SIAM J. COMPUT.
Vol. 16, No. 5, October 1987

DETERMINISTIC SIMULATION OF IDEALIZED PARALLEL COMPUTERS ON MORE REALISTIC ONES*

HELMUT ALT†, TORBEN HAGERUP‡, KURT MEHLHORN‡ AND FRANCO P. PREPARATA§

Abstract. We describe a nonuniform deterministic simulation of PRAMs on module parallel computers (MPCs) and on processor networks of bounded degree. The simulating machines have the same number n of processors as the simulated PRAM, and if the size of the PRAM's shared memory is polynomial in n, each PRAM step is simulated by $O(\log n)$ MPC steps or by $O((\log n)^2)$ steps of the bounded-degree network. This improves upon a previous result by Upfal and Wigderson. We also prove an $\Omega((\log n)^2/\log\log n)$ lower bound on the number of steps needed to simulate one PRAM step on a bounded-degree network under the assumption that the communication in the network is point to point.

As an important part of the simulation of PRAMs on MPCs, we use a new technique for dynamically averaging out a given work load among a set of processors operating in parallel.

Key words. parallel RAM, module parallel computer, processor network, deterministic simulation

AMS(MOS) subject classification. 68C25

1. Introduction and models of computation. Most parallel algorithms in the literature are designed to run on a PRAM (parallel RAM). The PRAM model was introduced by Fortune and Wyllie [FW]. A PRAM consists of some finite number n of sequential processors (RAMs), all of which operate synchronously on a common memory consisting of, say, m storage cells (also called "variables"), cf. Fig. 1. In every step of the PRAM, each of its processors executes a private RAM instruction. In particular, the processors may all simultaneously access (read from or write into) the common memory. Various types of PRAMs have been defined, differing in the conventions used to deal with read/write conflicts, i.e., attempts by several processors to access the same variable in the same step. In the most restrictive model, exclusive read-exclusive write or EREW PRAMs, no variable may be accessed by more than one processor in a given step. In contrast, CRCW (concurrent read-concurrent write) PRAMs allow simultaneous reading as well as simultaneous writing of each variable, with some rule defining the exact semantics of simultaneous writing.

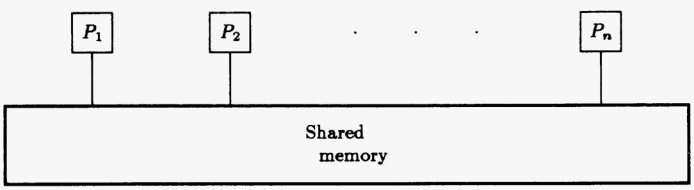

FIG. 1. *The PRAM model.* $P_1, \cdots, P_n$ *are processors.*

* Received by the editors April 28, 1986; accepted for publication (in revised form) November 14, 1986. This work was supported by the Deutsche Forschungegemeinschaft, SFB 124, TP B2, VLSI Entwurf und Parallelität, and by National Science Foundation grant ECS-84-10902. A preliminary and abridged version of this paper was presented at the 12th International Symposium on Mathematical Foundations of Computer Science, Bratislava, Czechoslovakia in August 1986.

† Fachbereich Mathematik, Freie Universität Berlin, Arnimallee 2-6, D-1000 Berlin 33, Federal Republic of Germany. Part of the research was done while the author was a member of the Mathematical Sciences Research Institute, Berkeley, California 94720.

‡ Fachbereich 10, Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Federal Republic of Germany.

§ Coordinated Science Laboratory, University of Illinois, Urbana, Illinois 61801.

PRAMs are very convenient for expressing parallel algorithms since one may concentrate on the problem of "parallelizing," i.e., decomposing the problem at hand into simultaneously executable tasks, without having to worry about the communication between these tasks. Indeed, any intermediate result computed by one of the processors will be available to all the others in the next step, due to the shared memory. Unfortunately, the PRAM is not a very realistic model of parallel computation. Present and foreseeable technology does not seem to make it possible for more than a constant number of processors to simultaneously access the same memory module. A model of computation that takes this problem into account is the MPC (*module parallel computer*, [MV]), cf. Fig. 2. An MPC consists of n processors (RAMs), each equipped with a memory module. Every processor may access every memory module via a complete network connecting the processors. However, the memory modules are sequential devices, i.e., able to satisfy only one request at a time. More precisely, a memory module M operates as follows: If several processors try in the same step to access a variable stored in M, exactly one of the processors is allowed to carry out its read or write instruction; the remaining access requests are discarded. All processors are informed of the success or failure of their access attempts. We make no assumptions about how the single successful processor is selected from among the processors competing to access M.

The MPC model is still not realistic for large n because of the postulated complete network connecting the processors. This leads us to consider a third model which we shall call the *network model*. Here the processors are connected via a network of bounded degree, i.e., each processor is linked directly to only a constant number of other processors, cf. Fig. 3. Since each step of a completely interconnected processor network may be simulated by $O(\log n)$ steps of a bounded-degree network ([AKS], [L]), efficient algorithms for the MPC model translate into asymptotically efficient algorithms for the network model.

The simulation of the idealized parallel machine, the PRAM, on the more realistic one, the MPC, has been considered in several previous papers. A naive approach represents each variable x of the PRAM by one variable $\psi(x)$ of the MPC. Now if a PRAM step accesses the variables $x_1, \cdots, x_l$, collisions may occur in the simulating

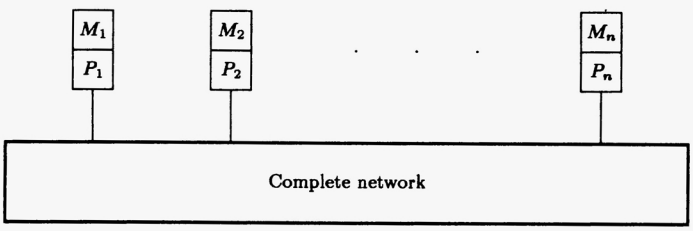

FIG. 2. *The MPC model.* $P_1, \cdots, P_n$ *are processors,* $M_1, \cdots, M_n$ *memory modules.*

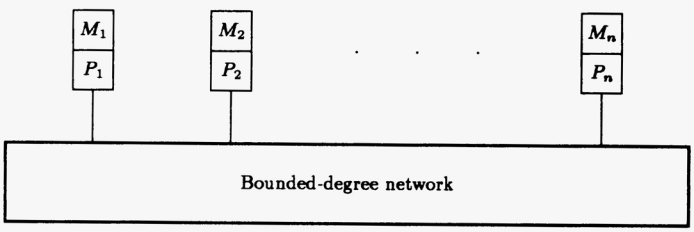

FIG. 3. *The network model.* $P_1, \cdots, P_n$ *are processors,* $M_1, \cdots, M_n$ *memory modules.*

machine because $\psi(x_1), \cdots, \psi(x_l)$ are not necessarily located in distinct memory modules. If $m \leqq n$, the m variables may be allocated to m different memory modules, and a trivial $O(1)$-time simulation is possible. However, we are concerned with the case in which m is considerably larger than n. Here a major problem is to find a memory correspondence between the PRAM and the MPC such that, for all possible access patterns of the PRAM, the maximum number of variables requested from a single MPC memory module is kept low. Note that, for specific PRAM algorithms such as matrix multiplication, there may be very efficient ways of assigning variables to modules; we refer the reader to § 4 of the survey paper by Kuck [K]. Here we are interested in universal simulations that work efficiently no matter which algorithm is executed by the PRAM.

Some results have been obtained previously using *probabilistic* methods: Mehlhorn and Vishkin [MV] used universal hashing to define the memory correspondence. They obtained several upper bounds, for example, an average of $O(\log n)$ MPC steps to simulate one PRAM step, with the total amount of memory used by the MPC larger than the PRAM memory by a factor of $O(\log n)$. Upfal [U] found a probabilistic simulation of $O((\log n)^2)$ average time for one PRAM step on a bounded-degree network; this was recently improved to $O(\log n)$ by Karlin and Upfal [KU].

This paper is concerned with *deterministic* simulations. We define the *slowdown* of a simulation as the number of steps needed by the simulating machine in the worst case to simulate one step of the simulated machine. Note that if $m \geqq n^2$, the simple scheme outlined above (x is represented by $\psi(x)$) performs poorly: An adversary could make the PRAM step access n variables $x_1, \cdots, x_n$ with $\psi(x_1), \cdots, \psi(x_n)$ all in the same module. Hence the slowdown is $\Omega(n)$. This reasoning shows that each PRAM variable must be represented by several "copies" stored in different modules. Mehlhorn and Vishkin [MV] showed that read instructions can be handled very efficiently using this idea. However, they did not know how to deal with write instructions. In a beautiful paper Upfal and Wigderson [UW] resolved this problem and exhibited a simulation which uses $\Theta(\log n)$ copies of each PRAM variable. If m is polynomial in n, the slowdown is $O(\log n(\log \log n)^2)$. They also showed an $\Omega(\log n/\log \log n)$ lower bound on the slowdown for a large class of simulations.

Using similar techniques, this paper improves the upper bound to $O(\log m)$. If m is polynomial in n, this is $O(\log n)$. Consequently, a PRAM step may be simulated in $O(\log n \log m)$ time on a bounded-degree network. On the other hand, we show that $\Omega(\log n \log m/\log \log m)$ time is necessary under certain assumptions on any bounded-degree network whose communication is restricted to be point to point. A similar result was also obtained by Karlin and Upfal [KU]. The assumption of point to point communication is not satisfied by our simulation algorithm which uses more general communication patterns.

The PRAM simulations which we consider will be based on *emulations* of the PRAM's shared memory. We conceptually retain the n PRAM processors while replacing (or, equivalently, implementing) the PRAM's (physically infeasible) shared memory by a (more feasible) suitably programmed MPC or bounded-degree network with n processors, called the emulating processors. Each PRAM processor, which was formerly connected to the shared memory, is now instead connected to one of the emulating processors called its *associated* processor, each emulating processor being associated with exactly one PRAM processor, cf. Fig. 4. We require the change to be completely transparent, i.e., all PRAM programs must run without change (though possibly slower) on the modified machine. Note that although the most direct PRAM simulation implied by a memory emulation as above uses a total of $2n$ processors, it

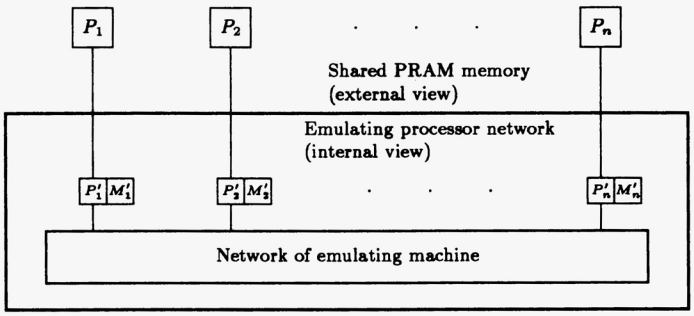

FIG. 4. *Emulation of the shared memory of a PRAM. For* $i = 1, \cdots, n$, P_i *is a PRAM processor and* P'_i *its associated emulating processor.*

is a trivial matter to reduce the number of processors to n by coalescing each pair of associated processors into a single processor. For expository reasons we prefer to keep the clean separation between PRAM processors and (emulated) shared memory.

Our simulation algorithms are nonuniform. This means that they are not given explicitly. Instead we merely prove that algorithms with the desired properties exist. For fixed values of n and m, such algorithms may be found by exhaustive search in a large but finite set. We return to this aspect in the concluding section.

