Parallel Sorting With Limited Bandwidth

Micah Adler[*] John W Byers[†] Richard M Karp[‡]

TR-TR-95-031

July 1995

Abstract

We study the problem of sorting on a parallel computer with limited communication bandwidth. By using the recently proposed PRAM(m) model, where p processors communicate through a small, globally shared memory consisting of m bits, we focus on the trade-off between the amount of local computation and the amount of inter-processor communication required for parallel sorting algorithms. We prove a lower bound of $\Omega(\frac{n \log m}{m})$ on the time to sort n numbers in an exclusive-read variant of the PRAM(m) model. We show that Leighton's Columnsort can be used to give an asymptotically matching upper bound in the case where m grows as a fractional power of n. The bounds are of a surprising form, in that they have little dependence on the parameter p. This implies that attempting to distribute the workload across more processors while holding the problem size and the size of the shared memory fixed will not improve the optimal running time of sorting in this model. We also show that both the upper and the lower bound can be adapted to bridging models that address the issue of limited communication bandwidth: the LogP model and the BSP model. The lower bounds provide convincing evidence that efficient parallel algorithms for sorting rely strongly on high communication bandwidth.

[*]Supported by a Schlumberger Foundation Graduate Fellowship.
[†]Supported by a GAANN Graduate Fellowship.
[‡]Supported by NSF grant number CCR-9005448

1 Introduction

A large body of theoretical research has concentrated on algorithms designed in the Parallel Random Access Machine (PRAM) model of computation. The PRAM allows processors to communicate with each other in unit time through a large globally shared memory, which leads to algorithms that have a high degree of parallelism but perform a great deal of inter-processor communication, an inexpensive operation in the PRAM model. This leaves unresolved the question of how to implement algorithms designed for a PRAM on machines which have limited inter-processor communication bandwidth.

Addressing this limitation has motivated the development of other models of parallel computation: the BSP model [V90a], the LogP model [CKP+93], and recently the PRAM(m) model [MNV94]. Provably efficient algorithms in the PRAM model are not necessarily the most efficient algorithms for these new models, so a host of problems need to be re-evaluated in this new framework. In this paper, we examine the problem of sorting in the context of parallel machines with limited communication bandwidth. We formalize the sorting problem as follows:

Definition 1 The Sorting Problem
Input: n *distinct keys* $k_1 \ldots k_n$, *with total order* $k_{(1)} < k_{(2)} < \ldots < k_{(n)}$.
Output at processor i: *A sorted list of keys:*

$$k_{(\frac{in}{p}+1)} \ldots k_{(\frac{(i+1)n}{p})}.$$

We concentrate first on the complexity of sorting in the PRAM(m) model. In this variant of the classical PRAM, p processors communicate only through a small, globally shared memory consisting of m bits, and the entire input is available to each processor in a globally shared read only memory (ROM). This model allows us to focus on the trade-off between the amount of information derived from local computation and the amount of information derived from inter-processor communication.

In the exclusive read variant of the PRAM(m) model, or ER PRAM(m), we prove a lower bound for sorting n distinct keys of

$$\Omega\left(\frac{n \log m}{m}\right).$$

The bound holds when $n > p^2$, which is the case of interest, since typical parallel applications involve problems where the input size is much larger than the number of processors. This lower bound does not rely on any restriction on the local computation of a processor, in contrast to sorting results which prove lower bounds on comparison based algorithms. The proof also extends to both Monte Carlo and Las Vegas randomized algorithms. Finally, the lower bound holds even when we allow processors the power to read concurrently from the ROM and the ability to write concurrently to the same memory location.

In order to prove the lower bound, we introduce the *oracle model of computation*, a model that allows us to quantify a tradeoff between local computation and information received from other processors. In this model, all inter-processor communication is simulated by an oracle of unlimited computational power, and processors no longer communicate with one another. We prove that even in this setting, local computation is of such limited utility that the oracle must provide a large amount of information in order to enable the processors to solve the sorting problem efficiently.

When m grows no faster than some function of n of the form $m = O(n^\beta)$, $\beta < 1$ we show that a version of Columnsort [L85] provides us with an upper bound of

$$O\left(\frac{n}{m} \log n(1-\beta)^{-3.42}\right).$$

For $n \gg m$, the case of greatest interest, the final term becomes a small constant and so in this setting the discrepancy between the upper and lower bounds is $\Theta(\frac{\log n}{\log m})$. The algorithm used in the upper bound runs with only a small constant factor slowdown on a machine model in which the input is distributed among all processors, and for which there is no globally shared ROM. This gives the algorithm more credibility from a practical standpoint.

We also show that our results can be generalized to two bridging models that incorporate limited bandwidth, the LogP model and the BSP model. In both of these models, the processors communicate using point-to-point messages, and a parameter g represents the average number of cycles between transmission of successive messages from a processor. Our technique used for the ER PRAM(m) model provides us with a lower bound on the number of bits which must be transmitted through the network in order to solve the sorting problem. Coupling this bound with model dependent lower bounds on the amount of time required to transmit a fixed number of bits results in lower bounds for sorting in these two models. Definitions of the models and the exact form of the bounds are deferred to the section where those results are discussed.

