HOW TO SHARE MEMORY IN A DISTRIBUTED SYSTEM

(A preliminary version)

Eli Upfal†

Stanford University
Stanford, CA 94305

Avi Wigderson‡

University of California
Berkeley, CA 94720.

ABSTRACT

We study the power of shared-memory in models of parallel computation. We describe a novel distributed data structure that eliminates the need for shared memory without significantly increasing the run time of the parallel computation. More specifically we show how a complete network of processors can deterministicly simulate one PRAM step in $O(\log n(\log\log n)^2)$ time, when both models use n processors, and the size of the PRAM's shared memory is polynomial in n. (The best previously known upper bound was the trivial $O(n)$). We also establish that this upper bounds is nearly optimal. We prove that an on-line simulation of T PRAM steps by a complete network of processors requires $\Omega(T\frac{\log n}{\log\log n})$ time.

A simple consequence of the upper bound is that an Ultracomputer (the only currently feasible general purpose parallel machine), can simulate one step of a PRAM (the most convenient parallel model to program), in $O((\log n \ \log\log n)^2)$ steps.

1. INTRODUCTION

The cooperation of n processors to solve a problem is useful only if the following two goals can be achieved:

1. Efficient parallelization of the *computation* involved.

2. Efficient *communication* of partial results between processors.

Models of parallel computation that allow processors to randomly access a large shared memory (e.g. PRAM) idealize communication and let us focus on the computation. Indeed, they are convenient to program and most parallel algorithms in the literature use them.

Unfortunately, no realization of such models seems feasible in foreseeable technologies. The only current feasible model is a *distributed system* - a set of processors (RAMs) connected by some communication network. As there is no shared memory, data items are stored in the processors' local memories, and information can be exchanged between processors only by messages. A processor can send or receive only one data item per unit time.

Let n be the number of processors in the system and m the number of data items. At every logical (e.g. PRAM) step of the computation, each processor can specify one data item it wishes to access (read or update). The execution time of the logical

† Part of this work was done while the first author was visiting U.C. Berkeley. Research supported by a Weizmann Post-Doctoral fellowship, and in part by DARPA Grant N00039-83-C-1036.

‡ Current address: IBM Research Center, IBM K55/281, 5600 Cottle rd. San Jose Ca 95193. Research supported by DARPA Grant N00039-82-C-0235.

0272-5428/84/0000/0171$01.00 © 1984 IEEE

step is at least the number of machine steps required to satisfy all these requests in parallel.

To illustrate the problem, assume $m \geq n^2$. A naive distribution of data items in local memories that uses no hashing or duplication will result in some local memory having at least n data items. Then, a perverse program can in every step force all processors to access these particular data items. This will cause an $\Omega(n)$ communication bottleneck, even if the communication network is complete. This means that *using n processors may not have an advantage over using just one, even when computation is parallelizable!*

We therefore see that it is a fundamental problem is to find a scheme to organize the data in the processors' memories such that information about any subset of n data items can be retrieved and updated in parallel as fast as possible.

This problem, called in several references 'the granularity problem of parallel memories', is discussed in numerous papers. The survey paper by Kuck [Ku] mentions 14 of them, all solving only part of the problem as they tailor data organization to particular families of programs. For a general purpose parallel machine, such as the NYU-Ultracomputer (Gottlieb et al. [GGK]), the PDDI machine (Vishkin [Vi]), and others, one would clearly like a general purpose organization scheme, that will be the basis of an automatic (compiler-like) efficient simulation of any program written for a shared memory model by a distributed model.

If the number of data items, m, is roughly the number of processors, n, then the fast parallel sorting algorithms, [AKS], and [Le], solve the problem. However, we argue that in most applications this is not the case. For example, in distributed databases, typically thousands of processors will perform transactions on billions of data items. Also, in parallel computation, appetite increases with eating; the more processors we can have in a parallel computers, the larger the problems we want to solve.

