Trade-offs Between Communication Throughput and Parallel Time

Yishay Mansour *Noam Nisan* *Uzi Vishkin*

November 23, 1998

Abstract

We study the effect of limited communication throughput on parallel computation in a setting where the number of processors is much smaller than the length of the input. Our model has p processors that communicate through a shared memory of size m. The input has size n, and can be read directly by all the processors. We will be primarily interested in studying cases where $n \gg p \gg m$.

As a test case we study the list reversal problem. For this problem we prove a time lower bound of $\Omega(\frac{n}{\sqrt{mp}})$. (A similar lower bound holds also for the problems of sorting, finding all unique elements, convolution, and universal hashing.) This result demonstrates that limiting the communication (i.e., small m) could have significant effect on parallel computation. We show an almost matching upper bound of $O(\frac{n}{\sqrt{mp}} log^{O(1)} n)$. The upper bound requires the development of a few interesting techniques which can alleviate the limited communication in some general settings. Specifically, we show how to emulate a large shared memory on a limited shared memory efficiently. The lower bound applies even to randomized machines, and the upper bound is a randomized algorithm.

We also argue that some standard methodology for designing parallel algorithms appears to require a relatively high level of communication throughput. Our results suggest that new alternative methodologies that need a lower such level must be invented for parallel machines that enable a low level of communication throughput, since otherwise

those machines will be severly handicapped as general purpose parallel machines. Although we do not rule that out, we cannot offer any encouraging evidence to suggest that such new methodologies are likely to be found,

1 Introduction

In this paper we study the communication needs of parallel computers. This measure is particularly meaningful in a setting where the number of processors, p, is smaller than the size of the input, n. While having received relatively little attention in the theoretical literature, such a setting is well-motivated by practical considerations.

Let us consider, informally, two extreme situations of the amount of communication allowed between processors: One extreme is where the processors cannot exchange any data among themselves. Clearly, the potential parallelism (due to having p processors) is completely useless in this case. The other extreme is where each processor can send and receive one data item in each time unit. Namely, the total communication in each time step is $O(p)$. Under this situation any parallel (PRAM) algorithm can be implemented, using known techniques, utilizing the full parallelism inherent in the algorithm. Thus such an amount of communication suffices for any parallel computation.

Suppose now that processors can exchange data among themselves, but at a rate, to be later defined and explicated as **communication throughput**, which is somewhere between these two extremes. To what extent does limiting the communication throughput of a parallel computer distant the performance from the ideal performance? This is the main question addressed in this paper.

The main conceptual contribution of this paper is the presentation of a model of computation which facilitates regulating the communication throughput. We present the model, justify it, and relate the definition to existing work.

The main technical contribution is the study of the communication throughput requirements of a representative problem, "list reversal". We establish lower bounds for parallel time as a function of the available communication throughput, and develop an algorithm whose performance almost matches the lower bounds.

Our lower bound technique is quite general and holds for other problems such as sorting. Our algorithm, while relying on some general techniques, is naturally specific to the list reversal problem. We leave as the main open problem of this paper the determination of the true communication throughput requirements of basic computational problems, in particular that of sorting.

1.1 The PRAM(m) model

Our model is a variant of the parallel-random access machine (PRAM), where the memory size is limited to m; we call it the PRAM(m) model. The machine model employs p synchronous processors, all have unit time access to a shared memory of size m. The shared memory is an abstraction of the communication media. The size of the shared memory should be interpreted as the number of packets that can be sent and delivered concurrently.

We allow several processors to read simultaneously from the same memory location, or write simultaneously into one. In the case of simultaneous writes the smallest numbered processor succeeds, i.e. the so-called "Priority concurrent-write conflict resolution". (Our results, however, do not depend on the way that the conflict resolution is performed.) Our model assumes that the input lies in a read only shared memory, and we denote the input size by n. We will be mostly interested in studying cases where $n \gg p \gg m$ (e.g., $p = n^{1/2}$ and $m = n^{1/4}$).

Let us sketch the rational behind some of these choices.

- The shared memory of size m is the medium through which processors can send messages to one another. The small size of the read/write shared memory m relative to the number of processors p will restrict the number of messages that can be concurrently transmitted among processors.
- The small number of processors p relative to the input size n represents the intent to run big problems on a given parallel computer.

- Putting the input in a read-only shared memory enables us to focus attention on the usage of the shared memory as a bottleneck for communication during the computation, rather than as means of access to the input. This makes strong lower bound results more significant than strong upper bounds. This also justifies the choice of the strong "Priority" resolution of write conflicts.

For our lower bound model results we assumed that the shared memory consists of m bits, instead of m words. This actually also widens the poly-logarithmic gap between our upper and lower bounds. More justification and discussion of this model can be found in the next section.

1.2 Extant work

The case where p is not so much smaller than n was studied in [VW85, LY86, Aza92]. The parameter m is called *communication width* in these papers. One result in that line of research has been that summing n numbers using a read/write shared memory of size $m = 1$ can be done no faster than $\Omega(\sqrt{n})$ time and this is tight for $p = \sqrt{n}$ processors. Typical additional results related **only** to cases where $p \geq \sqrt{n}$. The interesting cases for us are where the size of n compared to p is relatively big — reflecting a more realistic situation for the use of a parallel machine.

