PARALLEL SORTING WITH LIMITED BANDWIDTH

MICAH ADLER *, JOHN W. BYERS †, AND RICHARD M. KARP ‡

Abstract.
We study the problem of sorting on a parallel computer with limited communication bandwidth. By using the PRAM(m) model, where p processors communicate through a globally shared memory which can service m requests per unit time, we focus on the trade-off between the amount of local computation and the amount of inter-processor communication required for parallel sorting algorithms. Our main result is a lower bound of $\Omega(\frac{n \log m}{m \log n})$ on the time required to sort n numbers on the exclusive-read and queued-read variants of the PRAM(m). We also 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 lower and the upper 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 further convincing evidence that efficient parallel algorithms for sorting rely strongly on high communication bandwidth.

Key words. parallel sorting, limited bandwidth, PRAM, LogP, BSP

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 design algorithms for machines which have limited inter-processor communication bandwidth.

Addressing this limitation has motivated the development of other models of parallel computation, representative of which are the BSP model [33], the LogP model [15], and the PRAM(m) model [35]. Provably efficient algorithms in the PRAM model are not necessarily the most efficient algorithms for these models, so a host of problems need to be re-evaluated in this 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 1.1. **The Sorting Problem**
Input: *n distinct keys $k_1 \dots k_n$, with total order $k_{(1)} < k_{(2)} < \dots < k_{(n)}$.*
Output at processor i: *A sorted list of keys: $k_{\left(\frac{in}{p}+1\right)} \dots k_{\left(\frac{in}{p}+\frac{n}{p}\right)}$*

We concentrate on the complexity of sorting in the PRAM(m) model. In this variant of the classical PRAM model, p processors communicate through a globally shared memory consisting of m memory cells and the entire input, assumed to be of size n, is provided 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

*Department of Computer Science, University of Massachusetts, Amherst, Amherst, MA (micah@cs.umass.edu)

†Department of Computer Science, Boston University, Boston, MA (byers@cs.bu.edu)

‡Department of Computer Science, University of California, Berkeley and the International Computer Science Institute, Berkeley, CA (karp@cs.berkeley.edu) Research supported in part by National Science Foundation operating grant CCR-9005448. A large portion of this research was done while the authors were affiliated with the University of Califonia, Berkeley, and the International Computer Science Institute in Berkeley, California.

derived from local computation and the amount of information derived from interprocessor communication.

Each of the m shared memory cells consists of $\log n$ bits. As in traditional PRAM models, the resolution of contention for these shared memory cells can be defined in a variety of ways. In this paper, we focus on the exclusive-read and queued-read variants of the PRAM(m) model (ER PRAM(m) and QR PRAM(m) respectively); these are defined in Section 2. The main result of this paper is the proof of a lower bound that holds for both the ER PRAM(m) and the QR PRAM(m) on the time required to sort n distinct keys of

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

The bound holds when $n > p^2$, which is the case of primary 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. This is in contrast to sorting results which prove lower bounds on comparison based algorithms. The proof extends to both Monte Carlo and Las Vegas randomized algorithms, and to algorithms which allow for the m shared memory cells to employ a concurrent write contention resolution rule.

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, which is defined more fully in Section 2.1, processors are not required to transmit any information. Rather, all interprocessor communication is simulated by an oracle that is assumed to know the entire input before computation begins. The oracle is of unlimited computational power, and thus can pre-compute any function of the inputs before computation begins. 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 $m = O(n^\beta)$, for some $\beta < 1$, we show that a version of Columnsort [26] has a running time that is bounded by

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

For $n \gg m$, the case of greatest interest, the final factor becomes a small constant and so in this setting the ratio between the upper and lower bounds is $\Theta(\frac{\log n}{\log m})$. When sorting k-bit keys, the algorithm used in the upper bound runs with a slowdown of a factor of $O(\frac{k}{\log n})$ on a machine model in which the input is distributed among all processors rather than stored in a globally shared ROM. This gives the algorithm more credibility from a practical standpoint.

We also show that our results can be generalized to two other models that incorporate limited communication throughput, 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 minimum number of cycles between transmission of successive messages from a processor. To prove a lower bound for the ER PRAM(m) model, we show a lower bound on the number of bits which must be transmitted through the network in order to solve the sorting problem efficiently. Coupling this bound with model dependent lower bounds on the amount of time required to transmit a fixed number of bits in the BSP and LogP models results in lower bounds for sorting in these two models. We defer definitions of the models and the exact form of the bounds 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 or computational power 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) and QR PRAM(m) models 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 several results which also focus on inter-processor communication requirements. Using Thompson's VLSI model [32], Leighton, in [26], 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 $\Omega\left(\frac{n}{m}\right)$ 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 [12], where Borodin and Cook prove that sorting requires TIME $\cdot$ SPACE $= \Omega(\frac{n^2}{\log n})$. Aggarwal, Chandra and Snir show in [3] 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 [5] that sorting requires time $\Omega(\frac{n \log n}{p} + l \log p)$ in a model where reading or writing a block of size b from memory takes time $l + b$.

The PRAM(m) model was introduced in [35], and has been studied subsequently in [28], [19], [18], [9], [30], [1] and [10]. The case where $n \gg p$ was first examined in [30], where 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) and the QR PRAM(m).

An easy upper bound on the time required for sorting can be obtained by using a variant of Cole's parallel merge sort [13] for the PRAM. Cole's algorithm uses n processors to sort n keys in time $\mathrm{O}(\log n)$ time. This algorithm requires the use of a total of $O(n \log n)$ shared memory cells per time step, but by letting each of the m words of shared memory simulate $O(\frac{n \log n}{m})$ cells of the PRAM memory, we can run Cole's algorithm in time $\mathrm{O}(\frac{n \log^2 n}{m})$ on the ER PRAM(m). Related work on upper bounds includes [16], in which Cypher and Sanz discuss a recursive version of Columnsort, and introduce Cubesort, which can be used to obtain a running time of

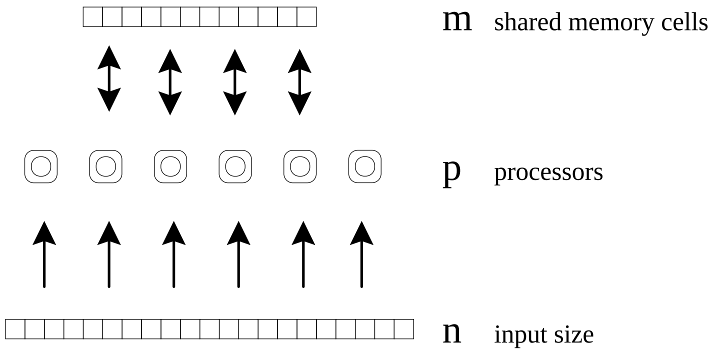

FIG. 2.1. *The PRAM(m) Model*

$O(\frac{n}{m}(1-\beta)^{-2})25^{\log^* n - \log^*(n/m)}$ for sorting on the ER PRAM(m), where $m = O(n^\beta)$. However, this algorithm has substantial overhead and is considerably more involved then the one presented in this paper. A recursive version of Columnsort is also used by Aggarwal and Huang in [6] to obtain an algorithm for sorting in fixed connection networks.