In a probabilistic sense, the problem is solved even for $m \gg n$. Melhorn and Vishkin [MV] propose distributing the data items using universal hashing. This guarantees that one parallel request for n data items will be satisfied in expected time $O(\frac{\log n}{\text{loglog } n})$. Upfal [U] presents a randomized distributed data structure that guarantees execution of any sequence of T parallel requests in $O(T \log n)$ steps with probability tending to 1 as n tends to ∞.

By contrast, if $m \gg n$, no deterministic upper bound better than the trivial $O(n)$ in known. Melhorn and Vishkin [MV], who provide an extensive study of this problem, suggest keeping several copies of each data item. In their scheme, if all requests are for 'read' instructions, the 'easiest' copy will be read, and all requests will be satisfied in time $O(kn^{1-\frac{1}{k}})$ where $m = n^k$. When update instructions are present, they cannot guarantee time better than $O(n)$, as all copies of a data item have to be updated.

In this paper we present a data organization scheme that guarantees a worst case upper bound of $O(\log n(\text{loglog } n)^2)$, for any m polynomial in n. Our scheme also keeps several copies of each data item. *The major novel idea is that not all of these copies have to be updated - it suffices that a majority of them are.* This idea allows the 'read' and 'update' operations to be handled completely symmetrically, and still allows processors to access only the 'easiest' majority of copies.

Our scheme is derived from the structure of a concentrator-like bipartite graph [Pi]. It is a long standing open problem to construct such graphs explicitly. However, a random graph from a given family will have the right properties with probability 1. As in the case of expanders and superconcentrators (e.g. [Pi]) this is not a serious drawback, as the randomization is done only once - when constructing the system.

One immediate application of the upper bound is to the simulation of ideal parallel computers by feasible ones. Since a bounded degree network

can simulate a complete network in $O(\log n)$ steps ([AKS], [Lei]), a typical simulation result which is derived from our upper bound is the following: *Any n-processors PRAM program that runs in T steps can be simulated by a bounded degree network of n processors (Ultracomputer [Sc]) that runs in deterministic time $O(T(\log n)^2(\log\log n)^2)$ steps.*

The scheme we propose has very strong fault-tolerance properties, which are very desirable in distributed systems. It can sustain up to $O(\log n)$ *maliciously chosen* faults and up to $(1-\epsilon)n$ random ones without any information or efficiency loss.

Finally we derive lower bounds for the efficiency of memory organizations schemes. We consider schemes that allow many copies of each data item, as long as each memory cell contains one copy of one data item. The redundancy of such a scheme is the average number of copies per data item.

Our lower bound gives a trade-off between the efficiency of a scheme and its redundancy. If the redundancy is bounded, we get an $\Omega(n^{\epsilon})$ lower bound on the efficiency. This result partially explains why previous attempts, that considered only bounded redundancy failed [MV], and why our scheme uses $O(\log n)$ copies per data item.

We also derive an $\Omega(\frac{\log n}{\log\log n})$ unconditional lower bound on the efficiency - almost matching our $O(\log n(\log\log n)^2)$ upper bound. This lower bound is the first result that separates models with shared memory from the feasible models of parallel computation that forbid it.

2. DEFINITIONS

To simplify the presentation, we shall concentrate on simulation of the *weakest* shared memory model - the EREW (Exclusive-Read Exclusive-Write) PRAM, by the *strongest* distributed system - a model equivalent to a complete network of processors. In the final version we shall fill in the details for a simulation of a the strongest PRAM model (the CRCW PRAM) by a bounded degree network of processors (an Ultracomputer).

An EREW PRAM consists of n processors $P_1, \ldots, P_n$, (RAMs) which operate synchronously on a set U of m shared variables (or data items). In a single PRAM step, a processor may perform some internal computation or access (read or update) one data item. Each data item is accessed by at most one processor at each step.

An MPC (Module Parallel Computer) [MV] consists of n synchronous processors, $P_1, \ldots, P_n$, and n *memory modules*, $M_1, \ldots, M_n$. Every module is a collection of memory *cells*, each of which can store a value of *one* data item.

