How to Share Memory in a Distributed System

ELI UPFAL

Stanford University, Stanford, California

AND

AVI WIGDERSON

IBM Research Laboratory, San Jose, California

Abstract. The power of shared-memory in models of parallel computation is studied, and a novel distributed data structure that eliminates the need for shared memory without significantly increasing the run time of the parallel computation is described. More specifically, it is shown how a complete network of processors can deterministically 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 $O(n)$). It is established that this upper bound is nearly optimal, and it is proved that an on-line simulation of T PRAM steps by a complete network of processors requires $\Omega(T(\log n/\log\log n))$ time.

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

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; *relations among modes*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Feasible model, parallel algorithm, parallel computation, simulation between models, theoretical model

1. *Introduction*

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

(1) efficient parallelization of the *computation* involved,
(2) efficient *communication* of partial results between processors.

A preliminary version of this work appeared in the *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, Singer Island, Florida, October 1984. © 1984 IEEE. Any portions of this paper that appeared in the original version are reprinted with permission.

Part of this work was done while the authors were visiting the University of California at Berkeley.

The research of E. Upfal was supported by a Weizmann postdoctoral fellowship and in part by DARPA grant N00039-83-C-1036. The research of A. Wigderson was supported by DARPA grant N00039-82-C-0235.

Authors' current addresses: E. Upfal, Department of Computer Science, Stanford University, Stanford, CA 94305-2095; A. Wigderson, Mathematical Sciences Research Institute, 1000 Centennial Road, Berkeley, CA 94720.

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. A feasible model is a *distributed system*—a set of processors (RAMs) connected by some communication network. Since 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 step is at least the number of machine steps required to satisfy all these requests in parallel.

To illustrate the problem, assume that $m \geq n^2$. 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 see, therefore, that it is a fundamental problem to find a scheme that organizes 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*, has been discussed in numerous papers. The survey paper by Kuck [7] mentions 14 of these papers, all solving only part of the problem as they tailor data organization to particular families of programs. For general-purpose parallel machines, such as the NYU-Ultracomputer [6], the PDDI machine [15], and others, one would clearly like a general-purpose organization scheme, which would 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 equal to the number of processors n, then the fast parallel sorting algorithms, [1, 8] 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 [9] propose distributing the data items using universal hashing. This guarantees that one parallel request for n data items will be satisfied in expected time $O(\log n/\log\log n)$. Upfal [14] 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 ∞.

In contrast, if $m \gg n$, no deterministic upper bound better than the trivial $O(n)$ is known. Melhorn and Vishkin [9], who provide an extensive study of this problem, suggest keeping several copies of each data item. In their scheme, if all requests are "read" instructions, the "easiest" copy will be read, and all requests will be satisfied in time $O(kn^{1-1/k})$ 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 $O(\log n(\log\log n)^2)$, for any m polynomial in n. Our scheme also keeps several copies of each data item. *The major novel idea is that all of these copies need not be updated—it suffices to update a majority of them.* 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 [10]. 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., [10]), this is not a serious drawback, since the randomization is only done once—when the system is constructed.

One immediate application of the upper bound is the simulation of ideal parallel computers by feasible ones. Since a bounded degree network can simulate a complete network in $O(\log n)$ steps [1, 8], a typical simulation result that 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* [12]) *that runs in deterministic time $O(T(\log n)^2\log\log n)$ steps.*

The scheme we propose has very strong fault-tolerant 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 a 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, which considered only bounded redundancy, failed [9], and why our scheme uses $O(\log n)$ copies per data item.

We also derive an $\Omega(\log n/\log\log n)$ 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 concentrate on simulation of the *weakest* shared memory model—the Exclusive-Read, Exclusive-Write (EREW) PRAM, by the *strongest* distributed system—a model equivalent to a complete network of processors. Extending this result to a simulation of the strongest PRAM model (the Concurrent-Read, Concurrent-Write (CRCW) PRAM) by a bounded degree network of processors (an Ultracomputer) requires standard techniques, which we shall mention at the end of Section 3.

The computational power of individual processors in the following definitions is of no importance, as long as it is the same for both the simulating and the simulated machines.

An EREW PRAM consists of n processors $P_1, \ldots, P_n$ (RAMs) that 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. A formal definition can be found in [3].

A module parallel computer (MPC) [9] 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 subinstructions, specifying the task of each of the n processors in this instruction. The subinstruction 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). Both the assignment and the protocol may be time dependent.

A scheme is *consistent* if, after the simulation of every PRAM instruction I_t, a protocol to read item u terminates with the value assigned to u by the latest previous write instruction.

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).

