

New Coding Techniques for Improved Bandwidth Utilization

Micah Adler
Computer Science Division
University of California Berkeley
and
International Computer Science Institute
micah@cs.berkeley.edu

January 11, 1998

1 Introduction

The introduction of parallel models that account for communication between processors has shown that interprocessor bandwidth is often the limiting factor in parallel computing. In this paper, we introduce a new coding technique for transmitting the XOR of carefully selected patterns of bits to be communicated which greatly reduces bandwidth requirements in some settings. This technique has broader applications. For example, we demonstrate that the coding technique has a surprising application to a simple I/O (Input / Output) complexity problem related to finding the transpose of a matrix.

Our main results are developed in the PRAM(m) model, a limited bandwidth PRAM model where p processors communicate through a small globally shared memory of m bits. We provide new algorithms for the problems of sorting and permutation routing. For the concurrent read PRAM(m), as p grows with m held constant, our sorting algorithm outperforms any previous algorithm by $\Omega(\log^c p)$ for any constant c . The combination of a lower bound from [ABK95] for sorting in the exclusive read PRAM(m) model and this algorithm implies that the concurrent read PRAM(m) is strictly more powerful than the exclusive read PRAM(m).

1.1 An I/O Complexity Application

We first describe an application of our technique in a simplified version of Aggarwal and Vitter's external memory model [AV88]. This application serves as a simple introduction to the new coding technique, and highlights the technique's power. The external memory model represents a computer with a small internal memory, and a larger, but much slower external memory. Each I/O transfers a continuous block of n words between the external memory and the internal memory, and the complexity of a problem is the number of I/Os. For the sake of simplicity, we assume that the internal memory also has size n .

Consider the following problem, which we call *single column transposition*. An $n \times n$ matrix is stored in the external memory in row major order, and T additional external memory locations are used to store additional information. The objective is to minimize the quantity $C = R + \frac{T}{\alpha}$, where α is a parameter of the problem, and R is the maximum over columns i of the number of I/Os required to bring column i into the internal memory. In other words, we want to minimize both the amount of additional memory required and the number of I/Os required to access any column, and α represents the relative importance of the two objectives. For ease of exposition, we here assume that $\alpha = n$; in this case the $\frac{T}{\alpha}$ term contributes the same amount to C as inputting the entire additional external memory.

A natural assumption about the additional external memory is that it can only be used to store duplicate entries of the original matrix. For example, this assumption has been used either explicitly or implicitly for I/O complexity lower bounds in [Flo72], [AV88], [CGG+95], and others. Using this assumption, we give the following easy lower bound for this problem.

Let $M(n)$ be the set of entries of the matrix that are stored in the additional external memory. There must exist some column of the matrix for which less than $\frac{T}{n}$ entries are in $M(n)$. When accessing such a column of the matrix, at least $n - \frac{T}{n}$ entries from that column must be read from the portion of the external memory that stores the matrix in row major order, which requires $n - \frac{T}{n}$ I/Os. Thus, $C \geq n - \frac{T}{n} + \frac{T}{\alpha} = n$.

In this paper, we show that at least for the single column transposition problem, the assumption that the additional external memory only stores duplicates of the original matrix is too limiting. We can instead use each additional external memory location to store the bitwise XORs (exclusive ORs) of carefully selected patterns of matrix entries. This results in an algorithm where $C = \frac{n}{2^{\sqrt{\log n - o(\log n)}}}$. The most interesting portion of the technique is selecting the pattern of matrix entries. The technique is described in section 2, and is the main technical contribution of this paper.

1.2 The PRAM(m) Model

The main application of the new coding technique in this paper is reducing the bandwidth requirements of algorithms in the PRAM(m), a limited bandwidth parallel model introduced by [MNV94]. Low communication throughput has been shown to be a primary bottleneck in parallel computation by several results, including [VW85], [MNV94], [ABK95], and [Goo96]. Thus, in many situations, using the available bandwidth as efficiently as possible is crucial.

In the PRAM(m), p processors communicate only through a small, globally shared memory consisting of m bits. The input, assumed to be of size n , is stored in a globally readable ROM. Therefore, the model focuses on the communication requirements of the actual computation, as opposed to the communication requirements of distributing the input. Processors have unlimited local memory, and at every time step, a processor can read or write one bit from the globally shared memory, access a single location in the ROM containing the input, or perform one step of local computation. Practical considerations make the case where $N \gg P \gg M$ the most interesting.

The PRAM(m) model allows us to study limited bandwidth in a setting that is almost as simple as the traditional PRAM. Ignoring other communication costs, such as message latency, that are incorporated into other, more precise parallel models such as the BSP model [Val90a] and the LogP model [CKP+93] is useful because limited bandwidth is easier to study in an environment that does not complicate the issue with these other costs. Furthermore, these other costs can often be hidden using techniques such as parallel slackness, but no such technique exists for limited bandwidth. For further motivation for the PRAM(m), we refer the reader to [MNV94]. We also point out that by eliminating the globally readable ROM, one can use a variant of the PRAM(m) in limited bandwidth analysis that includes the cost of distributing the input. In this case the input is distributed across the processors and all information processors receive passes through the shared memory cells. This model provides another useful tool for designing algorithms that use bandwidth efficiently.

As with the classical PRAM, two versions of the PRAM(m) have been proposed: the CR (concurrent read) PRAM(m) [MNV94] and the ER (exclusive read) PRAM(m) [ABK95]. For both models, we here assume that writing is exclusive, and that reading from the ROM containing the input may be performed concurrently. The two distinct models capture a fundamental difference in the power of limited bandwidth settings. The ER PRAM(m) models a system where the bandwidth limit is on the total amount of information processors can *receive* per time step: the maximum number of shared memory cells read per time step is m . An example of such a system is a distributed memory parallel machine where processors communicate via point-to-point messages and the bandwidth of the network carrying these messages is limited. Similarly, the CR PRAM(m) represents a system where the bottleneck is on the total amount of information processors can *send* per time

step. An example of such a system is a network of workstations communicating through a broadcast network with low bandwidth relative to the cycle speed of the workstations.