In each MPC step, a processor may perform some internal computation, or *request* an access to a memory cell in one of the memory modules. From the set of processors trying to access a specific module, exactly one will (arbitrarily) be granted permission. Only this processor can consequently access (read or update) exactly one cell in this module.

The task of the MPC is to execute a PRAM program. This program is a sequence of instructions I_t, $t=1, \ldots, T$. Each instruction is a vector of n sub-instructions, specifying the task of each of the n processors in this instruction. The sub-instruction of the processor P_i can be either to execute some local computation, or to access (read or update) a data item (shared variable) $u_i \in U$. In the case of an update, a new value v_i is also assigned.

For the simulation, each data item $u \in U$ may have several 'physical addresses' or *copies* in several memory modules of the MPC, not all of which are necessarily updated. Let $\Gamma(u)$ be the set of modules containing a copy of u. We sometimes refer to $\Gamma(u)$ also as the set of copies of u.

The essence of the simulation is captured by an *organization scheme* S. It consists of an assignment of sets $\Gamma(u)$ to every $u \in U$, together with a *protocol* for execution of read/update instructions (e.g. how many copies to access, in what order, etc.). Both the assignment and the protocol may be time

dependent.

The *efficiency* of a given scheme S is the worst case number of parallel MPC steps required to execute one PRAM instruction (according to the protocol). Note that the worst case is taken over all possible n-subsets of the set of data items U, and over all possible access patterns (read/write).

The notion of an *updated copy* of a data item $u \in U$ is very intuitive. At any step during the simulation, a copy $\gamma(u) \in \Gamma(u)$ is updated if it contains the value the data item u would have at this step if the program were executed by a PRAM.

Let the number of updated copies of u (at some step) be $\bar{\Gamma}(u)$.

A scheme is *consistent* if at every step, for every $u \in U$, $|\bar{\Gamma}(u)| \geq 1$

We define the *redundancy* $r(S)$ of S (at this step), to be $r(S) = \frac{\sum_{u \in U} |\Gamma(u)|}{|U|}$, the average number of copies of a data item in the scheme at this step.

3. UPPER BOUNDS

Our main results are given below.

THEOREM 3.1: *If m is polynomial in n then there exists a consistent scheme whose efficiency is $O(\log n (\log\log n)^2)$.*

Theorem 3.1 is a special case of:

THEOREM 3.2: *There is a constant $b_0 > 1$, s.t. for every $b \geq b_0$ and c satisfying $b^c \geq m^2$, there exists a consistent scheme with efficiency $O(b[c\ (\log c)^2 + \log n \log c])$.*

In our scheme, every item $u \in U$ will have exactly $2c-1$ copies, i.e. $|\Gamma(u)| = 2c-1$. Each copy of a data item is of the form <value, time-stamp>. We will show later how to locate the copies of each data item.

The protocol for accessing data item u at the t^{th} instruction is as follows:

1. To update u, access *any* c copies in $\Gamma(u)$, update their values and set their time-stamp to t.

2. To read u, access *any* c copies in $\Gamma(u)$, and read the value of the copy with the latest time-stamp.

This protocol completely symmetrizes the roles of read and update instructions, and gives a new application to the majority rule used in [Th] for concurrency control of distributed databases.

LEMMA 3.1: *Assume that before the execution of the first instruction all copies of each $u \in U$ contain identical values and are time-stamped '0'. Then the following invariant is maintained after the execution of each instruction:*

Every $u \in U$ has at least c copies that agree in value, and are time-stamped with the index of the last instruction in which a processor updated u.

The proof follows simply from the fact that every two c-subsets of $\Gamma(u)$ have a non-empty intersection, and therefore the protocol for read access, together with the invariant above, guarantee that a 'correct' value will be read.

COROLLARY 3.1: *The scheme is consistent.*

Let u_i be the data item requested by P_i, $1 \leq i \leq n$, at this step. Recall that c copies in $\Gamma(u_i)$ have to be accessed in order to read or update u_i. Denote the j^{th} copy in $\Gamma(u)$ by $\gamma_j(u)$. During the simulation of this instruction, we will say that $\gamma_j(u_i)$ is *alive* if this copy was not accessed yet. Also, say that u_i is *alive* if at least c copies in $\Gamma(u_i)$ are still alive. Notice that a request for u_i is satisfied when u_i is no longer alive. At this point the protocol for accessing u_i can terminate.