Finally, we define the *redundancy* $r(S)$ of S (at this step) to be $r(S) = \sum_{u \in U} |\Gamma(u)| / |U|$, 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 with efficiency $O(\log n (\log\log n)^2)$ and redundancy $O(\log n)$.*

Theorem 3.1 is a special case of

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

In our scheme, every item $u \in U$ will have exactly $2c - 1$ copies, that is, $|\Gamma(u)| = 2c - 1$. Each copy of a data item is of the form $\langle$value, timestamp$\rangle$; before the execution of the first instruction, all the copies of each data item have identical values and are timestamped "0." We show later how to locate the copies of each data item.

The protocol for accessing data item u at the tth instruction is as follows:

(1) To update u, access *any* c copies in $\Gamma(u)$, update their values and set their timestamp to t.
(2) To read u, access *any* c copies in $\Gamma(u)$ and read the value of the copy with the latest timestamp.

This protocol completely symmetrizes the roles of read and update instructions, and gives a new application to the majority rule used in [13] for concurrency control of distributed databases.

LEMMA 3.1. *The scheme is consistent.*

PROOF. We say that a copy $\gamma_j(u)$ of the data item u is updated after step t if it possesses the value assigned to u by the latest previous write instruction.

From the fact that each two c-subsets of $\Gamma(u)$ have a nonempty intersection, it follows by induction on t that, when the simulation of every instruction I_t terminates, at least c copies of every data item u are updated, these copies have the latest timestamp among all the copies of u, and a read u protocol will return their value. □

Let u_i be the data item requested by P_i, $1 \le i \le n$, 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 jth copy in $\Gamma(u)$ by $\gamma_j(u)$. During the simulation of this instruction, we say that $\gamma_j(u_i)$ is *alive* if this copy has not yet been accessed. 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 now ready 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 access these data items according to the protocol. It turns out to be efficient if at most $n/(2c-1)$ data items are processed at a time. Therefore, we partition the set of processors into $k = n/(2c-1)$ 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 is 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 the first time it is not alive (i.e., at least c of its copies have been accessed), this group will stop working on u_i. If the request is for a read, the copy with the latest timestamp will be computed and sent to P_j.