Subsequent to a preliminary version of this paper in [2], Adler [1] provides an algorithm for sorting in the CR PRAM(m) that is considerably faster than the lower bound for the ER PRAM(m) presented in this paper. Thus, that result together with the lower bound presented here imply that the CR PRAM(m) is strictly more powerful than the ER PRAM(m). [1] also slightly improves the ER PRAM(m) upper bound to $O(\frac{n \log p}{m \log n})$.

Finally, with respect to BSP algorithms for sorting, Gerbessiotis and Valiant [21] introduce a randomized algorithm for parallel sorting in the BSP model. Also, subsequent to an earlier version of this paper, the upper bound for sorting in the BSP model has been improved by both Goodrich [24], as well as by Gerbessiotis and Siniolakis [20]. The form of these bounds is deferred to Section 5.2, where they are described in the context of our description of the BSP model.

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. The primary goal of these models is to examine the effectiveness of parallel computation given a sharp limitation on inter-processor communication.

In a classical PRAM, p processors communicate by writing to and reading from a large globally shared memory in unit time. However, in practice, the available per-processor bandwidth to shared memory can be quite small. Access to shared memories is slowed by such factors as long message send overheads [15], contention at memory banks, the fact that memory banks are much slower than processors [11], and bandwidth limitations of the network connecting processors to memory banks. Similar difficulties exist in distributed memory parallel machines. The parameter m of the PRAM(m) model focuses attention on this bottleneck, by enforcing the condition that the shared memory can service only m requests per unit time, where $m < p$. This is modelled as a PRAM consisting of m shared memory cells, each of size $\log n$ bits. We note that all the results in this paper can easily be extended to a model in which each memory cell can hold a word of w bits, independent of the input size.

The input of size n is provided to the PRAM(m) in a read-only shared memory

(ROM) concurrently available to all of the processors. Conceptually, this is equivalent to having each processor begin with an identical copy of the input in its local memory. This capability serves 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. During each synchronized round of computation, every processor can perform one of four actions: it can 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.

As defined in [35], the PRAM(m) model allows processors concurrent read, concurrent write access to the globally shared memory. In this paper, we consider exclusive read and queued read variants of the PRAM(m) model. In the ER PRAM(m), two distinct processors are forbidden from reading the same memory cell at the same time step. We also define the QR PRAM(m), where read contention at a memory cell is resolved as follows: each step of an algorithm completes in k time steps, where k is the maximum number of processors reading the same memory location during that step of the algorithm. Finally, we define the asynchronous QR PRAM(m), where every memory cell services one request if any requests to that cell are pending, and all other requests are stored in a FIFO queue. We note that both queued contention resolution strategies are analogous to those devised in [22] and [23] for the standard PRAM. The contention resolution strategy for write access to the shared memory can be either concurrent write, queued write or exclusive write. Our upper and lower bounds are not affected by this choice, and thus, we leave this component of the model unspecified.

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 combined information a processor receives from all other processors is no more useful than the information it can receive from a single processor with unlimited computational resources and access to all the information the processors have.

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. The oracle transmits information to the processors through p *oracle memories* consisting of cells of size $\log n$ bits. Each of these memory cells is refered to as an *oracle word*. Processor i has read-only access to the ith oracle memory, but is not able to access any of the other oracle memories. We subject the oracle to the restriction that it compute and set all the values of the oracle memory before the processors begin computation. This restriction does not alter the power of the model; it only serves to simplify the analysis.

The processors access the input using a concurrently-readable ROM, which is identical to the ROM of the PRAM(m) model. During computation, at each synchronous time step every processor is allowed to perform one of three actions: it can read the contents of a cell from its oracle memory, read an input from the ROM, or perform local computation. The oracle knows the entire input and the programs

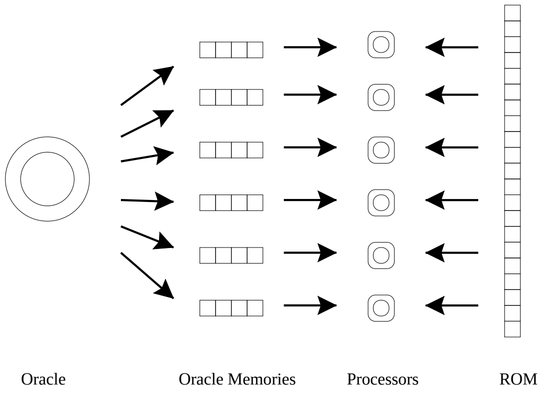

FIG. 2.2. *The flow of information in the oracle model*

executed by each of the processors. We are interested in the tradeoff between the maximum number of time steps required by any processor and the total number of cells read from the oracle memory. Lower bounds on the number of cells that must be read from the oracle memory by all processors combined with the limited throughput of the memory give corresponding lower bounds on the execution time.

More formally, consider algorithms A_e and A_o designed for the ER PRAM(m) and the oracle model, respectively. Let $r(A_e, i, p)$ denote the aggregate number of reads the p processors perform from the shared memory and the ROM on input i. Likewise, let $t(A_e, i, p)$ denote the number of time steps A_e runs on i with p processors. Define $r(A_o, i, p)$ and $t(A_o, i, p)$ similarly for algorithm A_o.