Our results show that fast parallel algorithms to solve the sorting problem must rely on large amounts of communication. Furthermore, we have the surprising result that both our upper and lower bounds are unaffected by attempting to distribute the work across an unlimited number of processors, while holding fixed the problem size, the size of the shared memory, and the number of processors that actually output the result. Therefore, to increase the speed of parallel sorting on a machine with limited communication bandwidth, increasing bandwidth is more likely to improve the running time than increasing the number of processors.

The remainder of the paper is organized as follows. In the rest of Section 1, we briefly compare our results with previous work in parallel sorting. In Section 2, we provide a complete description of both the PRAM(m) model, and our lower bound tool, the oracle model of communication. Sections 3 and 4 provide proofs of our lower and bound for deterministic and randomized algorithms for sorting in the ER PRAM(m) model of computation. Section 5 provides the matching upper bound, and Section 6 briefly describes extensions of those proofs to the LogP and BSP models.

1.1 Previous Work

From the large body of research in the realm of parallel sorting algorithms, we discuss a few results which also focus on inter-processor communication requirements. Perhaps the result most similar to ours is Leighton's in [L85]. Using Thompson's VLSI model [T80], he proves a lower bound of $\mathrm{AT}^2 = \Omega(n^2 \log^2 n)$ for sorting n keys of size $\Theta(\log n)$, where A is the area of a VLSI chip and T is the running time of the chip. His methods can be used to show bounds of the form $\Omega\left(\frac{n \log n}{m}\right)$ in a PRAM model with a globally shared memory of size m, but in which the input is evenly distributed across the p processors, rather than stored in a globally shared ROM. Indeed, an interesting question would be to determine whether we could apply our lower bound technique to a non-standard VLSI model in which the chip could receive each input in more than one location and at more than one time.

Other related work on parallel sorting includes [BC82], where Borodin and Cook prove that sorting requires TIME $\cdot$ SPACE $= \Omega(\frac{n^2}{\log n})$. Aggarwal, Chandra and Snir show in [ACS90] that any parallel comparison-based algorithm that sorts n words requires $\Omega(\frac{n \log n}{p \log(\frac{n}{p})})$ communication steps. Also, the same authors show in [ACS89] that sorting requires $\Omega(\frac{n \log n}{p} + l \log p)$ in a model where reading or writing a block of size b from memory takes time $l + b$.

When introducing the PRAM(m) model in [MNV94], Mansour, Nisan and Vishkin prove a lower bound of $\Omega(\frac{n}{\sqrt{mp}})$ for several problems, including sorting, in a concurrent read version of the PRAM(m), which implies the same bound in the ER PRAM(m). An easy upper bound can be obtained by using a variant of Cole's parallel merge sort for the PRAM which uses $O(n \log n)$ bits of memory and runs in $O(\log n)$ time. By letting each of the m bits of shared memory simulate $\frac{n \log n}{m}$ bits of the PRAM memory, we can run Cole's algorithm in time $O(\frac{n \log^2 n}{m})$ on the ER PRAM(m). Thus a large gap between upper and lower bounds for sorting remained prior to this work.

2 The PRAM(m) Model

In this section, we define the PRAM(m) model, and then describe a theoretical tool derived from the PRAM(m) model, the oracle model of communication.

In a classical PRAM, processors communicate by writing to and reading from a globally shared memory. The PRAM(m) model restricts communication by providing only m shared memory cells. The other distinguishing feature of the PRAM(m) is that the input is assumed to reside in a read-only shared memory (ROM) available to all of the processors via concurrent read access. During each synchronized round of computation, every processor can perform one of four actions: read the contents of a ROM location, read the contents of a globally shared memory location, write to a globally shared memory location, or perform local computation.

If n represents the input size, then practical considerations make the case where $n \gg p \gg m$ the most interesting. This models a machine running a large problem in which the processors communicate through a network that has insufficient bandwidth to handle simultaneous communication among any sizable subset of the processors. As defined in [MNV 94], the PRAM(m) model allows processors concurrent read, concurrent write access to the globally shared memory. In this paper, we discuss an exclusive read variant of the PRAM(m) model, the ER PRAM(m). Neither the upper nor the lower bounds are affected by whether or not processors are allowed concurrent write access to the shared memory, and thus this component of the model is left unspecified.

The primary goal of this model is to examine the effectiveness of parallel computation given a sharp limitation on inter-processor communication. The existence of the shared ROM allows us to concentrate our lower bound efforts on the amount of communication required for actual computation, rather than the amount required to distribute the input. Since such a ROM may be unrealistic from a practical standpoint, upper bounds achieved in this model that rely on use of the ROM are only applicable to problems in which the entire input is initially known by all the processors. A complete and detailed justification of the PRAM(m) model is provided in [MNV94].

We assume for the purpose of simplifying our exposition that each cell of the shared memory consists of only one bit. The previous work in the PRAM(m) model assumed that each shared memory cell could hold a word; our results can be extended to this case.

