Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors

Lars Arge
MADALGO[*]
University of Aarhus
Aarhus, Denmark
large@daimi.au.dk

Michael T. Goodrich
University of California, Irvine
Irvine, CA 92697, USA
goodrich@ics.uci.edu

Michael Nelson
University of California, Irvine
Irvine, CA 92697, USA
mjnelson@ics.uci.edu

Nodari Sitchinava
University of California, Irvine
Irvine, CA 92697, USA
nodari@ics.uci.edu

ABSTRACT

In this paper, we study parallel algorithms for private-cache chip multiprocessors (CMPs), focusing on methods for foundational problems that can scale to hundreds or even thousands of cores. By focusing on private-cache CMPs, we show that we can design efficient algorithms that need no additional assumptions about the way that cores are interconnected, for we assume that all inter-processor communication occurs through the memory hierarchy. We study several fundamental problems, including prefix sums, selection, and sorting, which often form the building blocks of other parallel algorithms. Indeed, we present two sorting algorithms, a distribution sort and a mergesort. All algorithms in the paper are asymptotically optimal in terms of the parallel cache accesses and space complexity under reasonable assumptions about the relationships between the number of processors, the size of memory, and the size of cache blocks. In addition, we study sorting lower bounds in a computational model, which we call the parallel external-memory (PEM) model, that formalizes the essential properties of our algorithms for private-cache chip multiprocessors.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms

Algorithms

[*]Center for Massive Data Algorithmics – a Center of the Danish National Research Foundation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'08, June 14–16, 2008, Munich, Germany.
Copyright 2008 ACM 978-1-59593-973-9/08/06 ...$5.00.

Keywords

Parallel External Memory, PEM, private-cache CMP, PEM sorting

1. INTRODUCTION

Advances in multi-core architectures are showing great promise at demonstrating the benefits of parallelism at the chip level. Current architectures have 2, 4, or 8 cores on a single die, but industry insiders are predicting orders of magnitude larger numbers of cores in the not too distance future [12, 20, 23]. Such advances naturally imply a number of paradigm shifts, not the least of which is the impact on algorithm design. That is, the coming multicore revolution implies a compelling need for algorithmic techniques that scale to hundreds or even thousands of cores. Parallelism extraction at the compiler level may be able to handle part of this load, but part of the load will also need to be carried by parallel algorithms. This paper is directed at this latter goal.

There is a sizable literature on algorithms for shared-memory parallel models, most notably for variations of the PRAM model (e.g., see [17, 18, 24]). Indeed, some researchers (e.g., see [26]) advocate that PRAM algorithms can be directly implemented in multicores, since separate cores share some levels of the memory hierarchy, e.g. the L2 cache or main memory. After all, an argument can be made that ignoring the memory hierarchy during algorithm design worked reasonably well for the single-processor architectures: in spite of recent developments in the cache-optimal models, most algorithms implemented and used by an average user are designed in the RAM model due to the small size of average input sets and relative simplicity of the RAM algorithms. However, we feel that to take advantage of the parallelism provided by the multicore architectures, problems will have to be partitioned across a large number of processors. Therefore, the latency of the shared memory will have a bigger impact on the overall speed of execution of the algorithms, even if the original problem fits into the memory of a single processor. The PRAM model contains no notion of the memory hierarchy or private memories beyond a few registers in the CPUs. It does not model the differences in the access speeds between the private cache

on each processor and the shared memory that is addressable by all the processors in a system. Therefore, it cannot accurately model the actual execution time of the algorithms on modern multicore architectures.

At the other extreme, the LogP model [9, 19] and bulk-synchronous parallel (BSP) [10, 13, 25] model, assume a parallel architecture where each processor has its own internal memory of size at least N/P. In these models there is no shared memory and the inter-processor communication is assumed to occur via message passing through some interconnection network.