It has been known since Adleman's work [A] that probabilistic algorithms may be converted into nonuniform deterministic ones. Hence the result by Karlin and Upfal [KU] automatically translates into a nonuniform deterministic simulation of PRAMs on a bounded-degree network. However, if the translation is based on Karlin and Upfal's analysis of their algorithm and uses known techniques, it introduces an $\Omega(n)$-increase in the product of time and number of processors [R, Thm. 6]. Since it is not difficult to devise a uniform deterministic simulation which uses $O(n^2/(\log n)^2)$ processors and has a slowdown of $O(\log n)$ (the construction is similar to one presented in the remark ending § 3), deterministic algorithms derived from Karlin and Upfal's probabilistic simulation are of little interest. The same is true of all other known probabilistic solutions.

The remaining part of the paper is structured as follows: In § 2 we describe our simulation of PRAMs on MPCs and show that its slowdown is $O(\log m)$. As part of the development of the algorithm, we define and solve a so-called "redistribution problem." Section 3 considers the simulation of PRAMs on bounded-degree networks and establishes upper and lower bounds of $O(\log n \log m)$ and $\Omega(\log n \log m/\log\log m)$, respectively. In § 4 we return to the redistribution problem and prove a stronger result than what was needed in § 2. Finally, § 5 addresses some interesting and important open issues.

2. Simulation of PRAMs on MPCs. In this section we describe an emulation with $O(\log m)$ slowdown of the shared memory of an EREW PRAM with n processors and m shared memory cells by an MPC with n processors. A standard method allows the emulation to be extended using only $O(\log n)$ extra time per simulated step to cover all common variants of CRCW PRAMs. We shall not give the details of this but refer the reader to [BH].

We will assume that $n < m \leqq 2^n$. This is no restriction: If $m \leqq n$, an $O(1)$-time solution is possible as argued in the previous section. And if $m > 2^n$, even the trivial sequential simulation on just one MPC processor is within the time bound of $O(\log m)$. We further allow our construction to fail for values of n and m smaller than some

(unspecified) constant number N. Again this is no restriction. Note that whenever we employ the "big-oh" notation, the limiting process implied is that of n and m tending to infinity while other parameters may take on arbitrary legal values.

For concreteness, let us assume that each PRAM processor P has two instructions for communicating with the shared memory: LOAD x, where x is a shared variable, replaces the contents of a distinguished register in P, called its accumulator, by the current value of x (which is held in the shared memory), and STORE x updates x with the contents of P's accumulator. We also assume that each PRAM variable x has a unique identification which we shall call its *name* and denote symbolically by "x".

We can now describe the computational task that must be performed by the emulating machine. In the beginning of each PRAM step, each emulating processor P' receives from its associated PRAM processor P as input one of the following:

(1) The name "x" of a PRAM variable x, meaning that P's current instruction is LOAD x. In this case we will say that P' *reads* x in the current (PRAM) step.

(2) A pair $\langle$"x", $a\rangle$, where "x" is the name of a PRAM variable x, meaning that P's current instruction is STORE x, and a is the contents of P's accumulator. We say that P' *writes* or *updates* x in the current step.

(3) A signal *NoOp*, meaning that P does not access the shared memory in the current step.

In cases (1) and (2), we will also say that P' is the *origin* of x and that x is *accessed* in the current PRAM step.

At the end of the simulation of each PRAM step, each emulating processor reading a variable x in that step must have computed the current value of x and must output it to the associated PRAM processor. The current value of x is defined in the obvious way as the second component of the pair $\langle$"x", $a\rangle$ with first component "x" most recently input by some emulating processor in a PRAM step prior to the current step (since we are considering only correct EREW PRAM programs, the pair $\langle$"x", $a\rangle$ is well-defined).

Our emulation algorithm is based on an idea introduced by Upfal and Wigderson [UW]:

Each variable x of the PRAM is represented by $2c-1$ memory cells $\psi_1(x), \cdots, \psi_{2c-1}(x)$ of the emulating MPC, where $c = \lfloor \log m \rfloor$.[1] These cells will be called the *copies* of x.

Whenever a processor of the PRAM executes the instruction STORE x, the MPC will access a majority, i.e., at least c, of x's copies and store the new value into them together with a time stamp indicating the number of PRAM steps simulated so far.

The simulation of the instruction LOAD x also consists of accessing at least c copies of x. Since this is again a majority, at least one of the copies must contain the current value of x, and the most recently updated copy is readily identified using the time stamps.

Remark. It suffices to count time modulo m if each PRAM variable is "cleaned" once every m simulated steps. Cleaning a variable means inspecting all its copies and replacing each of their time stamps by the appropriate one of two special values "invalid" and "oldest". Since this takes $O(c) = O(\log m)$ sequential time, one variable may be cleaned after each simulated step at only a constant-factor increase in simulation time. Furthermore, if variables are cleaned in a fixed cyclical order known to all MPC processors, the modified time stamps allow the MPC to determine the most recently

[1] "log" without subscript denotes the logarithm to base 2.

updated copy much as before. In the light of this observation, we shall assume that time stamps can be manipulated in unit time.

We partition the processors of the MPC into $k = \lceil n/(2c-1) \rceil$ *clusters* containing $2c-1$ processors each. (We assume that $2c-1$ divides n perfectly. If this is not the case, there will be an incomplete cluster. But if we let each MPC processor simulate two "virtual processors" at the price of increasing the execution time by at most a constant factor, there will be more than enough processors to fill the last cluster. Hence our assumption implies no loss of generality.) For $i = 1, \cdots, k$ and $j = 1, \cdots, 2c-1$, denote by $P_{i,j}$ the jth processor in the ith cluster, and for $r = 1, \cdots, 2c-1$, let $\mathcal{P}(r)$ be the set of the first r processors from each cluster, i.e., $\mathcal{P}(r) = \{P_{i,j} \mid 1 \leqq i \leqq k, 1 \leqq j \leqq r\}$. All processors within a cluster cooperate when an access to a variable is attempted by trying to access its different copies.

From now on consider the simulation of a single PRAM step called the *current step.* For each variable x accessed in the current step, there will be an MPC processor P *responsible* for x. Conversely, we shall say that x is in P's *custody* and sometimes think of x as residing in P. Initially, x is in the custody of its origin.

Responsibilities are changed during calls of the procedure REDISTRIBUTE to be described later. We maintain the invariant, however, that each processor at any given time is responsible for at most one variable.

Following is a description of the major subroutines used by our algorithm. In each case, all clusters operate in parallel.

ACCESS(j): ($1 \leqq j \leqq 2c-1$). Within the ith cluster, for $i = 1, \cdots, k$, all processors simultaneously try to access the different copies of the variable x_i which is in $P_{i,j}$'s custody. More specifically: For $l = 1, \cdots, 2c-1$, if $P_{i,l}$ has not previously been successful in an access to $\psi_l(x_i)$, it tries to access $\psi_l(x_i)$. If $P_{i,j}$ is currently not responsible for any variable, or if $P_{i,l}$ has already succeeded in accessing $\psi_l(x_i)$, $P_{i,l}$ remains idle. We shall say that the variables which are in the custody of some processor $P_{i,j}$ ($i = 1, \cdots, k$) are in the *range* of the call ACCESS(j).

Since processors from different clusters may attempt to access memory cells within the same module at the same time, in which case only one of them can be served, there will be successful and unsuccessful attempts. Each processor which is successful in accessing its copy records this fact in an internal table. If the copy is of a variable x which is read in the current step, the processor also records the value and the time stamp of the copy; if x is written in the current step, the processor updates the copy with the new value and time stamp.

By assumption the execution time of ACCESS is $O(1)$.

COUNT(r): ($1 \leqq r \leqq 2c-1$). For each variable x which is in the custody of some processor $P_{i,j}$ in $\mathcal{P}(r)$, this procedure counts the number of successful attempts by processors in the ith cluster to access copies of x. The variables for which this number is $\geqq c$ are declared *dead,* meaning that it is not necessary to access any more of their copies. COUNT uses for each cluster C a virtual complete binary tree T_C of processors, the processors of C being its leaves. Since T_C contains less than twice as many processors as C, the virtual processors in T_C may easily be simulated by the physical processors in C, each processor (except one) acting both as a leaf and as a particular internal node of T_C.

The counting is done for each variable x going up level by level in T_C, where C is the cluster containing the processor responsible for x. Initially C's processors acting as leaves of T_C transmit to their parents an integer, 1 if they have successfully accessed a copy of x, 0 otherwise (recall that this information was stored in an internal table

during the calls of ACCESS). Each internal node adds the two numbers received from its children and forwards the sum to its parent. After h steps, where h is the height of T_C, the root will have computed the total number of successful access attempts. If that number is $\geq c$ ($<c$), the root sends the information that x is dead (alive) down the tree.

If x is read in the current step, the values and time stamps of copies of x are also propagated up the tree. Each internal node selects among the (value, time stamp) pairs of its children the one with the most recent time stamp. If x is declared dead the root is able to determine its valid value and to propagate it to the leaves of T_C where it will be stored internally by the processor responsible for x.

When the above procedure has been carried out for all variables concerned, each processor which is responsible for a now dead variable x read in the current step sends the value of x to the origin of x (this is the processor actually needing the value of x). Since the values transferred go to distinct destinations, this step may be carried out in constant time. At this point all memory accesses involving dead variables have been successfully simulated, allowing us to forget about variables that died in this call of COUNT. In particular, we shall no longer consider them to be in the custody of any processor.

Since all clusters operate simultaneously, doing the counting for one variable per cluster takes $O(h) = O(\log c) = O(\log \log m)$ time. We are actually doing the counting for r variables per cluster. However, since each variable uses only one level at a time in the corresponding tree T_C, it is possible to *pipeline* the propagation of information up and down the tree. Hence a call COUNT(r) may be executed in $O(r + \log \log m)$ time.

BROADCAST(r): ($1 \leq r \leq 2c-1$). Before all processors within a cluster can attempt to access the different copies of a variable x, the name of x and, if x is written in the current step, its new value must be broadcast by the processor responsible for x to all other processors in the cluster.

This is done using the same trees T_C as in COUNT. The processor responsible for x sends the information about x up the tree, the root sends copies down to both subtrees, etc., so that the information reaches all processors in the cluster and may be stored in a table within each processor. BROADCAST(r) operates only on variables in the custody of processors in $\mathcal{P}(r)$. Since a pipelining technique similar to the one adopted in COUNT may be used, the time bound of $O(r + \log \log m)$ again applies.

BROADCAST-ACCESS-AND-COUNT(s, r): ($s \geq 1$, $1 \leq r \leq 2c-1$). This is a simple combination of the above procedures defined by the program segment:

```
BROADCAST(r);
do s times
  for j := 1 to r do
    ACCESS(j);
  od;
  COUNT(r);
od;
```

A call BROADCAST-ACCESS-AND-COUNT(s, r) first broadcasts information about the live variables which are in the custody of some processor in $\mathcal{P}(r)$ (say again that these are in the range of the call). It then goes s times through a cycle in which it first attempts to access all live variables in the range and then calls COUNT to

determine which of the access attempts were successful. The execution time of BROADCAST-ACCESS-AND-COUNT(s, r) is $O(s(r+\log\log m))$.

REDISTRIBUTE. Even if (responsibilities for) the live variables are distributed about evenly among the clusters at some point in time, we cannot guarantee that this will still be true after several calls of ACCESS and COUNT since the processors in some clusters may be much more successful than processors in other clusters. It is crucial for efficiency to periodically average out the work load among the clusters. We do this by means of two variants of a procedure REDISTRIBUTE.