1.3 Summary of New Results

In addition to the solution already mentioned for the I/O complexity single column transposition problem, we use the new coding technique to provide new algorithms for the problems of sorting and permutation routing in the CR PRAM(m). Our main result is an algorithm that, with probability at least $1 - \frac{1}{p^2}$, sorts a random permutation of n arbitrary real numbers in the CR PRAM(m) using time

$$O\left(\frac{n \log p}{\sqrt{m}} \cdot 2^{-\sqrt{\frac{1}{4} \log^2 m + \log p}} + \frac{n \log n}{p}\right),$$

provided that $n \geq mp^{2+\epsilon} \log p$ and $m < p^{1-\epsilon}$, for any constant $\epsilon > 0$. The fastest previous sorting algorithm for the PRAM(m) requires time $\Theta(\frac{n \log n}{m})$; as p and n grow with m held constant, the new algorithm is faster by $\frac{2^{\Omega(\sqrt{\log p})}}{\log p}$. The algorithm is especially noteworthy since all previous sorting algorithms require each key to be referenced in the shared memory at least once, which causes the size of the shared memory, and not the number of processors, to be the limiting factor on the running time of the algorithm. With the new algorithm, on the other hand, distributing the sorting problem across a larger number of processors improves the running time.

The running time of our algorithm also beats a lower bound of $\Omega(\frac{n \log m}{m})$ for the same problem in the ER PRAM(m) given in [ABK95]. Thus, in addition to showing that bound does not apply to the CR PRAM(m), the combination of the two results implies that the CR PRAM(m) is strictly more powerful than the ER PRAM(m). Again, as p and n grow with m held constant, the separation is $\frac{2^{\Omega(\sqrt{\log p})}}{\log p}$.

The algorithm relies on every processor being able to access the shared ROM containing the input, and thus is only directly applicable to sorting in an environment where every processor knows the entire input. An example of such an environment is a shared database stored on multiple processors, where each processor has a copy of the entire database. Our coding technique also serves as a good starting point for designing algorithms that reduce bandwidth requirements in more general concurrent read settings.

We also develop an improved upper bound for sorting in the ER PRAM(m) of $O(\frac{n \log p}{m})$. This is within a constant factor of the lower bound of [ABK95] when $p = m^c$ for any constant c .

The remainder of this paper is organized as follows. The following subsection gives a brief summary of relevant previous work. In section 2, we describe our new coding technique in the context of the single column transposition problem. In section 3, we give an outline of the CR PRAM(m) sorting algorithm. A description of the improved ER PRAM(m) sorting algorithm is given in section 4.

1.4 Previous Work

By using the results of [Sni85], we can show that parallel binary search on n keys requires time $\Omega(\log N)$ in the ER PRAM(m) model and can be performed in time $O(\log_p n)$ in the CR PRAM(m). This also implies a separation between the CR PRAM(m) and the ER PRAM(m). This result, however, is unsatisfying for two reasons. First, the difference is small compared to the complexity of several problems that have been shown to have a running time that is polynomial in n . Second, limited bandwidth has little to do with the running times in these results. The algorithm for the CR PRAM(m) is unchanged by limited bandwidth, and the ER PRAM(m) algorithm achieves within a constant factor of the lower bound for the classical EREW PRAM. Note that the broadcast problem does not imply the separation between the two models due to the presence of the globally readable ROM.

Leighton studies the bandwidth requirements of sorting in [Lei85]. Using Thompson's VLSI model [Tho80], he proves a lower bound of $AT^2 = \Omega(n^2 \log^2 n)$ 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. Leighton also shows that this bound is asymptotically optimal with his column sort algorithm. [ABK95] show that column sort can be used in both the ER and CR PRAM(m) to sort in time $O(\frac{n \log n}{m})$. This was the best known running time for sorting in both models prior to this paper.

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 the CR PRAM(m). They state the exact complexity of sorting as one of the most interesting open problems in the PRAM(m) model. Their result remains the best lower bound for sorting in the CR PRAM(m), and thus there is still a large gap. Other recent work on parallel sorting includes [Goo96], where Goodrich provides an algorithm for the BSP model which sorts using internal computation time $O(\frac{n \log n}{p})$ and $O(\frac{\log n}{\log(n/p+1)})$ communication rounds.

2 Single Column Transposition

We present a solution to the single column transposition problem. Recall that each I/O transfers a continuous block of n words between the external memory and the internal memory, and the internal memory has size $2n$. An $n \times n$ matrix is stored in the external memory in row major order and T additional external memory locations are used to store additional information. The objective is to minimize the quantity $C = R + \frac{T}{n}$, where R is the maximum over columns i of the number of I/Os required to bring column i into the internal memory. In this section, we prove the following theorem.

Theorem 1 *There is a solution to the single column transposition problem where*

$$C = \frac{n}{2\sqrt{\log n - o(\log n)}}.$$

Proof: The remainder of this section is devoted to proving the following technical lemma.

Lemma 1 *For any positive integer $k < \frac{\log n}{2}$, there is a solution to the single column transposition where $T \leq 4\frac{n^2}{2^k}$ and $R \leq 4kn^{1-1/k} + 4\frac{n}{2^k}$.*

Thus, $C \leq 4kn^{1-1/k} + 8\frac{n}{2^k}$, which is minimized when $k = \sqrt{\log n + o(\sqrt{\log n})}$, and theorem 1 follows. ■

Proof: (of lemma 1) The original $n \times n$ matrix is referred to as matrix A , and the additional external memory locations are called the additional memory. We say that a column is *accessed* when it is brought into the internal memory. We here assume that A is a binary matrix, but the results easily extend to any matrix of elements represented by a fixed number of bits.

In our solution, the additional memory stores XORs of entries in A . This allows the entries in columns other than column p to be used to determine the entries in column p . Consider for example the following protocol. The additional memory stores the XOR of the entries along the main diagonal of A : $D = (\sum_{i=1}^n A_{ii}) \bmod 2$. To access any column j , every row of the original matrix is input (transferred from external to internal memory) except row j . This directly determines all of column j except A_{jj} . It also determines all the entries $A_{ii}, i \neq j$, which, when combined with the bit D , determines the value of A_{jj} . Hence, one bit in the additional memory reduces the number of rows of A that are input by 1 regardless of which of the n columns is being brought into the internal memory. Informally, the single bit D provides n bits of utility.

Choosing which entries to XOR is an interesting question. We show that we can use any valid solution to the following combinatorial problem. Place tokens onto the matrix A , where the tokens are grouped together into sets of s tokens, subject to three constraints:

- (1) No two tokens are placed on the same matrix entry.

X	1	X	3
1	X	3	X
X	2	X	4
2	X	4	X

Figure 1: A poor layout

X	1	2	4
1	X	3	5
2	3	X	6
4	5	6	X

Figure 2: The optimal layout for pairs

- (2) No row or column of A has more than one token from the same set.
- (3) If row i and column j each contain a distinct token from the same set S , then no token from **any** set can be placed on matrix entry A_{ij} . We say that A_{ij} is blocked by S .