2.1 The Oracle Model of Communication

It is often the case in parallel computing that the amount of computation required by a processor is greatly reduced by receiving results of computations performed by other processors. In order to quantify a tradeoff between local computation and information received from other processors, we define the oracle model of computation. This lower bound tool uses the principle that the information a processor receives from other processors can be no better than information it receives from a single processor with unlimited computational resources.

In the oracle model, processors do not transmit any information. Rather, each processor only receives information from an oracle of unlimited computational power, and a read only memory (ROM) that contains the input. At the start of an algorithm, the oracle can access the entire input and it can instantaneously place some number of bits, called *oracle bits*, into p *oracle memories* which can only be written to at the start of an algorithm, and only by the oracle. Processor i has read-only access to the ith oracle memory, but not to any of the other oracle memories. We make no limitations on the size of the oracle memory, but the locations that are not written to by the oracle may be set arbitrarily, and thus contain no information.

The processors access the input using a ROM, which is identical to the ROM of the PRAM(m) model. The ROM can be read concurrently by every processor. Unless otherwise stated, the programs of the processors are deterministic, until Section 4, where we discuss a randomized oracle model. During computation, at each time step every processor is allowed to perform one of three actions: read an oracle bit, read an input word from the ROM, or perform local computation. The

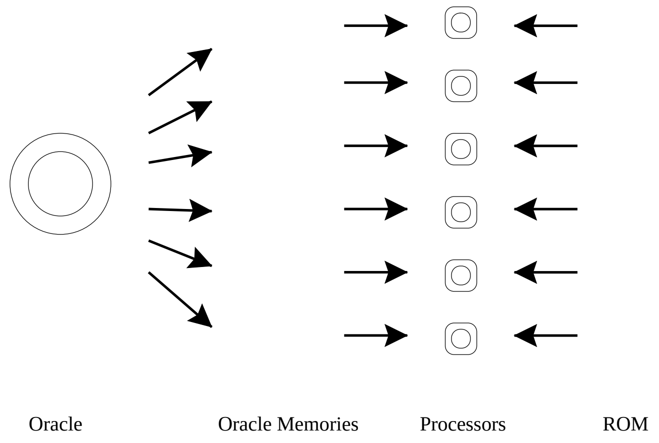

Figure 1: The Flow of Information in the Oracle Model

oracle knows the programs executed by each of the processors. We are interested in the tradeoff between the maximum number of time steps required by any processor versus the total number of bits placed into the oracle memory.

To see that the oracle model can be used to prove lower bounds in the ER PRAM(m) model, we show that the oracle model is at least as powerful as the ER PRAM(m) model. We note that a similar lemma shows that an oracle model with a single, concurrently readable oracle memory is at least as powerful as a PRAM(m) with a concurrently readable globally shared memory.

Lemma 2 *Any algorithm designed for the ER PRAM(m) model of computation can be simulated in the oracle model with no slowdown, using an algorithm where the total number of bits written into the oracle memory is no greater than the total number of reads from shared memory cells over the course of the ER PRAM(m) algorithm.*

Proof: The oracle first uses its knowledge of the input and the programs of all processors to simulate the entire ER PRAM(m) algorithm and to record the value of every bit read by each processor from the globally shared memory. For each processor k, the oracle then writes the values of all bits read by processor k into oracle memory k in sequential order, so that the jth bit read by processor k is in location j of oracle memory k. The number of bits written to the oracle memories is thus equal to the number of reads from the globally shared memory. Also, since the oracle has unlimited computational power, the simulation has thus far been instantaneous. Each processor then executes its program in the ER PRAM(m) algorithm. Writes to the shared memory are ignored, and whenever processor k is required to perform a read, it simply reads the first unread location in oracle memory k. Thus, the number of time steps required for an algorithm in the oracle model is no greater than the number of time steps for the same algorithm in the ER PRAM(m) model. ■

3 The Lower Bound

In this section, we prove the lower bound for deterministic algorithms for sorting in the oracle model by showing that even when all processors know the range of keys that need to be output by each processor, the task of locating those keys within the input is difficult. If we wish to sort n keys, and if all processors know the range of key values that will be output by each processor, but not the location within the ROM of the keys whose values fall into those ranges, then the work remaining can be formalized as the *ownership problem.* We show that any lower bound for this problem implies an identical lower bound for sorting.

Definition 3 The Ownership Problem
Input: *n elements, each designating a single processor which owns that element, such that each processor owns exactly $\frac{n}{p}$ elements.*
Output at each processor: *a list of elements it owns.*

Lemma 4 *Any lower bound for the ownership problem implies an identical lower bound for the sorting problem.*

Proof: It suffices to show that any algorithm that sorts n distinct keys can solve any instantiation of the ownership problem of size n. To do this, we interpret each element to be sorted as a key composed of the concatenation of the unique identifier of the processor that owns the element and the element's location within the ROM. Sorting these keys entails routing each element to the correct processor, and so it is sufficient to inform each processor of the elements which it owns. ■

Most of the remainder of this section is devoted to proving the following theorem.

Theorem 5 *For any oracle algorithm that solves the ownership problem in which each processor performs at most $\frac{n \log m}{m}$ ROM queries, where $n > p^2$, the oracle must on average provide at least $n \log m -$ $o(n \log m)$ bits in the oracle memories.*

We first work towards the result that the average number of oracle bits that the oracle must provide in each oracle memory is at least $\frac{n \log m}{p}$ - $o(\frac{n \log m}{p})$. We begin by developing some tools to prove this bound.