We are ready now to describe the algorithm. We start with an informal description.

Assume that the task of P_i is either to read u_i or to update its value to v_i. Processors will help each other to access these data items according to the protocol. It turns out to be efficient if at most $\frac{n}{2c-1}$ data items are processed at a time. Therefore, we

shall partition the set of processors into $k = \frac{n}{2c-1}$ groups, each of size $2c-1$. There will be $2c$ phases to the algorithm. In each of the phases, each group will work, in parallel, to satisfy the request of one of its members. This will be done as follows: The current distinguished member, say P_i, will broadcast its request (access u_i, and the new value v_i in case of a write request) to the other members of its group. Each of them will repeatedly try to access a fixed distinct copy of u_i. After each step, the processors in this group will check whether u_i is still alive, and at the first time it is not alive (i.e. at least c of its copies were accessed), this group will stop working on u_i. If the request was for a read, the copy with the latest time stamp will be computed and sent to P_i.

Each of the first $2c-1$ phases will have a time limit, that may stop the processing of the k data items while some are still alive. However, we will show that at most $\frac{k}{2c-1}$ from the k items processed in each phase will remain alive. Hence, after $2c-1$ phases at most k items will remain. These will be distributed, using sorting, one to each group. The last phase, that has no time limit, will handle them till all are processed.

For the formal presentation of the algorithm, let $P_{(l-1)(2c-1)+i}$, $i=1,\ldots,2c-1$ denote the processors in group l, $l=1,\ldots,k$, $k=\frac{n}{2c-1}$. The structure of the j^{th} copy of the data items u is, as before, $\langle value_j(u), time-stamp_j(u)\rangle$

```
Phase (i,time_limit):
begin
  l:=⌈ processor_no / (2c-1) ⌉
  f:=(l-1)(2c-1);
  P_{f+i} broadcast its request
  (read(u_{f+i}) or update(u_{f+i},v_{f+i}))
  to P_{f+1},...,P_{f+2c-1};
  live(u_{f+i}):=true;
  count:=0;
  while live(u_{f+i}) and count < time_limit do
      count := count+1;
      P_{f+j} tries to access γ_j(u_{f+i});
      if permission granted then
          if read request then
              read ⟨value_j(u_{f+i}), time_stamp_j(u_{f+i})⟩;
          else (update request)
              ⟨value_j(u_{f+i}), time_stamp(u_{f+i})⟩ := ⟨v_{f+i},t⟩
      if less than c copies of u_{f+i} are still alive then
          live(u_{f+i}):=false;
  end while
  if a read request then
      find and send to P_{f+i} the value with the
      latest time_stamp;
end  Phase i;


The algorithm:
begin
    for i=1 to 2c-1 do
        run Phase(i,log_η 4c);
    [ for a fixed η (to be calculated later),
      there are at most k live request at this
      point of the algorithm]
    sort the k' live requests and route them to
    the first processors in the k' first groups,
    one to each processor;
    run Phase(1,log_η n);
end algorithm.
```

Consider now one iteration of the **while** loop in an execution of a phase in the algorithm. The number of requests sent to each module during the execution of this iteration is equal to the number of live copies of live data item this module contains. The module may receive all the requests together and therefore process only one of them, thus we can only guarantee that the number of copies processed in each iteration of the while loop is equal to the number of memory modules containing live copies of data items that were alive before this iteration.

Let $A \subseteq U$ denote the set of live data items at the start of a given iteration. Let the set $\Gamma'(u) \subseteq \Gamma(u)$ denote the set of live copies of $u \in U$ at this time. Since u is alive, $|\Gamma'(u)| \geq c$. The number of live copies at the start of this iteration is given by $\sum_{u \in U} |\Gamma'(u)|$. The number of memory modules containing live copies of live data items, and thus a lower bound for the number of copies processed during this iteration is given by $|\Gamma'(A)| = |\bigcup_{u \in A} \Gamma'(u)|$.