Given such a placement of tokens, we use the following protocol \mathcal{P} . There is a bit in the additional memory for each set of tokens, where the value of this bit is the XOR of all entries in the matrix covered by the tokens from the set. When determining column j , we input the entire additional memory and all rows i of A such that A_{ij} is not covered by a token.

Claim 1 For any column j , protocol \mathcal{P} inputs sufficient information to the internal memory to reconstruct column j .

Proof: Consider any entry A_{ij} that is not read directly from A by the protocol. Since A_{ij} is not read, it is covered by a token from some set S . To determine the value of A_{ij} when the entire additional memory is known, it is sufficient to know the other $s - 1$ entries in A that are covered by tokens from the set S . By (2), we know that none of these entries can be in row i or in column j . Let r be any row that contains some other token in S . By (3), entry A_{rj} is not covered by any token, and so row r is read. Thus, all entries except A_{ij} covered by a token from set S are read. ■

Let m be the maximum over all columns of the number of entries in the column not covered by a token. Since the additional memory requires approximately $\frac{n^2}{s}$ bits, and the protocol requires us to input m rows of A , the goal is to simultaneously maximize s and minimize m . Note that a set of size s blocks $s^2 - s$ entries of the matrix. Thus, we can easily define a placement of $\frac{n^2}{s^2}$ sets of s tokens in an $n \times n$ matrix, where no two sets block the same entry. For $s = 2$, the 4×4 case is shown in Figure 1, where tokens from the same set are represented by the same digit, and Xs represent blocked entries.

This, however, requires us to input $\frac{n}{2}$ rows even for the case where $s = 2$. To do better, the entries blocked by different sets have to overlap. We start with the case where $s = 2$; we call a set of two tokens a *pair*. The solution is to place pairs so that all the blocked entries are on the main diagonal of the matrix. This is done by placing pairs of the form $(A_{ij}, A_{ji}), i > j$. The pair (A_{ij}, A_{ji}) blocks only the entries A_{ii} and A_{jj} . The 4×4 case is depicted in Figure 2.

Note that this strategy does work for the single column transposition problem. When determining column p , we input row p of A , and this, combined with the additional memory, contains enough information to reconstruct column p of the matrix. This is a protocol that requires only 1 row to be input for determining any column, and only $\frac{n^2-n}{2}$ bits to be stored in the additional memory.

	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">1</td><td style="width: 25%; height: 25%; border: 1px solid black;">2</td><td style="width: 25%; height: 25%; border: 1px solid black;">4</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">1</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">3</td><td style="width: 25%; height: 25%; border: 1px solid black;">5</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">2</td><td style="width: 25%; height: 25%; border: 1px solid black;">3</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">6</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">4</td><td style="width: 25%; height: 25%; border: 1px solid black;">5</td><td style="width: 25%; height: 25%; border: 1px solid black;">6</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td></tr> </table>	×	1	2	4	1	×	3	5	2	3	×	6	4	5	6	×		
×	1	2	4																
1	×	3	5																
2	3	×	6																
4	5	6	×																
	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">1</td><td style="width: 25%; height: 25%; border: 1px solid black;">2</td><td style="width: 25%; height: 25%; border: 1px solid black;">4</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">1</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">3</td><td style="width: 25%; height: 25%; border: 1px solid black;">5</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">2</td><td style="width: 25%; height: 25%; border: 1px solid black;">3</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td><td style="width: 25%; height: 25%; border: 1px solid black;">6</td></tr> <tr><td style="width: 25%; height: 25%; border: 1px solid black;">4</td><td style="width: 25%; height: 25%; border: 1px solid black;">5</td><td style="width: 25%; height: 25%; border: 1px solid black;">6</td><td style="width: 25%; height: 25%; border: 1px solid black;">×</td></tr> </table>	×	1	2	4	1	×	3	5	2	3	×	6	4	5	6	×		
×	1	2	4																
1	×	3	5																
2	3	×	6																
4	5	6	×																

Figure 3: A good layout for $s = 4$

Since every column that contains an entry in any pair must also contain an entry that is blocked by that pair, this construction is optimal for pairs in terms of minimizing the number of rows of the matrix that need to be input. However, the additional memory is still fairly large, and a large number of inputs are required to access the correct bits that appear in the additional memory. In order to do better we use larger values of s . We next consider the case where $s = 4$.

We provide a recursive construction using the placement of pairs. For some $t \ll n$, we partition the matrix A into $(\frac{n}{t})^2$ contiguous $t \times t$ non-overlapping regions called *boxes*. We combine sets of tokens into *neighborhoods*, consisting of $\frac{t^2-t}{2}$ sets of tokens each. All the tokens in a neighborhood are placed into 2 boxes, where each set has 2 tokens in each of the boxes. Within each box, we are left with the recursive problem of placing $\frac{t^2-t}{2}$ pairs into a matrix of size $t \times t$, for which we simply use the solution for pairs described above. We choose the 2 boxes for each neighborhood with the same algorithm: the boxes are treated as entries in an $\frac{n}{t} \times \frac{n}{t}$ matrix, and the two boxes containing entries from a given neighborhood are treated as a pair. The result is that all the blocked entries resulting from tokens that are in different boxes lie in the boxes along the diagonal of the matrix A . In Figure 3, we see an example for the case where $n = 16$, and $t = 4$. In this figure, we refer to the 4×4 sub-matrix containing two tokens from each of sets i through j by $M_{i:j}$.

If we continue this process, recursing k levels results in a construction for $s = 2^k$. To determine how large to make the boxes at the different levels of the recursion, we count the number of blocked entries in each column after k levels of recursion. Let x_1 be the side length of the largest boxes, and in general, for $1 \leq i \leq k-1$, let x_i be the side length of the i^{th} largest boxes. Thus, $n > x_1 > x_2 > \dots > x_{k-1}$.