We define an *input configuration* for processor i to be one of the $\binom{n}{n/p}$ distinct ways of distributing the $\frac{n}{p}$ elements owned by processor i in the n ROM locations. In order to guarantee a correct solution to the ownership problem, each processor must be able to reach a distinct state for each of the $\binom{n}{n/p}$ input configurations. We say that an input configuration c for processor i is consistent with S, a setting of the oracle bits for processor i, in algorithm A if c is a possible outcome of A when the oracle bits for i are set to S. We will show that for any algorithm A that solves the ownership problem, and for any setting of the oracle bits S, the number of input configurations consistent with S in A is small, which implies that the number of oracle bits provided to processor i must on average be large.

The lower bound employs the "little birdie" principle: giving a processor additional information never increases the complexity of the problem that the processor must solve. We provide each processor i with some additional information for free, in the form of a matrix H^i, called a *hidden matrix.* Providing H^i to processor i allows us to restrict the amount of information provided by each ROM query, as we shall see momentarily. To describe this matrix, we introduce a possible representation of the ownership problem. We represent the ownership problem as a bit matrix B with n rows and p columns. If processor j owns element i then $B_{ij} = 1$, otherwise $B_{ij} = 0$. Rows of B correspond to elements, and columns of B correspond to processors, so each column of B has exactly $\frac{n}{p}$ ones, and there is exactly one 1 in each row. A ROM query reveals exactly one row of this matrix. In order to solve its portion of the ownership problem, processor i must be able to specify column i of B exactly.

The hidden matrix H^i given to processor i is based on the matrix B, but with the 1's in some of the rows moved to new columns. In order to hide the location of the elements owned by processor i, we obtain H^i by performing a transformation on the matrix B, which we call a *hide.* The hide transformation for processor i takes the matrix B, and replaces each 1 in column i with a 1 in the same row, but in another column. Each column $j \neq i$ receives exactly $\frac{n}{p(p-1)}$ 1s from column i. The hide transformation for processor i is chosen uniformly at random from all possible hide transformations.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Figure 2: Ownership Problem Input: $n = 6$, $p = 3$.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 3: A Hidden Matrix for Processor 1

We say that an input configuration c for processor i is consistent with a hidden matrix H^i if there exists a matrix B with an ith column defined by c such that some hide transformation on B results in H^i. When $n > p^2$, it is straightforward to see that for any hidden matrix H^i, there exist exactly

$$\binom{\frac{n}{p-1}}{\frac{n}{p(p-1)}}^{p-1}$$

input configurations for processor i consistent with H^i. Let $n_i(S, A, H^i)$ be the number of input configurations for processor i that are consistent with S in A and that are consistent with H^i.

Lemma 6 *For any processor i given matrix H^i, oracle bits S and algorithm A which solves the ownership problem using at most r ROM queries, $n_i(S, A, H^i) \leq \sum_{k=1}^{\frac{n}{p}} \binom{r}{k}$.*

Proof: After processor i observes H^i and S, we can model the remainder of processor i's actions by a decision tree, in which each node of the decision tree corresponds to a ROM query, and each leaf of the tree corresponds to a processor state achievable after performing ROM queries. In order for the processor to successfully produce x distinct results (determine x separate input configurations), this decision tree must contain at least x leaves.

Row j of the matrix H^i gives processor i a great deal of information about the owner of element j. Either row j of H^i is identical to row j of B, in which case processor i was already provided with the correct owner of element j, or row j is incorrect, in which case processor i is the owner of element j. Therefore, a ROM query to location j only gives processor i the answer to the following question: "Is element j an element that I, processor i, really own, or is it owned by the processor indicated in H^i?" Thus, the decision tree has branching factor two. Since processor i only owns $\frac{n}{p}$ elements, at most $\frac{n}{p}$ of these ROM queries can result in the answer "It is owned by processor i." Thus any path from root to leaf in the ROM query decision tree can have at most $\frac{n}{p}$ "processor i" branches. This limits the possible number of leaves to at most $\binom{r}{0} + \binom{r}{1} + \ldots + \binom{r}{n/p}$. ■

Proof: (of Theorem 5) At the beginning of any algorithm A, we can provide each processor i with a matrix H^i chosen uniformly at random from those consistent with processor i's input configuration. We work first towards a lower bound on u_i, the average number of oracle bits in oracle memory i, when processor i is given such a matrix H^i. After receiving H^i, processor i must still be capable of producing all

$$\binom{\frac{n}{p-1}}{\frac{n}{p(p-1)}}^{p-1}$$

distinct results consistent with H^i. Furthermore, by Lemma 6, once processor i has seen both H^i and the oracle bits S, and given that processor i may perform at most $\frac{n \log m}{m}$ ROM queries, we have the following bound on the number of distinct results processor i can still produce:

$$n_i(S, A, H^i) \leq \sum_{k=1}^{\frac{n}{p}} \binom{\frac{n \log m}{m}}{k}$$

Therefore, a lower bound on u_i follows directly:

$$u_i \geq log \left(\frac{\binom{\frac{n}{p-1}}{\frac{n}{p(p-1)}}^{p-1}}{\sum_{k=1}^{\frac{n}{p}} \binom{\frac{n \log m}{m}}{k}} \right).$$