We first show that a good organization scheme can guarantee that $|\Gamma'(A)|$ is not too small.

LEMMA 3.2: For every $b \geq 4$, *if*

$m \leq (\frac{b}{(2e)^4})^{\frac{c}{2}}$ *then there is a way to distribute the* $2c-1$ *copies of each of the* m *shared data items among the* n *modules s.t. before the start of each iteration of the 'while' loop* $|\Gamma'(A)| \geq \frac{|A|}{b}(2c-1)$.

PROOF: It is convenient to model the arrangement of the copies among the memory models in terms of a bipatite graph $G(U,N,E)$, where U represents the set of m shared data items, N the set of n memory modules, and $\Gamma(u)$, the set of neighbors of a vertex $u \in U$ represents the set of memory modules storing a copy of the data item u. We use a probabilistic construction in order to prove the existence of a good memory allocation.

Let $G_{m,n,c}$ be the probabilistic space of all bipartite graphs $G(U,N,E)$ s.t. $|U| = m$, $|N| = n$ and the degree of each vertex $u \in U$ is $2c-1$. Give all graphs in the space equal probability.

Say that a graph $G(U,N,E) \in G_{m,n,c}$, is 'good' if for all possible choices of the sets $\{\Gamma'(u) : \Gamma'(u) \subseteq \Gamma(u), |\Gamma'(u)| \geq c, u \in U\}$ and for all $A \subseteq U$, $|A| \leq \frac{n}{2c-1}$, the inequality $|\Gamma'(A)| \geq \frac{1}{b}(2c-1)|A|$ holds. This condition captures the property that for any set A of live data items, no matter which of their copies are still alive, the set of all the copies of data items in A are distributed among at least $\frac{1}{b}(2c-1)|A|$ memory modules.

$$Pr\{ G \in G_{m,n,c} \text{ is not 'good'}\} \leq$$

$$\sum_{q \leq \frac{n}{(2c-1)}} \binom{m}{q} \left[\frac{n}{\frac{q}{b}(2c-1)} \right] \left[\binom{2c-1}{c} \right]^q \left[\frac{(2c-1)q}{bn} \right]^{qc}$$

$$= o(\frac{1}{n}), \text{ for } m \leq (\frac{b}{(2e)^4})^{\frac{c}{2}}, \text{ and } b \geq 4$$

In what follows we assume that the algorithm is applied to a memory organization that possesses the properties proven in Lemma 3.2.

LEMMA 3.3: *If the number of live items at the beginning of a phase is* w $(\leq k)$, *then after the first* s *iterations of the* while loop at most $2(1-\frac{1}{b})^s w$ *live copies remain.*

PROOF: At the beginning of a phase there are w live items, and all their copies are alive, so there is a total of $(2c-1)w$ live copies. By lemma 3.2, after s iterations, the number of live copies remaining is $\leq (1-\frac{1}{b})^s(2c-1)w$. Since $|\Gamma'(u)| \geq c$ for each live item, these can be the live copies of at most $(1-\frac{1}{b})^s \frac{2c-1}{c} w \leq 2(1-\frac{1}{b})^s w$ items.

COROLLARY 3.2: *Let* $\eta = (1-\frac{1}{b})^{-1}$.

1. After the first $\log_\eta(4c-2)$ *iterations of the* while *loop in a phase, at most* $\frac{k}{2c-1}$ *live items remain alive (establishes the fact that the last phase has to process no more than k requests).*

2. After $\log_\eta 2k \leq \log_\eta n$ *iterations in a phase, no live items remain (establishes the correctness of the last phase).*

To complete the analysis, observe that each group needs during each phase to perform the following operations: broadcast, maximum (for finding the latest time stamp) and summation (testing whether u_i is still alive). Also, before the last phase, all the requests that are still alive are sorted.