DEFINITION 2.1. *An oracle algorithm A_o* exactly simulates *a PRAM(m) algorithm A_e if for all values of p and on all inputs i, $r(A_o, i, p) = r(A_e, i, p)$, $t(A_o, i, p) = t(A_e, i, p)$, and A_o computes the same output as A_e.*

LEMMA 2.2. *Given any ER PRAM(m) algorithm A_e, there is an oracle algorithm A_o such that A_o exactly simulates A_e.*

Proof. Consider the execution of the ER PRAM(m) algorithm A_e on input i. Let $w(u, v)$ denote the contents of the cell processor u would read at time v. The oracle can compute $w(u, v)$ for all u, v instantaneously in advance of the simulation by using its unlimited resources. To perform the simulation, the oracle simply furnishes $w(u, v)$ in oracle memory u at time v for all u, v. The processors then execute their ER PRAM(m) algorithms, ignoring all write operations and reading from their oracle memory in place of reading from shared memory locations. The simulation has zero slowdown and the processors read the same total number of cells in both executions. ∎

Note that the oracle model can also exactly simulate any QR PRAM(m) algorithm, as well as any asynchronous QR PRAM(m) model. Furthermore, a similar lemma shows that an oracle model in which processors have concurrent read access to a single, m cell oracle memory can exactly simulate any CR PRAM(m) algorithm, but proving strong lower bounds for sorting in this model remains open. The oracle model can also be used to prove lower bounds for randomized algorithms. To allow the oracle to simulate the programs of processors in such a setting, we give the oracle access to the random bits used by each processor prior to the execution of the algorithm.

3. Lower Bounds. In this section, we prove lower bounds for sorting algorithms 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 value of these keys, nor their location within the ROM, then the work remaining can be formalized as the *permutation routing problem.*

DEFINITION 3.1. **The Permutation Routing Problem**
Input: *n memory locations, each containing a processor id, such that each processor id appears exactly $\frac{n}{p}$ times.*
Output at processor i: *a list of the locations where i appears.*

LEMMA 3.2. *Any algorithm for the ER PRAM(m) that sorts n distinct keys in time T can solve any instance of the permutation routing problem of size n in time T. The same is true for the QR PRAM(m).*

Proof. We derive from each location of the permutation routing problem a key to be sorted, where the key is the concatenation of the processor id stored at that location and the location index within the ROM. Sorting these keys is sufficient to inform each processor i of the locations where i appears. $\square$

Thus, any lower bound for the permutation routing problem implies an identical lower bound for sorting. In order to prove our lower bounds for this problem in different scenarios, we first prove a lower bound on a simpler problem in the oracle model. This problem is called the *processor d permutation routing problem*, and is defined as follows, with d any processor id between 1 and p. The input is chosen uniformly at random from the set of all possible n element inputs to the permutation routing problem. Processor d is required to determine the list of locations in the ROM where d appears, but the remaining processors are not required to do anything. We are interested in the average, over all possible n element inputs to the permutation routing problem, of the number of oracle words processor d reads, given a limitation on how many ROM locations processor d reads.

LEMMA 3.3. *In any deterministic algorithm for the oracle model that solves the processor d permutation routing problem when $n > p^2$, if the average number of oracle words read by processor d is at most $\frac{n \log m}{8p \log n}$, and processor d never reads more than $\frac{n \log m}{m \log n}$ ROM locations, then on an input chosen uniformly at random, processor d produces an incorrect result with probability at least $\frac{1}{4}$.*

The proof of this lemma is the main technical portion of our lower bound, and is deferred to Section 3.1. We first discuss its implications.

THEOREM 3.4. *For any deterministic ER PRAM(m) algorithm that solves the sorting problem for any set of n distinct keys, where $n > p^2$, the average over all permutations of the input keys of the time required by the algorithm is at least $\frac{n \log m}{8m \log n}$. The same is true for any deterministic algorithm for the QR PRAM(m).*

Proof. We assume there is a sorting algorithm A for the ER PRAM(m), where the average over all permutations of the input keys of the time to perform A is less than $\frac{n \log m}{8m \log n}$, and reach a contradiction. Let $w(A, C)$ be the total number of words read from the shared memory when algorithm A is executed on input C. Let $r(A, C, i)$ be the number of ROM locations read by processor i when A is executed on input C. We shall use $w(A, C)$ and $r(A, C, i)$ to represent these quantities for both the PRAM(m) as well as the oracle model.

Since at most m words can be read from the shared memory at any time step, when C ranges over all permutations of the input keys, the average of $w(A, C)$ is less than $\frac{n \log m}{8 \log n}$. Also, the average over all such C of $\max_i r(A, C, i)$ is less than $\frac{n \log m}{8m \log n}$. By Lemmas 2.2 and 3.2, this implies the existence of an oracle algorithm A' for the permutation routing problem, where the average over all input permutations C of $w(A', C)$ is less than $\frac{n \log m}{8 \log n}$, and the average over all input permutations of $\max_i r(A, C, i)$ is less than $\frac{n \log m}{8m \log n}$.

For any such A', there is some processor d such that the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, and the average over all inputs of $r(A', C, d)$ is at most $\frac{n \log m}{8m \log n}$. By Markov's inequality, in A', the fraction of inputs C where $r(A', C, d) \geq \frac{n \log m}{m \log n}$ is at most $\frac{1}{8}$. Thus, we can use A' to construct algorithm A'' for the oracle model. In A'', processor d behaves the same as in A', except that it only performs at most $\frac{n \log m}{m \log n}$ ROM queries. If in A', processor d requires more ROM queries on a given input, on that input in A'', processor d returns an arbitrarily chosen permutation. Since algorithm A' responds correctly on all inputs, A'' responds correctly on at least $\frac{7}{8}$ of all possible inputs. This contradicts Lemma 3.3, and thus there does not exist such an algorithm A for the ER PRAM(m). The proof for the QR PRAM(m) is identical. □

For Las Vegas algorithms, or randomized strategies that are guaranteed to provide a correct solution with a bound only on the expected running time, we have a lower bound which follows from a direct application of Yao's lemma [36].

THEOREM 3.5. *For any Las Vegas algorithm A_v for the ER PRAM(m) where $n > p^2$, there is some permutation of the inputs I for the sorting problem, such that A_v requires expected time at least $\frac{n \log m}{8m \log n}$ to solve I. Also, the expected running time of any Las Vegas algorithm on an input chosen uniformly at random from the set of all inputs is at least $\frac{n \log m}{8m \log n}$. The same is true for the QR PRAM(m).*

Proof. Yao's lemma [36] states that if there is a distribution over the inputs such that every deterministic algorithm requires time at least L for that distribution, then for any randomized algorithm there exists an input for which the expected running time is at least L. This combined with Theorem 3.4 directly implies the first claim of Theorem 3.5. The second claim follows from Theorem 3.4 and the fact that any Las Vegas algorithm is actually a distribution over deterministic algorithms, and thus cannot fare better than the best deterministic algorithm. □

For Monte Carlo algorithms, or randomized strategies with bounded running time which provide a correct solution with probability greater than $\frac{3}{4}$, we have the following lower bound.