As we argue below, the redistribution problem is reducible to that of sorting the variables. This observation together with a fast parallel sorting algorithm [AKS] immediately gives us an $O(\log n)$-time procedure EXACT-REDISTRIBUTE for doing a perfect redistribution of the live variables. However, since we cannot always afford to spend $\Theta(\log n)$ time, we also need another redistribution procedure which is considerably faster but, in exchange, less accurate. Here the construction is more complicated. A comparatively short argument given below proves an upper bound on the time needed for an approximate redistribution. The bound, although weak, suffices for our purposes. Since we consider the redistribution problem to be interesting and important in its own right, however, we reconsider the redistribution problem in § 4 and obtain a much stronger result. Both of our proofs are inspired by the construction of ε-halvers in [AKS]. Let us first state the problem precisely. We distinguish between a weak and a strong form. The weak form will be used in the present section and the strong form in § 4.

DEFINITION. Consider n locations divided into k equal-size groups. Each location may hold a record, and given a set of n records, we call a particular association between the locations and the records a *configuration.* For integers R and y, the *weak* (*strong*) *redistribution problem* with n elements, upper bound R, and y allowable errors is as follows: Given a configuration of n records, each containing a key equal to either 0 or 1 and such that the set U of records with keys equal to 0 contains at most R elements, permute the records so that, in the resulting configuration for some set $V \subseteq U$ with $|U \setminus V| \leqq y$, there is no group whose locations contain more than $\lceil R/k \rceil$ ($\lceil |U|/k \rceil$, respectively) records belonging to V.

In our application the groups correspond to clusters of processors, and the records to names of variables together with any other relevant information (e.g., if a variable is read in the current step, an identification of its origin). A key equal to 0 indicates a live variable. Hence the strong redistribution problem asks for the live variables to be evenly redistributed among the clusters such that no more than y live variables "go wrong." The redistribution problem is closely related to that of approximate data compaction:

DEFINITION. Consider n locations arranged in a fixed order and each capable of holding a record. For integers R and y, the *weak* (*strong*) *approximate data compaction problem* with n elements, upper bound R, and y allowable errors is as follows: Given a configuration of n records, each with a key equal to either 0 or 1 and such that the number Q of records with keys equal to 0 is at most R, permute the records such that in the resulting configuration at most y records with keys equal to 0 are not in one of the first R(Q, respectively) locations.

We shall consider algorithms for the above problems consisting of a sequence of *parallel comparison-exchange steps.* A *comparison-exchange operation* compares the keys of two records A and B, with the location of A, say, preceding the location of B. If B's key is smaller than A's key, the records A and B are swapped. A parallel

comparison-exchange step on a set $\mathcal{L}$ of locations consists of a number of comparison-exchange operations executed in parallel on disjoint pairs of locations in $\mathcal{L}$. It is clear that if each of n MPC processors holds one of n locations, the MPC can execute any parallel comparison-exchange step on the n locations in constant time.

Suppose that we are given a redistribution problem. If we number the groups $1, \cdots, k$ and for $i = 1, \cdots, k$ denote the locations in the ith group by $L_{i,1}$, $L_{i,2}, \cdots, L_{i,n/k}$, then it is easy to see that solving the weak (strong) approximate data compaction problem with upper bound R and y allowable errors relative to the location order

$$L_{1,1}, L_{2,1}, \cdots, L_{k,1}, L_{1,2}, \cdots, L_{k,2}, \cdots, L_{1,n/k}, \cdots, L_{k,n/k}$$

also solves the weak (strong) redistribution problem with upper bound R and y allowable errors. Hence we need only consider the approximate data compaction problem which allows of a more convenient terminology. Note that data compaction is a special case of sorting in which there are only two distinct key values.

For all $p, q \in \mathbb{N}$, let

$$[p]_q = p(p-1)\cdots(p-q+1).$$

We shall make repeated use of the inequalities given in the proposition below of which one expresses a simple combinatorial fact and the other follows from Stirling's approximation of the factorial function.

PROPOSITION 1. (1) *Given sets A, A', B, B' with $A' \subseteq A$, $B' \subseteq B$ and $|A| \leqq |B|$, the fraction, among all injective functions $A \to B$, of those injective functions $A \to B$ which map each element of A' to an element of B' is*

$$\frac{[|B'|]_{|A'|}}{[|B|]_{|A'|}} \leqq \left(\frac{|B'|}{|B|}\right)^{|A'|}.$$

(2) *For all $p, q \in \mathbb{N}$ with $q \geqq 1$,*

$$\binom{p}{q} \leqq \left(\frac{ep}{q}\right)^q.$$

LEMMA 2.1. *For all n, R, and y with $1 \leqq y \leqq R \leqq n/2$, the weak approximate data compaction (or redistribution) problem with n elements, upper bound R, and y allowable errors is solved by an algorithm consisting of $\lceil 4(n/y)\log(n/y)\rceil$ parallel comparison-exchange steps on n elements.*

Proof. Let us consider the approximate data compaction problem and call it that of "compacting with at most y errors." Let the ordered sequence of locations be $L_1, \cdots, L_n$. We shall henceforth ignore the actual information to be permuted and consider each record to consist of a single bit, its key. Following [AKS], we translate the problem into a graph-theoretic setting and show the existence of certain expander-like graphs. Let $A = \{L_1, \cdots, L_R\}$, $B = \{L_{R+1}, \cdots, L_n\}$, and $\nu \in \mathbb{N}$. For each choice of ν injective functions $\pi_1, \cdots, \pi_\nu$ mapping A into B, we may construct a labeled bipartite graph on the node sets A and B by drawing an edge labeled l from a to $\pi_l(a)$ for all $l = 1, \cdots, \nu$, $a \in A$. Let $\mathcal{G}_\nu$ denote the set of all such labeled graphs obtained by varying $\pi_1, \cdots, \pi_\nu$.

To each graph in $\mathcal{G}_\nu$ there corresponds an algorithm consisting of ν parallel comparison-exchange steps on $L_1, \cdots, L_n$: Interpret an edge labeled l linking $L_i \in A$ and $L_j \in B$ as a comparison-exchange operation between L_i and L_j to be performed in the lth step. Note that by the ordering of the locations, the smaller value will be

placed in L_i. For $a \in A' \subseteq A$ and $b \in B' \subseteq B$, we will say that the edge (a, b) (if it exists) joins A' and B'. Now call a graph $G \in \mathcal{G}_\nu$ "good" if it has the following property:

> For all sets $A' \subseteq A$, $B' \subseteq B$ with $|A'| = |B'| = y$, there is at least one edge in G joining A' and B'.

We claim that the algorithm $\mathcal{A}$ corresponding to a good graph compacts with at most y errors. Suppose the contrary. Then there is a set $B' \subseteq B$, $|B'| = y$, such that all locations in B' contain a 0 after the execution of $\mathcal{A}$. Since the total number of 0's is at most R, there is also a set $A' \subseteq A$, $|A'| = y$, such that all locations in A' contain a 1 after the execution of $\mathcal{A}$. Since the graph is good, there is at least one edge joining A' and B'. Suppose that such an edge is labeled l and links $L_i \in A$ with $L_j \in B$. Then after the lth step of the algorithm, the value held by L_j is no less than the value held by L_i by the interpretation of the edge. On the other hand, L_i holds a 1 and L_j holds a 0 after the last step. But this is impossible since the values held by locations in A never increase, while those held by locations in B never decrease.

It still remains to show that $\mathcal{G}_\nu$ contains good graphs for all n for sufficiently small values of ν. If a graph in $\mathcal{G}_\nu$ is not good (call it "bad"), there are sets $A' \subseteq A$, $B' \subseteq B$, $|A'| = |B'| = y$, such that no edge joins A' and B'. Hence the fraction f_ν of bad graphs in $\mathcal{G}_\nu$ is bounded by

$$\binom{R}{y}\binom{n-R}{y}\left(\frac{[(n-R)-y]_y}{[n-R]_y}\right)^\nu.$$

Here $\binom{R}{y}$ is the number of possible choices of A', $\binom{n-R}{y}$ is the number of possible choices of B', and $[(n-R)-y]_y/[n-R]_y$, for fixed A' and B', is by Proposition 1 the fraction of injective mappings of A into B with the property that no element of A' is mapped to an element of B'. Using again Proposition 1, we get

$$f_\nu \leqq \left(\frac{eR}{y}\right)^y \left(\frac{e(n-R)}{y}\right)^y \left(\frac{(n-R)-y}{n-R}\right)^{\nu y} \leqq \left(\frac{e^2 n^2}{y^2}\left(1-\frac{y}{n}\right)^\nu\right)^y,$$

and it is clear that $f_\nu < 1$ provided that

$$\nu > \frac{2\log(en/y)}{-\log(1-y/n)}.$$

Finally observe that since

$$-\log\left(1-\frac{y}{n}\right) \geqq \frac{y}{n}\log e \quad \text{and} \quad \frac{n}{y} \geqq 2,$$

we have

$$\frac{2\log(en/y)}{-\log(1-y/n)} \leqq 2\frac{n}{y}\left(\log e + \log\frac{n}{y}\right)\frac{1}{\log e} = 2\frac{n}{y}\log\frac{n}{y}\left(\frac{1}{\log(n/y)} + \frac{1}{\log e}\right) < 4\frac{n}{y}\log\frac{n}{y}.$$

□

Let us denote a call of the MPC algorithm implied by Lemma 2.1 by APPROXIMATE-REDISTRIBUTE(R, y). As a technical point, define APPROXIMATE-REDISTRIBUTE(R, 0) for arbitrary R to be equivalent to EXACT-REDISTRIBUTE. Note that we may assume that a call APPROXIMATE-REDISTRIBUTE(R, y) moves all except y live variables to processors in $\mathcal{P}(\lceil R/k \rceil)$ and analogously for EXACT-REDISTRIBUTE, i.e., the live variables are moved to the *first* processors in each cluster.

The algorithm for simulating the memory accesses of one PRAM step consists of three parts: Part 1 decreases the number of live variables from at most n to $O(n \log\log m/\log m)$, Part 2 reduces it further to $O(m2^{-\log m/\log\log m})$, and Part 3 eliminates all remaining live variables.

The following program segment describes Part 1 of the algorithm. It uses a variable R that will be an upper bound on the number of live variables. a_1 and γ are design parameters to be determined in the subsequent analysis of the algorithm. Let us call an execution of lines (4)-(6) a stage. Part 1 of the algorithm consists of an initialization phase followed by $u = \lceil \log\log m - \log\log\log m \rceil$ stages.

Part 1

```
(1) For each origin P of a variable x,
      declare P to be responsible for x;
(2) R := n;
(3) do ⌈log log m − log log log m⌉ times
(4)   BROADCAST-ACCESS-AND-COUNT(a_1, ⌈R/k⌉);
(5)   R := ⌊R/2⌋;
(6)   APPROXIMATE-REDISTRIBUTE(R, ⌊γR⌋);
(7) od;
```

So far we have said nothing about how the $2c-1$ copies of each variable are to be distributed among the memory modules of the MPC. Any such arrangement may be represented by a Boolean $m \times n$ matrix whose rows correspond to variables and whose columns correspond to memory modules. The ijth entry is 1 if and only if the jth memory module contains a copy of the ith variable (i.e., each row in the matrix has exactly $2c-1$ entries equal to 1). Such an arrangement is called a *memory organization scheme* (MOS).

By a counting argument, Upfal and Wigderson proved the following lemma:

LEMMA 2.2 (Lemma 3.3 in [UW], slightly adapted). *There exists a constant η, $0 < \eta < 1$, such that for all n and m there is an* MOS *with the following property:*

> *For any call of BROADCAST-ACCESS-AND-COUNT(s, r), if the number of live variables in the range of the call before the call is w, then after the call it is at most $2\eta^s w$.*

Furthermore, the fraction of MOS*'s not having the above property among all possible* MOS*'s is $o(1/n)$.*

We are now in a position to analyze Part 1 of the algorithm. Choose a constant η and an MOS as given by Lemma 2.2. Then let a_1 be an integer large enough to make $2\eta^{a_1}$ strictly less than $\frac{1}{2}$ and put $\xi = 2\eta^{a_1}$, $\gamma = (\frac{1}{2} - \xi)/(1-\xi)$. Denote for $i = 0, 1, \cdots$ by R_i the value of the variable R after exactly i stages, and let $r_{i+1} = \lceil R_i/k \rceil$. We will show by induction that the invariant

At the beginning of the ith stage, there are at most R_{i-1} live variables

holds for $i = 1, 2, \cdots$. Since $R_0 = n$, the assertion is trivially true for $i = 1$. Thus let $i \geqq 1$ and assume that at the beginning of the ith stage, there are at most R_{i-1} live variables. We must show that the ith stage reduces the number of live variables to R_i.