LEMMA 3.4: *Any subset of* p *processors of the MPC, using only* p *of the memory modules, can perform maximum, summation, and sorting of* p *elements, and can broadcast one message in* $O(\log p)$ *steps.*

PROOF: The only non-trivial case is the sorting and this can be done using Leighton's sorting algorithm [Le].

THEOREM 3.2: *For every* $b \geq 4$, *if* $m \leq (\frac{b}{(2e)^4})^{\frac{c}{2}}$, *then there exists a memory organization scheme with efficiency*

$$O(bc(\log c)^2 + b(\log n)(\log c)).$$

PROOF: In each iteration of the while loop each processor performs up to one access to a memory module, and each group of $2c-1$ processors computes the summation and the maximum of up to $2c-1$ elements. Thus, each iteration takes $O(\log c)$ steps. The first $2c-1$ phases perform $\log_\eta c$ iteration each, therefore together they require

$$O\left(\frac{(2c-1)(\log c)^2}{\log \eta}\right)$$

parallel steps.

The sorting before the last phase takes $O(\log n)$ steps, and the last phase consists of $O(\log_\eta n)$ while iterations, hence requires $O((\log_\eta n)(\log c))$ steps. As $\log \eta = \log (1-\frac{1}{b})^{-1} = O(\frac{1}{b})$ the total number of steps is $O(bc(\log c)^2 + b(\log n)(\log c))$

We conclude this section with some remarks:

1. **Fault tolerance:** A variant of our scheme, in which every processor tries to access $(2-\varepsilon)c$ copies rather than c, guarantees that even if up to $(1-2\varepsilon)c$ of the copies of each data item are destroyed by an adversary, no information or efficiency loss will occur.

2. **Explicit construction:** The problem of explicit construction of a good graph in $G_{m,n,c}$ remains open. This problem is intimately related to the long standing open problem of explicit construction of (m,n)-concentrators (e.g. [DDPW]), when $m \gg n$.

4. LOWER BOUNDS

The fast performance of the organization scheme presented above depends on having at least $O(\log n)$ updated copies of each data item, distributed among the modules. A natural question to ask here is whether this redundancy in representing the data items in the memory is essential. In this section we give a positive answer to this question. We prove a lower bound relating the efficiency of any organization scheme to the redundancy in it. Using this trade-off we derive a lower bound for any on-line simulation of ideal models for parallel computation with shared memory by feasible models that forbid it.

We assume without loss of generality that each processor of the MPC has only a constant number, d, of registers for internal computation. (This is no restriction as P_i can use M_i as its local memory). In what follows we consider only schemes that allow a memory cell or an internal register to contain one value of one data item (no encoding or compression are allowed).

THEOREM 4.1: *The efficiency of any organization scheme with m data items, n memory modules and redundancy r is $\Omega((\frac{m}{n})^{\frac{1}{2r}})$.*

PROOF: Let S be a scheme with m data items, n modules, and redundancy r. If the efficiency of the scheme S is less than some number h then there is no set of n data items such that all their updated copies are concentrated in a set of $h^{-1}n$ modules. Otherwise, it would have taken at least h steps to read these data items, since only one data item can be read per step at each module.

Recall that r is the average number of updated copies of a data items in the scheme. Therefore, there are at least $\frac{m}{2}$ data items with no more than $2r$ copies. At most dn out of these items appear in the internal registers of processors.

There are $\left\lfloor \binom{n}{h^{-1}n} \right\rfloor$ sets of $h^{-1}n$ modules, and each set can store all the copies of no more than $n-1$ data items. If a data item has at most $2r$ copies then all its copies are included in at least $\left\lfloor \binom{n-2r}{h^{-1}n-2r} \right\rfloor$ sets of $h^{-1}n$ modules. Counting the total number of data items with at most $2r$ copies that are stored by the scheme, we get

$$\frac{\left\lfloor \frac{n}{h^{-1}n} \right\rfloor (n-1)}{\left\lfloor \frac{n-2r}{h^{-1}n-2r} \right\rfloor} \geq \frac{m}{2} - dn$$

which implies $h = \Omega((\frac{m}{n})^{\frac{1}{2r}})$

Using the result of theorem 4.1 we can now derive a lower bound for the on-line simulation of a PRAM program by the MPC model.