THEOREM 3.6. *For any Monte Carlo algorithm A_m for the ER PRAM(m) where $n > p^2$, there is some input I for the sorting problem, such that A_m requires time at least $\frac{n \log m}{8m \log n}$ to solve I. Also, for the uniform distribution over all possible inputs, the running time of any Monte Carlo algorithm is at least $\frac{n \log m}{8m \log n}$. The same is true for the QR PRAM(m).*

We prove this theorem using an alternate formulation of Yao's lemma, provided in [29]:

LEMMA 3.7. *Let P_1 be the success probability of a T step randomized algorithm solving problem B, where the success probability is taken over the random choices made by the algorithm, and minimized over all possible inputs. Let P_2 be the success probability over a distribution $\mathcal{D}$ of inputs, maximized over all possible T step* *deterministic algorithms to solve B. Then,* $P_1 \leq P_2$.

Theorem 3.6 follows from a direct application of Lemma 3.7 to the following lemma.

LEMMA 3.8. *For any deterministic ER PRAM(m) algorithm A that solves the sorting problem for any set of n distinct keys, where* $n > p^2$*, if A always uses less than* $\frac{n \log m}{8m \log n}$ *time steps, then when the input is chosen uniformly at random from the set of all possible permutations of the inputs, the probability that every processor successfully produces the correct output is* $\leq \frac{3}{4}$*. The same is true for any deterministic algorithm for the QR PRAM(m).*

Proof. We assume that there is such a deterministic algorithm A_d which produces the correct output with probability greater than $\frac{3}{4}$, and reach a contradiction. In such an algorithm, for every input C, $w(A_d, C) < \frac{n \log m}{8 \log n}$ and $\max_i r(A_d, C, i) < \frac{n \log m}{8m \log n}$. By Lemmas 2.2 and 3.2, this implies the existence of an oracle algorithm A'_d for the permutation routing problem, where the total number of oracle words read by all of the processors is less than $\frac{n \log m}{8 \log n}$, and $\max_i r(A'_d, C, i) < \frac{n \log m}{8m \log n}$. For any such A', there is some processor d such that the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, and $r(A'_d, C, d) \leq \frac{n \log m}{8m \log n}$. However, this implies the existence of an algorithm for the processor d permutation routing problem, where the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, $r(A'_d, C, d) < \frac{n \log m}{8m \log n}$, and yet processor d responds correctly with probability $> \frac{3}{4}$. This contradicts Lemma 3.3, and thus there does not exist such an algorithm A_d. □

3.1. The Processor d Permutation Routing Problem. In this subsection, we prove Lemma 3.3, restated here for convenience.

Lemma 3.3 *In any deterministic algorithm for the oracle model that solves the processor d permutation routing problem when* $n > p^2$*, if the average number of oracle words read by processor d is at most* $\frac{n \log m}{8p \log n}$*, and processor d never reads more than* $\frac{n \log m}{m \log n}$ *ROM locations, then on an input chosen uniformly at random, processor d produces an incorrect result with probability at least* $\frac{1}{4}$.

For concreteness and simplicity, we represent an input to the permutation routing problem as a bit matrix B with n rows and p columns. If processor id j appears in location i in the permutation routing problem instance, then $B_{ij} = 1$; otherwise $B_{ij} = 0$. Rows of B correspond to locations, 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 to location k reveals row k of this matrix. In order to correctly solve the processor d permutation routing problem, processor d must be able to specify column d of B exactly.

The proof of this lower bound employs the "little birdie" principle: giving a processor additional information never increases the complexity of the problem that the processor must solve. The side information that the "little birdie" reveals *a priori* to processor d is a perturbed representation of the input specification B called a *hidden matrix*. Providing this matrix to processor d allows us to prove lower bounds on the amount of information subsequent ROM queries provide to processor d.

The hidden matrix is chosen as follows. A permutation routing problem instance

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

FIG. 3.1. *Permutation routing problem input: $n = 6$, $p = 3$.*

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

FIG. 3.2. *A hidden matrix for processor 1 consistent with input in Figure 3.1*

B is chosen uniformly at random, and revealed to the oracle. Then, based on the choice of B, a hidden matrix H is chosen and revealed to processor d and the oracle. To construct the hidden matrix H, we first choose an $n \times p$ matrix G uniformly at random from binary matrices whose d^{th} column is identical to the d^{th} column of B, and the remaining columns each have either $\lfloor \frac{n}{p(p-1)} \rfloor$ 1's or $\lceil \frac{n}{p(p-1)} \rceil$ 1's, such that every row with a 1 in column d has exactly one 1 in some other column, and all other rows have no 1's. The hidden matrix H is defined to be $H = B \oplus G$, where $\oplus$ denotes the bitwise XOR of the two matrices. This mapping evenly redistributes the 1's from column d of B across the other columns while leaving all other rows unchanged. A pictorial representation of an input matrix and a possible hidden matrix constructed from it are given in Figures 3.1 and 3.2. Based on B and H, the oracle places some number of words in processor d's oracle memory. Processor d then executes its deterministic algorithm, and produces its output.

We say that an input C is *consistent* with a hidden matrix H if there is a matrix G perturbing C as defined above such that $C \oplus G = H$. Let $\mathcal{C}(H)$ denote the set of inputs that are consistent with hidden matrix H. For deterministic algorithms, the pair $<C, H>$, where C is an input consistent with hidden matrix H, uniquely determines S, the setting of oracle memory d. We say that S is consistent with hidden matrix H, if there is $C \in \mathcal{C}(H)$ such that H and C determine S. Let $\mathcal{S}(H)$ be the set of all settings of the oracle memory S that are consistent with H. Also, we say that input C is consistent with both H and S if H and C determine S. Let $\mathcal{C}(H, S)$ be the set of all inputs C that are consistent with H and S. Finally, for a hidden matrix H and an input $C \in \mathcal{C}(H)$, let the indicator variable $R(H, C) = 1$ if processor d produces the correct output on C when given H, and 0 otherwise.

We first demonstrate that for any given hidden matrix, setting of the oracle memory and algorithm, there cannot be too many inputs for which the algorithm produces the correct result.