Using the inequality

$$\left(\frac{a}{b}\right)^b \leq \binom{a}{b} \leq \left(\frac{ae}{b}\right)^b,$$

and the fact that the sum $\sum_{k=1}^{\frac{n}{p}} \binom{\frac{n \log m}{m}}{k}$ is dominated by the final term, this gives us

$$u_i \geq \frac{n}{p} \log m - o\left(\frac{n}{p} \log m\right).$$

Let $u_i(H^i, A)$ be the average number of oracle bits provided in oracle memory i in algorithm A, where processor i does not receive the matrix H^i but the average is still taken over input configurations consistent with H^i. By the little birdie principle, the number of oracle bits required does not increase when we do not give processor i the matrix H^i, and thus $u_i(H^i, A) \geq u_i$. Also, since each hidden matrix is consistent with the same number of input configurations, $u_i(H^i, A)$ is also the average number of oracle bits provided in oracle memory i in algorithm A, averaging over all inputs. By the linearity of expectations, the average number of oracle bits provided in all of the oracle memories is at least $n \log m -$ $o(n \log m)$. ■

Corollary 7 *Any deterministic ER PRAM(m) algorithm that solves the sorting problem where $n > p^2$, requires, on average, time at least $\frac{n \log m}{m} -$ $o(\frac{n \log m}{m})$.*

Proof: From Lemma 4, we know that a lower bound on the ownership problem implies a lower bound on the sorting problem. We can assume that for any algorithm solving the ownership problem in time $\leq \frac{n \log m}{m}$, each processor performs no more than $\frac{n \log m}{m}$ ROM queries. Thus, from Theorem 5, any oracle algorithm for the ownership problem writes at least $n \log m -$ o$(n \log m)$ bits to the oracle memories on average. From Lemma 2, any ER PRAM(m) algorithm must perform at least that many reads from the global shared memory, and since only m bits can be read at any time step, this requires time at least $\frac{n \log m}{m} -$ o $(\frac{n \log m}{m})$. ■

4 Randomized Algorithms

In addition, the lower bound proof can be extended to randomized algorithms. We here discuss how the lower bound can be extended to Monte Carlo algorithms, or randomized strategies with a guarantee on the running time of the algorithm that provide a correct solution with probability greater than and bounded away from $\frac{1}{2}$. The proof is similar for Las Vegas algorithms, or randomized strategies with a guarantee on the expected running time which always provide a correct solution.

To extend the lower bound to randomized algorithms we introduce a randomized version of the oracle model. The proof that the oracle model is capable of simulating the ER PRAM(m) model relies on the fact that the oracle can simulate the behavior of each processor. For the randomized oracle model, this is done by allowing the oracle access to each processor's random bits. Since the oracle model is a lower bound tool, giving it this extra power can only strengthen the corresponding lower bound. For randomized algorithms, Lemma 6 can be replaced by the following, which is sufficient to prove a randomized version of Theorem 5.

Lemma 8 *For any processor i given matrix H^i, oracle bits S and randomized algorithm A which solves the ownership problem with probability $> \frac{1}{2}$ using at most r ROM queries, $n_i(S, A, H^i) \leq 2\sum_{k=1}^{\frac{n}{p}} \binom{r}{k}$.*

Proof: The ROM queries again result in a decision tree with $\sum_{k=1}^{\frac{n}{p}} \binom{r}{k}$ leaves. The lemma follows from the fact that on each of the $n(S, A, H^i)$ input configurations consistent with S and H^i, the algorithm must answer correctly with probability $> \frac{1}{2}$. ■

5 The Upper Bound

We show that a version of Leighton's Columnsort [L85], can match the lower bound. In the case where the keys are of size $O(\log m)$, this gives us an asymptotically optimal algorithm where all processors only read from distinct portions of the ROM of size $\frac{n}{p}$. Thus, this algorithm can be run in a model where there is no ROM for the input, and the input is distributed across the processors' local memories.

Theorem 9 *There exists an ER PRAM(m) algorithm for sorting n keys which runs in time*

$$O(\frac{n \log n}{m}(1-\beta)^{-3.42})),$$

provided m grows no faster than some fixed function of n of the form n^β, $\beta < 1$.

Proof: In Columnsort, the n keys are thought of as elements in a matrix M. The elements are sorted using seven steps, where each step is one of three types: steps that sort the columns of the matrix, steps that perform an odd-even transposition sort along the rows of the matrix and steps that route a fixed permutation of the matrix elements, where each column routes an equal number of elements to every other column. There is an additional requirement on the aspect ratio of M: if M is an $s \times r$ matrix, then s must be larger than r^2.