In an on-line simulation, the MPC is required to finish executing the t^{th} PRAM instruction before reading the $t+1^{th}$. Of course it can perform other operations as well during the execution of the t^{th} instruction, but these can not depend on future instructions.

We shall assume, w.l.o.g., that the initial value of all data items (and all MPC memory cells) are zero. Since we have m data items and n processors, it makes sense to consider PRAM programs of length $\Omega(\frac{m}{n})$, otherwise some items were redundant.

THEOREM 4.2: *Any on-line simulation of T steps of a PRAM with n processors and m shared variables on an MPC with n processors and n memory modules requires $\Omega(T \frac{\log n}{\log\log n})$ parallel MPC steps.*

PROOF: We will construct a PRAM program of length T as follows: The first $\frac{m}{n}$ instructions will assign new values to all the data items. Subsequent instructions will alternate between a *hard read* and a *hard write* instructions.

Consider the redundancy r_t of the scheme after the execution of the t^{th} instruction. A *hard read* instruction will essentially implement theorem 4.1 - it will assign processors to read n items that all of their updated copies are condensed among a small number of modules. A *hard write* instruction will assign new values to the n items with the highest number of updated copies. Clearly there are always n data items with at least r_t updated copies (as $m \gg n$)

For simplicity consider each pair of a hard read followed by a hard write as one PRAM instruction. Let s_t be the number of MPC steps used while executing the t^{th} instruction. For the first $\tau = \frac{m}{n}$ instructions, at most $\sum_{t=1}^{\tau} s_t$ memory locations were accessed, and hence

$$r_\tau \leq \frac{n}{m} \sum_{t=1}^{\tau} s_t. \qquad (1)$$

Recall that r_τ is the redundancy when we start alternating reads and writes. Let $t > \tau = \frac{m}{n}$. By theorem 4.1, at least $\tau^{\frac{1}{2r_{t-1}}} = \beta_{t-1}$ of the s_t MPC steps were used by each processor to execute the hard read instruction. Hence, at most $(s_t - \beta_{t-1})n$ cells were accessed for write instructions. Also, the value of n data items, with $\geq r_{t-1}$ updated copies each, was changed, thus, we have

$$r_t \leq r_{t-1} + (s_t - \beta_{t-1} - r_{t-1})\frac{n}{m},$$

for $t = \tau+1, \ldots, T$.

Summing all these inequalities we get

$$\sum_{t=\tau+1}^{T} r_t \leq \sum_{t=\tau+1}^{T} r_{t-1} + \frac{n}{m} \sum_{t=\tau+1}^{T} (s_t - \beta_{t-1} - r_{t-1}).$$

Using simple manipulation we get:

$$\frac{m}{n} r_\tau + \sum_{t=\tau+1}^{T} s_t \geq \frac{m}{n} r_T + \sum_{t=\tau+1}^{T} (\beta_{t-1} + r_{t-1}),$$

and using (1),

$$\sum_{t=1}^{T} s_t = \sum_{t=1}^{\tau} s_t + \sum_{t=\tau+1}^{T} s_t \geq \frac{m}{n} r_\tau + \sum_{t=\tau+1}^{T} s_t \geq$$

$$\frac{m}{n} r_T + \sum_{t=\tau}^{T-1} (\beta_t + r_t) \geq \sum_{t=\tau}^{T-1} \beta_t + r_t$$

Where $\sum_{t=1}^{T} s_t$ is the total simulation time.