CLAIM 3.9. *In any algorithm for the processor d permutation routing problem, where processor d is provided with any hidden matrix H and any setting of the oracle memory S, if processor d performs no more than $\frac{n \log m}{m \log n}$ ROM queries, then*

$$\sum_{C \in \mathcal{C}(H,S)} R(H,C) \leq Z, \text{ where } Z = \sum_{r=1}^{\frac{n}{p}} \binom{\frac{n \log m}{m \log n}}{r}$$

Proof. After the setting of the oracle memory and the hidden matrix have been fixed, we can model 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 at most $\frac{n \log m}{m \log n}$ ROM queries. The number of distinct results that processor d can produce is at most the number of leaves in this tree. We show that for any non-redundant algorithm, i.e. an algorithm that only examines each ROM location once, the tree has at most Z leaves. Any redundant algorithm can be simulated by a non-redundant algorithm, and thus the number of distinct results processor d can produce is at most Z for all algorithms.

Suppose processor d reads the value of row i of C from the ROM. Then, either (a) row i of H is identical to row i of C, or (b) C has a 1 in column d of row i whereas H has a 1 in some other column. Since processor d knows H at the start of the algorithm, the decision tree has branching factor two. Since processor id d only appears in $\frac{n}{p}$ elements, at most $\frac{n}{p}$ of these ROM queries can discover elements where processor id d appears. Mapping successful discoveries to left branches and unsuccessful queries to right branches ensures that any algorithm which is non-redundant has a decision tree where any path from root to leaf can have at most $\frac{n}{p}$ left branches. The number of distinct leaves of the decision tree that can be reached by a path from root to leaf with k left branches is at most $\binom{\frac{n \log m}{m \log n}}{k}$. Thus, the possible number of leaves in the decision tree is at most $\binom{\frac{n \log m}{m \log n}}{0} + \binom{\frac{n \log m}{m \log n}}{1} + \ldots + \binom{\frac{n \log m}{m \log n}}{n/p}$. □

For any algorithm for the processor d permutation routing problem, let $a(H)$ be the average number of oracle words provided to processor d, where the average is taken over all inputs in $\mathcal{C}(H)$.

CLAIM 3.10. *Consider any algorithm for the processor d permutation routing problem where $n > p^2$, where the little birdie provides processor d with a hidden matrix and where processor d performs at most $\frac{n \log m}{m \log n}$ ROM queries. For any H, if $a(H) < \frac{n \log m}{4p \log n}$, then on an input chosen uniformly at random from the set $\mathcal{C}(H)$, the probability that processor d produces the correct output is $< \frac{1}{2}$.*

Proof. The total number of inputs in $\mathcal{C}(H)$ on which processor d responds correctly is at most:

$$\sum_{C \in \mathcal{C}(H)} R(H,C) = \sum_{S \in \mathcal{S}(H)} \sum_{C \in \mathcal{C}(H,S)} R(H,C),$$

and so the probability of a successful response is at most

$$\frac{1}{|\mathcal{C}(H)|} \sum_{S \in \mathcal{S}(H)} \sum_{C \in \mathcal{C}(H,S)} R(H,C).$$

But, by Claim 3.9, this is at most Y, where

$$Y = \sum_{S \in \mathcal{S}(H)} \min\left(\frac{Z}{|\mathcal{C}(H)|}, \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}\right).$$

We assume that $Y \geq \frac{1}{2}$, and reach a contradiction by showing that this implies that $a(H) \geq \frac{n \log m}{4p \log n}$. First, note that $Y \geq \frac{1}{2}$ implies that $|\mathcal{S}(H)| \geq \frac{|\mathcal{C}(H)|}{2Z}$. Let $|S|$ denote the number of words in oracle memory setting S. Because there are at most $2^{r \log n}$ settings of the oracle memory with $\leq r$ words, there are at least $\frac{3|\mathcal{C}(H)|}{8Z}$ oracle memory settings $S \in \mathcal{S}(H)$ such that

$$|S| \geq \frac{\log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{\log n}.$$

We call such oracle memory settings *large settings*. We use this to minimize $a(H)$ subject to $Y \geq \frac{1}{2}$. Note that

$$a(H) = \sum_{S \in \mathcal{S}(H)} |S| \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}.$$

By counting the total contribution to $a(H)$ of the large settings, we see that

$$a(H) \geq \frac{3|\mathcal{C}(H)|}{8Z} \cdot \frac{\log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{\log n} \cdot \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}.$$

We can assume that for any $S \in \mathcal{S}(H)$, $|\mathcal{C}(H,S)| \geq Z$, since given any valid solution, any sets $\mathcal{C}(H,S)$ of smaller cardinality can be combined into sets of cardinality Z without changing the value of Y, and without increasing $a(H)$. Thus,

$$a(H) \geq \frac{3 \log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{8 \log n}.$$

Using the fact that $n > p^2$, we have that for any H,

$$|\mathcal{C}(H)| \geq \binom{\lfloor \frac{n}{p-1} \rfloor}{\lfloor \frac{n}{p(p-1)} \rfloor}^{p-1}.$$

Using the inequality $\left(\frac{a}{b}\right)^b \leq \binom{a}{b} \leq \left(\frac{ae}{b}\right)^b$, and the fact that $m \leq p$ implies that the sum expressed by Z is dominated by the final term, this gives us

$$a(H) \geq \frac{3n \log m}{8p \log n} - o\left(\frac{n \log m}{p \log n}\right),$$

which is a contradiction. Thus, the probability of a correct response from processor d is less than $\frac{1}{2}$. □

Proof. (of Lemma 3.3) The number of hidden matrices consistent with a given input is invariant over the choice of input and the number of inputs consistent with a given hidden matrix is invariant over the choice of matrix. Thus, if the average, over all inputs, of the number of oracle words read by processor d is at most $\frac{n \log m}{8p \log n}$, then the average of $a(H)$ over all H is at most $\frac{n \log m}{8p \log n}$. By Markov's inequality, this implies that $a(H) \leq \frac{n \log m}{4p \log n}$ for at least $\frac{1}{2}$ of the hidden matrices H. The input to the permutation routing problem is chosen uniformly at random from the set of all inputs. One method for producing this uniform distribution is as follows: first a hidden matrix H_i is chosen uniformly at random from the set of all hidden matrices that can be given to d. Then, an input C_i is chosen uniformly at random from $\mathcal{C}(H_i)$. With probability at least $\frac{1}{2}$, for the resulting choice of H_i, $a(H_i) \leq \frac{n \log m}{4p \log n}$. By Claim 3.10, if $a(H_i) \leq \frac{n \log m}{4p \log n}$, then the probability that processor d responds correctly is at most $\frac{1}{2}$. Therefore, the probability that processor d responds correctly on an input chosen uniformly at random, when given a hidden matrix, is at most $\frac{3}{4}$. By the little birdie principle, the probability that processor d responds correctly on an input chosen uniformly at random, when not given a hidden matrix, is also at most $\frac{3}{4}$. □

4. The Upper Bound. We show that a version of Leighton's Columnsort [26] performs well in both the ER PRAM(m) and the QR PRAM(m). Moreover, this algorithm runs in a model where there is no globally shared ROM for the input (which may not always be realistic in practice), but instead the input is distributed across the processor's local memories.