Each of the first $2c-1$ phases will have a time limit, which may stop the processing of the k data items while some are still alive. However, we show that at most $k/(2c-1)$ 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, which has no time limit, will handle all of them until they are all 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, $l = 1, \ldots, k$, $k = n/(2c-1)$. The structure of the jth copy of the data items u is, as before, $\langle value_j(u), time-stamp_j(u)\rangle$.

```
Phase ($i$, $time\_limit$):
begin
  $l := \left\lceil \frac{processor\_no}{2c - 1} \right\rceil$
  $f := (l - 1(2c - 1)$;
  $P_{f+1}$ broadcast its request
  [read($u_{f+i}$) or update($u_{f+i}$, $v_{f+i}$)]
  to $P_{f+1}, \ldots, P_{f+2c-1}$;
  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 $\gamma_j(u_{f+i})$;
    if permission granted then
      if read request then
      read$\langle value_j(u_{f+i}), time\_stamp_j(u_{f+i})\rangle$
      else (update request)
    $\langle value_j(u_{f+i}), time\_stamp(u_{f+i})\rangle := \langle v_{f+i}, t\rangle$;
    if fewer 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($i$, $\log_\eta 4c$);
  [for a fixed $\eta$ (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, $\log_\eta n$);
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 the k live data item handled during this iteration that 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 the k data items handled during this iteration that were alive before the execution of 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 $m \leq (b/(2e)^4)^{c/2}$, then there is a way to distribute the $2c - 1$ copies of each of the m shared data items among the n modules such that before the start of each iteration of the* **while** *loop* $|\Gamma'(A)| \geq (|A|/b)(2c-1)$.

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 represents 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 $G(U, N, E)$ such that $|U| = m$, $|N| = n$, and the degree of each vertex $u \in U$ is $2c - 1$. Let all graphs in the space have equal probability.

A graph $G(U, N, E) \in G_{m,n,c}$, is said to be "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 $A \subseteq U$, $|A| \leq n/(2c-1)$, the inequality $|\Gamma'(A)| \geq (1/b)(2c-1)|A|$ 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 $(1/b)(2c-1)|A|$ memory modules.

$$\Pr\{G \in G_{m,n,c} \text{ is not "good"}\} \leq \sum_{q \leq n/(2c-1)} \binom{m}{q} \binom{n}{(q/b)(2c-1)} \left(\binom{2c-1}{c}\right)^{q} \left(\frac{(2c-1)q}{bn}\right)^{qc}$$
$$= o\left(\frac{1}{n}\right),$$

for $m \leq (b/(2e)^4)^{c/2}$, and $b \geq 4$. □

In what follows we assume that the algorithm is applied to a memory organization that possesses the properties proved 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 $2(1 - 1/b)^s w$ live copies remain.*

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

COROLLARY 3.1. *Let $\eta = (1 - 1/b)^{-1}$.*

(1) *After the first $\log_\eta(4c - 2)$ iterations of the* **while** *loop in a phase, at most $k/(2c-1)$ 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 finding (for finding the latest timestamp), 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 finding, summation, and sorting of p elements and can broadcast one message in $O(\log p)$ steps.*

PROOF. The only nontrivial case is the sorting and this can be done using Leighton's sorting algorithm [8]. □

THEOREM 3.3. *For every $b \geq 4$, if $m \leq (b/(2e)^4)^{c/2}$, 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 requiring $O((\log_\eta n)(\log c))$ steps. Since $\log \eta = \log(1 - 1/b)^{-1} = O(1/b)$, the total number of steps is $O(bc(\log c)^2 + b(\log n)(\log c))$. □

We mention how to extend the result of this section to a simulation of a concurrent-read, concurrent-write (CRCW) PRAM by an Ultracomputer. The CRCW PRAM differs from the EREW PRAM (defined in Section 2) in having no restrictions on memory access. When several processors try to write into the same memory cell, the one with the smallest index succeeds.

An Ultracomputer is a synchronized network of n processors, connected together by a fixed bounded-degree network. At each step each processor can send and receive only one message through one of the lines connecting it to a direct neighbor in the network. We assume a network topology that enables sorting of n keys, initially one at each processor, in $O(\log n)$ steps [8].

THEOREM 3.4. *Any program that requires T steps on a CRCW PRAM with n processors and m shared variables (m polynomial in n) can be simulated by an n processor Ultracomputer within $O(T(\log n)^2 \log\log n)$ steps.*

PROOF (sketch). There are two logical parts to the simulation of each instruction. Both parts rely on the capability of the Ultracomputer to sort n items in $O(\log n)$ steps. The first part (which involves pre- and postprocessing) translates at CRCW PRAM instruction into an EREW PRAM instruction at the *additive* cost of $O(\log n)$ MPC steps. In the preprocessing we sort to eliminate multiple (read or write) access to the same item. In the postprocessing, each read value is broadcast by the processor that actually read it to all the processors that had requested this value.

The preprocessing phase leaves us with an EREW PRAM instruction to simulate. This can be done (by Theorem 3.1) in $O(\log n(\log\log n)^2)$ MPC steps. Each MPC instruction is executed in $O(\log n)$ Ultracomputer steps. Sorting is used both to eliminate multiple access to the same module (which is now a local memory of one of the processors) and to route the resulting partial permutation in $O(\log n)$ steps. This gives a total of $O((\log n \log\log n)^2)$ Ultracomputer steps for the whole simulation.

To save a factor of $\log\log n$, as promised by the theorem, we need only to observe that in the simulation, $O(\log c)$ MPC steps were used for broadcast, summation, and maximum finding. Performing these operations directly on the Ultracomputer, instead of simulating each of the $O(\log c)$ steps, requires only $O(\log n)$ steps, rather than $O(\log n \log c)$ steps. □

We conclude this section with some remarks:

(1) *Fault tolerance.* A variant of our scheme, in which every processor tries to access $(2 - \epsilon)c$ copies rather than c, guarantees that even if up to $(1 - 2\epsilon)c$ of the copies of each data item are destroyed by an adversary, no information or significant 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., [4, 5]) when $m \gg n$.