If $i > 1$, the inductive hypothesis implies that the call of APPROXIMATE-REDISTRIBUTE which ended the previous stage was applied to a legal input. In particular, the first argument R is always at most $n/2$ as required by our construction. Hence we may conclude that $y \leqq \gamma R_{i-1}$, where y is the number of live variables outside the range of the call of BROADCAST-ACCESS-AND-COUNT in the ith stage. But

then the total number Q of live variables at the end of the ith stage is at most

$$y+\xi(R_{i-1}-y)=(1-\xi)y+\xi R_{i-1}\leqq((1-\xi)\gamma+\xi)R_{i-1}=R_{i-1}/2,$$

and since Q is an integer, we have in fact $Q\leqq\lfloor R_{i-1}/2\rfloor=R_i$, completing the inductive step. Thus Part 1 of the algorithm reduces the number of live variables to at most

$$n2^{-u}\leqq n\frac{\log\log m}{\log m}.$$

As for the running time $T(n, m)$ of Part 1, we have

$$T(n, m)=O\left(\sum_{i=1}^{u}(r_i+\log\log m+S_i)\right)$$

where S_i is the time spent in the ith call of APPROXIMATE-REDISTRIBUTE. Now

$$\sum_{i=1}^{u}r_i\leqq\sum_{i=1}^{u}\left(\frac{R_{i-1}}{k}+1\right)\leqq\frac{n}{k}\sum_{i=0}^{\infty}2^{-i}+u=O(\log m),$$

and since $n/(\gamma R_i)\leqq 2^{i+1}/\gamma=O(2^i)$, we have $S_i=O(i2^i)$ by Lemma 2.1 and

$$\sum_{i=1}^{u}S_i=O(u2^u)=O(\log m).$$

Hence $T(n, m)=O(\log m)$.

Part 2 of the algorithm is as follows:

Part 2

EXACT-REDISTRIBUTE;
BROADCAST-ACCESS-AND-COUNT($\lceil a_1\log m/\log\log m\rceil$, $\lceil 2\log\log m\rceil$);

The exact redistribution together with the fact that Part 1 leaves at most $n(\log\log m/\log m)\leqq 2k\log\log m$ live variables guarantees that all live variables are in the range of the call of BROADCAST-ACCESS-AND-COUNT. Hence the number of live variables is reduced to at most

$$n\frac{\log\log m}{\log m}2\eta^{a_1\log m/\log\log m}\leqq n2^{-\log m/\log\log m}.$$

The running time is $O(\log n+(\log m/\log\log m)\log\log m)=O(\log m)$. Summing up, we have shown:

LEMMA 2.3. *For a suitable choice of an MOS and of the constants a_1 and γ, Parts 1 and 2 together reduce the number of live variables to at most $n2^{-\log m/\log\log m}$. The execution time of each part is $O(\log m)$.*

Part 3 is described by the program segment:

Part 3

```
(1)  EXACT-REDISTRIBUTE;
(2)  BROADCAST(1);
(3)  do ⌈a2 log n/log log m⌉ times
(4)    do ⌈a3 log log m⌉ times
(5)      ACCESS(1);
(6)    od;
(7)    COUNT(1);
(8)  od;
```

Call an execution of lines (4)-(7) a stage. a_2 and a_3 are design parameters to be chosen later. Since the execution time of ACCESS is $O(1)$ and that of COUNT(1) is

$O(\log\log m)$, Part 3 clearly requires no more than $O(\log n)$ time. It remains to show that Part 3 eliminates all remaining live variables for suitable constants a_2 and a_3.

Notice that Part 3 uses a strategy different from the one adopted in Parts 1 and 2. A variable is accessed *several* (in fact, $\Theta(\log\log m)$) times before a COUNT is done to find out if it has died. Thus variables may be accessed even though they have fewer than c remaining live copies (copies that were not yet accessed). This means that Lemma 2.2 is not applicable. We will now show, however, that the idea of Lemma 2.2 still works as long as the variables being accessed have an average of at least c^q live copies each for some $q>0$.

LEMMA 2.4. *For all constants p, q and η with $0<p,\ q,\ \eta<1$ and $p+q>1$, there is a constant N such that for all n, m with $m>n\geqq N$, there exists an MOS with the following property:*

> *If a call of ACCESS attempts to access exactly $Q\leqq n2^{-(\log m)^p}$ variables and the total number of live copies of these variables before the call is $W\geqq Qc^q$, then after the call at most ηW of the copies are still alive.*

Furthermore, the fraction of MOS's not having the above property among all possible MOS's tends to 0 as n tends to infinity.

Proof. Call an MOS "good" if for any choice of $Q\leqq Q_{\max}=\lfloor n2^{-(\log m)^p}\rfloor$ variables and any choice of $W\geqq Qc^q$ live copies of these variables, the number of modules containing at least one of the W live copies is at least $(1-\eta)W$. It is clear that a good MOS guarantees the elimination of at least $(1-\eta)W$ live copies. Hence we need only show that there are sufficiently many good MOS's.

For each choice of Q and W ($1\leqq Q\leqq Q_{\max}$, $Qc^q\leqq W\leqq Q(2c-1)$) denote by $f_{Q,W}$ the fraction of MOS's that fail to have the above property because they map some set of W copies of Q variables to fewer than $(1-\eta)W$ modules. We now derive an upper bound on $f_{Q,W}$. There are

$\binom{m}{Q}$ ways to choose the set of Q variables,

$\binom{(2c-1)Q}{W}$ ways to choose the W live copies from the set of all copies of the Q variables, and

$\binom{n}{\lfloor(1-\eta)W\rfloor}$ ways to choose $\lfloor(1-\eta)W\rfloor$ modules to contain all the live variables.

For each choice of W live copies and $\lfloor(1-\eta)W\rfloor$ modules, the fraction of MOS's that map the chosen live copies to the chosen modules is by Proposition 1 at most

$$\left(\frac{\lfloor(1-\eta)W\rfloor}{n}\right)^W.$$

Hence

$$\begin{aligned}
f_{Q,W} &\leqq \binom{m}{Q}\binom{(2c-1)Q}{W}\binom{n}{\lfloor(1-\eta)W\rfloor}\left(\frac{\lfloor(1-\eta)W\rfloor}{n}\right)^W\\
&\leqq \left(\frac{em}{Q}\right)^Q\left(\frac{e(2c-1)Q}{W}\right)^W\left(\frac{en}{\lfloor(1-\eta)W\rfloor}\right)^{\lfloor(1-\eta)W\rfloor}\left(\frac{\lfloor(1-\eta)W\rfloor}{n}\right)^W\\
&\leqq [e^{2-\eta+Q/W}(1-\eta)^{\eta}m^{Q/W}Q^{1-Q/W}(2c-1)W^{\eta-1}n^{-\eta}]^W\\
&\leqq \left[2e^{2-\eta+Q/W}m^{Q/W}c\left(\frac{Q}{W}\right)^{1-\eta}Q^{\eta}n^{-\eta}\right]^W
\end{aligned}$$

$$\leqq \left[2e^3 m^{c^{-q}} c^{1-q(1-\eta)} \left(\frac{Q}{n}\right)^{\eta} \right]^W,$$

where the relation $Q/W \leqq c^{-q}$ was used in the last step. We will show that the quantity g_Q in square brackets is dominated by its last factor $(Q/n)^{\eta}$. Consider for this purpose its logarithm:

$$\log g_Q = 1 + 3 \log e + c^{-q} \log m + (1 - q(1-\eta)) \log c + \eta \log \frac{Q}{n}.$$

Now

$$c^{-q} \log m = O((\log m)^{1-q}) \quad \text{and} \quad (1 - q(1-\eta)) \log c = O(\log \log m)$$

while

$$\log \frac{Q}{n} \leqq -(\log m)^p.$$

Since $1 - q < p$, we may conclude that for all χ with $0 < \chi < \eta$,

$$\log g_Q \leqq -\chi (\log m)^p$$

and hence

$$f_{Q,W} \leqq 2^{-\chi W (\log m)^p}$$

for all sufficiently large values of m.

Now the fraction f of bad MOS's (for some values of Q and W) is bounded as follows:

$$f \leqq \sum_{Q=1}^{Q_{\max}} \sum_{W=\lceil Qc^q \rceil}^{(2c-1)Q} f_{Q,W} \leqq Q_{\max} \sum_{W=\lceil c^q \rceil}^{\infty} 2^{-\chi W (\log m)^p}$$

$$\leqq Q_{\max} 2^{-\chi c^q (\log m)^p} \sum_{W=0}^{\infty} (2^{-\chi (\log m)^p})^W \leqq Q_{\max} 2^{-\kappa (\log m)^{p+q}}$$

for some constant $\kappa > 0$ (and sufficiently large values of m). Using $Q_{\max} \leqq n$, we finally get

$$\log f \leqq -\kappa (\log m)^{p+q} + \log n,$$

and since $p + q > 1$ and $m \geqq n$, this shows that $f \to 0$ as $n \to \infty$. In particular, good graphs exist for all sufficiently large values of n. This concludes the proof of Lemma 2.4. □

For $i = 1, 2, \cdots$, let Q_i be the number of live variables immediately before the ith stage of Part 3 of the algorithm. By Lemma 2.3,

$$Q_1 \leqq n 2^{-\log m / \log \log m}.$$

This is less than $k = n/(2\lfloor \log m \rfloor - 1)$ and also less than $n 2^{-(\log m)^p}$ for any $p < 1$ for sufficiently large values of m. Hence all remaining live variables are in the range of the calls of ACCESS, and Lemma 2.4 is applicable. Let p, q and η be constants with $0 < p, q, \eta < 1$ and $p + q > 1$ and choose an MOS as given by Lemma 2.4.

We want the following assertion (1) to be true for all i:

(1) The number of live copies decreases in the ith stage from at most $Q_i(2c-1)$ to at most $Q_i c^q$.

As long as there are more than $Q_i c^q$ live copies, each call of ACCESS will decrease their number by a factor of at least η by Lemma 2.4. Hence s iterations of the inner loop are sufficient to guarantee (1) if

$$\eta^s Q_i(2c-1) \leqq Q_i c^q,$$

i.e., if

$$s \geqq \log_{1/\eta} \frac{2c-1}{c^q}.$$

Since $c \leqq \log m$, we have $\log_{1/\eta}((2c-1)/c^q) = O(\log \log m)$. Hence (1) holds given only that a_3 is chosen large enough.

