H \textbf{ do} \\
& \qquad \textit{Decimate}(S,\, q_h)
\end{aligned}
$$

The values of $q_1, q_2, \ldots, q_H$ and H will be specified later so that after the H^{th} iteration S is empty, i.e. all the variables in X have been accessed.

To complete the definition of the memory access protocol, it remains to specify an implementation for the procedure *Decimate*. One of the tools which will prove useful in the implementation is that of *balancing*. The goal of a balancing operation is to take a set of elements distributed among the nodes of the BDN and rearrange them so that each nodes has essentially the same number of elements. The next subsection is devoted to a description to the operation of *balancing* and sorting. The implementation and the analysis of *Decimate* are given in the subsequent subsection. The final subsection provides an analysis of the running time of the entire memory access protocol.

4.1 Balancing and Sorting

Let $\mathbf{k} = (k_1, k_2, \ldots, k_n)$. We say that a set Z is $\mathbf{k}$-distributed if the elements of Z are distributed among the n nodes of a BDN with k_i elements at node i. We say that a $(k_1, k_2, \ldots, k_n)$-distributed set has *degree* K if $\max(k_1, k_2, \ldots, k_n) = K$. A $\mathbf{k}$-distributed set is *balanced* if $K = \lceil |Z|/n \rceil$.

Peleg and Upfal[PU86] exhibit an algorithm for balancing a set by redistributing its elements among the nodes. (Similar, but not entirely equivalent, balancing schemes appear in [AHMP87] and [CV88].) The algorithm of Peleg and Upfal works on a BDN $(V, E_{AKSL} \bigcup E_{PU})$, where (V, E_{AKSL}) is the graph underlying Leighton's modification of the AKS sorting network [AKS83,Lei85], and (V, E_{PU}) is a square-root expander graph[PU86,LPS86]. The result of [PU86] is stated in a model where only communication steps are counted, and where the set to be balanced has at most n elements. Restated in our model and generalized to sets of larger size, the result becomes the following one.

Lemma 3 *A $(k_1, \ldots, k_n)$-distributed set Z, with degree K and $|Z| = n^{O(1)}$, can be balanced on the BDN $(V, E_{AKSL} \bigcup E_{PU})$ of n nodes in time $O(\log n \log K + K + \lceil |Z|/n \rceil \log n)$.*

The protocol of [PU86] has three parts. During the first part a number $\log K$ of subgraphs $G_1, G_2, \ldots, G_{\log K}$ of (V, E_{PU}) are identified in $O(\log n \log K)$ time. No elements are moved during this part. The second part consists of $\log K$ phases one executed on each of $G_1, G_2, \ldots, G_{\log K}$. Each phase reduces the maximum number of elements at any node by some fixed constant fraction. The entire second part runs in $O(\log n + K)$ time. At its conclusion no node has more than $O(\lceil |Z|/n \rceil)$ elements. The third part involves sorting to ensure that no node holds more than $\lceil |Z|/n \rceil$ elements.

Let Z be a $(k_1, \ldots, k_n)$-distributed set and let Z' be a $(k'_1, \ldots, k'_n)$-distributed set. We say that Z *dominates* Z' if $k_i \geq k'_i$ for $i = 1, 2, \ldots, n$. It turns out that the first part (identifying the subgraphs) of Peleg and Upfal's algorithm to balance Z can be reused to balance Z', provided that $|Z| \leq n$. Exploiting this fact, we obtain the following lemma.

Lemma 4 *Let Z, $|Z| \leq n$, be a $(k_1, \ldots, k_n)$-distributed set with degree K. Having balanced Z, any set Z' dominated by Z can be balanced at an extra cost of $O(\log n + K)$ time.*

Various stages of the implementation of *Decimate* will consist of sorting a distributed set Z, that is rearranging the elements of Z so that the $\lceil |Z|/n \rceil$ smallest elements are at node 1, the next $\lceil |Z|/n \rceil$ are at node 2 and so on. Sorting can be accomplished by first balancing Z, and then using the AKSL sorting algorithm[AKS83,Lei85] modified according to the techniques of [BS78] to handle the case $|Z| > n$. Sorting Z after balancing will take $O(\lceil |Z|/n \rceil \log(|Z| + n))$ time. We can summarize Lemmas 3 and 4, and the preceding discussion as follows , where we assume that $|Z|$ is polynomial in n since this simplifies the expressions and is not restrictive for our applications.