Each column has x_1 blocked entries in one of the largest boxes. For $2 \leq i \leq k-2$, each column has x_i blocked entries in each of the boxes with side length x_{i-1} , of which there are at most $\frac{n}{x_{i-1}}$. There is 1 blocked entry in each of the $\frac{n}{x_{k-1}}$ smallest boxes. Thus the total number of blocked entries per column is at most

$$x_1 + x_2 \cdot \frac{n}{x_1} + \dots + x_{k-1} \cdot \frac{n}{x_{k-2}} + \frac{n}{x_{k-1}}.$$

This quantity is minimized, subject to $x_i > 0$, when $x_i = n^{1-i/k}$, yielding the value $kn^{1-1/k}$. However, the x_i must be integral and x_i must divide x_{i+1} . To ensure that this is the case, the algorithm is performed on a $2^{\lceil \log n \rceil} \times 2^{\lceil \log n \rceil}$ matrix containing the matrix A in the upper left corner and 0s everywhere else. This increases the number of rows that are input by a factor of at most 2 and the size of the additional memory by no more than a factor of 4. We let the side lengths of the boxes be as follows: $x_i = 2^{\lceil \log n - \frac{i \log n}{k} \rceil}$. With

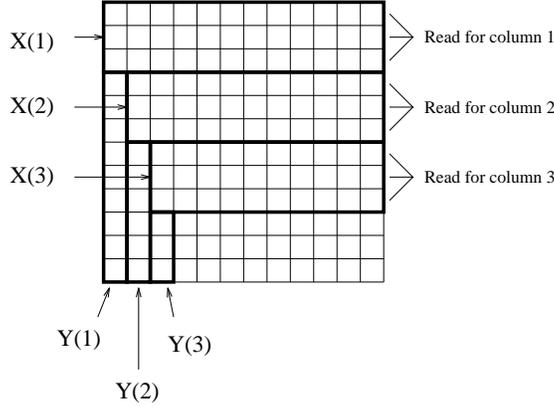


Figure 4: The lower bound argument

these choices of the x_i ,

$$x_i \frac{n}{x_{i-1}} \leq 2n^{1-\frac{i}{k}} \frac{n}{n^{1-\frac{i-1}{k}}}$$

and so this yields a placement of tokens with at most $2kn^{1-1/k}$ blocked entries per column.

In order to ensure that $\frac{x_{i-1}}{x_i} \geq 2$, we require that $\frac{n^{1-\frac{i-1}{k}}}{n^{1-\frac{i}{k}}} \geq 4$, or that $k \leq \frac{\log n}{2}$. This provides a solution to the single column transposition problem where for any parameter $k \leq \frac{\log n}{2}$, the additional memory stores $4\frac{n^2}{2^k}$ bits and any column can be determined by reading at most $4kn^{1-1/k}$ rows of A . The additional memory is read n bits at a time, and so the total number of I/Os is $4kn^{1-1/k} + 4\frac{n}{2^k}$. ■

2.1 A Lower Bound

We provide the following information theoretic lower bound.

Theorem 2 *For the single column transposition problem, $C = \Omega(\sqrt{n})$.*

This is much smaller than the upper bound just described, where C approaches $\frac{n}{2^{\sqrt{\log n}}}$. Closing this gap is one of the most interesting open problems associated with the new coding technique.

Proof: We show that one of the following hold: $R \geq \frac{n}{2}$, $T \geq n^2$, or $RT = \Omega(n^2)$. Let $R(i)$ be the set of rows of A that are input when accessing column i . Let $X(i)$ be the set of bits that are in $R(i)$ and also in columns i through n of the input matrix A . Let $Y(i)$ be the bits in column i of A that are not in $\bigcup_{j=1}^i X(j)$. For $i \geq 1$, $|Y(i)| \geq n - iR$, since no more than R rows are read when accessing any column.

Whenever $T < n^2$, there must be some column i such that at least 1 row is input when accessing column i , and thus we assume that $R > 0$. Let $\mathcal{Y} = \bigcup_{j=1}^{\lceil \frac{n}{R} \rceil} Y(j)$. We have $|\mathcal{Y}| \geq \frac{R \lfloor \frac{n}{R} \rfloor \lfloor \frac{n}{R} - 1 \rfloor}{2}$. Let $\mathcal{X} = \bigcup_{j=1}^{\lceil \frac{n}{R} \rceil} X(j)$. Let $\tilde{\mathcal{X}}$ be the set of bits in the matrix A that are not in \mathcal{Y} . Note that $\mathcal{X} \subset \tilde{\mathcal{X}}$ and that $|\tilde{\mathcal{X}}| \leq n^2 - \frac{R \lfloor \frac{n}{R} \rfloor \lfloor \frac{n}{R} - 1 \rfloor}{2}$. Let \mathcal{Z} be the set of bits stored in the additional memory.

Claim 2 *For any valid protocol, the bits in $\mathcal{X} \cup \mathcal{Z}$ are sufficient to reconstruct all the bits in \mathcal{Y} .*

Proof: We show by induction that for all i , the bits in $\mathcal{Z} \cup \bigcup_{j=1}^i X(j)$ are sufficient to reconstruct $Y(i)$, from which the claim follows. For any correct protocol, the bits in $X(1) \cup \mathcal{Z}$ are sufficient to determine $Y(1)$, and this provides the base case for the induction. By the inductive hypothesis, the bits in $\mathcal{Z} \cup \bigcup_{j=1}^{i-1} X(j)$

are sufficient to reconstruct all of $\bigcup_{j=1}^{i-1} Y(j)$, and thus the bits in $\mathcal{Z} \cup \bigcup_{j=1}^{i-1} X(j) \cup X(i) \cup \mathcal{Z}$ are sufficient to reconstruct all of the rows that are input when accessing column i . For the protocol to be correct, this must be enough information to reconstruct column i , and therefore, the bits in $\mathcal{Z} \cup \bigcup_{j=1}^i X(j)$ are sufficient to reconstruct $Y(i)$. ■

By Claim 2, $\bar{\mathcal{X}} \cup \mathcal{Z}$ is enough information to reconstruct the entire matrix A . Therefore, $|\mathcal{Z}| \geq \frac{R \lfloor \frac{n}{R} \rfloor \lfloor \frac{n}{R} - 1 \rfloor}{2}$, and so $RT \geq \frac{R^2 \lfloor \frac{n}{R} \rfloor \lfloor \frac{n}{R} - 1 \rfloor}{2}$. Whenever $R < \frac{n}{2}$, we have $RT = \Omega(n^2)$. ■

3 Sorting in the CR PRAM(m)

In this section we show how to use the technique presented in Section 2 to develop our main result for the CR PRAM(m). We consider the following natural sorting problem:

Definition 1 The Sorting Problem

Input: n distinct keys $k_1 \dots k_n$, with total order $k_{(1)} < k_{(2)} < \dots < k_{(n)}$, in random order.

Output at processor i : A sorted list of keys:

$$k_{(\frac{i}{p}+1)} \dots k_{(\frac{i}{p} + \frac{n}{p})}$$

Theorem 3 *In the CR PRAM(m) model, there is an algorithm that on a random permutation of n arbitrary keys, sorts these keys with probability at least $1 - \frac{1}{p^2}$, in time*

$$O\left(\frac{n \log p}{\sqrt{m}} \cdot 2^{-\sqrt{\frac{1}{4} \log^2 m + \log p}} + \frac{n \log n}{p}\right),$$

provided that there is a constant $\epsilon > 0$ such that $n \geq mp^{2+\epsilon} \log p$ and $m < p^{1-\epsilon}$.