By (1), the number of live copies at the beginning of the $(i+1)$st stage, for $i = 1, 2, \cdots$, is at most $Q_i c^q$. And each live variable has at least c live copies since the ith stage ended with a call of COUNT. But then

$$Q_{i+1} \leqq \frac{Q_i c^q}{c} = Q_i c^{q-1} \leqq Q_1 (c^{q-1})^i,$$

where the last step followed by induction. Hence t stages will eliminate all live variables if

$$Q_1 (c^{q-1})^t < 1,$$

i.e., if

$$t > \frac{\log Q_1}{(1-q)\log c}.$$

Since $Q_1 \leqq n$ and $\log c = \Omega(\log \log m)$, the smallest solution t is $O(\log n / \log \log m)$. In conclusion, we have proved:

LEMMA 2.5. *For a suitable choice of an MOS and of the constants a_2 and a_3, Part 3 eliminates all live variables remaining after the execution of Parts 1 and 2. Its execution time is $O(\log n)$.*

In order to combine Parts 1, 2 and 3 of the algorithm we must use an MOS that satisfies both Lemma 2.2 and Lemma 2.4 (i.e., has the properties described in the lemmas). But since the fraction of MOS's not satisfying Lemma 2.2 goes to 0 as n goes to infinity, as does the fraction not satisfying Lemma 2.4, the proportion of MOS's satisfying both conditions tends to 1, i.e., such MOS's certainly exist. Even a randomly chosen MOS for large n with high probability has the desired properties.

Summarizing our results about Parts 1, 2 and 3 as expressed in Lemmas 2.3 and 2.5, we have the following theorem:

THEOREM 1. *The shared memory of any* EREW PRAM *with n processors and m cells of shared memory may be emulated by an n-processor* MPC *with a slowdown of* $O(\log m)$.

Remark 1. As indicated in the first paragraph of § 2, Theorem 1 remains true even if we allow the simulated machine to be a CRCW PRAM.

Remark 2. It follows from the discussion towards the end of § 1 that for any PRAM program H running on an n-processor CRCW PRAM and using at most m cells of shared memory, there exists an equivalent MPC program which runs on an n-processor MPC and simulates the execution of H with a slowdown of $O(\log m)$.

3. Simulation of PRAMs on bounded-degree networks. This section considers emulations of PRAM memories by bounded-degree networks. We give an upper bound

on the slowdown which follows as a corollary to Theorem 1 and devote most of the section to the proof of a lower bound of $\Omega(\log n \log m/\log\log m)$. A similar lower bound was shown independently by Karlin and Upfal [KU].

For added clarity, let us use the word "cycle" instead of "step" when talking about network computations. In this section we will say that a data item is stored in a network processor meaning that it is stored in that processor's memory module.

THEOREM 2. *The shared memory of any CRCW PRAM with n processors and m cells of shared memory may be emulated by an n-processor bounded-degree network with a slowdown of* $O(\log n \log m)$.

Proof. This is a consequence of Theorem 1, the remark following it, and the general result that any step of an n-processor MPC may be simulated by $O(\log n)$ cycles of an n-processor bounded-degree network. The latter fact in turn follows from the existence of algorithms which sort on bounded-degree networks in $O(\log n)$ time ([AKS], [L]), together with the observation that a scheme for doing partial routing within the same time bound may be derived from any such algorithm.

Let us now turn to the lower bound and first describe our model and the assumptions made. We consider an n-processor bounded-degree network emulating the shared memory of an n-processor PRAM and identify the network processors with the nodes and the links between network processors with the edges of an undirected graph $G=(V, E)$. The distance $\text{dist}(u, v)$ between two network processors u and v is the number of edges on a shortest path in G from u to v.

Recall that when a PRAM processor executes the instruction STORE x, it presents to its associated network processor a pair $\langle$"x", $a\rangle$, where a is the new value of x. We call such a pair a *copy* of x. The copy is said to be *valid* until the next PRAM step in which x is updated (forever, if x is never again updated). We will assume that any network processor reading a PRAM variable x in a given PRAM step must contain a valid copy of x at the end of that step. We also assume that the network treats copies of variables as indivisible entities capable only of being input from or output to PRAM processors, of being routed through the network and of being stored in network processors (cf. Fig. 4). In particular, the network cannot "synthesize" copies of variables.

Regarding the cost of communication, we will assume that for each pair (u, v) of network processors, sending a copy of a variable from u to v needs at least $\text{dist}(u, v)$ *atomic actions* by the network, where an atomic action is a "processor-step," i.e., the amount of work associated with a single network processor executing a single RAM instruction. This is reasonable since each processor on a path in G from u to v must devote at least one cycle to passing on the copy.

We finally require all communication in the network to be *point to point.* Note that while our other assumptions were innocuous, this is a serious restriction. Point to point communication means that copies of variables are physical entities that cannot be subjected to duplication by the network. We allow a network processor writing a PRAM variable x to obtain an arbitrary number of copies of x from its associated PRAM processor; but any network processor which receives a copy of x from a network neighbor may forward the copy to at most one neighboring (PRAM or network) processor. As an example, if a network processor u writing a PRAM variable x wants to send a copy of x to each of h processors lying close together in the network but far from u, then u must obtain h separate copies from its associated PRAM processor and have them routed through the network. A less restrictive model would allow u to send just one copy that would be replicated by the network in the vicinity of the h destinations. We will use the assumption of point to point communication to conclude

that if during the simulation there are h instances of a processor sending a copy of a variable to a processor at a distance of at least q, then the network must perform a total of at least hq atomic actions.

Most of our assumptions may be visualized by imagining copies of variables to be contained in sealed envelopes which are carried by the network between processors of the PRAM.

Note that the assumption of point to point communication is not satisfied by our emulation algorithm, nor by the one by Upfal and Wigderson. Both algorithms use a form of load redistribution in which conceptually variables, not copies of variables, are transported through the network. This corresponds to "bundles" of $2c-1$ copies of the same variable being transported together at unit cost or, equivalently, to the replication of copies after they have been sent, something which is forbidden by the assumption of point to point communication. Thus the lower bound which we are about to derive does not apply to any of these cases. Partly due to the well-known difficulty of obtaining any sort of lower bounds for most natural problems, we nevertheless believe the result to be interesting.

Upfal and Wigderson showed in [UW] an $\Omega(\log n/\log\log n)$ lower bound on the slowdown associated with a simulation of a PRAM on an MPC. Our proof is a variation of their argument which we will therefore briefly review.

They start out with the observation (also made in [MV]) that if there is an average of r valid copies of each variable, then some PRAM steps in which all processors read need $\Omega((m/n)^{1/(2r)})$ cycles for their simulation. On the other hand, if an average of r valid copies is maintained, then the average cost of simulating a writing step is $\Omega(r)$. Thus one cannot do better than

$$\Omega\left(\min_r\left(\left(\frac{m}{n}\right)^{1/(2r)}+r\right)\right)=\Omega\left(\frac{\log n}{\log\log n}\right),$$

the desired lower bound. Of course, this informal argument ignores a lot of detail.

The main ideas behind our modified proof are as follows: Suppose that a processor u sends copies of some variable to a number of other processors. We will then (in the lower bound argument) count only those copies that go to processors at a distance of at least $q\approx\log n/(2\log d)$ from u, where d is the maximum degree of the network; transmission of all other copies is considered free. Since every counted copy requires $\Omega(\log n)$ atomic actions, the network will need $\Omega(r\log n)$ cycles to simulate a writing step updating n variables if an average of r counted copies per variable is to be kept.

What happens to reading steps? There is now for every variable an average of r counted copies and at most $\approx d^q\approx\sqrt{n}$ free copies. However, the free copies always cluster in a small "sphere" in the network, and the free copies of many variables cluster in the same sphere. This allows us to prove that the free copies are of little use in the sense that the simulation of a reading step may require $\Omega((m/(4n^2))^{1/(4r)})$ cycles. Thus the slowdown is

$$\Omega\left(\min_r\left(\left(\frac{m}{4n^2}\right)^{1/(4r)}+r\log n\right)\right)=\Omega\left(\frac{\log n\log m}{\log\log m}\right).$$

The details follow.

Before we state the lower bound, let us define a single instruction-multiple data (SIMD) program for a PRAM to be a program whose execution never causes two processors to carry out different types of instructions in the same step.

THEOREM 3. *If $m=\Omega(n^{2+\varepsilon})$ and $T\geqq(1+\varepsilon)(m/n)$ for some fixed $\varepsilon>0$, 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

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

for any on-line emulation of the shared PRAM memory by an n-processor bounded-degree network that uses only point to point communication.

Remark 1. In particular, the (worst-case) slowdown is $\Omega(\min\{\sqrt{n \log n}, \log n \log m/\log \log m\})$, but the theorem is slightly stronger: There are arbitrarily long programs that cause an "average slowdown" of $\Omega(\min\{\sqrt{n \log n}, \log n \log m/\log \log m\})$.

Remark 2. Under the assumptions of Theorem 3, the simulation time is $\Omega((\log n)^2/\log \log n)$. If m is polynomial in n, this statement is equivalent to that of Theorem 3.

Remark 3. The emulation being on-line means that each PRAM step must be simulated without knowledge of memory requests in later steps. If this requirement is relaxed, it is possible in some cases to beat the lower bound; see [VW].

Proof. Let $\tau = \lfloor (T - \lceil m/n \rceil - 1)/2 \rfloor$ and observe that $\tau = \Omega(T)$. We consider PRAM programs in which the ith processor, for $i = 1, \cdots, n$, executes a program of the following form:

```
(1) LOADINDEX;
(2) for t := 1 to ⌈m/n⌉ do
(3)    STORE y_I^{t,i};
(4) od;
(5) for t := 1 to τ do
(6)    LOAD y_R^{t,i};
(7)    STORE y_W^{t,i};
(8) od;
```

Here the first instruction loads a well-defined value (the processor index, say) into the accumulator, and the various y's are PRAM variables. The above program as given of course does not have the straight-line format, but may be thought of as a compact representation of a straight-line program whose length is bounded by T.

We will call an execution of lines (6)-(7) a stage. The tth stage, for $t = 1, \cdots, \tau$, consists of a reading part which inspects all variables in the set $Y_R^t = \{y_R^{t,1}, \cdots, y_R^{t,n}\}$, followed by a writing part which updates all variables in the set $Y_W^t = \{y_W^{t,1}, \cdots, y_W^{t,n}\}$.

Let X be the set of the PRAM's m shared variables. Lines (2)-(4) simply assign values to all variables in X in some arbitrary fixed way.

Suppose now that programs of the above form are run on a PRAM whose shared memory is emulated by a bounded-degree network satisfying the assumptions of Theorem 3 and running for at most S cycles. We will demonstrate that S is large by exhibiting a particular choice of

$$(Y_R^1, Y_W^1, \cdots, Y_R^\tau, Y_W^\tau)$$

which forces the simulation to be slow. We fix the Y's one by one, choosing Y_R^t and Y_W^t depending on the state of the emulating network at the beginning of the tth stage. The salient parameters of the state of the network are the *redundancies* of variables in X as defined below.

Each network processor which writes a PRAM variable x may send copies of x to other processors. For $t = 1, \cdots, \tau + 1$ and $x \in X$, denote by Γ_x^t the set of network processors which have a valid copy of x at the beginning of the tth stage ("at the

beginning of the $(\tau+1)$st stage" should be interpreted to mean at the end of the τth stage). Identify the network with a graph $G=(V, E)$ as in the definition of our model, let $d \geq 2$ be the maximum node degree of G, and for $u \in V$, let $B(u)$ be the sphere of radius $q=(\log n-2 \log \log n)/(2 \log d)$ around u, i.e.,

$$B(u)=\{v \in V \mid \operatorname{dist}(u, v) \leq q\}.$$

For $t=1, \cdots, \tau+1$ and $x \in X$, the redundancy r_x^t of x at the beginning of the tth stage is defined as

$$r_x^t=\min_{u \in V}|\Gamma_x^t \backslash B(u)|;$$

i.e., take the sphere $B(u)$ best fitting the copies of x and count the number of copies outside $B(u)$. The total redundancy R^t and the average redundancy r^t at the beginning of the tth stage are

$$R^t=\sum_{x \in X} r_x^t$$

and

$$r^t=R^t/m.$$

The following Lemma 3.1 generalizes Theorem 4.1 in [UW] and § III.1 of [MV].