Lemma 5 *Let Z, $|Z| = n^{O(1)}$, be a $(k_1, \ldots, k_n)$-distributed set with degree K. Z can be sorted on the BDN $(V, E_{AKSL} \bigcup E_{PU})$ in $O(\log n \log K + K + \log n \lceil |Z|/n \rceil)$ time. Furthermore if $|Z| \leq n$ and Z' is dominated by Z, then sorting Z' can be done at an extra cost of $O(\log n + K)$ time, having sorted Z.*

4.2 Implementation of *Decimate*

We outline an implementation of *Decimate(S,q)* on the BDN $(V, E_{AKSL} \bigcup E_{PU} \bigcup E_{Tree})$ where E_{AKSL} and E_{PU} are as defined in the previous section and (V, E_{Tree}) is a balanced binary tree. Recall that S is a set of PRAM variables, at most one per processor. A processor holding a variable of S has formed $2c - 1$ request packets, one for each copy of that variable, with the format described in Step II of the memory access protocol.

The overall idea is as follows. For each destination $v \in V$ we *select* exactly $\min(s_v, q) \triangleq d_v$ requests (recall that s_v represents the number of requests directed at module v). Destination v generates exactly d_v *dummies.* The key step involves sorting both the set of selected requests and the set of dummies. There are exactly the same number of selected requests destined for v and dummies which originated at v. By pairing corresponding items in the two sorted lists we match each selected request with a dummy which originated at its destination. If, during this sorting step, the dummies had recorded their paths this information can be used to guide the selected requests to their destinations: they simply retrace the steps taken the dummy during the sorting step. The details follow.

Decimate(S,q)

1 Sort the request packets by destination.

2 For each destination v, determine d_v. Select d_v request packets from those destined for node v.

3 Sort the selected requests by destination.

4 For each v generate a packet containing d_v and labelled v. Sort these packets by label and route them to the corresponding nodes.

5 For each node v generate d_v dummy tokens, each labelled v.

6 Sort the dummy tokens by label.

7 Match each request with a dummy which originated at its destination.

8 Route the request packets to their destinations by retracing the steps taken by the corresponding dummy during step 6.

9 Load values and timestamps into request packets.

10 Route the selected requests (now satisfied) back to their origins tracing the paths taken in steps 1, 3 and 6 backwards.

11 Let each origin count the number of packets received during Step 10 and if they are at least c in number, retain the value of the packet(s) with the most recent timestamp and remove the variable from S.

Lemma 6 Decimate(S,q) *can be implemented on the BDN* $(V, E_{AKSL} \bigcup E_{PU} \bigcup E_{Tree})$ *with running time* $O(\log n \log(cg) + c + q + \log n \lceil c|S|/n \rceil)$.

Proof Firstly Steps 5, 7, 9 and 11 are sequential computations local to a given node and take $O(q)$, $O(1)$, $O(q)$, and $O(c)$ time respectively.

Step 2 can be reduced to prefix computations [KRS85] on sequences of length $(2c-1)$ $|S|$ which can be executed on (V, E_{Tree}) in $O(\log n + \lceil c|S|/n \rceil)$ time.

Steps 1, 3 and 4 consist of sorting a distributed set of size no larger than $(2c-1)|S|$ and with at most $2c-1$ elements per node. By Lemma 5, these take $O(\log n \log c + c + \log n \lceil c|S|/n \rceil)$ time. (Step 4 also includes a routing step but this does not alter the time bound.) Step 6 consists of sorting a distributed set of size no larger than $(2c-1)|S|$ an with at most q elements per node. By Lemma 5, these steps take $O(\log n \log q + q + \log n \lceil c|S|/n \rceil)$ time.

Step 8 involves retracing the steps taken by the dummy during step 6 and has the same running time. Step 10 is a combination of Steps 1, 3 and 6 and has the same running time. To accomplish this, each node need simply record the arrival and departure (the time and the edge involved) of each packet which passes through it.