The benefit of utilizing local, faster memory in single-processor architectures was advocated by researchers almost 20 years ago. Aggarwal and Vitter [1] introduced the external memory model (see also [5, 16, 27]), which counts the number of block I/Os to and from external memory. They also describe a parallel disk model (PDM), where D blocks can simultaneously be read or written from/to D different disks, however, they do not consider multiple processors. Nodine and Vitter [22] describe several efficient sorting algorithms for the parallel disk model. Interestingly, Nodine and Vitter [21] also consider a multiprocessor version of the parallel disk model, but not in a way that is appropriate for multicores, since they assume that the processors are interconnected via a PRAM-type network or share the entire internal memory (see, e.g., [28, 29, 30]). Assuming that processors share internal memory does not fit the current practice of multicore. But it does greatly simplify parallel algorithms for problems like sorting, since it allows elements in internal memory to be sorted using well-known parallel sorting algorithm, such as Cole's parallel merge sort [6].

Cormen and Goodrich [8] advocated in a 1996 position paper that there was a need for a bridging model between the external-memory and bulk-synchronous models, but they did not mention anything specific. Dehne *et al.* [11] described a bulk-synchronous parallel external memory model, but in their model the memory hierarchy was private to each processor with no shared memory, while the inter-processor communication was conducted via message passing as in regular BSP model.

More recently, several models have been proposed emphasizing the use of caches of modern CMPs [2, 4, 3]. Bender *et al.* [2] propose a concurrent cache-oblivious model, but focus on the correctness of the data structures rather than efficiency. Blelloch*et al.* [3] (building on the work of Chen *et al.* [4]) consider thread scheduling algorithms with optimal cache utilization for a wide range of divide-and-conquer problems. However, the solutions they consider are of a more moderate level of parallelism.

In contrast to this prior work, in this paper we consider designing massively parallel algorithms (in PRAM sense) in the presence of caches. Indeed, our algorithms are extremely scalable: if we set the cache sizes to be constant or non-existent, our algorithms turn into corresponding PRAM algorithms. At the other extreme, if we have only a single processor, our algorithms turn into solutions for the single-disk, single-processor external-memory model of Aggarwal and Vitter [1].

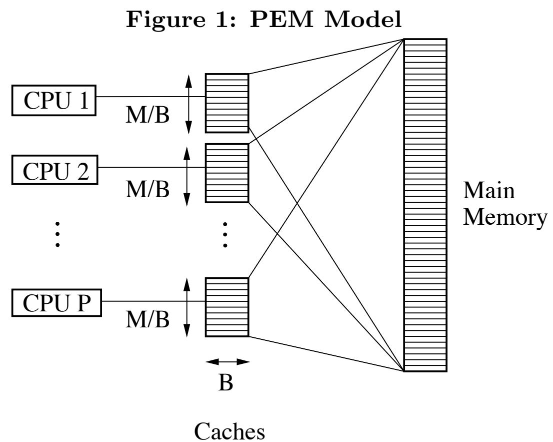
We study a number of fundamental combinatorial problems, which have been studied extensively in other parallel models (so as to establish them accurately being called "fundamental"), as well as the sorting problem. We provide algorithms that have efficient parallel I/O complexity, including two parallel sorting algorithms—a distribution sort and a mergesort. We also provide a sorting lower bound for our model, which shows that our sorting algorithms are optimal for reasonable numbers of processors.

In the following section we formally define the parallel external-memory model. Next we present algorithms for several fundamental building blocks in parallel algorithm design: prefix sums and multi-way selection and partitioning. In section 4 we describe two PEM sorting algorithms: a distribution sort and a mergesort. In section 5 we prove the lower bounds on sorting in the PEM model. We conclude with some discussion and open problems.

2. THE MODEL

The model we propose is a cache-aware version of the model proposed by Bender *et al.* [2]. It can be viewed as a variation of the single-disk, multiple-processor version of the model of Nodine and Vitter [21]. We call it the *parallel external memory (PEM)* model to underline the similarity of the relationship of the PEM model and the single-processor, single-disk *external memory (EM)* model of Aggarwal and Vitter [1] to the PRAM and the RAM models, respectively. The *Parallel External Memory (PEM)* model is a computational model with P processors and a two-level memory hierarchy. The memory hierarchy consists of the external memory (*main memory*) shared by all the processors and P internal memories (*caches*). Each cache is of size M, is partitioned in blocks of size B and is exclusive to each processor, i.e., processors cannot access other processors' caches. To perform any operation on the data, a processor must have the data in its cache. The data is transferred between the main memory and the cache in blocks of size B. (See Figure 1.)