LEMMA 3.1. *Let $1 \leq t \leq \tau$ and suppose that $Y_R^1, Y_W^1, \cdots, Y_R^{t-1}, Y_W^{t-1}$ (and hence r^t) are fixed. Then it is possible to choose Y_R^t in such a way that the tth stage of the simulation (in fact, its reading part) must consist of $\Omega(g(r^t))$ cycles, where*

$$g(r)=\begin{cases}\min\left\{\sqrt{n} \log n, \dfrac{n}{r}, \left(\dfrac{m}{4n^2}\right)^{1/(4r)}\right\} & \text{if } r>0,\\ \sqrt{n} \log n & \text{if } r=0.\end{cases}$$

Proof. Let us drop the superscript t from Γ_x^t, r_x^t, R^t and r^t. Let $\{X_v \mid v \in V\}$ be a partition of X (empty sets allowed) such that

$$\forall x \in X_v: r_x=|\Gamma_x \backslash B(v)|.$$

Then there is an X_v which is not much smaller than average and whose redundancy is not much larger than average, i.e.,

Claim 1.

$$\exists v \in V: |X_v| \geq \frac{m}{2n} \quad \text{and} \quad \sum_{x \in X_v} r_x \leq 2r|X_v|.$$

Proof. Assume otherwise and let $U=\{v \in V \mid |X_v| \geq m/(2n)\}$. Then $U \neq \varnothing$ and

$$R \geq \sum_{v \in U} \sum_{x \in X_v} r_x > \sum_{v \in U} 2r|X_v| > 2r\frac{m}{2}=R$$

since

$$\sum_{v \notin U}|X_v|<n\frac{m}{2n}=\frac{m}{2}.$$

This is a contradiction. □

Choose $v \in V$ as given by Claim 1, i.e., $|X_v| \geqq m/(2n)$ and $\sum_{x \in X_v} r_x \leqq 2r|X_v|$. Then there is a set $A \subseteq X_v$ with $|A| \geqq |X_v|/2 \geqq m/(4n)$ such that $\forall x \in A$: $r_x \leqq 4r$. Let $f = |B(v)|$. A simple combinatorial argument shows that

$$f \leqq d^q + 1 = \frac{\sqrt{n}}{\log n} + 1.$$

Now for any set $W \subseteq V \backslash B(v)$, let

$$A_W = \{x \in A \mid \Gamma_x \subseteq B(v) \cup W\}.$$

Claim 2. For any k with $0 \leqq k \leqq n - f$, there is a subset W of $V \backslash B(v)$ with $|W| = k$ and

$$|A_W| \geqq |A| \binom{n - f - \lfloor 4r \rfloor}{k - \lfloor 4r \rfloor} \Big/ \binom{n - f}{k}.$$

Proof. Let $\mathcal{W} = \{W \subseteq V \backslash B(v) \mid |W| = k\}$. For any $x \in A$, there are exactly

$$\binom{n - f - r_x}{k - r_x}$$

sets $W \in \mathcal{W}$ with $x \in A_W$. Since $r_x \leqq \lfloor 4r \rfloor$ and hence

$$\binom{n - f - r_x}{k - r_x} \geqq \binom{n - f - \lfloor 4r \rfloor}{k - \lfloor 4r \rfloor},$$

we have

$$\sum_{W \in \mathcal{W}} |A_W| \geqq |A| \binom{n - f - \lfloor 4r \rfloor}{k - \lfloor 4r \rfloor}.$$

Since

$$\mathcal{W} = \binom{n - f}{k},$$

Claim 2 now follows by the pigeonhole principle. □

We can now finish the proof of Lemma 3.1. We consider only the case $r > 0$ and leave the verification for $r = 0$ to the reader. Let

$$k = \left\lceil 4r + n \left(\frac{m}{4n^2} \right)^{-1/(4r)} \right\rceil.$$

We may assume that $r \leqq n/8$ and $(m/(4n^2))^{1/(4r)} \geqq 3$ since otherwise what we are claiming is trivial. This allows us to also assume that n is large enough to make $n - f \geqq k$. Then

$$\frac{|A|}{n} \geqq \frac{m}{4n^2} \geqq \left(\frac{n}{k - 4r} \right)^{4r} \geqq \left(\frac{n}{k - \lfloor 4r \rfloor} \right)^{\lfloor 4r \rfloor}$$

$$\geqq \frac{[n - f]_{\lfloor 4r \rfloor}}{[k]_{\lfloor 4r \rfloor}} = \binom{n - f}{k} \Big/ \binom{n - f - \lfloor 4r \rfloor}{k - \lfloor 4r \rfloor},$$

and Claim 2 provides us with a subset W of $V \backslash B(v)$ with $|W| = k$ and $|A_W| \geqq n$. Choose Y'_R as any n-element subset of A_W. By construction, all valid copies of variables in Y'_R are contained in a total of $f + k$ processors. Hence the network cannot possibly

output copies of all variables in Y_R^t, i.e., simulate line (6), in fewer than $n/(f+k)$ cycles. But since

$$f+k \leqq \frac{\sqrt{n}}{\log n}+1+4r+n\left(\frac{m}{4n^2}\right)^{-1/(4r)}+1 = O\left(\max\left\{\frac{\sqrt{n}}{\log n}, r, n\left(\frac{m}{4n^2}\right)^{-1/(4r)}\right\}\right),$$

we have

$$\frac{n}{f+k} = \Omega\left(\min\left\{\sqrt{n}\log n, \frac{n}{r}, \left(\frac{m}{4n^2}\right)^{1/(4r)}\right\}\right). \qquad \square$$

We now continue the proof of Theorem 3 and first give a rule for choosing Y_R^t and Y_W^t. For $t=1,\cdots,\tau$, Y_R^t is chosen as in Lemma 3.1, and Y_W^t is chosen among all sets of n variables to maximize $\sum_{x\in Y_W^t} r_x^t$. Clearly $\sum_{x\in Y_W^t} r_x^t \geqq nr^t$. Note that this means that the writing part of stage t invalidates at least nr^t of those copies of variables which are counted in the total redundancy. On the other hand, writing such a copy costs $\Omega(q)=\Omega(\log n)$ atomic actions on the average, and each copy must be paid for separately by the assumption of point to point communication. To see this, consider a network processor u which writes a PRAM variable x in the tth stage, and let $t_1 = \min(\{\tau+1\}\cup\{t'>t \mid x\in Y_W^{t'}\})$ (i.e., x is next updated in the t_1th stage). Assume that totalled over stages $t,\cdots,t_1-1$, exactly h valid copies of x leave $B(u)$. The cost of this is clearly $\Omega(hq)=\Omega(h\log n)$ by the assumption of point to point communication. On the other hand, $r_x^{t'}$, for $t<t'\leqq t_1$, can never exceed h since $r_x^{t'}$ by definition is the minimum number of valid copies of x outside any sphere $B(v)$, $v\in V$. Hence the average cost per counted copy is $\Omega(\log n)$.

Since $R^1=0$, $R^{\tau+1}\geqq 0$, and the total number of atomic actions performed by the network is at most Sn, we have the relation

$$\frac{Sn}{\log n} - \sum_{t=1}^{\tau} nr^t \geqq 0$$

or

$$S \geqq \tau \hat{r} \log n \tag{2}$$

where $\hat{r} = (1/\tau)\sum_{t=1}^{\tau} r^t$ is the value of r^t averaged over all stages.

Relation (2) expresses the cost of writing. In order to investigate the cost of reading as given by Lemma 3.1, we shall consider separately the various "cases" of the function g. Let

$$g_1(r) = \sqrt{n}\log n,$$

$$g_2(r) = \begin{cases} n/r & \text{if } r>0, \\ \infty & \text{if } r=0, \end{cases}$$

$$g_3(r) = \begin{cases} (m/(4n^2))^{1/(4r)} & \text{if } r>0, \\ \infty & \text{if } r=0, \end{cases}$$

and let $\{C_1, C_2, C_3\}$ be a partition of $\{1,\cdots,\tau\}$ such that for $i=1,2,3$,

$$\forall t\in C_i: g(r^t) = g_i(r^t).$$

Next choose $l\in\{1,2,3\}$ such that C_l is a biggest set in the partition, i.e., $|C_l| = \max_{1\leqq j\leqq 3}|C_j|$. Then there is a subset C' of C_l with

$$|C'| \geqq \frac{\tau}{6} \quad \text{and} \quad \forall t\in C': r^t \leqq 6\hat{r}.$$

By Lemma 3.1 and (2), we have

$$S=\Omega\left(\max\left\{\tau\hat{r}\log n,\ \sum_{t\in C'} g_l(r^t)\right\}\right)=\Omega(T(\hat{r}\log n+g_l(6\hat{r}))).$$

Consider three cases depending on l. For $l\in\{1,2\}$, clearly

$$\min_{\hat{r}\geq 0}(\hat{r}\log n+g_l(6\hat{r}))=\Omega(\sqrt{n\log n}).$$

As for the case $l=3$, observe first that $m=\Omega(n^{2+\varepsilon})$ implies $m/(4n^2)=\Omega(m^{\varepsilon/(2+\varepsilon)})$. Letting $\rho=\varepsilon\log m/(48(2+\varepsilon)\log\log m)$, we find

$$\min_{\hat{r}\geq\rho}(\hat{r}\log n+g_3(6\hat{r}))\geq\rho\log n=\Omega\left(\frac{\log n\log m}{\log\log m}\right)$$

and

$$\begin{aligned}\min_{0\leq\hat{r}\leq\rho}(\hat{r}\log n+g_3(6\hat{r}))\geq g_3(6\rho)&=\left(\frac{m}{4n^2}\right)^{2(2+\varepsilon)\log\log m/(\varepsilon\log m)}\\&=\Omega((m^{\varepsilon/(2+\varepsilon)})^{(2+\varepsilon)2\log\log m/(\varepsilon\log m)})\\&=\Omega(2^{2\log\log m})=\Omega((\log m)^2)=\Omega\left(\frac{\log n\log m}{\log\log m}\right).\end{aligned}$$

Putting the various cases together finally yields

$$S=\Omega\left(T\min\left\{\sqrt{n\log n},\frac{\log n\log m}{\log\log m}\right\}\right).$$ □

Remark. The lower bound of Theorem 3 makes it natural to wonder whether a slowdown of close to $O(\sqrt{n})$ is achievable for arbitrary values of m. This is indeed the case. Assume for simplicity that n is a perfect square and consider the n emulating processors as arranged in a $\sqrt{n}$-by-$\sqrt{n}$ array. We use the following idea: When a PRAM processor P updates a variable x, its associated processor places a copy of x in each processor in its row (call this a *row operation*). When P wants to read the value of a variable x, the associated processor obtains a copy of x from each processor in its column which has a copy of x (a *column operation*). Clearly one of the copies will be valid, and it may be found using time stamps as in § 2.

It remains to determine the time needed for row and column operations. Observe that the procedure BROADCAST of § 2 actually performs a row operation if we consider each row as a cluster and execute a call BROADCAST($\sqrt{n}$). In the same way, a column operation is part of what happens in the execution of COUNT($\sqrt{n}$) with the columns considered as clusters. Hence if the emulating machine is an MPC, the slowdown is $O(\sqrt{n}+\log\sqrt{n})=O(\sqrt{n})$. Since COUNT and BROADCAST both use a fixed boundeded-degree interconnection pattern, namely a complete binary tree for each cluster, the same slowdown is attainable on a bounded-degree network. However, any such network must of course employ communication which is not point to point since otherwise it would violate the lower bound of Theorem 3 for sufficiently large values of m. On a bounded-degree network using only point to point communication, a slowdown of $O(\sqrt{n}\log n)$ is achievable. In order to see this, note that the necessary communication can be realized by a constant number of rounds, where in a round each processor sends at most one message to each processor in its row (or column). A round may clearly be implemented by $\sqrt{n}$ suitable permutations within each row

(column), e.g., by cyclic shifts by $1, 2, \cdots$, and finally by $\sqrt{n}$ positions, and each such permutation may be routed in $O(\log n)$ time if the processors within each row (column) are connected by, e.g., a cube-connected cycles network [PV].