This information will allow a subsequent step to retrace the steps taken. The details are straightforward.

By summing the contributions of all the steps it is easy to see that the stated running time is achieved. □

4.3 Simulation Time

We now turn our attention to the running time of Step III of the memory access protocol consisting of a sequence of H decimations with parameters $q_1, q_2, \ldots, q_H$. Let L_i be the set of live variables after the i^{th} decimation.

For the first decimation, applied to the set X of PRAM variables, we choose $q_1 = \max(2/\lambda, 4)cg$. By Lemma 2, $|L_1| < (1-\lambda)n/(cg)$. By Lemma 6 , *Decimate(S,q_1)* runs in $O(\log n \log(cg) + cg + c\log n)$ time.

For the remaining decimations we choose $q_2 = q_3 = \ldots = q_H = 2g+1$. By Lemma 2, for $i \geq 2$, $|L_i| \leq (1-\lambda)|L_{i-1}|/g$. Thus, for $i \geq 2$, $|L_i| < (1-\lambda)^{i-1}|L_1|/g^{i-1} \leq (1-\lambda)^i n/g^i$. Choosing $H = \lceil \log n / \log(g/(1-\lambda)) \rceil$ ensures that $L_H = \emptyset$.

To analyse running time, let us observe that, for each $i \geq 2$, the set of requests at Step 1 and the set of dummies at Step 6 in the i^{th} decimation are dominated by the corresponding sets in the $(i-1)^{st}$ decimation. Furthermore, all these sets have size smaller than n. Therefore, by Lemma 6, *Decimate(L_2, q_2)* runs in time $O(\log n \log(cg) + c + g + \log n)$. By Lemma 5, each of the remaining $H-2$ decimations will take $O(c + g + \log n)$ time.

The total time for Step III, and indeed for the entire simulation is

$$O(\log n \log(cg) + cg + c\log n + (c + g + \log n)(\log n/\log g)).$$

Choosing $g = \log n$ and recalling that $c = \beta \log m/\log g$, we arrive at the following result.

Theorem 2 *An arbitrary step of a PRAM with n processors and $m \geq n$ memory cells can be simulated by the BDN $(V, E_{AKSL} \bigcup E_{PU} \bigcup E_{Tree})$ in time $O((\log n \log m)/\log\log n)$ time.*

Notice that in our protocol each request is treated as a separate entity by the routing network: requests are generated only at their origins, while the other processors serve only to pass the requests from their origins to their respective destinations and back. This form of communication has been referred to as *point-to-point* communication in the literature [AHMP87]. Lower bounds for simulations obeying the point-to-point assumption are discussed in the next section.

5 Lower Bound

In [AHMP87] and [KU88] the following lower bound was established independently. (All of the bounds in this section refer to worst case simulation time of a PRAM with n processors and m cells of shared memory on a BDN of n processors.)

Theorem 3 *If* $m = \Omega(n^{2+\epsilon})$ *and* $T \geq (1+\epsilon)(m/n)$ *for some* $\epsilon > 0$*, then the worst-case simulation time for a straight-line SIMD program running for* T *steps on an* n*-processor EREW PRAM with* m *cells of shared memory is*

$$\Omega(T\,min(\sqrt{n \log n}, \frac{\log m \log n}{\log \log m})).$$

The bound has been derived under the *point-to-point* assumption: different copies of a variable generated by a given BDN processor must be transmitted by the network as distinct entities. In other words during the simulation of a writing step copies can be synthesized only at the writing processor, others can serve to pass the copies along unaltered to their destinations. During a reading phase copies may be made only at the nodes holding valid copies of a particular variable. Under the same assumption, by refining the proof technique of [AHMP87,KU88], we can extend the lower bound to all values of $An \leq m \leq n^{2+\epsilon}$, for some $A = O(1)$.