THEOREM 4.1. *There is an ER PRAM(m) algorithm for sorting n keys which runs in time*

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

provided $p \geq m \log n$ *and* $m = O(n^{\beta})$, *for some* $\beta < 1$. *This algorithm has the same running time on the QR PRAM(m).*

Proof. It is sufficient to provide a sorting algorithm for the ER PRAM(m). To do so, we use a recursive version of Columnsort, which we describe below. In Columnsort, the n keys are thought of as elements in a matrix M. There is a requirement on the aspect ratio of M: if M is an $s \times r$ matrix, then s must be larger than r^2. The elements are sorted using seven phases, where each phase is one of three types: phases that sort the columns of the matrix, phases that perform an odd-even transposition sort along the rows of the matrix and phases that route a fixed permutation of the matrix elements, where each column routes an equal number of elements to every other column. The following simple description of Columnsort is provided in [27]. In phases 1, 3, and 7, the columns are sorted into increasing order. In phase 5, odd columns are sorted into increasing order and even columns are sorted into decreasing order. In phase 2, the matrix is "transposed": the items are picked up in column major order and set down in row-major order (preserving the shape of the matrix). Phase 4 applies the reverse of the permutation applied in phase 2, and phase 6 performs two steps of odd-even transposition sort to each row.

We specify a call to recursive Columnsort in the ER PRAM(m) by two parameters: k, the number of keys to be sorted and a, the number of memory cells dedicated to this function call. We develop the following recurrence relation for the running time of recursive Columnsort, where the keys are contained in a ROM of size n:

$$RC(k,a) = \mathrm{O}\left(\frac{k}{a}\right), \text{ if } k \geq a^3 \log n$$
$$RC(k,a) = 4\ RC\left(k^{2/3}, ak^{-1/3}\right) + \mathrm{O}\left(\frac{k}{a}\right), \text{ otherwise.}$$

In the version of Columnsort we use to obtain this recurrence, each of the m memory cells is assigned a set of $\log n$ processors. These $m \log n$ processors sort the n keys, and inform each of the $p - m \log n$ remaining processors of the range of keys

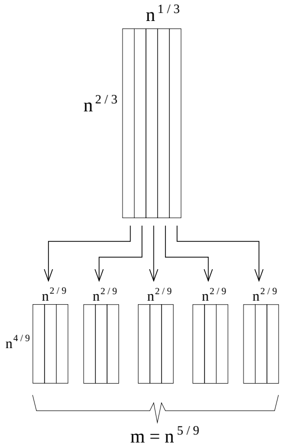

FIG. 4.1. *The recursive algorithm when* $m = n^{5/9}$

that they need to output. Note that since the algorithm makes effective use of only $m \log n$ processors, this algorithm is consistent with the observation that increasing communication throughput, as opposed to adding processors, is required for faster parallel sorting.

The base case for the recurrence, where $k \geq a^3 \log n$, works as follows. The k keys are thought of as being entries in a matrix M of keys of size $\frac{k}{a \log n} \times a \log n$; this matrix satisfies the aspect ratio requirement of Columnsort. We have $a \log n$ available processors, and each of these assigned to one column of M. Thus, a memory cell serving a set of $\log n$ processors handles data transfer for $\log n$ columns. Recall that each memory cell is assigned to the set of $\log n$ columns to which its processors are assigned. We now show that we can implement each of the three types of phases of Columnsort on the PRAM(m) in time $\mathrm{O}(\frac{k}{a})$. Sorting the columns can be performed by each of the a processors locally in time $O(\frac{k \log k}{a \log n})$ by any of a variety of known serial algorithms, which is at most $O(\frac{k}{a})$.

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 is routed by the source and destination processors using the shared memory location that is assigned to the destination processor. The source processor writes to this memory cell the key's address in the ROM, and then the destination processor reads this address. Since each memory cell handles $\log n$ columns, and each column contains $\frac{k}{a \log n}$ keys, the total number of addresses written to each memory cell is $O(\frac{k}{a})$. Thus, the entire permutation can be routed in time $O(\frac{k}{a})$. One phase of odd-even transposition sort can be performed by each processor routing all the keys currently in its column to the processor that is assigned to the neighboring column. This can also be done in time $O(\frac{k}{a})$.

For the case where $k < a^3 \log n$, we sort the keys as follows. The keys are thought of as the elements in a matrix M of size $k^{2/3} \times k^{1/3}$. This matrix satisfies the aspect ratio of Columnsort, and thus we use Columnsort to sort this matrix as well. We can still route the permutations of the matrix M and perform the phases of odd-even transposition sort in time $\mathrm{O}(\frac{k}{a})$. This follows from the fact that for each permutation only k keys need to be routed through the shared memory and this operation can be performed without conflict while making use of each cell during each of the $\mathrm{O}(\frac{k}{a})$ time steps. For the phases which sort the columns of the matrix, we employ parallel calls to recursive Columnsort, one call per column. Each column consists of $k^{2/3}$ keys, and we evenly distribute the a available memory cells (with their associated processors) across the columns. Thus, each of the 4 sorting phases takes time $\mathrm{RC}\left(k^{2/3}, ak^{-1/3}\right)$. An example which graphically describes one level of this recursive procedure is given in Figure 4.1.