Proof: We provide such a sorting algorithm. We start with a brief summary of this algorithm. Let an *exact output partition* be a set of keys $y_0 \dots y_p$ such that for $0 \leq i \leq p-1$, the number of keys that lie between y_i and y_{i+1} , is exactly $\frac{n}{p} - 1$. Let an *approximate output partition* be a set of keys $z_0 \dots z_p$ such that for $1 \leq i \leq p-1$, the number of keys that lie between y_i and y_{i+1} , is at most $\frac{n}{p} + o(\frac{n}{p})$. The algorithm first finds and broadcasts an approximate output partition. This is straightforward using standard random sampling techniques. Processor i is then informed of the location in the input ROM of all the keys that lie between y_{i-1} and y_i . This phase of the algorithm is called *permutation routing*, and is the difficult portion of sorting in the PRAM(m) model.

We use the new XOR coding technique to perform permutation routing more efficiently than the obvious algorithm of allocating $\log n$ <time unit, memory cell> pairs to each key. Once permutation routing has been performed, processors locally sort the keys in their range. The algorithm is not done yet, however, since we require each processor to output *exactly* $\frac{n}{p}$ keys, and we started with only an approximate output partition. Thus, each processor i counts the number of keys in the range $[y_{i-1}, y_i]$, and the processors then perform a parallel prefix on these counts. This gives each processor i enough information to determine the exact rank of each key in the range $[y_{i-1}, y_i]$ and so the processors can easily find and broadcast an exact output partition. The permutation routing algorithm is then used again to inform each processor of the location of all the keys that it must output. Finally, these keys are sorted locally at each processor.

It is easily seen that the local sorting steps, the parallel prefix, and the broadcast of p keys can each be performed in time $O(\frac{n \log n}{p} + \frac{p}{m} + p)$. For the range of model parameters in question, this is $O(\frac{n \log n}{p})$. We next show that we can find an approximate output partition efficiently.

Lemma 2 *Given n randomly permuted arbitrary keys, when $n \geq p^2 \log p$ and there is some constant $\epsilon > 0$ such that $m < p^{1-\epsilon}$, there is an algorithm that in time $O(\frac{n \log p}{mp^\epsilon})$, with probability at least $1 - \frac{1}{n}$, finds keys $z_0 \dots z_p$ that form an approximate output partition.*

Proof: The first $p^{3/2}\sqrt{m}\log n$ input keys, denoted \mathcal{K} , are sorted using for example the sorting algorithm of [ABK95]. This requires time $O\left(\frac{p^{3/2}\log n(\log p + \log \log n)}{\sqrt{m}}\right)$, which by the assumptions that $n \geq p^2 \log p$ and $m < p^{1-\epsilon}$ is $O\left(\frac{n \log p}{mp^\epsilon}\right)$. The keys $z_1 \dots z_p$ are then chosen to be the largest of these keys at each processor in the sorted order, and z_0 is the smallest key at processor 1. Let $B_n(a, b)$ be the set of keys k in the n input keys such that $a < k < b$. Let $B_{\mathcal{K}}(a, b)$ be the set of keys k in \mathcal{K} such that $a < k < b$. We want to show that for any z_i , $|B_n(z_i, z_{i+1})| = \frac{n}{p} + o\left(\frac{n}{p}\right)$.

For any pair of keys z_i and z_{i+1} , $B_{\mathcal{K}}(z_i, z_{i+1}) = \sqrt{pm} \log n$. However, if z_i and z_{i+1} are such that $|B_n(z_i, z_{i+1})| \geq \frac{n}{p} + \frac{4n}{p^{5/4}m^{1/4}}$, then in a random permutation of the keys, $E[|B_{\mathcal{K}}(z_i, z_{i+1})|] \geq X = \sqrt{mp} \log n + (pm)^{1/4} \log n$. Using Lemma 5, given in Section A, for any given pair of keys a and b , where $E[|B_{\mathcal{K}}(a, b)|] \geq X$ the probability that $|B_{\mathcal{K}}(a, b)| \leq \sqrt{pm} \log n$ is at most $\frac{1}{n^3}$. Thus, with probability at least $1 - \frac{1}{n}$, no such pair of keys exist. ■

The keys $z_0 \dots z_p$ are broadcast to all the processors. We then perform permutation routing.

Definition 2 : The Permutation Routing Problem

Input: N processor names, each appearing $\frac{N}{p} + o\left(\frac{N}{p}\right)$ times.

Output at processor i : The locations where i appears.

Permutation routing can be expressed using the following matrix representation. The input is an $n \times p$ binary *permutation matrix*, where each row contains one 1 and every column contains at most $\frac{n}{p} + o\left(\frac{n}{p}\right)$ 1s. The rows are randomly permuted. Processor i outputs **column** i of the matrix, and each read of a ROM location reveals one **row** of this matrix. We first show how to solve the closely related matrix transposition problem, where the $p \times p$ input matrix can be any of the 2^{p^2} possible binary matrices.

Definition 3 : The Matrix Transposition Problem

Input: $p \times p$ matrix A , each row stored in one ROM location.

Output at processor i : Column i of A .

For this problem, we assume that m additional processors perform all the writes of the algorithm: one per memory location. When we use the solution to this problem to solve the permutation routing problem, this assumption is removed. For any solution A to the matrix transposition problem, we are concerned with four measures of efficiency.

- $e(A, p)$: the total steps of local computation performed by the m additional processors.
- $f(A, p)$: the total time required for communication.
- $g(A, p)$: the maximum number of rows read by any of the p processors that output columns of the matrix.
- $h(A, p)$: the maximum number of steps of local computation performed by any of the p processors that output columns of the matrix.

The matrix transposition problem is very similar to the single column transposition problem described in Section 2. We show that the solution to the coding problem provided in Section 2 lead to the following lemma.

Lemma 3 For any positive integer $k \leq \frac{\log p}{2}$, there is an algorithm A for the CR PRAM(m) that solves the matrix transposition problem, such that $f(A, p) = O\left(\frac{p^2}{m^{2k}}\right)$, $g(A, p) = O(kp^{1-1/k})$, $e(A, p) = p^2$, and $h(A, p) = O(2^k kp^{1-1/k})$.