An alternative model of shared memory, which is closer than the PRAM to what is considered technologically feasible, is the *module parallel computer (MPC)* as defined in [MV84]. There, the shared memory is partitioned into m memory modules and only one memory cell of each module can be accessed in a given time unit. If there are several access requests to different addresses within the same module they are queued up and performed one at a time. That paper shows that an MPC, for which $m \geq p$, is strong enough to efficiently simulate, using randomization, a PRAM algorithm which is designed for a much bigger shared memory. Additional simulation results with respect to the MPC model are referenced in [Lei92]. Again, these simulations are targeted only for the case where $m \geq p$. The case $m \ll p$ which is of interest to us is very different.

More on the relation between the PRAM(m) model and the MPC model can also be found in the next section.

1.3 The concrete problem

We study the communication throughput of a very elementary problem, the **list reversal problem**, which is defined as follows. An array of n entries which contains a linked list is given in the read-only shared memory; each entry has a pointer to its successor in the linked list; the problem is to find for each entry its predecessor in the list.

This problem can be trivially solved in $O(1)$ time on a PRAM with n processors and n common memory cells. The trivial algorithm requires a communication throughput of n which is used in a very simple manner: each processor sends one message to some other processor. Is this throughput really necessary?

Our results give tight bounds, up to logarithmic factors, in terms of tradeoffs between running time and the communication throughput. The tradeoff is of a somewhat unexpected form since it allows trading communication throughput with the number of processors.

Theorem 1.1 *Any (deterministic or randomized) $PRAM(m)$ algorithm with p processors that solves the list reversal problem requires time $\Omega(n/\sqrt{mp})$.*

The lower bound is instructive since it proves that limited communication throughput may degrade the effectiveness of a parallel machine. In particular optimal speedup is not possible when $m = o(p)$. Similar lower bounds apply to other problems such as sorting, finding unique elements, convolution, and universal hashing.

The lower bound is related to lower bounds and tradeoffs in several other models of computation, including the problem of inverting a black-box permutation studied in [Hel80, Yao90]. A discussion of some of these connections appears in a later subsection.

Theorem 1.2 *The list reversal problem can be solved on a randomized $PRAM(m)$ with p processors (where $m \leq p \leq n$) in time $O(n \cdot log^{O(1)}(n)/\sqrt{mp})$ with high probability.*

This upper bound is instructive since it demonstrates how to partially alleviate, in certain cases, limited communication throughput. An important ingredient of the algorithm is a "virtual memory" construction which allows emulating a larger memory by a smaller sized memory. Yet, we do not know whether algorithms with similar performance can be designed for problems such as sorting or convolution.

Organization of the paper Further motivation for the model is given in the next section. The lower bound is given in section 3. Some interesting connections to lower bounds in other models of computation are discussed there as well. The upper bound is given in section 4, which also describes the virtual memory.

Postscript Since the first publication of our research in ACM-STOC'94, the $PRAM(m)$ model received considerable attention. The dissertation [Adler-96] and the papers [ABK95] and [AGMR97], are a few representative examples.
In August 1998, the U.S. President's Information Technology Advisory Committee (PITAC) [PITAC-98] wrote, "There is substantive evidence that current scalable parallel architectures are not well suited for a number of important applications, especially those where the computations are highly irregular or those where huge quantities of data must be transferred from memory to support the calculation". In our opinion, the current paper, whose conference version appeared in the 1994 ACM Symposium on Theory of Computing (STOC), anticipated these limitations of current parallel architectures.

2 More motivation

2.1 On Communication Throughput

The following background information motivates the present paper:

- Physical limits on the speed of "serial machines" imply that computing systems must turn to parallelism for improved performance.

- Many parallel computer systems have been built. Typically, these systems allow a relatively low level of communication throughput (CT).
- A wealth of parallel algorithmic methods have been developed. Many of these algorithms appear to need a relatively high level of CT.
- There appears to be a typical gap between some algorithmic methods and many computer systems concerning CT; a relatively low CT of the system is often a bottleneck for improvement in the running time of a parallel algorithm whose CT needs are relatively high.
- Increasing CT is very expensive.

Therefore, it makes sense to quantitatively study CT as a resource similar to parallel time and processors.

Such study can affect the future of parallel computing, since it may lead to one of the following three (mutually exclusive) conclusions:

1. Build high-CT parallel machines.
2. The high-CT demand algorithmic methods can be alleviated in general-purpose computing. This may be shown by either providing alternative parallel algorithmic methods (for the same problems) whose CT demand is relatively low, or by giving general methods for efficiently implementing high-CT algorithms on low-CT systems.
3. Parallel computers with a low-CT level will be handicapped as general purpose computing systems. This may further imply that a parallel system should be designed around the level of CT that its budget allows.

The lower bound in this paper, which is probably the first of its kind, formally shows that low-CT computers are indeed handicapped, at least when it comes to the list reversal problem as well as for the problems of sorting, finding all unique elements, convolution, and universal hashing, for which a similar lower bound holds.