We first describe how to sort the keys in the case where $n \geq m^3$. The matrix M will be of size $\frac{n}{m} \times m$, and thus the aspect ratio requirement of M is satisfied. The portion of the algorithm that actually performs the sorting only requires m processors, which lends additional weight to the observation that increasing bandwidth, as opposed to adding processors is required for faster parallel sorting. Each of these m processors, which we refer to as active processors, are responsible for $\frac{n}{m}$ distinct keys at each step, where each processor's keys constitute a column of the matrix M to be sorted using Columnsort. We show that each of the three types of steps of Columnsort can

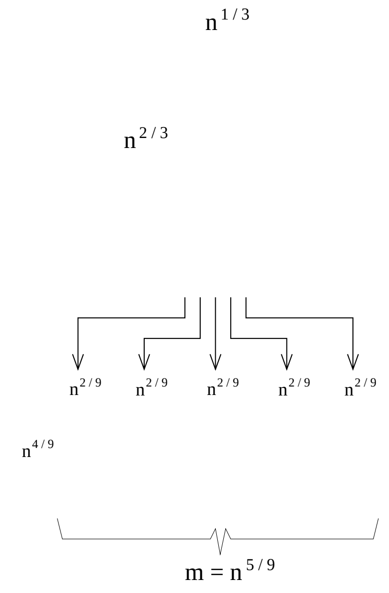

Figure 4: The recursive algorithm when $m = n^{5/9}$

be performed in time $O(\frac{n \log n}{m})$, and thus the entire algorithm runs in time $O(\frac{n \log n}{m})$. After the completion of Columnsort, each key is routed to the processor required to output that key, which also requires time $O(\frac{n \log n}{m})$.

Sorting the columns can be performed by each of the m active processors locally in time $O(\frac{n \log n}{m})$ by any of a variety of known serial algorithms (see for example [AHU74]). Routing the fixed permutation on the matrix elements requires each processor to send an identical number of keys to every other processor, and thus can be done with a single pass through all the entries. A single element will be routed by the source and destination processors using a single shared memory location for $\log n$ consecutive steps, during which the key's address in the ROM is transmitted. Thus, the entire permutation is routed in time $O(\frac{n \log n}{m})$. One step of odd-even transposition sort can be performed by each active processor i by informing processors $i+1$ and $i-1$ of all elements known to processor i. This can be done in time $\frac{2n \log n}{m}$, by cycling each element through the shared memory twice.

When $n < m^3$, the described method of dividing the keys into columns does not meet the aspect ratio requirement of Columnsort. We can, however, still use a recursive form of Columnsort. We first describe the case where $n^{1/3} < m \leq n^{5/9}$. In this case, we will use $n^{5/9}$ processors. The initial matrix will be of the shape $n^{2/3} \times n^{1/3}$. Each column is further divided into contiguous pieces of size $n^{4/9}$. Each processor will be responsible for one of these contiguous pieces, and reads that piece from the ROM.

We sort each column recursively in time $O(\frac{n \log n}{m})$ by assigning each column i of M to a smaller matrix M_i of size $n^{4/9} \times n^{2/9}$, where there is a one to one correspondence between the columns of the M_is, and the contiguous pieces read from the ROM by distinct processors. Then, we assign $\frac{m}{n^{1/3}}$ memory locations to matrix M_i, and let each memory location simulate $\frac{n^{5/9}}{m}$ memory locations with a corresponding slowdown of $\frac{n^{5/9}}{m}$. The number of simulated memory locations used for each matrix M_i is $\frac{m}{n^{1/3}} \times \frac{n^{5/9}}{m} = n^{2/9}$. This allows us to sort all the $n^{4/9} \times n^{2/9}$ M_i matrices recursively in time $\mathrm{O}(n^{4/9} \log n) \cdot \frac{n^{5/9}}{m} = \mathrm{O}(\frac{n \log n}{m})$.

We also need to show that in this case, we can route the permutations of the matrix M, as well as perform the steps of odd-even transposition sort, in time $\mathrm{O}(\frac{n \log n}{m})$. But, the number of processors containing keys that need to be routed (or compared in the case of transposition sort) is at least as large as the number of memory cells available, and each processor has an equal number of keys. Thus, each memory cell can be put to productive use at each of $O(\frac{n\log n}{m})$ time steps. Thus, the time to perform these operations has not changed from the case where $m \le n^{1/3}$.

When $m > n^{5/9}$, we need to use further levels of recursion to satisfy the Columnsort aspect ratio requirement. The process is repeated until the columns are small enough to be sorted in time $\frac{n\log n}{m}$. Columnsort uses four sorting steps, and thus, this algorithm requires time $\mathrm{O}(\frac{n\log n}{m}4^i)$, where i is the number of levels of recursion required. When m is a fixed function of n where $m = \mathrm{O}(n^\beta)$ for $\beta < 1$, this i is the smallest i that satisfies $n^{((2/3)^i)} \le \frac{n}{m} \le n^{1-\beta}$, and the following bound on 4^i follows directly:

$$4^i \ge (1-\beta)^{\frac{2}{\log\frac{2}{3}}} \approx (1-\beta)^{-3.42}$$

Substituting in to the formula above gives an upper bound on the running time of $O(\frac{n\log n}{m}(1-\beta)^{-3.42})$. As m approaches n, this function becomes much larger than $\frac{n\log n}{m}$. But this is not surprising, given the known lower bound for sorting of $\Omega(\log n)$ on any n processor PRAM, and the fact that our bound of the form $O(\frac{n\log n}{m})$ in the bit model becomes a bound of the form $O(\frac{n}{m})$ in the word model.