The algorithm starts with a call to recursive Columnsort with n keys and m memory cells (with each assigned $\log n$ processors). After the n keys are sorted, the $m \log n$ processors can inform each of the remaining $p - m\log n$ processors of the correct sorted list of $\frac{n}{p}$ keys to output in the same amount of time as is required to route one permutation. To analyze the running time of sorting n keys on the ER PRAM(m), we evaluate the recurrence $\mathrm{RC}\,(n, m)$. The running time of the algorithm is dominated by the time for sorting at the bottom of the recursion at level j: $\mathrm{O}(\frac{n}{m}4^j)$. When m is a fixed function of n where $m = \mathrm{O}(n^\beta)$ for $\beta < 1$, this j is the smallest integer that satisfies $n^{((2/3)^j)} \leq n^{1-\beta}$, and the following bound on 4^j follows directly:

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

Substituting in to the formula above gives an upper bound on the running time of $O(\frac{n}{m}(1-\beta)^{-3.42})$. □

5. 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 throughput. We give a brief discussion of translating the ER PRAM(m) lower bound for sorting into the LogP model [15] and the BSP model [33] and state the best known upper bounds for sorting in these models.

5.1. The LogP model. In the LogP model, limited communication throughput 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 using the global shared memory used in the PRAM(m) model. We make the additional assumption that the point-to-point messages, or 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

$$m_L = \left\lceil \frac{wP}{g\log n} \right\rceil$$

$\log n$-bit words per machine cycle. We denote this expression for throughput by m_L to make plain its correspondence with m in the PRAM(m) model. In order to prove a lower bound for sorting in the LogP model, we first show that any LogP model algorithm can be simulated in the oracle model.

LEMMA 5.1. *If $w > \log g$, then given any LogP algorithm A_l that completes in time T, there is an oracle algorithm A_o that computes the same function as A_l, also in time T, and A_o writes at most $2m_LT$ words to each of the oracle memories.*

Proof. We partition the time steps of the LogP algorithm A_l into *epochs*, where each epoch consists of g consecutive time steps. Note that each processor receives at most one message during any epoch. We number the bits of each oracle memory, ignoring word boundaries. We can simulate A_l with an oracle algorithm A_o, where epoch i in A_l is represented in A_o by the oracle utilizing bits $(i-1)(\lfloor \log g \rfloor + 1 + w) + 1$ through $i(\lfloor \log g \rfloor + 1 + w) + 1$ in each oracle memory. The first $\lfloor \log g \rfloor + 1$ of the bits for each epoch are used to inform processor j of whether or not a message arrives during that epoch, and in the case of an arrival, the exact time step of the arrival during the epoch. In the case of an arrival, the remaining w bits contain the message contents. In A_o, the processors execute their algorithms for A_l, ignoring steps where messages are sent, and reading from the oracle memory at the start of every epoch. The total number of bits read is $P(\lfloor \log g \rfloor + 1 + w)\lceil \frac{T}{g} \rceil$. When $w > \log g$, this is $O(m_L T \log n)$, and thus at most $O(m_L T)$ words are required in each oracle memory. □

We briefly point out why we require that the oracle give each processor the timing information provided by the $\lfloor \log g \rfloor + 1$ additional bits used for each epoch. The LogP model is an asynchronous model, and thus processors cannot use the timing information to ensure the correctness of the algorithm. However, for the purpose of running time analysis, it is assumed that each processor behaves synchronously. Thus, in the optimal algorithm, it is possible that a processor is able to use the timing information to achieve a better running time than an algorithm that does not make inferences based on this information. Note that any algorithm that does not use this timing information can be simulated in the oracle model using at most $O(m_L T)$ words, even in the case where $w \leq g$.

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, even in the case where every processor knows all the keys in advance. Let $T_s(n)$ be the optimal sequential time required to sort the n keys.

THEOREM 5.2. *In the LogP model, sorting n distinct keys requires expected time*

$$\Omega\left(\frac{T_s(n)}{P} + \frac{n \log m_L}{m_L \log n} + L + o + g\right),$$

provided that $n > P^2$, $w > \log g$ and $T_s(n) \geq L$. This bound holds even in the case that every processor has access to every key at the start of the algorithm.

Proof. We first assume there exists an algorithm A_l for the LogP model, where the average over all inputs of the time to perform A_l is at most $\frac{n \log m_L}{16 m_L \log n}$, and reach a contradiction. By Lemma 5.1, such an algorithm A_l implies the existence of an oracle model algorithm where the average number of oracle words used is at most $\frac{n \log m_L}{8 \log n}$, and the average of the maximum number of ROM queries by any processor is at most $\frac{n \log m_L}{8 m_L \log n}$. However, as we saw in Theorem 3.4, this leads to a contradiction of Lemma 3.3, and thus there does not exist such an algorithm A_l for the ER PRAM(m). The lower bound then follows from the fact at least one transmission, which requires time at least $\max(L, o, g)$, is required in any algorithm that sorts in time faster than time $T_s(n)$, and that the time to sort on P processors is no faster than the optimal time to sort on one processor divided by P. □

A recursive implementation of Columnsort similar to that presented in Section 4 can be tuned to deliver the following asymptotic performance:

THEOREM 5.3. *In the LogP model, sorting n keys known to all processors can be* *completed in time*

$$O\left(T_s\left(\frac{n}{P}\right)+\frac{n}{m_L}+L+o+g\right),$$

provided that $P=O(n^{\beta})$ for some constant $\beta<1$, and that $w\leq\log n$.

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. Otherwise, the algorithm, as well as the resulting bounds, are the same.

5.2. The BSP model. We now briefly describe analogous bounds for sorting in the bulk-synchronous parallel (BSP) model proposed by Valiant [33][34]. The model consists of a set of processors capable of transmitting point- to-point messages through a communication network and facilities for performing barrier synchronization across any subset of the processors. The three parameters of the model are P, the number of processors, L, the minimum number of local computation steps between successive synchronization operations and g, the ratio between the throughput of local computation to the throughput at which a processor may inject point-to-point messages into the network. As in the LogP model, g enforces a limit on the communication throughput available to each processor. We assume that each transmitted packet is at most w bits in size.

Computation in the BSP model proceeds in *supersteps*, wherein each processor in parallel executes a task consisting of some number of local computation steps, message transmissions and message receipts, subject to the constraints imposed by the parameter g. The superstep lasts for kL time steps, where k is the minimum integer such that all processors have completed their tasks before time kL. As in the LogP model, the total communication throughput in the BSP model is $m_B=\left\lceil\frac{wP}{g\log n}\right\rceil$ words per step of local computation, where each word consists of $\log n$ bits.

THEOREM 5.4. *In the BSP model, sorting n distinct keys requires time*

$$\Omega\left(\frac{T_s(n)}{P}+\frac{n\log m_B}{m_B\log n}+L+o+g\right),$$

when $n>P^2$ and $T_s(n)\geq L$.