By way of motivation we explain what we mean by the communication throughput of a given parallel computer with p processors. Suppose that in some given time interval each of the p processors sends at most x messages to other processors, and each processor is the address of at most x messages which are sent by other processors. Let T_x be the worst case time for completing the delivery of all these messages to their destinations. One would expect that for a given machine T_x will be of the form $a + bx$, where $a + b$ represents the latency, counting time cycles, until a first message is delivered and b represents the incremental number of time cycles to deliver another message. We refer to $1/b$, which is also the limit of x/T_x where x tends to infinity (note that p is fixed), as the communication throughput (CT) of the computer. Note that the "incremental rate" p/b upper bounds the total number of additional messages which are delivered in a time unit. To articulate the **communication throughput debate**, we note two extreme cases: should $(1/b)$ be close to 1 or can it can be much smaller?

2.2 CT and the PRAM(m) Model

The PRAM(m) model forces a bound on the number of messages sent within a given time period. We suggest that the ratio m/p in the PRAM(m) corresponds to $1/b$ in the motivating problem, where $p \geq m$.

Studying the resource of communication throughput in the context of general parallel algorithmic techniques, where input availability is not a problem, is of theoretical interest. In addition, such study can be justified as follows. Making the input available at the local memory of each processor can be done by a broadcast of the input to all processors, which appears to be less time demanding, using current technology, than a straightforward emulation of a large shared memory, in which every element needs to be accessed at least once. Note also that our lower bounds extend to problems, such as sorting, where the computational effort is relatively greater than for the list reversal problem (and therefore making the input available to all processors becomes even more relatively affordable).

The issue of communication throughput is reflected in various models of computation, which are designed to be closer to currently available machines such as the LogP model of [CKP+93].

We give an additional justification for considering the range $m < p$ for the size of the read/write shared memory m.

One of the most useful methodologies for describing PRAM algorithms is the data-parallel style, as suggested in [SV-82]. A similar style was dubbed "data-parallel" in [HS86] and has been used extensively in [J-92], as one example. A parallel algorithm is described in terms of a sequence of steps, and for each step the set of operations to be performed (concurrently) in that step is given. Using an informal extension of a theorem by Brent [B74], a data-parallel style algorithm which uses a total of W operations in T steps can "typically" be emulated on a PRAM with any number of $p \leq W/T$ processors in $O(W/p)$ time. Incorporating the simulations of [MV84], this emulation needs a shared memory of size $m = p$ for achieving time proportional to W/p, while ignoring poly-logarithmic factors. (Such time bound is denoted $\tilde{O}(W/p)$.) However, we already mentioned that the list reversal algorithm has an $O(1)$ time algorithm using n processors (with $m = n$), and our lower bound results actually imply that such $\tilde{O}(n/p)$ result is impossible where $m \ll p$. This sheds some doubts on the extent to which data-parallelism will remain a useful methodology for such relatively small values of m.

This means that either new alternative methodologies that need a low-CT level will be invented, or that low-CT level parallel machines will be severly handicapped as general purpose parallel machines. Inventing such alternative methodologies is a fundamental open question in parallel computing. Currently, we cannot even offer any encouraging evidence that such new methodologies are likely to be found.

2.3 The PRAM(m) Model and the MPC Model

Next, we show that the MPC model with m memory modules (as defined in the introduction above), where the input is in a read-only shared memory can be efficiently simulated on a PRAM(m) model (where the Priority concurrent-write conflict resolution is used), and vice versa. We are only interested in the case $m \ll p$. Emulating a PRAM(m) on such an MPC is trivial; just store one shared item per module. For emulating such an MPC on a PRAM(m) we let the local memories of processors $p_1, \ldots, p_m$ emulate one memory module each and let each cell of the read/write shared memory regulate the access to local memory of one of these processors. We only have to show how to deal with queue of access requests to the same memory module. All the processors that want access to memory module i try repeatedly to write simultaneously into shared memory location i. All the ones that fail, keep trying, and then do a normal computation step after each try. This discussion implies that both our lower and upper bounds apply to the MPC model as well.

Finally, we note that the PRAM(m) model suppresses a potential technological opportunity to arrange the communication in batches, where larger amounts of data are included in one fetch unit (e.g., a memory page, or memory line) and whose transmission can be done essentially within the same efficiency bounds as the transmission of a single datum. Note that the BPRAM [ACS89] model addresses this technological opportunity.

3 Lower Bound

The concrete problem considered is the **list reversal problem**. An array with n entries that compose a linked list is given in a read-only shared memory. Cell i, for $1 \leq i \leq n$, contains the integer $next(i)$, which is a pointer to the next entry in the list. For simplicity we assume that the list is circular, although our lower bounds also hold for the more natural non-circular list starting, say, at location 1 and ending with $NULL$.

The output should be, for each i, the index $prev(i)$ such that $next(prev(i)) = i$. We assume that we have p processors and that at the end of the algorithm each processor j should hold the answer $prev(i)$ for $(n/p)(j-1)+1 \leq i \leq (n/p)j$ (for simplicity we assume that $p|n$). In this section we prove a lower bound for the time needed to solve the list reversal problem as a function of n, p and m.

The processors share read/write common memory. For the lower bound, we make no assumption regarding the structure of the common memory, but only that its total size is known. We denote by m the total size of the common memory (measured in bits). This deviates from the $PRAM(m)$ model which allows m cells of $O(\log n)$ bits each by only a logarithmic factor. We assume the Priority Concurrent-Read Concurrent-Write (Priority-CRCW, in short) simultaneous access arbitration with respect to this read-write memory; that is several processors can read simultaneously from the same memory location, and in case several processors attempt to write simultaneously into the same memory location the smallest numbered among these processors succeeds. In each time step each processor may read exactly one input location $next(i)$, but our lower bound allows it to read from and write to as many common memory cells as it wants to.