Ralph Werchner has pointed out similar work by Cypher and Sanz. In [CS92] they allude to a recursive version of Columnsort, and introduce Cubesort, which can be used to obtain a running time of $O(\frac{n\log n}{m}(1-\beta)^{-2})25^{\log^* n-\log^*(n/m)}$ for sorting on the PRAM(m). This algorithm, however, is somewhat more involved then the one presented here. ∎

6 Other limited bandwidth models

We can use the techniques discussed for the PRAM(m) to derive bounds in other parallel models which address the issue of limited communication bandwidth. We give a brief discussion of the complexity of sorting in the LogP model [CKP+93] and the BSP model [V90a].

6.1 The LogP model

In the LogP model, limited communication bandwidth in a parallel machine is enforced by requiring that each processor must wait for a gap of at least g cycles between the transmission of consecutive point-to-point messages. The three other LogP parameters are P, the number of processors, L, the latency of a message in the network, and o, the overhead (in cycles) to place a fixed-size message onto the network. Note that this model uses point-to-point messages for communication, as opposed to the global shared memory used in the PRAM(m) model. We also make the assumption that the point-to-point messages, also called packets, have a maximum size w, measured in bits.

In this model, only P packets can be issued into the network each g time steps, and thus the throughput of the network is $m_L = \frac{wP}{g}$ bits per machine cycle. We denote this expression for throughput by m_L to make plain its correspondence with m in the PRAM(m) model. The same techniques give asymptotically tight bounds for sorting in the case where m_L grows as $\Theta(n^\beta)$, $\beta < 1$. In this paper we provide the form of the bounds, and a sketch of the proof in the case where the entire input is known to each processor. The case where the input is distributed across the processors requires long keys to be sent in their entirety, rather than sending just the original index of the key, but the resulting bounds are similar.

We begin by proving a lower bound for sorting in the LogP model. As in the ER PRAM(m) model, when m_L grows as a fractional power of n, the time required to sort n keys is asymptotically no less than the time required to route all n keys through the network.

Theorem 10 *In the LogP model, sorting n distinct keys requires time $\Omega(\frac{n}{m_L}\log m_L + L + o)$, provided that $n > P^2$ and that $L + o < n\log n$.*

Proof: (sketch) A lemma similar in flavor to Lemma 2 holds which proves that any LogP algorithm could be simulated with no slowdown in an oracle model in which the cells of the oracle memories store words of size w. To simulate a LogP algorithm using such a model, the oracle could simply write the value of each message received by any processor into the oracle memory for that processor.

Now, by applying Theorem 5 in conjunction with the lemma sketched above, any LogP algorithm where no processor examines more than $\frac{n \log m_L}{m_L}$ input keys will require a total of at least $\Omega(n \log m_L)$ bits, or $\Omega\left(\frac{n}{w} \log m_L\right)$ packets, to be transmitted through the network. The lower bound then follows from the fact that only P packets can be sent every g time steps, so transmission of n packets takes time $\Omega(\frac{ng}{p})$ and transmission of a single packet takes time at least $\Omega(L + o)$.

Although LogP is most often thought of as an asynchronous model of computation, the stated bounds also hold in a synchronous version of LogP. In such a synchronous version, a processor could conceivably obtain some information from the fact that it did not not receive any messages at a given time step. This can be incorporated into a strengthened version of the oracle model in which the cells of the oracle memories store words from $\{0, 1, -\}^w$, where $-$ represents a bit that was not written to by the oracle. This has the net effect of changing the lower bound on the running time by at most a constant factor. ∎

Theorem 11 *In the LogP model, sorting n keys known to all processors can be completed in time $O(\frac{n}{m_L} \log n + Pg + L + o)$, provided that $m_L = O(n^\beta)$ for some $\beta < 1$.*

Proof: When $m_L = O(n^\beta)$ for some $\beta < 1$, we can use the recursive version of Columnsort. We again show that each of the three kinds of steps of Columnsort can be performed in the stated time bound. The communication time to perform the odd-even transposition sort is at most twice that required to perform the permutation routing step. The permutation routing step requires each processor to send $\frac{n}{P(P-1)}$ keys to every other processor. This requires a total of $\lceil \frac{n}{P(P-1)} \frac{\log n}{w} \rceil (P-1)$ packets to be sent from each processor, and thus takes time $O(\lceil \frac{n \log n}{w P^2} \rceil Pg + L + o) = O(\frac{n}{m_L} \log n + Pg + L + o)$. The requirement that $m_L = O(n^\beta)$ for some $\beta < 1$ also allows us to recurse until the columns are small enough that a local sort of each column can be performed in the stated time as well. ∎

6.2 The BSP model

We briefly describe bounds on sorting for a variant of the BSP model, the XPRAM [V90b]. In this model, P processors perform computation in *supersteps*, where each superstep consists of each processor executing local computation followed by reads and writes to a globally shared memory. The globally shared memory is akin to that of the PRAM and consists of memory cells of size w. The model has the additional parameters L and g, which are similar to the parameters with the same names in the LogP model. In particular, during superstep j, if processor i performs a_i local operations, b_i reads, and c_i writes, then let $t_j = max_{0 \leq i < p}(\frac{a_i}{g} + b_i + c_i)$. Then, superstep j requires time $\lceil \frac{t_j}{L} \rceil Lg$. As in the LogP model, the throughput of the XPRAM is $m_X = \frac{wP}{g}$ bits per machine cycle.