Proof: We use the range of solutions to the single column transposition problem described in lemma 1 for a $p \times p$ binary matrix. For all i , processor i reads the rows of the matrix A corresponding to the rows that are read when determining column i in the single column transposition problem. Every bit stored in the additional external memory when solving the single column transposition problem is written by one of the additional m processors to the shared memory of the PRAM(m). Each processor i reads all the bits that are relevant to column i . Since m bits are written at each time step, in the case where $m > 1$, we need to schedule the bits so that no processor is required to read more than one bit that appears in the shared memory at any given time step. This is described in Section 3.2. The bound on $e(A, p)$ follows from the fact that no bit appears in more than 1 XOR. The bound on $h(A, p)$ comes from the fact that each processor needs to reconstruct $kp^{1-1/k}$ bits that are not read directly, and each reconstruction involves taking the XOR of 2^k bits. ■

The solution to the matrix transposition problem is used to solve the permutation routing problem. For any algorithm B for this problem, we are concerned with the following three measures of efficiency.

- $f'(B, p)$: the total time required for communication.
- $g'(B, p)$: the maximum number of rows read by any processor.
- $h'(B, p)$: the maximum number of steps of local computation performed by any processor.

We prove the following lemma in Section 3.1.

Lemma 4 *Given any algorithm A for the CR PRAM(m) that solves the matrix transposition problem, we can produce algorithm B that solves the permutation routing problem on the CR PRAM(m) for all but a fraction of $\frac{1}{2p^2}$ of the possible permutations, provided that there is a constant $\epsilon > 0$ such that $n \geq mp^{2+\epsilon} \log p$. During algorithm B , $f'(B, p) = O(\frac{n \log p}{p^2} f(A, p))$, $g'(B, p) = O(\frac{n}{p} g(A, p))$, and $h'(B, p) = O(\frac{n \log p}{p} g(A, p) + \frac{n \log p}{p^2} h(A, p) + \frac{n \log p}{mp^{2+\epsilon}} \epsilon(A, p) + \frac{n \log p}{mp^\epsilon})$.*

Combining Lemmas 2, 3, and 4, we obtain an algorithm for sorting that runs in time

$$O\left(\frac{n \log p}{m2^k} + \frac{kn \log p}{p^{1/k}} + \frac{n \log n}{p} + \frac{n \log p}{mp^\epsilon}\right).$$

By setting the parameter k as follows,

$$k = \frac{1}{2} \left(\sqrt{\log^2 m + 4 \log p} - \log m \right),$$

we obtain an algorithm with running time

$$O\left(\frac{n \log p}{\sqrt{m}} \cdot 2^{-\sqrt{\frac{1}{4} \log^2 m + \log p}} + \frac{n \log n}{p}\right).$$

Theorem 3 has the following corollary.

Corollary 1 *The CR PRAM(m) is strictly more powerful than the ER PRAM(m). As p and n grow with m held constant, while maintaining the property that*

$$mp^{2+\epsilon} \log p \leq n \leq 2 \left(\frac{p \log p}{m2^{\sqrt{\log p}}} \right),$$

the separation is

$$2^{\Omega(\sqrt{\log p})}.$$

Proof: We see that the upper bound of theorem 3 and the lower bound given in [ABK95] of $\frac{n \log m}{8m}$ imply that the separation is $\Omega\left(\frac{\log m}{\log p} 2^{\sqrt{\frac{1}{4} \log^2 m + \log p} - \frac{1}{2} \log m}\right)$. ■

3.1 Performing Permutation Routing

In this subsection we provide the proof of Lemma 4. Note that using the matrix transposition algorithm to directly perform permutation routing is wasteful of communication throughput since each p bit row of the permutation matrix can be represented by $\log p$ bits. Processors instead perform the matrix transposition algorithm on a smaller *condensed matrix*, which contains the information of the permutation matrix in a more efficient form. This condensed matrix is not stored in the input ROM. Rather, to simulate reading a set of rows in the condensed matrix, processors read a larger set of rows in the permutation matrix stored in the input ROM, and then compute the values of the rows of the condensed matrix.

To obtain the condensed matrix from the permutation matrix, we partition the permutation matrix into *regions* of $p \log p$ consecutive rows. Call a region *full* if there is any column with more than $4 \log p$ 1's in the region. Since the total number of 1's in any column is assumed to be $\frac{n}{p} + o(\frac{n}{p})$, by Lemma 5 given in Section A, the probability that the region is full is at most $\frac{1}{2p^3}$. Thus, the expected number of full regions is at most $\frac{n}{2p^4 \log p}$. By Markov's inequality, with probability at least $1 - \frac{1}{2p^2}$, there are at most $\frac{n}{p^2}$ full regions. We replace each region by a *condensed region*, consisting of $4 \log^2 p$ rows. If a region is not full, then each column in the condensed region describes the location of the 1's in the column in the original region. If a region is full, then the $4 \log^2 p$ rows are used to inform the processors that they must read every row of that region. If there are at most $\frac{n}{p^2}$ such regions, this causes a total of no more than $\frac{n \log p}{p}$ rows to be read by any processor.

To derive the permutation routing algorithm B , we use the given matrix transposition algorithm A on $p \times p$ *blocks*, disjoint subsets of p rows of the condensed matrix. This is sufficient to inform every processor i of column i of the condensed matrix, from which processor i is able to determine the location of all the 1's in column i of the permutation matrix. To compute $4 \log^2 p$ rows of the condensed matrix, a processor reads $p \log p$ rows of the permutation matrix from the input ROM. Thus, to allow processors to read exactly the correct subset of the rows in a block required to perform the matrix transposition algorithm, we need to be careful about which rows of the condensed matrix are placed in the same block. So, we partition the condensed matrix into *sections* of $4p \log^2 p$ rows where each section contains all the rows from p condensed regions $C_1 \dots C_p$. We combine the rows in a section into blocks $L_1 \dots L_{\log^2 p}$, by placing the i^{th} row of the condensed region C_j in the j^{th} row of block L_i . In this way, no two rows of the same condensed region are placed in the same block. Within a section, any given processor always reads the same subset of the rows in each of the blocks $L_1 \dots L_{\log^2 p}$.

To verify that the algorithm B can be performed within the stated bounds, note that the total number of rows read by a single processor within each block is $g(A, p)$. There are a total of $\frac{4n \log p}{p^2}$ blocks in the condensed matrix, so a total of $\frac{4n \log p}{p^2} g(p)$ rows of the condensed matrix need to be read. This corresponds to reading $\frac{n}{p} g(A, p)$ rows of the permutation matrix. Each read only affects $4 \log p$ bits of the condensed matrix, requiring $\frac{4n \log p}{p} g(A, p)$ steps of local computation for converting the permutation matrix into the condensed matrix. In addition, for each of the $\frac{4n \log p}{p^2}$ blocks, $h(A, p)$ steps of local computation are required for the matrix transposition algorithm, resulting in another $O(\frac{n \log p}{p^2} h(A, p))$ steps of local computation. Similarly, the total time for communication used for each block is $f(A, p)$, resulting in a total of $\frac{4n \log p}{p^2} f(A, p)$ time for communication.