4. An algorithm for approximate redistribution. This section establishes an upper bound on the parallel complexity of the strong redistribution problem. We also show how this result allows our simulation of PRAMs on MPCs to be somewhat simplified.

Our motivation for studying the redistribution problem is that it captures some aspects of a problem which is fundamental in parallel computing, namely that of distributing the work to be done evenly among the available processors in order to keep them all busy for as long as possible. We believe that the results and methods of this section may have applications outside of our simulation algorithm.

The main steps in the proof are as follows: We first show the existence of certain expander graphs and then indicate how these may be used to construct fast ε-halving procedures. This part of the proof parallels arguments in [AKS] except that we are being more explicit and want to correct a few inaccuracies. We then give an algorithm based on repeated ε-halving, bound its execution time and finally show that it solves the strong approximate data compaction problem.

Given an undirected graph $G=(V, E)$ and a set $U \subseteq V$ of nodes, let us write $\Gamma_G(U)$ or $\Gamma(U)$ for the set of nodes adjacent to a node in U, i.e., $\Gamma_G(U) = \{v \in V \mid (u, v) \in E \text{ for some } u \in U\}$.

DEFINITION. A bipartite graph on the node sets V_1 and V_2 is called an *expander graph* with parameters $(n_1, n_2, \lambda, \alpha, \nu)$ if $|V_1| = n_1$, $|V_2| = n_2$, and

(1) $\forall v \in V_1 \cup V_2$: $|\Gamma(\{v\})| \leqq \nu$,

(2) $\forall A \subseteq V_i$: $|A| \leqq \alpha n_{3-i} \Rightarrow |\Gamma(A)| \geqq \lambda |A|$, for $i = 1, 2$.

LEMMA 4.1. *For all $b > 0$, there is a constant K such that for all $\lambda \geqq 1$ ($\lambda \in \mathbb{R}$), $n \geqq 1$ and $\delta \in \{0, 1\}$ there exists an expander graph with parameters $(n+\delta, n, \lambda, 1/(\lambda+b), K\lambda^2)$.*

Proof. This is again a counting argument. We first consider the case $\delta = 0$. Let A and B be disjoint sets with $|A| = |B| = n$. Given ν bijections $\pi_1, \cdots, \pi_\nu$ from A to B, we may as in the proof of Lemma 2.1 construct a bipartite graph of maximum degree ν on the node sets A and B by drawing an edge from a to $\pi_l(a)$ for all $l = 1, \cdots, \nu$, $a \in A$. Let $\mathcal{G}_\nu$ denote the set of all such graphs and call a graph in $\mathcal{G}_\nu$ good if it is an expander graph with parameters $(n, n, \lambda, 1/(\lambda+b), \nu)$.

If a graph in $\mathcal{G}_\nu$ is not good, then there is a subset U of either A or B such that $1 \leqq |U| \leqq n/(\lambda+b)$ and $\Gamma(U) \subseteq V$ for some set V with $|V| = \lfloor \lambda |U| \rfloor$. For $z = 1, \cdots, \lfloor n/(\lambda+b) \rfloor$, let $f_{\nu,z}$ denote the fraction of graphs in $\mathcal{G}_\nu$ for which there is such a set U with $|U| = z$. As in the proofs of Lemmas 2.1 and 2.4 it is easy to see that (the factor 2 is due to the fact that one may choose $U \subseteq A$ or $U \subseteq B$)

$$f_{\nu,z} \leqq 2\binom{n}{z}\binom{n}{\lfloor \lambda z \rfloor}\left(\frac{[\lfloor \lambda z \rfloor]_z}{[n]_z}\right)^\nu \leqq 2\left(\frac{en}{z}\right)^z \left(\frac{en}{\lfloor \lambda z \rfloor}\right)^{\lfloor \lambda z \rfloor} \left(\frac{\lfloor \lambda z \rfloor}{n}\right)^{\nu z} \leqq 2\left[e^{\lambda+1}\frac{\lfloor \lambda z \rfloor^{\nu-\lambda}}{n^{\nu-\lambda-1}z}\right]^z.$$

Proceeding under the assumption $\nu \geqq \lambda + 1$, we further get

$$f_{\nu,z} \leqq 2\left[e^{\lambda+1}\left(\frac{\lambda z}{n}\right)^{\nu-\lambda-1}\lambda\right]^z \leqq 2\left[e^{\lambda+1}\left(\frac{\lambda}{\lambda+b}\right)^{\nu-\lambda-1}\lambda\right]^z,$$

where $z \leqq n/(\lambda+b)$ was used in the last step. Let

$$g_\nu = e^{\lambda+1}\left(\frac{\lambda}{\lambda+b}\right)^{\nu-\lambda-1}\lambda.$$

Then

$$\log g_\nu = (\lambda+1)\log e - (\nu-\lambda-1)\log\left(1+\frac{b}{\lambda}\right)+\log\lambda$$

$$\leqq (\lambda+1)\log e - \frac{(\nu-\lambda-1)}{\lambda}\log(1+b)+\log\lambda$$

which is $\leqq -2$ if

$$\nu \geqq \frac{\lambda(\lambda+1)\log e+\lambda\log\lambda+2\lambda}{\log(1+b)}+\lambda+1.$$

Hence if $\nu \geqq K\lambda^2$ for a suitable constant K depending only on b, we have

$$g_\nu \leqq \tfrac{1}{4}$$

and hence

$$f_{\nu,z} \leqq \frac{2}{4^z}.$$

The total fraction of bad graphs in $\mathcal{G}_\nu$ is then bounded by

$$\sum_{z=1}^{\lfloor n/(\lambda+b)\rfloor} f_{\nu,z} \leqq \sum_{z=1}^{\infty}\frac{2}{4^z} < 1$$

which proves the lemma for the case $\delta = 0$.

We now turn to the case $\delta = 1$. Suppose that we must produce an expander graph with parameters $(n+1, n, \lambda, 1/(\lambda+b), K\lambda^2)$ for given b, λ and n. First take an expander graph G with node sets A and B and parameters $(n+1, n+1, \lambda+b/2, 1/(\lambda+b), K\lambda^2)$ which exists by the proof for the case $\delta = 0$ for some constant K depending only on b. We may assume (increasing K as necessary) that extra edges have been added to achieve that each node in G has at least $2\lambda/b$ incident edges.

Now remove one node $v \in B$ and its incident edges. We claim that the resulting graph G' is an expander graph with parameters $(n+1, n, \lambda, 1/(\lambda+b), K\lambda^2)$. We must show that a sufficiently small subset U of either A or $B\backslash\{v\}$ has $|\Gamma_{G'}(U)| \geqq \lambda|U|$. Since this is trivial for $U \subseteq B$, we may assume that $U \subseteq A$, $1 \leqq |U| \leqq n/(\lambda+b)$. Then

$$|\Gamma_{G'}(U)| \geqq |\Gamma_G(U)| - 1 \geqq \left(\lambda+\frac{b}{2}\right)|U| - 1,$$

and if $|U| \geqq 2/b$, this allows the desired conclusion $|\Gamma_{G'}(U)| \geqq \lambda|U|$. On the other hand, if $|U| \leqq 2/b$, then an arbitrary node u in U by construction has a sufficient number of neighbors, i.e.,

$$|\Gamma_{G'}(U)| \geqq |\Gamma_{G'}(\{u\})| \geqq \frac{2}{b}\lambda \geqq \lambda|U|. \qquad \square$$

DEFINITION. A permutation π of $\{1, \cdots, n\}$ is said to be *ε-halved* if

(1) $\forall l \in \{1, \cdots, \lceil n/2\rceil\}$: $|\{i \mid 1 \leqq i \leqq l \text{ and } \pi(i) > \lceil n/2\rceil\}| \leqq \varepsilon l$;

(2) $\forall l \in \{\lceil n/2\rceil + 1, \cdots, n\}$: $|\{i \mid l \leqq i \leqq n \text{ and } \pi(i) \leqq \lceil n/2\rceil\}| \leqq \varepsilon(n-l+1)$.

Consider again n locations arranged in a fixed order and each containing a record with a key. The configuration may be said to represent a permutation π of $\{1, \cdots, n\}$ if the record stored in the $\pi(i)$th location, for $i = 1, \cdots, n$, contains the key i. A procedure is called an ε-halver on n elements if it takes as input a configuration

representing an arbitrary permutation of $\{1, \cdots, n\}$ and produces as output a configuration representing an ε-halved permutation. An ε-halver thus approximately separates the small and the big keys.

LEMMA 4.2. *There is a constant K such that for all $n \geq 1$ and all $\varepsilon > 0$, there exists an ε-halver on n elements consisting of at most $K(1/\varepsilon)^2$ parallel comparison-exchange steps on n elements.*

Proof. Assume without loss of generality that $\varepsilon \leq \frac{1}{2}$. We consider two cases.

Case 1: $n \leq 8/\varepsilon^2$. Here Batcher's well-known construction [B] gives us not only an ε-halver, but actually a sorting algorithm. For some constant K', it consists of at most $K'(\log n)^2 \leq 25K'(\log(1/\varepsilon))^2 \leq 25K'(1/\varepsilon)^2$ parallel comparison-exchange steps.

Case 2: $n > 8/\varepsilon^2$. Now let $\lambda = (1-\varepsilon)/\varepsilon$ and consider an ordered sequence $L_1, \cdots, L_n$ of n locations whose contents represent a permutation of $\{1, \cdots, n\}$. We assume that each record consists only of its key.

By Lemma 4.1, there is an expander graph G with parameters $(\lceil n/2 \rceil, \lfloor n/2 \rfloor, \lambda, 1/(\lambda + \frac{1}{2}), K\lambda^2)$ for some constant K. As in the proof of Lemma 2.1 we may identify the node sets of G with $A = \{L_1, \cdots, L_{\lceil n/2 \rceil}\}$ and $B = \{L_{\lceil n/2 \rceil + 1}, \cdots, L_n\}$ and consider each edge (L_i, L_j) as a comparison-exchange operation to take place between L_i and L_j. Since no node in G has more than $K\lambda^2$ incident edges, it is possible to carry out all comparison-exchange operations in at most $K\lambda^2 \leq K(1/\varepsilon)^2$ parallel comparison-exchange steps (this is true without additional assumptions on G; see [O, Thm. 7.5.6]. However, the reader may also imagine G to have been constructed as in the proof of Lemma 4.1, in which case the claim becomes trivial).

Suppose that the resulting algorithm $\mathcal{A}$ is not an ε-halver. We will show that this leads to a contradiction. Assume without loss of generality (the other situation being similar) that for some $l \in \{1, \cdots, \lceil n/2 \rceil\}$, more than εl keys $\leq l$ are in locations in B after the execution of $\mathcal{A}$. Call a key "small" exactly if it is $\leq l$ and let B', $|B'| = \lceil \varepsilon l \rceil$, be a set of locations in B containing only small keys after the execution of $\mathcal{A}$. Then

$$|B'| \leq \lceil \varepsilon \lceil n/2 \rceil \rceil \leq \varepsilon \lfloor n/2 \rfloor + 2 \leq \frac{\lfloor n/2 \rfloor}{\lambda + \frac{1}{2}},$$

where the last inequality may be shown correct by elementary manipulations using the relation $n > 8/\varepsilon^2$ and the definition of λ. But this means that B' is small enough to be "expanded" by G, i.e.,

$$|\Gamma(B')| \geq \lambda |B'| \geq (1-\varepsilon) l.$$

Since more than εl small keys are in locations in B after the execution of $\mathcal{A}$, fewer than $l - \varepsilon l = (1-\varepsilon) l$ are in locations in A. But then some location in B' is necessarily linked by an edge to a location in A containing a bigger key. As in the proof of Lemma 2.1, this is impossible. Hence $\mathcal{A}$ is an *ε-halver.* □

THEOREM 4. *There exists a constant K such that for all n, R, and y with $1 \leq y < R \leq n$, the strong approximate data compaction (or redistribution) problem with n elements, upper bound R, and y allowable errors is solved by an algorithm consisting of at most*

$$K\left(\frac{R}{y}\right)^2 \left\lceil \log \frac{n}{y} \right\rceil^3$$

parallel comparison-exchange steps on n elements.