We can show that the variant of the oracle model used to derive the LogP lower bound is also capable of simulating any XPRAM algorithm with no slowdown. If we assume that each cell of the memory is w bits in size, then when $n > P$, we can again apply Theorem 5. Therefore, any algorithm that sorts n keys in the XPRAM model where each processor performs $O(\frac{n \log m_X}{m_X})$ local operations must perform $\Omega(\frac{n \log m_X}{w})$ reads from the shared memory. Combining this bound with the Columnsort algorithm gives us the following theorem, the proof of which is omitted from this extended abstract.

Theorem 12 *In the XPRAM model, sorting n distinct keys requires time $\Omega(\frac{n}{m_X} \log m_X + gL)$,* *provided $n > P^2$, and can be completed in time $O(\frac{n \log n}{m_X} + Pg + gL)$, provided that $m_X = O(n^\beta)$ for some $\beta < 1$.*

7 Conclusion

We have examined the problem of sorting on parallel models with limited communication bandwidth. Our main results include upper and lower bounds for sorting in an exclusive read variant of the PRAM(m) model which are asymptotically optimal for many practical settings of the parameters and are otherwise asymptotically tight to within at most a logarithmic factor. The form of our bound is noteworthy in that it demonstrates that all efficient parallel algorithms for sorting in this limited bandwidth model depend on large amounts of inter-processor communication. The techniques used to develop the bounds also apply to the LogP model and the BSP model, bridging models which consider the effect of limited bandwidth on parallel computation. For all three models of computation considered, when $m = \Omega(n^\beta)$, the time to sort and the time to transmit all the keys through the shared memory (or the network) are asymptotically equivalent, even in the case where the enitre input is known to each of the processors. Furthermore, as long as $n > p^2$, the bounds do not depend on the parameter p, so when attempting to improve the performance of parallel sorting on machines with limited communication bandwidth, increasing communication bandwidth is more likely to be beneficial than increasing the number of processors.

The lower bound, however, does not apply to the concurrent read version of the PRAM(m) originally introduced by Mansour, Nisan and Vishkin in [MNV94], and thus the asymptotic complexity of sorting in this model remains an open question. Another, perhaps related question of interest is to find further applications for the lower bound technique employing the oracle model in which an oracle simulates the communication between processors.

8 Acknowledgements

We would like to thank Ralph Werchner for his useful comments and suggestions.

References

[ACS90] A. Aggarwal, A. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science* 71: pp 3-28, 1990.

[ACS87] A. Aggarwal, A. Chandra and M. Snir. Hierarchical Memory with Block Transfer. In *Proc. 28^{th} IEEE Symp. on Foundations of Computer Science*, pp. 204-216, 1987.

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

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley: Reading, MA, 1974.

[AKS83] M. Ajtai, J. Komlós and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica* 3: pp. 1 - 19, 1983.

[BC82] A. Borodin and S. Cook. A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation. *SIAM J. of Computing*, 11(2): pp. 287 - 297, 1982.

[C88] R. Cole. Parallel Merge Sort. *SIAM J. of Computing*, 17(4): pp. 770 - 785, 1988.

[CD82] S. Cook, C. Dwork, and R. Reischuk. Upper and Lower Bounds for Parallel Random Access Machines Without Simultaneous Writes. *SIAM J. of Computing* 15: pp. 87-97, 1985.

[CKP+93] D. Culler, R. M. 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 SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 1-12, January 1993.

[CS92] R. Cypher and J. Sanz. Cubesort: A Parallel Algorithm for Sorting N Data Items with S-Sorters. *Journal of Algorithms 13*: pp. 211-234, 1992.

[D94] A. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report: UCB/CSD-94-829, May 1994.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., pp. 869-941. Elsevier Science Publishers: Amsterdam, The Netherlands, 1990.

[L85] T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. on Computers*, c-34(4): pp. 344-354, 1985.

[MNV94] Y. Mansour, N. Nisan and U. Vishkin. Trade-offs Between Communication Throughput and Parallel Time. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*: pp. 372-381, 1994.

[T80] C. Thompson. A Complexity Theory for VLSI. *PhD Thesis.* Carnegie-Mellon University, Pittsburgh, PA, 1980.

[V90a] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8): pp 103-111, August 1990.

[V90b] L. Valiant. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., pp. 943-971. Elsevier Science Publishers: Amsterdam, The Netherlands, 1990.

[VW85] U. Vishkin and A. Wigderson. Trade-Offs between Depth and Width in Parallel Computation. *SIAM Journal of Computing*, 14(2): pp. 303 - 314, 1985.