The behavior of each processor depends on the values written in the common memory locations and on the input locations that it accesses. The dependence on the input locations may be modeled by a *decision tree*, where each node of the decision tree represents an access to one of the input locations i, and then there is a branch according to the n different possible values of $next(i)$. The combinatorial core of our lower bound will thus consist of the following simple lemma considering decision trees.

Lemma 3.1 *Any decision tree of depth d which attempts to compute the values $prev(1)...prev(k)$ is correct on at most $n\binom{d+k}{k}/\binom{n}{k}$ fraction of all circular lists.*

Proof: Given a decision tree of depth d which aims to compute $prev(1)...prev(k)$, we can add the queries to the locations $prev(1)...prev(k)$, which are found at each leaf, and obtain a decision tree of depth at most $d+k$ which actually queries all locations in $prev(1)...prev(k)$. We shall see that if the value of d is "relatively small", such a decision tree could exist for only a small fraction of the circular lists which are possible.

Fix this decision tree of depth $d+k$. We will describe a process that yields a uniformly distributed random permutation, but in a way that makes it easy to compute the probability that $prev(1)...prev(k)$ were accessed. This is done by following the decision tree and for each query i choosing a random value for $next(i)$ as the answer (among all values that were still not given as answers.) The values for all the yet un-queried $next(i)$'s are then chosen at random among the remaining possibilities.

It should be clear that our random choice has resulted in a uniformly distributed permutation (whatever the decision tree was). Also it should be clear that the probability of accessing $prev(1)...prev(k)$ is equal to the probability of us giving answers $1...k$ during the $d+k$ questions, an event whose probability can readily be computed as $\binom{d+k}{k}/\binom{n}{k}$. (We are choosing at random $d+k$ balls, without repetitions, out of n numbered balls. What is the probability that we choose all balls numbered $1..k$?).

However, this fraction is the fraction of all possible permutations. We want the fraction out of all circular lists, so, for an upper bound, using simple conditional probability, we have to multiply by a factor of n (since the probability that a permutation is a circular list is exactly $1/n$). $\square$

We are now ready to prove our lower bound.

Theorem 2: Any (deterministic or randomized) PRAM algorithm with p processors and m memory cells reversing an n-element circular list, where $1 \leq m \leq p \leq n$, requires time $t = \Omega(n/\sqrt{pm})$.

Proof: Let t denote the running time of the algorithm. Define the "total common view" of a given computation to be the set of all values written in the common memory. It consists of mt bits of information. Fix now a value for the total common view, and consider all inputs that result in this total common view. On these inputs, the behavior of each processor is now given by a depth t decision tree that computes n/p values.

We now look at the first $n/(3t)$ processors. Their common behavior (on these inputs) may be given by a decision tree of depth $n/3$ which computes $prev(1)...prev(n^2/(3pt))$. We can now apply the previous lemma with $d = n/3$ and $k = n^2/(3pt)$. With these values, and after simplification, the lemma states that such a decision tree may be correct for at most $2^{-\Omega(k)} = 2^{-\Omega(n^2/(3pt))}$ fraction of all circular lists. (The simplification uses also the trivial bound $pt \geq n$ which implies $k \leq n/3$.) It follows that only for that fraction of circular lists may the total common view indeed have been as fixed.

The total number of different total common views is 2^{mt}, so, since each circular list gives some value for the common view, we get the equation $2^{mt}2^{-\Omega(n^2/(3pt))} \geq 1$. This implies the lower bound for deterministic algorithms.

To get the lower bound for randomized algorithms note that by replacing the constant 1 by the constant $2/3$ in the last inequality we get the essentially same lower bound for a deterministic algorithm that is correct on even 2/3 of the possible circular lists. Since a randomized algorithm is a probability distribution over deterministic ones, a standard averaging argument implies the lower bound for randomized algorithms that are correct with probability 2/3 on every input. □

Comment: It is not difficult to see that the same type of argument gives a lower bound of $t = \Omega(\sqrt{n}/\sqrt{m})$ for any value of p. This tradeoff for the case $p \geq n$ is similar in form to the tradeoff derived in [VW85].

3.1 Connections of lower bounds to other models

The lower bound proved in this paper has interesting connections with questions regarding tradeoffs in several other models of computation. We briefly point to them here.

3.1.1 Inverting a Permutation

The problem of reversing a list is quite similar to the problem of inverting a black-box permutation studied in [Hel80, Yao90]. In fact our lemma regarding decision trees suffices to prove the lower bound in [Yao90].

3.1.2 Time-Space tradeoffs for Branching Programs

It is interesting to compare our decision tree lemma to similar lemmas that are used as the standard way (due to [BC82]) of proving time-space tradeoffs for branching programs [BC82, Bea89, Yes84, MNT90, Abr86].

It is not difficult to see that the lemmas proved in some of above-mentioned papers also suffice for our purposes. In these papers the main combinatorial lemmas state that a small depth decision tree computing k output bits of the function considered may only be correct on an exponentially small in k fraction of all inputs. Using one of these lemmas instead of our lemma 3.1 in the proof of theorem 2, we can obtain the same $t = n/\sqrt{pm}$ lower bounds for the problems of sorting [BC82], finding all unique elements [Bea89], convolution [Yes84], or universal hashing [MNT90].