(3) *Memory requirement.* The simulation algorithm requires that every processor knows the location of all the $2c - 1$ copies of all the m data items. This might require $O(cm)$ memory cells in the local memory of each processor. Finding an efficient simulation that requires much less memory remains an interesting open problem.

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 answer this question positively. 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, since 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 $\Omega((m/n)^{1/2r})$.*

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 data items in the scheme. Therefore, there are at least $m/2$ data items with no more than $2r$ copies. At most dn out of these items appear in the internal registers of processors.

There are $\binom{n}{h^{-1}n}$ 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 $\binom{n-2r}{h^{-1}n-2r}$ 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

$$\frac{\binom{nh^{-1}}{n}(n-1)}{\binom{(n-2r)h^{-1}}{n-2r}} \geq \frac{m}{2} - dn,$$

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

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 tth PRAM instruction before reading the $(t + 1)$th. Of course, it can perform other operations as well during the execution of the tth instruction, but these cannot depend on future instructions.

We assume, without loss of generality, 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 $\Omega(m/n)$; 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* $\Omega(T(\log n/\log\log n))$ *parallel MPC steps.*

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

Consider the redundancy r_t of the scheme after the execution of the tth instruction. A *hard read* instruction will essentially implement Theorem 4.1; that is, it will assign processors to read n items all of whose 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 (since $m \gg n$).

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 tth instruction. For the first $\tau = m/n$ instructions, at most $\sum_{t=1}^{\tau} s_t$ memory locations were accessed, and hence

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

Recall that r_τ is the redundancy when we start alternating reads and writes. Let $t > \tau = m/n$. By Theorem 4.1, at least $\tau^{1/2r_{t-1}} = \beta_{t-1}$ 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

$$r_t \leq r_{t-1} + (s_t - \beta_{t-1} - r_{t-1})\frac{n}{m}, \qquad \text{for} \quad t = \tau + 1, \ldots, T.$$

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 $\bar{r} = 1/(T - m/n) \sum_{t=\tau}^{T-1} rt$ be the average redundancy in the last $T - m/n$ steps. Notice that $\beta(r) = (m/n)^{1/2r}$ is a convex function in r for $r \geq 0$. Hence, by Jensen's inequality [11, pp. 211–216],

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

Hence,

$$\sum_{t=1}^{T} s_t \geq \left(T - \frac{m}{n}\right)\left(\bar{r} + \left(\frac{m}{n}\right)^{1/2\bar{r}}\right) = \Omega\left(\left(T - \frac{m}{n}\right) \frac{\log(m/n)}{\log\log(m/n)}\right).$$

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

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.

We believe that the new notion of consistency suggested by our scheme can have major impact on the theory and design of systems such as those used for databases. Also, there seem to be 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 [9] or [14]. Another application is to asynchronous systems. Although a similar scheme was suggested in this context [13], we believe that the potential of this idea was not fully exploited, and we plan to continue research in this direction.

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

REFERENCES

(Note: Reference [16] is not cited in text.)

1. AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An $O(\log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on the Theory of Computing* (Boston, Mass., Apr. 25–27). ACM New York, 1983, pp. 1–9.
2. AWERBUCH, B., ISRAELI, A., AND SHILOACH, Y. Efficient simulation of PRAM by Ultracomputer. Preprint, Technion, Haifa, Israel. 1983.

3. COOK, S. A., AND DWORK, C. Bounds on the time for parallel RAM's to compute simple functions. In *Proceedings of the 14th ACM Symposium on the Theory of Computing.* ACM, New York, 1982, pp. 231-233.
4. DOLEV, D., DWORK, C., PIPPENGER, N., AND WIGDERSON, A. Superconcentrators, generalizers and generalized connectors with limited depth. In *Proceedings of the 15th Symposium on the Theory of Computing* (Boston, Mass., Apr. 25-27). ACM, New York, 1983, pp. 42-51.
5. GABBER, O., AND GALIL, Z. Explicit construction of linear-sized superconcentrators. *J. Comput. Syst. Sci. 22* (1981), 407-420.
6. GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU Ultracomputer—Designing a MIMD shared memory parallel machine. *IEEE Trans. Comput. C-32*, 2 (1983), 175-189.
7. KUCK, D. J. A survey of parallel machines organization and programming. *ACM Comput. Surv. 9*, 1 (1977), 29-59.
8. LEIGHTON, T. Tight bounds on the complexity of parallel sorting. In *Proceedings of the 16th ACM Symposium on the Theory of Computing* (Washington, D.C., Apr. 30-May 2). ACM, New York, 1984, pp. 71-80.
9. MELHORN, K., AND VISHKIN, U. Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories. In *9th Workshop on Graph Theoretic Concepts in Computer Science.* Fachbereich Mathematik, Univ. Osnabruck, Osnabruck, West Germany, June 1983.
10. PIPPENGER, N. Superconcentrators. *SIAM J. Comput. 6*, 2 (1977), 298-304.
11. ROBERTS, A. W., AND VARBERG, D. E. *Convex Analysis.* Academic Press, Orlando, Fla., 1973.
12. SCHWARTZ, J. T. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2* (1980), 484-521.
13. THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180-209.
14. UPFAL, E. A probabilistic relation between desirable and feasible models of parallel computation. In *Proceedings of 16th ACM Symposium on the Theory of Computing* (Washington, D.C., Apr. 30-May 2). ACM, New York, 1984, pp. 258-265.
15. VISHKIN, U. A parallel-design distributed-implementation general-purpose computer. *J. Theor. Comput. Sci. 32*, 1-2 (1984), 157-172.
16. VISHKIN, U. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms 4*, 1 (1983), 45-50.

RECEIVED OCTOBER 1984; REVISED OCTOBER 1985; ACCEPTED JUNE 1986