Proof. If $y = 1$, we may use Batcher's sorting network. Hence assume that $y \geq 2$. Let the locations $L_1, \cdots, L_n$ contain the records to be permuted, and assume as usual that each record is just a one-bit key. We will use repeated ε-halving, first on the whole set $L_1, \cdots, L_n$ of locations, then on its left and right halves, etc., until the pieces become smaller than y. More formally, when $I = \{a, \cdots, b\}$ is a finite set of consecutive

integers (call this an *interval*), we denote by I_l and I_r its left and right halves, i.e.,

$$I_l = \left\{a, \cdots, \left\lfloor \frac{a+b}{2} \right\rfloor\right\}, \qquad I_r = \left\{\left\lfloor \frac{a+b}{2} \right\rfloor + 1, \cdots, b\right\}.$$

We will also use the notation $L_I = \{L_i \mid i \in I\}$. Let $S_0 = \{\{1, \cdots, n\}\}$, and for $t = 1, 2, \cdots$, let

$$S_t = \{J \mid J = I_l \text{ or } J = I_r \text{ for some } I \in S_{t-1}\}.$$

Choose T minimal such that $|I| \leqq y$ for all $I \in S_T$ and let

$$\varepsilon = \frac{y}{3R(T+1)}.$$

The algorithm for approximate data compaction now is

```
(1) for t := 0 to T do
(2)   for all I ∈ S_t pardo (* do in parallel *)
(3)     Run an ε-halver on L_I;
(4)   odpar;
(5) od;
```

We call an execution of line (3) an ε-halving step. Each L_I should of course be considered ordered with $L_i < L_j \Leftrightarrow i < j$.

According to Lemma 4.2, the above algorithm may be implemented by $(T+1)K'(1/\varepsilon)^2$ parallel comparison-exchange steps for some constant K'. Since $T+1 \leqq \log(n/y)+2 \leqq 3\lceil \log(n/y) \rceil$ and hence $1/\varepsilon \leqq 9(R/y)\lceil \log(n/y) \rceil$, this is less than

$$K\left(\frac{R}{y}\right)^2 \left\lceil \log \frac{n}{y} \right\rceil^3$$

for a suitable constant K.

It remains only to show that the algorithm is correct, i.e., solves the strong approximate data compaction problem. We need a few definitions. First, denote by Q the total number of keys equal to 0. Then for $t = 0, \cdots, T+1$ and $I \in S_t$, let $\mu_t(I)$ be the number of 0's in (locations in) L_I after t ε-halving steps, let $\hat{\mu}(I)$ be the number of 0's that would result from a perfect data compaction, and let $\phi_t(I)$ be the "surplus" of 0's. More precisely,

$$\mu_t(I) = |\{i \in I \mid L_i \text{ contains a 0 after exactly } t \text{ } \varepsilon\text{-halving steps}\}|,$$
$$\hat{\mu}(I) = |I \cap \{1, \cdots, Q\}|, \text{ and}$$
$$\phi_t(I) = \max\{\mu_t(I) - \hat{\mu}(I), 0\}.$$

A global measure of the error accumulated after t ε-halving steps is given by

$$\Phi(t) = \sum_{I \in S_t} \phi_t(I).$$

We claim that Φ increases by at most εQ in a given ε-halving step, say the tth. Let $I \in S_{t-1}$. First it is obvious that if $Q \notin I$, then $\phi_t(I_l) + \phi_t(I_r) = \phi_{t-1}(I)$. Hence we need only consider the unique interval $I \in S_{t-1}$ with $Q \in I$ and show that

$$\phi_t(I_l) + \phi_t(I_r) \leqq \phi_{t-1}(I) + \varepsilon Q.$$

Note that ε-halvers were designed to sort permutations of $\{1, \cdots, n\}$. Since we consider algorithms consisting exclusively of comparison-exchange steps, ε-halving may also be applied to sequences of 0's and 1's. In order to deduce what happens, it is useful to imagine each one-bit key to have been augmented by a unique suffix and the comparisons to take place using the lexicographic order on the augmented keys. This would not change the resulting distribution of zero keys. Hence one may think of each

key as belonging in one particular position in the sorted sequence. It is now easy to see that the distribution of 0's between L_{I_l} and L_{I_r} achieved by any ε-halving of L_I may be obtained from the distribution that would result from a perfect sorting of L_I by moving at most εq 0's from L_{I_l} to L_{I_r} or vice versa, where q is the number of 0's in L_I. Furthermore, this would change $\phi_t(I_l)+\phi_t(I_r)$ by at most εq. On the other hand, if the tth ε-halving step happens to sort L_I perfectly, then $\phi_t(I_l)+\phi_t(I_r)=\phi_{t-1}(I)$. Hence

$$\phi_t(I_l)+\phi_t(I_r)\leqq\phi_{t-1}(I)+\varepsilon q\leqq\phi_{t-1}(I)+\varepsilon Q,$$

proving our claim.

Now for $t=0,\cdots,T+1$, let y_t be the number of errors in the configuration obtained after t ε-halving steps, i.e.,

$$y_t=\{i\in\mathbb{N}\,|\,Q<i\leqq n \text{ and } L_i \text{ contains a 0 after exactly } t\ \varepsilon\text{-halving steps}\}.$$

If $I\in S_t$ is the interval containing Q, and $J\in S_t$ is an interval to the right of I (i.e., $i<j$ for all $i\in I, j\in J$), then $\phi_t(J)=\mu_t(J)$. Hence $y_t\leqq\Phi(t)+|I|$. In particular,

$$y_{T+1}\leqq\Phi(T+1)+\left\lceil\frac{y}{2}\right\rceil\leqq(T+1)\varepsilon Q+\left\lceil\frac{y}{2}\right\rceil\leqq(T+1)\varepsilon R+\left\lceil\frac{y}{2}\right\rceil\leqq\frac{y}{3}+\left\lceil\frac{y}{2}\right\rceil\leqq y. \qquad \square$$

Consider now the implications on the simulation algorithm in § 2. Armed with the result of Theorem 4, we are no longer constrained to run Part 1 of the simulation for just $O(\log\log m)$ stages. In fact, suppose that we increase the number of stages in Part 1 to $v=\lceil(\log m)^{1/4}\rceil$. Then the total time spent in calls of APPROXIMATE-REDISTRIBUTE is

$$\sum_{i=1}^{v}S_i=O\left(\sum_{i=1}^{v}(\log 2^i)^3\right)=O\left(\sum_{i=1}^{v}i^3\right)=O(v^4)=O(\log m),$$

and the same bound holds for the time spent in the remainder of Part 1. The number of live variables is reduced to at most

$$n2^{-v}\leqq n2^{-(\log m)^{1/4}}.$$

This suffices for an input to Part 3 of the simulation, as may be seen by reviewing the proof of Lemma 2.5. Hence Theorem 4 allows us to simplify the simulation algorithm by removing Part 2.

5. Conclusion. In this paper we achieved an upper bound of $O(\log m)$ on the slowdown incurred by an n-processor MPC emulating the shared memory of an n-processor PRAM with m shared memory cells. Our scheme leaves open a number of important problems.

The most troublesome unsolved problem is that of efficient (polynomial-time) construction of good MOS's. This computation needs to be carried out only once for given values of n and m. However, we know no method for doing this except systematically generating one MOS after another and testing "goodness" by brute force, and this is not a feasible approach for interesting values of n and m like $n=10^3$ and $m=10^6$.

Finding a polynomial-time construction algorithm appears to be very difficult because of the close connection to the problem of efficient construction of expander graphs. A polynomial-time algorithm was discovered for the latter problem after a considerable effort [GG]; the MOS construction problem, however, seems more difficult (cf. Lemma 2.4). In fact, the graph should maintain the expanding property even when an adversary is allowed to eliminate all but c^q of the $2c-1$ edges incident on each node representing a PRAM variable.

We assume, as do Upfal and Wigderson, that each MPC processor, when presented with the name of a variable x, is able to tell which of the n memory modules contains "its" copy of x (i.e., $\psi_l(x)$ if the processor is the lth in its cluster). If this mapping has no regularity to be taken advantage of, we need $\Omega(m \log n)$ extra bits per processor to store it in the form of a table, a very high overhead. Therefore it is important to find a good MOS with a short description.

There are two ways in which one might want to strengthen the lower bound of Theorem 3. First, we would like to get rid of the factor $\log \log m$ in the lower bound in order to obtain matching upper and lower bounds. In order to do this, it seems to be necessary to refine the analysis to not only count the routing steps necessary to bring copies to where they are needed, but also somehow take into account the difficulty of deciding where a valid copy may be found. Note that our emulation algorithm realizes a read instruction by accessing many copies and using time stamps to find a valid copy.

The other obvious improvement would be to obviate the need for the assumption of point to point communication. Even without this assumption, the cost of setting up r copies of some variable is at least proportional to the number of edges in a smallest Steiner tree spanning the processors containing the r copies. We believe that if the processors containing the copies are chosen to often make this quantity significantly smaller than $r \log n$, i.e., writing is cheap, then the copies necessarily cluster, in a sense still to be appropriately defined, and reading must be expensive.

REFERENCES

[A] L. ADLEMAN, *Two theorems on random polynomial time*, Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 75-83.

[AKS] M. AJTAI, J. KOMLÓS AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 1-9.

[B] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Computer Conference, 32 (1968), pp. 307-314.

[BH] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 338-344.

[FW] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th Annual ACM Symposium on Theory of Computing, 1978, pp. 114-118.

[GG] O. GABBER AND Z. GALIL, *Explicit constructions of linear size concentrators and superconcentrators*, Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, pp. 364-370.

[K] D. J. KUCK, *A survey of parallel machine organization and programming*, Comput. Surveys, 9 (1977), pp. 29-59.

[KU] A. R. KARLIN AND E. UPFAL, *Parallel hashing—an efficient implementation of shared memory*, Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 160-168.

[L] T. LEIGHTON, *Tight bounds on the complexity of parallel sorting*, Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 71-80.

[MV] K. MEHLHORN AND U. VISHKIN, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories*, Acta Inform., 21 (1984), pp. 339-374.

[O] O. ORE, *Theory of Graphs*, American Mathematical Society, Providence, RI, 1962.

[PV] F. P. PREPARATA AND J. VUILLEMIN, *The cube-connected cycles: A versatile network for parallel computation*, Comm. ACM, 24 (1981), pp. 300-309.

[R] J. H. REIF, *On the power of probabilistic choice in synchronous parallel computations*, Proc. 9th Internat. Coll. Automata, Languages and Programming, 1982, pp. 442-450.

[U] E. UPFAL, *A probabilistic relation between desirable and feasible models of parallel computation*, Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 258-265.

[UW] E. UPFAL AND A. WIGDERSON, *How to share memory in a distributed system*, Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, 1984, pp. 171-180.

[VW] U. VISHKIN AND A. WIGDERSON, *Dynamic parallel memories*, Inform. and Control, 56 (1983), pp. 174-182.