On the other hand, one should notice that the list-reversal problem is easier to compute than all of the above mentioned problems and indeed may be easily computed by log-space linear-time branching programs (as opposed to all of the above mentioned problems which require a time-space tradeoff of $TS \geq \Omega(n^2)$ for branching programs.) What our lemma does suffice to prove though is a $TS \geq \Omega(n^2)$ tradeoff for output-oblivious branching programs, i.e. branching programs which must compute the bits of their output in a pre-defined order.

3.1.3 Size-Depth tradeoffs for Circuits

The most interesting challenge regarding lower bounds suggested by this paper is proving better lower bounds for some explicit function, perhaps lower bounds of the form $\Omega(n/m)$. However we notice that such lower bounds would imply size-depth tradeoffs for circuits, a long standing open problem.

Lemma 3.2 *Let $f : \{0,1\}^n \to \{0,1\}^n$ be a boolean function computed by boolean circuits of $O(n)$ size and $O(\log n)$ depth. Then, for any $\delta > 0$, f may be computed by a PRAM with $p = n^\delta$ processors and $m = 1$ common memory cells in time $O(n/\log\log n)$.*

Proof: We use Valiant's result [Val77] stating that in a linear-size log-depth circuit for f there exist $O(n/\log\log n)$ wires in the circuit such that the value of each output bit and of each one of these wires depends on at most n^ϵ of the input bits and the other wires (this is for any $\epsilon > 0$, and we take an arbitrary $\epsilon < \delta$.) Furthermore this dependence is easy to compute and is given by an n^ϵ size boolean formula.

The PRAM simulation of such a circuit will be as follows: The contents of the single common memory cell will in turn hold the values of all of these special wires. This will be done ordered by the level in the circuit in which the wire's value is computed. The wires output by each level will be partitioned amongst the $p = n^\delta$ processors, and each processor is responsible to write the value of its assigned wires. Between any two levels of the circuit, each processor computes the values of all of its assinged wires. This takes time $n_1/n^\delta \cdot n^\epsilon$, where n_l is the total number of these wires in this level. Since the sum of all n_l's is $O(n/\log\log n)$, and $\epsilon < \delta$, the total time is $O(n/\log\log n)$.

At this point each output bit is a function the wire values already seen by all processors and at most n^ϵ input bits. Each of the processors can thus compute the value of all output bits assigned to it in $n^\epsilon \cdot n/n^\delta \leq O(n/\log\log n)$ time. $\Box$

If the circuit size is $c \cdot n$ for some non-constant c, then the simulation presented in the proof requires a factor of c more time. In particular, bounds of the form $t = \Omega(n)$ in our model with $p = n^\delta$ and $m = 1$ yield an $\Omega(n \log\log n)$ size lower bound for logarithmic depth circuits.

4 Algorithm

We mention again the problem. A shared array with n entries that compose a linked list. Each entry points to the next entry in the list. There are p processors; each can read from the shared array. There are m words of shared memory which the processors can both read and write. (As mentioned before each word consists of $O(\log n)$ bits.) Each processor has a prespecified set of entries for which it need to compute the previous element in the list. Namely, processor p_i needs to compute the predecessor of the entries $i\frac{n}{p}, \ldots, (i+1)\frac{n}{p} - 1$. (Recall that the interesting case is $n \geq p \geq m$.) We start with a high level overview of the algorithm and later give the details of the algorithm.

4.1 Preview

For implementing the algorithm below, we use a **virtual memory concept**, which supports three operations: *INSERT*, *RETRIEVE*, and *UPDATE*. The virtual memory enables us to store much more data than the size of the shared memory seems to allow. An overview of the algorithm is followed by an overview of the virtual memory concept.

High-level description of the algorithm.

For concreteness, we demonstrate the algorithm for $p = n^{1/2}$, and $m = n^{1/4}$. The asymptotic running time of the algorithm will be proportional to $n^{5/8}$ $(= \frac{n}{\sqrt{mp}})$, ignoring poly-logarithmic factors. Such running time is denoted $\tilde{O}(n^{5/8})$.

- *A sample S of $n^{7/8}$ $(= \frac{n\sqrt{m}}{\sqrt{p}})$ elements is chosen at random.* Each processor separately chooses at random $n^{3/8}$ $(= \frac{n\sqrt{m}}{p\sqrt{p}})$ elements for S. The distance in the linked list between an element of S and the next element in S is, with high probability, $\tilde{O}(n^{1/8})$ $(= O(\sqrt{\frac{m}{p}}))$, as we expect the sample to be roughly equally distributed over the linked list. Each processor INSERTs its elements to the virtual memory, hence, the virtual memory holds the set S.

- *For each element $s_1 \in S$, find its predecessor $s_2 \in S$.* For each element $s_2 \in S$, trace its successors in the list till the first successor which is in S, s_1. UPDATE the entry of s_1 in the virtual memory to include its predecessor s_2. (Each processor will need $\tilde{O}(n^{1/8})$ $(= \tilde{O}(\sqrt{\frac{m}{p}}))$ *RETRIEVE* queries for each of its $n^{3/8}$ sample elements.)