Proof. Using the technique from Lemma 5.1, we see that the oracle model is also capable of simulating any BSP algorithm. Thus, the existence of any algorithm that sorts faster than $\frac{n\log m_B}{16m_B\log n}$ again implies the existence of an algorithm that contradicts Lemma 3.3. □

Using recursive columnsort, it is straightforward to show that in the BSP model, sorting any n keys can be completed in time $\mathrm{O}\left(T_S\left(\frac{n}{P}\right)+\frac{n}{m_B}+L+g+o\right)$, provided that $P=\mathrm{O}(n^{\beta})$ for some constant $\beta<1$, and that $w\leq\log n$. This result for sorting in the BSP model compares with the previous best randomized methods of Gerbessiotis and Valiant [21] for the BSP model with the assumption that each packet that is transmitted consists of exactly one key. Their algorithms run in time $\mathrm{O}\left(\frac{n\log n}{P}+gp^{\epsilon}+\frac{gn}{P}+L\right)$, with high probability, for any positive constant $\epsilon<1$, and for $P\leq n^{1-\delta}$, where δ is a small constant depending on ϵ. After the preliminary version of this paper appeared in [2], work on this problem by Goodrich [24] has tightened the bounds for sorting on the BSP, giving deterministic algorithms which run in time $\mathrm{O}\left(\frac{n\log n}{P}+(L+\frac{gn}{P})(\log n/\log(n/P))\right)$ for all values of P, coupled with a matching lower bound. Other recent work by Gerbessiotis and Siniolakis [20] gives a deterministic algorithm for sorting on the BSP which runs in time $(1 + o(1))(\frac{n \log n}{P} + L) + O(\frac{gn}{P})$ for $P = n^{1-\epsilon}$, $0 < \epsilon < 1$, and uses 1-optimal local computation.

6. 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 on exclusive and queued-read variants 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 entire 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 [30], and thus the asymptotic complexity of sorting in this model remains an open question.

7. Acknowledgements. We would like to thank Ralph Werchner for his useful comments and suggestions on an earlier version of this paper and for the valuable and insightful suggestions of the anonymous SICOMP referees.

REFERENCES

[1] M. Adler, *New Coding Techniques for Improved Bandwidth Utilization*, Proc. 37^{th} IEEE Symp. on Foundations of Computer Science, pp. 173-182, 1996.

[2] M. Adler, J. Byers and R. Karp, *Parallel Sorting with Limited Bandwidth*, Proc. 7th ACM Symp. on Parallel Algorithms and Architectures, pp. 129 - 136, 1995.

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

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

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

[6] A. Aggarwal and M. Huang, *Network Complexity of Sorting and Graph Problems and Simulating CRCW PRAMs by Interconnection Networks*, Proc. of 3rd Aegean Workshop on Computing, LNCS , pp. 339-350, 1988.

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

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

[9] Y. Azar, *Lower Bounds for Threshold and Symmetric Functions in Parallel Computation*, SIAM J. on Computing, 21(2): pp. 329 - 338, 1992.

[10] P. Beame, F. Fich, and R. Sinha, *Separating the Power of CREW and EREW PRAMs with Small Communication Width*, Information and Computation, vol. 138, no. 1, October 1997, pp. 89-99.

[11] G. Blelloch, P. Gibbons, Y. Matias and M. Zagha, *Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors*, Proc. 7^{th} ACM Symposium on Parallel Algorithms and Architectures, pp. 84-94, 1995.

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

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

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

[15] D. Culler, R. M. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, *LogP: A Practical Model of Parallel Computation*, Communications of the ACM, November 1996, Vol. 39, No.11. Preliminary version in Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pp. 1-12, January 1993.

[16] 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.

[17] A. Dusseau, D. Culler, K. Schauser and R. Martin, *Fast Parallel Sorting Under LogP: Experience with the CM-5*, IEEE Transactions on Parallel and Distributed Systems, Volume 7, Number 8, pp. 791-805, August 1996,

[18] F. Fich, M. Li, P. Ragde, and Y. Yesha, *Lower Bounds for Parallel Random Access Machines with Read Only Memory*, Information and Computation, vol. 83, no. 2, November 1989, pages 234-244.

[19] F. Fich, P. Ragde, and A. Wigderson, *Relations Between Concurrent-Write Models of Parallel Computation*, SIAM J. on Computing, vol. 17, no. 3, June 1988, pages 606-627.

[20] A. Gerbessiotis and C. Siniolakis, *Deterministic Sorting and Randomized Median Finding on the BSP Model*, Proc. 8th ACM Symposium on Parallel Algorithms and Architectures, 1996.

[21] A. Gerbessiotis and L. Valiant, *Direct Bulk-Synchronous Algorithms*, Journal of Parallel and Distributed Computing, 22:251-267, 1994.

[22] P. Gibbons, Y. Matias, and V. Ramachandran, *The QRQW PRAM: Accounting for Contention in Parallel Algorithms*, Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, pp. 638-648, January 1994.

[23] P. Gibbons, Y. Matias, and V. Ramachandran, *Efficient Low-Contention Parallel Algorithms*, Proc. 6th ACM Symposium on Parallel Algorithms and Architectures, pp. 236-247, June 1994.

[24] M. Goodrich, *Communication-Efficient Parallel Sorting*, Proc. 28^{th} Annual ACM Symposium on Theory of Computing, pp. 247 - 256, 1996.

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

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

[27] T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, 1992.

[28] M. Li and Y. Yesha, *Separation and Lower Bounds for ROM and Non-deterministic Models of Parallel Computation*, Proc. 18th ACM Symposium on Theory of Computing, pp. 177-187, 1986.

[29] P. MacKenzie, *Lower Bounds for Randomized Exclusive Write PRAMs*, Proc. 7th ACM Symposium on Parallel Algorithms and Architectures, pp. 254-263, July 1995.

[30] Y. Mansour, N. Nisan and U. Vishkin, *Trade-offs Between Communication Throughput and Parallel Time*, Proc. of the 26^{th} Annual ACM Symposium on Theory of Computing, pp. 372-381, 1994.

[31] 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.

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

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

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

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

[36] A. Yao, *Probabilistic Computations: Toward a Unified Measure of Complexity*, Proc. 18^{th} IEEE Symp. on Foundations of Computer Science, pp. 222-227, 1977.