In order to actually produce the communication bits described, processors 1 through mp^ϵ , called the *writing processors*, each read a set of $\frac{n}{mp^\epsilon}$ rows of the permutation matrix (stored in the input ROM), and write all the communication bits associated with these rows. These bits are written by the m additional processors in the matrix transposition problem solution. This requires that $\frac{n}{mp^\epsilon}$ is at least one full section, or that $n \geq mp^{2+\epsilon} \log p$.

We next see how much local computation is required of the writing processors. Each row of the ROM effects at most $4 \log p$ bits of the condensed matrix, and thus the number of steps of local computation required for the writing processors to convert the required portion of the permutation matrix into the

condensed matrix is $O(\frac{n \log p}{mp^\epsilon})$. The number of blocks for which each writing processor computes the communication bits is $O(\frac{n \log p}{mp^{2+\epsilon}})$, and thus this portion of the algorithm requires no processor to perform more than $O(\frac{n \log p}{mp^\epsilon} + \frac{n \log p}{mp^{2+\epsilon}} e(A, p))$ steps of local computation.

The writes need to be scheduled so that no processor is ever required to write more than one bit per time step. In order to facilitate this, each writing processor writes each bit into the same memory cell that it is written to in the matrix transposition problem solution, and the bits written to a given memory cell are written in the same order as they are written in the matrix transposition problem solution. All writing processors use the same algorithm to determine what order to write the required bits, and thus, bit $b(i, j, k)$, defined to be the j^{th} bit written by processor i to memory cell k , is read by the same set of processors, regardless of the value of i . Furthermore, this bit is not read by any processor that reads any bit $b(i', j, k')$ for any i' , and for any $k' \neq k$.

We partition the writing processors into *write sets* of size m , where for any processors i_1 and i_2 in the same write set, $\lceil \frac{i_1}{m} \rceil = \lceil \frac{i_2}{m} \rceil$. Each step is allocated to a single write set \mathcal{W} . During the t^{th} step allocated to \mathcal{W} , processor $i \in \mathcal{W}$ writes bit $b(i, \lceil \frac{t}{m} \rceil, i \bmod m)$ to memory cell $i \bmod m$. This ensures that no writing processor is required to write more than one memory cell at any time step, exactly one processor writes to each memory cell at each time step, and no processor is required to read more than one memory cell at any time step.

3.2 Scheduling the Communication Bits

A processor is only able to read one shared memory location at any time step. Thus, during the matrix transposition algorithm, when $m > 1$, we have to schedule the times that the communication bits are written to the shared memory carefully. Each processor i requires the value of each bit written to the shared memory that is the XOR of any set of bits containing some bit in column i . We describe how to construct a *scheduling partition* of the communication bits required for a single $p \times p$ block of the condensed matrix: a partition into sets of bits, where each set contains at least m bits, such that no processor is required to read more than one bit from each set. Given a construction of such a partition, it is easy to schedule the communication bits in such a manner that no processor is required to read more than one bit that appears at any time step.

Our construction of the scheduling partition uses the recursive structure of the placement of tokens. We show how to partition the described placement of pairs in an $n \times n$ matrix into sets of size $\geq \lfloor \frac{n-1}{2} \rfloor$ such that no processor uses more than one pair in any set. If a recursive construction using k levels of recursion has boxes of size $x_1, x_2 \dots x_{k-1}$, then this can be used to obtain a scheduling partition with sets of size

$$\geq \left\lfloor \frac{\frac{p}{x_1} - 1}{2} \right\rfloor \left\lfloor \frac{\frac{x_1}{x_2} - 1}{2} \right\rfloor \left\lfloor \frac{\frac{x_2}{x_3} - 1}{2} \right\rfloor \dots \left\lfloor \frac{x_{k-1}}{2} \right\rfloor \geq \frac{p}{4^k}$$

Thus, each set has size at least m , provided that $p > m2^{2k}$, which holds for any $k = o(\log p)$, provided that there is an $\epsilon > 0$ such that $m < p^{1-\epsilon}$.

We first partition the pairs into sets where set i contains all the pairs with tokens on entries of the form $A_{j(i-j)}$. That is, partition the pairs into sets of pairs that appear on the same diagonal parallel to the skew diagonal of the matrix. No two pairs in the same set have entries in the same column. Note that set i has size $\lfloor \frac{i-1}{2} \rfloor$ for $3 \leq i \leq p+1$, $\lfloor \frac{2p-i+1}{2} \rfloor$ for $p+1 \leq i \leq 2p$ and is empty otherwise. To obtain sets of size $\geq \lfloor \frac{p-1}{2} \rfloor$, we merge sets l and $l+p$, for $3 \leq l \leq p-1$. We obtain a scheduling partition since the largest entry in set l appears in column $l-1$, and the smallest entry in set $l+p$ appears in column l .

This is demonstrated in Figure 5, where $p = 6$, and the entries comprising each set are circled.

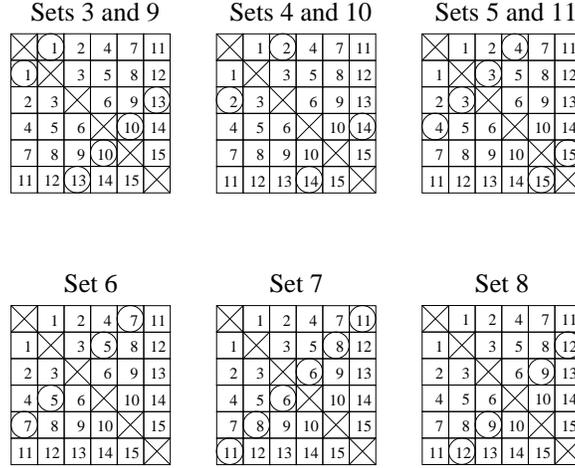


Figure 5: The scheduling partition

4 An Improved Bound for EREW Sorting

Theorem 4 *There is an algorithm that sorts all but $\frac{1}{p^2}$ permutations of n arbitrary real keys in the ER PRAM(m) in time $O(\frac{n \log p}{m})$, provided that $n > p^2 \log p$ and that there is some $\epsilon > 0$ such that $m < p^{1-\epsilon}$.*

Proof: We show here how to perform permutation routing on a random permutation in the stated running time with probability at least $1 - \frac{1}{p^2}$. The remainder of the sorting algorithm is similar to the CR PRAM(m) algorithm, with the additional constraint that broadcasting the p keys that partition the inputs requires time $O(\frac{p^2}{m})$. Motivated by the techniques of Section 3.1, we use the random permutation of the input to perform one transmission using only $O(n \log p)$ bits. We partition the input into $\frac{n}{p \log p}$ equal sized regions. For each region, one processor reads all the rows in that region and informs each processor i where in the region input keys bound for processor i are located. Each key is referred to by its address in its region, requiring only $O(\log p)$ bits.