Theorem 4 *If* $An \leq m \leq n^{2+\epsilon}$ *and* $T \geq (1+\epsilon)(m/n)$ *for some* $\epsilon > 0$*, then the worst-case simulation time for a straight-line SIMD program running for* T *steps on an* n*-processor EREW PRAM with* m *cells of shared memory is*

$$\Omega(T\frac{\log^2(m/n)}{\log\log(m/n)}).$$

We first review the ideas behind the proof of Theorem 3, and then indicate then modifications necessary to obtain Theorem 4. For ease of reference, we will use the same notation as in [AHMP87].

The approach consists of constructing an adversary program of $(1+\epsilon)(m/n)$ instructions, the first m/n of which are used to initialize all m PRAM variables, and the remaining $\epsilon m/n$ form $\tau \triangleq \epsilon m/2n$ *stages* each consisting of a read instruction followed by a write instruction. The variables to be accessed in each of these instructions are chosen, by the adversary, according to the state of the BDN memory so as to make each read and write as time-consuming as possible.

In order to construct such a sequence of hard reads and writes, the lower bounds of [AHMP87,KU88] define the notion of the *redundancy* of a variable. Loosely speaking, the idea is as follows. During the simulation of a write step, each writing processor will dispatch copies to various nodes on the network. For the purposes of analysing the cost of a write step, only those which travel to nodes at least a certain minimum distance q from the writing processor are counted, the others are considered free. It turns out that when considering the cost of a read step the free copies can also be ignored essentially, since the free copies of many variables will be concentrated in a small number of nodes, and consequently at most a small number of processors can read free copies without causing massive congestion. In order to formalize some of these notions, denote by Γ_u^t the set of nodes which contain a valid copy of the variable u at the start of the t^{th} stage. Let $G=(V,E)$ be the graph representing the structure of the network as described in Section 4. For each node $v \in V$, let $B(v)$ represent the set $\{x : \delta(v,x) \leq q\}$ of nodes, where q is a parameter to be specified later, and δ represents the usual graph-theoretic distance function. The *redundancy* of a variable u is defined as follows:

$$r_u^t \triangleq \min_{v \in V} |\Gamma_u^t - B(v)|.$$

The total redundancy of the entire set of variables is $R^t \triangleq \sum_{u \in U} r_u^t$, and the average redundancy R^t/m is denoted by r^t.

The arguments in [AHMP87] establish the following lemma, although the lemma is not stated explicitly in this form. Let $\Omega(g(r))$ be a lower bound for the worst-case simulation time of a read step, where r is the average redundancy at the beginning of the step.

Lemma 7 *With the above notation, the last τ stages of the adversary PRAM program can be chosen so that the total simulation time S satisfies*

$$S = \Omega(T max(\hat{r}q, g(\hat{r}))),$$

where $\hat{r} = (1/\tau)\sum_{t=1}^{\tau} r^t$, is the average redundancy over the entire simulation.

Intuitively, the term $\hat{r}q$ captures the cost of writing $\hat{r}$ copies outside a ball of radius q centred at the writing processor, and the term $g(\hat{r})$ is the cost of reading variables when there is an average of $\hat{r}$ copies per variable kept outside a given ball of radius q.

While [AHMP87,KU88] obtain a function g for all $m = \Omega(n^{2+\epsilon})$, we shall obtain a different g for all $An \leq m \leq n^2$. The two following refinements of the proof technique enable its application to the lower range of m: (i) the value of q is chosen as a function of both m and n, instead of just n, and (ii) in constructing the set of variables which are hard to read, the union of several balls is considered, rather than just one ball. The details follow.

Lemma 8 *Let d be the maximum degree of a a node in the BDN. For $An \leq m \leq n^2$, $A = 2d^4$, if at the beginning of a read step the average redundancy is r, then there exists a set of n variables that will require*

$$g(r) = \Omega((m/2n)^{1/(8r+2)})$$

time to be read by the simulating BDN.

Proof We will select the value of q to be $\lfloor \log(m/2n)/(2\log d) \rfloor - 1$. Note that since $m \geq 2d^4 n$ we have $q \geq 1$. Also, for this particular choice of q, $|B(v)| \leq d^{q+1} \leq (m/2n)^{1/2}$, for all $v \in V$. Let $U' = \{x \in U : r_x \leq 2r\}$. Clearly $|U'| \geq m/2$. Partition U' into disjoint sets X_v, $v \in V$, so that $X_v = \{x \in U' : r_x = |\Gamma_x - B(v)|\}$. (Note that some of the X_v may be empty.) We will deal separately with the case $r < 1/2$ and the case $r \geq 1/2$.