Figure 1: PEM Model



The model's complexity measure, the *I/O complexity*, is the number of *parallel* block transfers between the main memory and the cache by the processors. For example, an algorithm with each of the P processors simultaneously reading one (but potentially different) block from the main memory would have an I/O complexity of $O(1)$, not $O(P)$.

Unlike previous parallel disk models, there is no direct communication network among the processors. All the communication among the processors is conducted through writing and reading to/from the shared memory. Multiple processors can access distinct blocks of the external memory

concurrently. When it comes to accessing the same block of the external memory by multiple processors, just like in the PRAM model, we differentiate the following three variations:

- Concurrent Read, Concurrent Write (CRCW): multiple processors can read and write the same block in the main memory concurrently.
- Concurrent Read, Exclusive Write (CREW): multiple processors can only read the same block concurrently, but cannot write to it.
- Exclusive Read, Exclusive Write (EREW): there is no simultaneous access of any kind to the same block of the main memory by multiple processors.

In the cases of the EREW and CREW simultaneous write to the same block of main memory by $P \leq B$ processors can be handled in various ways. A simple conflict resolution strategy is to serialize the writes resulting in a total of P I/Os. With access to an additional auxiliary block of main memory by each processor, the basic serialization strategy can be improved to $O(\log P)$ parallel I/O complexity. The idea is to gradually combine the data into a single block by the P processors in parallel by scheduling the writes in a binary tree fashion: in the first round half of the processors combine the contents of their memory into a single block by utilizing their auxiliary block, in the second round a quarter of the processors combine the resulting blocks, and so on for $O(\log P)$ rounds. Finally, one of the processors writes out the combined block into the destination address.

For the CRCW PEM model, the concurrent writes of disjoint sets of items of the same block can still be accomplished concurrently. However, during the writes of the same items of a block by different processors, we can use rules corresponding to one of the conflict resolution policies of the analogous CRCW PRAM model.

In this paper, we consider only the CREW PEM model and leave concurrent writes and their implication in the PEM model for future work.

3. THE FUNDAMENTALS

Simulating a PRAM algorithm with the PRAM time complexity $T(N,P)$ on the PEM model results in a trivial PEM algorithm with the I/O complexity $O(T(N,P))$. This is very inefficient in cases when $P < N/B$, where we can get an $O(B)$ factor speedup in the I/O complexity over the simple simulation. However, for some problems a PRAM algorithm is also an efficient PEM algorithm as it is, without any modifications. For example, consider the all prefix sums problem.

Definition 1. *Given an ordered set A of N elements, the* all-prefix-sums *operation returns an ordered set B of N elements, such that $B[i] = \sum_{j=0}^{i} A[j]$, $0 \leq i < N$.*

Theorem 1. *If the input set A is located in contiguous main memory, the all-prefix-sums problem can be solved in the PEM model with the optimal $O(N/PB + \log P)$ PEM I/O complexity.*[1]

[1]For simplicity of exposition by $\log x$ we denote the value $\max\{1, \log x\}$.

Algorithm 1 PEM_MULTI_PARTITION($A[1{:}N], \mathcal{M}, d, P$)

1: **for each** processor i **in parallel do**
2: Read the vector of pivots $\mathcal{M}$ into the cache.
3: Partition S_i into d buckets and let vector $\mathcal{J}_i = \{j_1^i, \ldots, j_d^i\}$ be the number of items in each bucket.
4: **end for**
5: Run PEM Prefix Sums on the set of vectors $\{\mathcal{J}_1, \ldots, \mathcal{J}_P\}$ simultaneously.
6: **for each** processor i **in parallel do**
7: Write elements S_i into memory locations offset appropriately by $\mathcal{J}_{i-1}$ and $\mathcal{J}_i$
8: **end for**
9: Using the prefix sums stored in $\mathcal{J}_P$ the last processor P calculates the vector $\mathcal{B}$ of bucket sizes and returns it.