- *For each element in the list, find its predecessor.* For each element e in the list: trace its successors in the list till the first successor $s_e \in S$. The entry of s_e includes its predecessor in S, p_e. Advance from p_e in the list till we reach e, the element before last is the predecessor of e. (Each processor will need $\tilde{O}(n^{1/8})$ $(= \tilde{O}(\sqrt{\frac{m}{p}}))$ *RETRIEVE* queries for each of its $n^{1/2}$ $(=\frac{n}{p})$ elements.)

The virtual shared memory.

We are interested in storing a set S of elements in the virtual memory such that later they could be retrieved efficiently. What we are implementing is a dictionary that is separated to pages.

A random hash function h maps the elements of S into an array of size $\tilde{O}(|S|)$, so that at most $\tilde{O}(1)$ elements of S are mapped into the same location of the array. Denote this new array by V.

The idea is to put it in equal chunks of V into the local memories of processors $p_1, p_2, \ldots, p_m$. Locations j such that $j \bmod m = i$ will be in the local memory of processor p_i, $1 \leq i \leq m$. In order to enable the processors to have access to all entries of V we will *"flush" the entries of V through the physical shared memory of size m in $|V|/m$ rounds.* In round 1, locations $1, 2, \ldots m$ (called also the first *block*) in V are put in the shared memory, and in general, in round i, $1 \leq i \leq |V|/m$, the locations are $(i-1)m + 1, (i -$

$1)m + 2, \ldots, im - 1$ (the ith block) are put into the shared memory. To achieve our design goals, each processor will have to access at most $\tilde{O}(1)$ locations in each block, which we achieve by: (i) in each of the three steps of the algorithm we flush the blocks of V through the shared memory of size m once in $|V|/m$ rounds; (ii) prior to such flush, each processor computes into its local memory all the locations in V that it plans to access and only when the block with the appropriate locations resides in the shared memory, it accesses it. We actually require a careful selection of the hash function so that each processor will need to access only $\tilde{O}(1)$ locations in each block, since we have to wait till all the processors completed their readings before we continue to the next block.

4.2 The Virtual Memory

We have a physical memory of m words. We want to use it to implement a virtual memory of V words. This is done by having V/m blocks of m words each. We support three operations: *INSERT*, *RETRIEVE*, and *UPDATE*. In order to demonstrate the generality of our techniques, we give a more refined analysis of the virtual memory performance than actually needed for the list reversal algorithm.

For the performance of the virtual memory we are interested in batches of operations of all the p processors. We show the following properties:

Claim 4.1 *If each processor performs ℓ* INSERT *or* UPDATE *operations to the virtual memory, with high probability, the total time is $\tilde{O}(\ell p/m + V/m)$.*

Claim 4.2 *If each processor performs ℓ* RETRIEVE *operations to the virtual memory, with high probability, the total time is $\tilde{O}(\ell + V/m)$.*

At first sight it seems peculiar that we have different performances for read versus write operations. The real reason behind the difference is that while we can only write m values at a time, to the virtual memory, we can read p values in each time step.

(*Comment:* Note that in the list reversal algorithm the total number of *INSERT* (or *UPDATE*) operations (measured by ℓp) does not exceed V, and therefore $\ell p/m$ and V/m are about equal; hence, the analysis of the list reversal algorithm does not articulate well the separation of "time complexity charges" into $\ell p/m$ and V/m, as per Claim 1. In addition, the total number of *RETRIEVE* operations in the list reversal algorithm is $\frac{np}{\sqrt{pm}} > n$, and therefore Claim 2 implies a time of $\tilde{O}(n/\sqrt{mp} \ + \ V/m)$ per processor; as we shall see $n/\sqrt{pm}$ is of the same order of V/m and the time complexity at that stage is $\tilde{O}(1)$ amortized per operation.)

Let $u = V/m$ be the number of blocks. During the insert we perform a hashing both for the block number and the location within the block.

We specify first how we compute the location of an entry. We take its index i and compute two values: The virtual block number, $h_1(i) \in \{1, \ldots, u\}$, and the location in the block, $h_2(i) \in \{1, \ldots, m\}$. We need to show that the following two properties hold.

Given $\ell \geq u$ indices, with probability $2^{-\gamma}$, the number of indices that h_1 maps to a given value is bounded by $\tilde{O}(\ell\gamma/u)$. (For $\ell < u$ we use the bound of $\ell = u$.)

Given m indices, with probability $2^{-\gamma}$, the number of indices that h_2 maps to the same value is bounded by γ.

Clearly if we can implement the second condition we can implement the first condition, since we can map the ℓ indices to ℓ locations. With probability $2^{-\gamma}$ no location has more than γ elements. Therefore, any set of ℓ/u locations has at most $\ell\gamma/u$ elements.

Claim 4.3 *Let h be a γ-wise independent hash function mapping m elements from $\{1, \ldots N\}$ to $\{1, \ldots, m\}$. Then the probability that more than γ elements are mapped to the same value is bounded by $\binom{m}{\gamma} m^{-\gamma} \leq 2^{-\gamma}$.*

Let us elaborate how the access to the virtual memory is implemented. Assume that we "know" that at most γ elements map to a given location. (We chose γ to be $O(\log n)$, and the claim will hold with high probability.) Only processors $p_1, \ldots, p_m$ are used to maintain the virtual memory. Each entry of the virtual memory is composed from two cells. The first is the key (the name of the entry) and the second is its predecessor (with respect to the linked list). In order to do an operation on the shared memory, the processor needs that the appropriate block be in the shared memory. In our implementation we are interested only in large batches of operations (which are all of the same type). We perform all the operations in one global stage. The blocks of the virtual memory are placed in the shared memory in a round robin fashion. Each block is kept in the shared memory enough time so that all the operations that are to be performed on it, in this stage, would be completed.