By Lemma 5 given in Section A, during any transmission, every processor with probability at least $1 - \frac{1}{p^4}$ receives no more than $O(\log p)$ of the inputs in each region, and so, each processor only needs to be allocated $O(\log^2 p)$ bits of shared memory per region. If a region has too many 1's in a particular column, then that processor is informed that it must read every row of that region. By Markov's inequality, with probability at least $1 - \frac{1}{p^2}$, this causes a total of at most $\frac{n}{p}$ rows to be read by any processor. ■

5 Conclusion

We have introduced a new coding technique for making better use of limited bandwidth, and shown that this technique can be used to improve the efficiency of sorting in the PRAM(M) model, as well as the efficiency of a simple problem related to finding the transpose of a matrix in an I/O complexity model.

Open problems include finding further applications for the new technique, such as other problems in the PRAM(M) model and in I/O complexity models. It would also be interesting to show the existence of parallel algorithms that use the new technique for improved bandwidth efficiency but do not rely on the presence of the globally readable ROM containing the input. These algorithms would be useful in the more general situation where every processor does not necessarily know the entire input to start with.

As mentioned in the lower bound of section 2.1, there is a large gap in the upper and lower bounds of the single column transposition problem, and closing this gap is one of the most interesting open problems

related to the new coding technique. For example, it is possible that there exists a better placement of tokens that leads to an improved savings in the amount of additional memory required. However, we conjecture that the lower bound can be improved. On the other hand, it is also interesting to note that the lower bound of section 2.1 implies a lower bound for using the technique described in this paper for sorting in the CR PRAM(M) which is identical to that given in [MNV94]. Related to this open problem is finding the exact complexity of sorting in the CR PRAM(M).

6 Acknowledgments

The author would like to thank Satish Rao for suggesting applying the coding technique to I/O complexity. Also thanks to John Byers, Mike Luby, Rajmohan Rajaraman and Satish Rao for useful comments on previous drafts of this paper.

References

- [ABK95] M. Adler, J. Byers, R. Karp. Parallel Sorting with Limited Bandwidth *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*: pp. 129 - 136, 1995.
- [ACS90] A. Aggarwal, A. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science* 71: pp 3-28, 1990.
- [AV88] A. Aggarwal, J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, pp 116-127, September 1988.
- [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.
- [CGG+95] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, J. Vitter. External-Memory Graph Algorithms *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pp. 139 - 149, 1995.
- [C88] R. Cole. Parallel Merge Sort. *SIAM J. of Computing*, 17(4): pp. 770 - 785, 1988.
- [CKP+93] D. Culler, R. M. Karp, D. Patterson, A. Sahay, K. E. Schauer, 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.
- [Flo72] R. Floyd. Permuting Information in Idealized Two-Level Storage. In *Complexity of Computer Calculations*, R. Miller and J. Thatcher, ed., Plenum, pp. 105 - 109, 1972.
- [Goo96] M. Goodrich. Communication-Efficient Parallel Sorting *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*: pp. 247 - 256, 1996.
- [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.
- [Lei85] T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. on Computers*, c-34(4): pp. 344-354, 1985.

- [LRT93] C. Leiserson, S. Rao, S. Toledo. Efficient Out-Of-Core Algorithms for Linear Relaxation Using Blocking Covers. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pp. 704-713, 1993.
- [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.
- [MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.
- [Rei85] R. Reischuk. Probabilistic Parallel Algorithms For Sorting and Selection. *SIAM Journal on Computing* 14:2 pp 396-409, 1985.
- [Sni85] M. Snir. On Parallel Searching. *SIAM Journal of Computing*: 14: pp. 688-708, 1985.
- [Tho80] C. Thompson. A Complexity Theory for VLSI. *PhD Thesis*. Carnegie-Mellon University, Pittsburgh, PA, 1980.
- [Val90a] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8): pp 103-111, August 1990.
- [Val90b] 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.
- [VS94] J. Vitter, E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12(2/3): pp. 110-147, 1994.

Appendix

A A Technical Lemma

For completeness, we here prove an easy lemma that was used several times in this paper.

Lemma 5 *Let X red balls and $N - X$ blue balls be placed one ball each into N bins, of which Y bins are marked, and let a permutation, chosen uniformly at random from the set of all permutations on N elements, be applied to the balls. Let the random variable Z represent the resulting number of red balls in marked bins. Then,*

$$(1) \Pr \left[Z > (1 + \epsilon) \frac{XY}{N - X} \right] \leq \left[\frac{e^\epsilon}{(1 + \epsilon)^{(1 + \epsilon)}} \right]^{\frac{XY}{N - X}}$$

$$(2) \Pr \left[Z < (1 - \epsilon) \frac{X(Y - \frac{XY}{N})}{N} \right] \leq \exp \left(- \frac{\epsilon^2 X(Y - \frac{XY}{N})}{2N} \right)$$

Proof: For some canonical order on the balls, let $X_i = 1$ if the i^{th} red ball is in a marked bin and let $X_i = 0$ otherwise. After the locations of $i - 1$ balls have been determined, there are still at most Y marked bins left. Thus,

$$\Pr[X_i = 1 \mid X_0 \dots X_{i-1}] \leq \frac{Y}{N - i} \leq \frac{Y}{N - X}.$$

Equation (1) follows from a Chernoff bound (see for example [MR95]).

Let $X'_i = 1$ if either the i^{th} red ball is in a marked bin or at least $\frac{XY}{N}$ of the first $i - 1$ balls are in marked bins and let $X'_i = 0$ otherwise. If $\sum X'_i \geq (1 - \epsilon) \frac{X(Y - \frac{XY}{N})}{N}$, then the number of red balls in marked bins is also at least $(1 - \epsilon) \frac{X(Y - \frac{XY}{N})}{N}$. After the locations of $i - 1$ balls have been determined, either there are at least $Y - \frac{XY}{N}$ empty marked bins where the i^{th} ball could be placed, or there are already $\frac{X(Y - \frac{XY}{N})}{N}$ red balls in marked bins. Thus,

$$\Pr[X'_i = 1 \mid X'_0 \dots X'_{i-1}] \geq \frac{Y - \frac{XY}{N}}{N}.$$

Equation (2) follows from a Chernoff bound. ■