Let $\tilde{r} = \frac{1}{(T - \frac{m}{n})} \sum_{t=\tau}^{T-1} r_t$ be the average

redundancy in the last $T-\frac{m}{n}$ steps. Notice that $\beta(r) = (\frac{m}{n})^{\frac{1}{2r}}$ is a convex function in r, for $r \geq 0$. Hence by Jensen's inequality [RV,211-216],

$$\sum_{t=\tau}^{T-1} \beta_t = \sum_{t=\tau}^{T-1} (\frac{m}{n})^{\frac{1}{2r_t}} \geq (T-\frac{m}{n})(\frac{m}{n})^{\frac{1}{2\tilde{r}}}.$$

Hence,

$$\sum_{t=1}^{T} s_t \geq (T-\frac{m}{n})(\tilde{r}+(\frac{m}{n})^{\frac{1}{2}\tilde{r}}) =$$

$$\Omega((T-\frac{m}{n})\frac{\log \frac{m}{n}}{\text{loglog}\frac{m}{n}}.$$

For $m \geq n^{1+\epsilon}$, and $T \geq (1+\epsilon)\frac{m}{n}$, the simulation time is $\Omega(T\frac{\log n}{\text{loglog } n})$.

5. CONCLUSIONS

We describe a novel scheme for organizing data in a distributed system, that admits highly efficient retrieval and update of information in parallel.

This paper concentrates on applications to synchronized models of parallel computation, and specifically to the question of the relative power of deterministic models with and without shared memory. Quite surprisingly, we show that these two families of models are nearly equivalent in power, and therefore we justify the use of shared memory models in the design of parallel algorithms.

There are other applications of our scheme that we did not pursue in this paper. One application is to probabilistic simulation. An interesting open problem, which we are considering, is whether our scheme can improve the probabilistic results in [MV] or [U].

Another application we did not pursue here is to asynchronous systems. Although a similar scheme was suggested in this context [Th], we believe that the potential of this idea was not fully exploited there, and we plan to continue research in this direction. However, we believe that the new notion of consistency suggested by our scheme can have a major impact on the theory and design of such systems, in particular for distributed database systems. We intend to continue research in this direction.

Acknowledgments:

We thank Dick Karp for helpful discussions, and Edna Wigdersn, Oded Goldreich, and David Shmoys for their comments on earlier version of this paper.

REFERENCES

[AKS] M. Ajtai, J. Komlos and E. Szemeredi. An $O(\log n)$ sorting network. *Proc. of the Fifteenth ACM STOC*, 1983. 1-9.

[AIS] B. Awerbuch, A. Israeli and Y. Shiloach. Efficient simulation of PRAM by Ultracomputer. Preprint, Technion, Haifa, Israel. 1983.

[DDPW] D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson. Superconcentrators, generalizers and generalized connectors with limited depth. *Proc. of the Fifteenth ACM STOC*, 1983. 42-51.

[GG] O. Gabber and Z. Galil. Explicit construction of linear-sized superconcentrators. J. Comp. and Sys. Sci. 22, 1981. 407-420.

[GGK] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing a MIMD shared memory parallel machine. *IEEE Trans. on Comp.* C-32, 2, 1983. 175-189.

[Ku] D.J. Kuck. A survey of parallel machines organization and programming. *Computer Surveys*, Vol 9, No. 1, 1977. 29-59.

[Le] T. Leighton. Tight bounds on the complexity of parallel sorting. *Proc of the Sixteenth ACM STOC*, 1984. 71-80.

[MV] K. Melhorn and U Vishkin. Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories. Ninth Workshop on Graph Theoretic Concepts in Computer Science, Fachbereich Mathematic, Universitat Osnabruck, June 1983.

[Pi] N. Pippenger. Superconcentrators. *SIAM J. on Computing*, 6, 2, 1977. 298-304.

[RV] A.W. Roberts and D.E. Varberg. *Convex Analysis*. Academic Press, New York, London 1973.

[Sc] J. T. Schwartz. Ultracomputers. *ACM TOPLAS* 2 (1980) 484-521.

[Th] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy database. *ACM Tran. on Database Systems*. 4 (1979) 180-209.

[Vi] U. Vishkin. A parallel-design distributed-implementation general-purpose computer. Preprint, Courant Institute, New York University. 1983. To appear in J. TCS.

[U] E. Upfal. A probabilistic relation between desirable and feasible models of parallel computation. *Proc. of Sixteenth ACM STOC* 1984. 258-265.