For the INSERT and UPDATE operations, the processor computes the hash value of the block. It waits for the appropriate block to be placed in the shared memory. Then it computes the hash value of the location in the block, and writes in the location specified by the hash. With high probability, each processor has only $\tilde{O}(\ell\gamma/u)$ entries to write in each block, the total number of entries written to a block is $\tilde{O}(\ell\gamma p/u)$, and in each time step, with high probability, at least $\tilde{O}(m)$ entries are written. This implies that the time per block is $\tilde{O}(\ell\gamma p \log\gamma/mu)$, where the $\log\gamma$ factor is due to the fact that there might be γ entries per location, and we need to search for the correct one. The total time is $\tilde{O}(\frac{\ell p}{m}\gamma\log\gamma + u)$. Since $\gamma = O(\log n)$, the time is $\tilde{O}(\ell p/m + u)$.

The RETRIEVE operation: the processor computes the hash value of the block. It waits for the appropriate block to come to the shared memory. Then it computes the hash value of the location in the block. Each RETRIEVE operation requires $O(\gamma)$ time. The time for a block is $\tilde{O}(\ell\gamma^2/u)$ and the total time is $\tilde{O}(\ell\gamma^2 + u)$. Since $\gamma = O(\log n)$, the time is $\tilde{O}(\ell + u)$.

4.3 The algorithm itself

The algorithm uses two notations: one is for the distance of two elements as the number of pointers that have to be traversed to reach one of them from the other. Formally, $dist(x,y) = 1$ if $y = next(x)$. In general, if at location x we have the value y, then $dist(x,z) = 1 + dist(y,z)$. (In case that z is not reachable from x then $dist(x,z) = \infty$.) The second notation gives the set of all elements which are at a given distance from a specific element: $Forward(k,x) = \{s | dist(x,s) \leq k\}$. In the algorithm we use a parameter α, which would be later fixed to $c\log n$, for some constant c.

The algorithm for processor p_i:

1. Processor p_i selects k entries $s_{i,1} \ldots s_{i,k}$ at random. The selection among processors is pairwise independent. With high probability, each selected element will have another selected element $s_{i',j'}$ whose distance on the list is $\tilde{O}(\frac{n}{kp})$. (The time is $O(k)$).

2. Processor p_i performs INSERT of its k selected values into the virtual memory. (Virtual memory size is $\tilde{O}(kp)$, time $\tilde{O}(kp/m)$.)

3. In this step of the computation we establish the relationship between the $s_{i,j}$'s that were chosen. Mainly, the ordering between them, as induced by the list.

   Processor p_i, for each of the k values $s_{i,1}, \ldots s_{i,k}$, computes the set $Forward(\alpha\frac{n}{kp}, s_{i,j})$. (Time $O(\frac{n}{p})$).

   Let $S_i = \cup_j Forward(\alpha\frac{n}{kp}, s_{i,j})$, which are all the values that the processor p_i is considering. (Note that $|S_i| = \alpha\frac{n}{p}$.) For each value in S_i we perform a RETRIEVE operation on the virtual memory. (Time $\tilde{O}(\frac{n}{p})$.)

   Let $z \in Forward(\alpha\frac{n}{kp}, s_{i,j})$ be a value that is RETRIEVEd from the virtual memory and equals some $s_{i',j'}$ (in case there is more than one, processor p_i chooses the one with the smallest distance from $s_{i,j}$.). An UPDATE query is performed with respect to entry $s_{i',j'}$ in which entry $s_{i,j}$ is recorded as its predecessor. This establishes the predecessor relationship between the $s_{i,j}$'s.

4. For each of the $\frac{n}{p}$ values, $x_{i,1} \ldots x_{i,\frac{n}{p}}$, for which processor p_i needs to compute the predecessor, it advances $\alpha\frac{n}{kp}$ pointers. Namely, it computes $Forward(\alpha\frac{n}{kp}, x_{i,j})$. (Total $\tilde{O}(\frac{n^2}{kp^2})$ values and time.)

   Let $R_i = \cup_j Forward(\alpha\frac{n}{kp}, x_{i,j})$, which are all the values that the processor p_i is considering in this step.

   Processor p_i RETRIEVEs those values in the virtual memory. For each of the $x_{i,j}$ values one of the elements of $Forward(\alpha\frac{n}{kp}, x_{i,j})$ is located in the virtual memory, let it be $s_{i',j'}$. Denote by $b_{i,j}$ the pointer that is stored in $s_{i',j'}$ as the predecessor pointer.

   For each $x_{i,j}$ a search is started from $b_{i,j}$ to $x_{i,j}$ and the value that points to $x_{i,j}$ is the output of $x_{i,j}$.

4.4 Correctness

The output of the algorithm is correct, with high probability.

After step 1, with high probability, for each $s_{i,j}$ there is $s_{i',j'}$ such that the distance between them is $\tilde{O}(\frac{n}{kp})$.

In step 2, with high probability, the hash functions map all the elements into the appropriate number of blocks.