If $r < 1/2$, then all the elements of U' have redundancy 0, since each individual variable has redundancy strictly less than one and individual variables can only have integral redundancies. Consider a ball $B(v)$ such that $|X_v|$ is as large as possible, which implies that $|X_v| \geq \lceil m/2n \rceil$. Clearly reading $\lceil m/2n \rceil$ the variables in X_v will require at least

$$\frac{|X_v|}{|B(v)|} \geq \frac{\lceil m/2n \rceil}{(m/2n)^{1/2}} = \Omega((m/2n)^{1/2})$$

steps.

Now suppose that $r \geq 1/2$. Without loss of generality we assume that $r \leq (\log(m/2n) - 2)/8$, since otherwise the bound on $g(r)$ is trivially satisfied. Let $L \subseteq V$ be the set of indices of the $n/(m/2n)^{1/2+1/8r}$ largest X_v, and let $X_L = \bigcup_{v \in L} X_v$. Clearly

$$|X_L| \geq (m/2n)|L| = n(m/2n)^{1/2-1/8r}.$$

We also let $B_L = \bigcup_{v \in L} B(v)$. Now

$$|B_L| \leq |L|(m/2n)^{1/2} = \frac{n}{(m/2n)^{1/8r}}.$$

Now for any $W \subseteq V$, let

$$A_W = \{x \in X_L : \Gamma_x \subseteq B_L \bigcup W\}.$$

Recall that each variable in U' has redundancy at most $2r$ and hence at most $\lfloor 2r \rfloor$ and so a simple combinatorial argument given in [AHMP87] establishes the following: for any k with $0 \leq k \leq n$, there is a subset W of V with $|W| = k$ and

$$|A_W| \geq |X_L| \binom{n - \lfloor 2r \rfloor}{k - \lfloor 2r \rfloor} \Big/ \binom{n}{k} \geq \left(\frac{k - \lfloor 2r \rfloor + 1}{n}\right)^{\lfloor 2r \rfloor}.$$

We now choose

$$k = \left\lceil \lfloor 2r \rfloor - 1 + n(m/2n)^{-(1/2-1/8r)(1/\lfloor 2r \rfloor)} \right\rceil.$$

Straightforward manipulations making use of the assumptions $m \leq n^2$ and $r \geq (\log(m/2n) - 2)/8$ show that $k \leq n$. Use of the chosen value of k in the lower bound for $|A_W|$ yields $|A_W| \geq n$. Then we can choose the set of variables to be read in the current step as any n-element subset of A_W. Since all of the valid copies of these variables are contained in $|B_L \bigcup W| \leq |B_L| + |W|$ nodes, it follows that the simulation of that read step (in the case where $r \geq 1/2$) will require at least

$$\frac{n}{|B_L| + k} = \Omega\left(\min\left(\frac{n}{2r}, \left(\frac{m}{2n}\right)^{(\frac{1}{2}-\frac{1}{8r})\frac{1}{\lfloor 2r \rfloor}}, \left(\frac{m}{2n}\right)^{\frac{1}{8r}}\right)\right) = \Omega(\left(\frac{m}{2n}\right)^{\frac{1}{8r}}).$$

time. Combining the results for the two subcases $r < 1/2$ and $r \geq 1/2$, we obtain the stated result. $\square$

Theorem 4 now follows by using the result of Lemma 8 in Lemma 7, after a few straightforward manipulations.

A comparison of the lower bounds of Theorems 3 and 4 with the upper bound provided by Theorem 2 shows that the simulations of Section 4 are optimal whenever $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^\alpha})$ for some $\epsilon > 0$ and $\alpha > 0$, as claimed in the Introduction.

It should be noted that, for $m < n2^{\sqrt{O(\log n \log\log n)}}$, the lower bound of Theorem 4 is weaker than the straightforward $\Omega(\log n)$ lower bound based on diameter considerations.