In step 3, with high probability, each $s_{i,j}$ is updated with its predecessor $s_{i',j'}$ in the list.

In step 4, each element x, with high probability, is at distance $\tilde{O}(\frac{n}{kp})$ from some $s_{i,j}$. Therefore, when we advance from x the set $Forward(\alpha\frac{n}{kp}, x)$ includes $s_{i,j}$. When we retrieve $s_{i,j}$ we find the pointer to its predecessor $s_{i',j'}$. Then we scan from $s_{i',j'}$ to x, and find the predecessor of x. The total number of pointers that we traverse is $\tilde{O}(\frac{n}{kp})$.

4.5 Time complexity

We compute the expected time of each step. (Recall that $\alpha = O(\log n)$ and therefore does not appear in the $\tilde{O}$ notation.)

1. Only local computation. Time $O(k)$.

2. Each processor performs k INSERTs to the virtual memory. (Note that $V = kp$.) Time $\tilde{O}(kp/m)$.

3. Local computation (the *Forward* sets) is $\tilde{O}(n/p)$.

   Each processor has n/p RETRIEVE operations on the virtual memory. Total time is $\tilde{O}(n/p)$.

   Each processor has k UPDATE operations on the virtual memory. Total time is $\tilde{O}(kp/m)$.

4. Local computation (the *Forward* sets) is $\tilde{O}(n^2/kp^2)$.

   Each processor has $\tilde{O}(n^2/kp^2)$ RETRIEVE operations on the virtual memory. Total time is $\tilde{O}(n^2/kp^2)$.

Setting $k = \frac{n}{p}\sqrt{\frac{m}{p}}$, we have the time bounded by $\tilde{O}(\frac{n}{\sqrt{mp}})$. (Note that since $m < p$, we have that $n/p < n/\sqrt{mp}$.)

Acknowledgement: Helpful discussions with Yossi Azar, and comments by Richard Cole, Rajeev Raman, Suleyman Sahinalp and Jeff Westbrook are gratefully acknowledged.

References

[AKP91] F. Abolhassan, J. Keller and W. Paul. On the cost-effectiveness and realization of the theoretical PRAM model, technical report 09/91, FB Informatik, universitat des saarlandes. 1991.

[Abr86] K. Abrahamson. Time-space tradeoffs for branching programs constructed with those for straight line programs. In *Proc. 27^{th} Annual Symposium on Foundations of Computer Science, Toronto, Ontario, Canada*, pages 402–409, October 1986.

[Adler-96] M. Adler. Limited Bandwidth Parallel Computation. Ph.D. Dissertation in Computer Science, University of California at Berkeley, 1996.

[ABK95] M. Adler, J. Byers and R. Karp. Parallel sorting with limited bandwidth. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 129–136, 1995.

[AGMR97] M. Adler, P.B. Gibbons, Y. Matias and V. Ramachandran. Modeling Parallel Bandwidth: Local vs. Global Restrictions. In *Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–105, 1997.

[ACS89] A. Aggarwal and A. Chandra and M. Snir. On Communication Latency in PRAM Computations. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[Aza92] Y. Azar. Lower bounds for threshold and symmetric functions in parallel computation. *SIAM J. Comput.*, 21(2):329–338, 1992.

[B74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2): 201-208, 1974.

[BC82] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 1(2):287–297, 1982.

[Bea89] P. Beam. A general sequential time space tradeoff for finding unique elements. In *Proceedings of the 21^{st} Annual ACM Symposium on Theory of Computing, Seattle, Washington*, pages 197–203, May 1989.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM PPOPP*, pages 1–12, May 1993.

[Hel80] M. E. Hellman. A cryptanalytic time-memory tradeoff. *IEEE Trans. Infor. Theor.*, 26:401–406, 1980.

[HS86] W.D. Hillis, and G.L. Steele. Data parallel algorithms. *Comm. ACM*, 29(12):1170-1183, 1986.

[J-92] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[PITAC-98] R. Joy and K. Kennedy. *President's Information Technology Advisory Committee (PITAC) - Interim Report to the President.* National Coordination Office for Computing, Information and Communication, 4201 Wilson Blvd, Suite 690, Arlington, VA 22230, August 10, 1998.

[Lei92] T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24^{th} Annual ACM Symposium on Theory of Computing*, pages 77–96, 1992.

[LY86] M. Li and Y. Yesha. New lower bounds for parallel computation. In *Proceedings of the 18^{th} Annual ACM Symposium on Theory of Computing, Berkeley, California*, pages 177–187, 1986.

[MNT90] Y. Mansour, N. Nisan, and P. Tiwari. The computational complexity of universal hashing. In *Proceedings of the 22^{nd} Annual ACM Symposium on Theory of Computing*, 1990.

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

[SV-82] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128-146, 1982.

[Val77] L. Valiant. Graph theoretic arguments in low-level complexity, technical report CS 13-77, University of Edinburgh, UK. 1977.

[VW85] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM J. Comput.*, 14,2:303–314, 1985.

[Yao90] A. Yao. Coherent functions and program checkers. In *Proceedings of the* 22^{nd} *Annual ACM Symposium on Theory of Computing*, 1990.

[Yes84] Y. Yesha. A time-space tradeoff for matrix multiplication and the discrete Fourier transform on a general sequential random access computer. *J. Comp. and Syst. Sci.*, 29:183–197, 1984.