6 Conclusions

In this paper optimal point-to-point simulations have been given for a wide range of the memory size m. It is natural to try to close the gap between upper and lower bounds for the remaining values of m. Other important problems related to deterministic PRAM simulations on BDNs are the explicit construction of generalized expanders for the memory organization, characterization of the complexity of non point-to-point memory access protocols and the development of simulations on BDNs with a simpler structure than those proposed in this paper.

References

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Simulation of idealized parallel computers on more realistic ones. *SIAM Journal of Computing*, 16(5):808–835, Nov 1987.

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(\log n)$ sorting network. In *Proceedings of the* 15^{th} *Annual Symposium on the Theory of Computing,* Boston, Massachusetts, pages 1–9, Apr 1983.

[BH85] A. Borodin and J.E. Hopcroft. Routing, merging, and sorting on a parallel model of computation. *Journal of Computers and Systems Sciences*, 30(1):130–144, Feb 1985.

[BS78] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, c-27(1):84–87, Jan 1978.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling part 1: The basic technique with application to optimal parallel list ranking in logarithmic time. *SIAM Journal of Computing*, 17(1):128–142, Feb 1988.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the* 10^{th} *Annual Symposium on the Theory of Computing, ,* San Diego, California , pages 114–118, May 1978.

[Gol82] L.M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(3):1073–1086, Oct 1982.

[KR88] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California, Berkeley, California, Mar 1988.

[KRS85] C.P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, c-34(10):965–968, Oct 1985.

[KU88] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. *SIAM Journal of Computing*, 35(4):876–892, Oct 1988.

[Lei81] F. T. Leighton. New lower bound techniques for VLSI. In *Proceedings of the 22^{nd} Symposium on the Foundations of Computer Science,* Nashville, Tennessee, pages 1–12, Oct 1981.

[Lei85] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, c-34(4):344–354, April 1985.

[LPP87a] F. Luccio, A. Pietracaprina, and G. Pucci. An efficient probabalistic simulations of PRAMs in VLSI. Technical Report S-87-8, Dipartmento di Informatica, Universita di Pisa, April 1987.

[LPP87b] F. Luccio, A. Pietracaprina, and G. Pucci. A new scheme for deterministic simulations of PRAMs in VLSI. Technical Report S-87-11, Dipartmento di Informatica, Universita di Pisa, June 1987.

[LPS86] A. Lubotzky, R. Phillips, and P. Sarnak. Explicit expanders and the ramanujan conjectures. In *Proceedings of the 18^{th} Annual Symposium on the Theory of Computing,* Berkeley, California, pages 240–246, May 1986.

[LPV81] G.F. Lev, N. Pippenger, and L.G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, c-30(2):93–100, Feb 1981.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, Nov 1984.

[NMB83] D.D. Nath, S. N. Maheshwari, and P.C.P. Bhatt. Efficient VLSI networks for parallel processing based on orthogonal trees. *IEEE Transactions on Computers*, c-32(6):569–581, June 1983.

[PU86] D. Peleg and E. Upfal. The token distribution problem. In *Proceedings of the 27^{th} Annual Symposium on the Foundations of Computer Science,* Toronto, Canada, pages 185–194, Oct 1986.

[Ran87] A.G. Ranade. How to emulate shared memory. In *Proceedings of the 28^{th} Annual Symposium on the Foundations of, Computer Science,* Los Angeles, California, pages 185–192, Oct 1987.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, 1980.

[Sny86] L. Snyder. Type architectures, shared memory and the corollary of modest potential. Technical Report TR86-03-04, Department of Computer Science, University of Washington, Mar 1986.

[SV81] Y. Shilaoch and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, Jan 1981.

[Upf84] E. Upfal. A probabalistic relation between desirable and feasible models of parallel computation. In *Proceedings of the* 16^{th} *Annual ACM Symposium on the Theory of Computing,* Washington, D.C., pages 258–265, May 1984.

[UW87] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, Jan 1987.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models which forbid it. *Journal of Algorithms*, 4(1):45–50, 1983.

[Vis84] U. Vishkin. A parallel-design distributed-implementation(PDDI) general-purpose computer. *Theoretical Computer Science*, 32(1):157–172, 1984.