Proof. The PEM solution is by simulating an optimal PRAM algorithm for $P < N$ processors. The PRAM algorithm consists of four phases: sum N/P elements in parallel, up-sweep, down-sweep, and distribute the results of the down-sweep phase across N/P elements in parallel (see [24] for details). The resulting PRAM complexity is $O(N/P + \log P)$, with the first and the last phases contributing the $O(N/P)$ term, while the up- and down-sweep phases contributing the $O(\log P)$ term.

Since the input set A is in the contiguous memory, the first and the last phases can be accomplished in $O(N/PB)$ I/O complexity in the PEM model. The up-sweep and the down-sweep phases can be simulated in the PEM model with each PRAM memory access counting as a single PEM I/O for a total $O(\log P)$ I/Os. Combining the I/O complexities of the four phases yields the theorem. □

In all the algorithms that follow, we assume that the input is partitioned into $\frac{N}{P}$-sized contiguous segments $S_1, \ldots, S_P$ that each processor primarily works on. We say that processor i is *responsible* for the items in the address space spanned by S_i.

3.1 Multiway Partitioning

Let $\mathcal{M} = \{m_1, \ldots m_{d-1}\}$ be a vector of $d-1$ pivots sorted in increasing order and let A be an unordered set of N elements. The goal of *d-way partitioning* is to partition A into d disjoint subsets, or *buckets*, such that all the elements of the i-th bucket A_i are greater than m_{i-1} and are at most m_i.[2] In addition, we require for the final contents of each bucket to be located in contiguous memory.

Algorithm 1 presents a PEM solution to the multiway partitioning problem. Note that in line 5, P processors run d separate Prefix Sums simultaneously. In particular, each processor i loads the vector $\mathcal{J}_i$ into its cache and the P processors simulate a PRAM all-prefix-sums algorithm on the set $\{j_1^1, \ldots, j_1^P\}$ in $O(\log P)$ I/Os.

Theorem 2. *If the input set A and the set of $d = O(M)$ pivots $\mathcal{M}$ are located in contiguous memory, then the d-way partitioning problem can be solved in the PEM model with $O(N/PB + \lceil d/B \rceil \log P + d \log B)$ I/O complexity.*

Proof. Let's analyze the total I/O complexity of Algorithm 1.

[2]For the first and d-th bucket, we only require that all the elements of A_1 are at most m_1 and all the elements of A_d are greater than m_{d-1}.

Algorithm 2 PEM_SELECT(A[1:N], P, k)

1: **if** $N \leq P$ **then**
2: PRAM_SORT(A, P); **return** $A[k]$.
3: **end if**
4: **for each** processor i **in parallel do**
5: $m_i = \text{SELECT}(S_i, N/2P)$ {Find median of each S_i}
6: **end for**
7: PRAM_SORT$(\{m_1, \dots, m_P\}, P)$ {Sort the medians}
{Partition around the median of medians}
8: $t = \text{PEM_PARTITION}(A, m_{P/2}, P)$
9: **if** $k \leq t$ **then**
10: **return** PEM_SELECT$(A[1:t]$, P, $k)$.
11: **else**
12: **return** PEM_SELECT$(A[t+1:N]$, P, $k-t)$.
13: **end if**

Since the input array A and $\mathcal{M}$ are in the contiguous memory, Lines 1 through 4 run in $O(\lceil N/PB \rceil)$ I/O. In line 5, each step of PRAM simulation, requires reading and writing of vector $\mathcal{J}_i$, which is stored in contiguous memory. Thus, the I/O complexity of line 5 is $O(\lceil d/B \rceil \log P)$.

After prefix sums have been calculated, $\mathcal{J}_{i-1}$ and $\mathcal{J}_i$ define the start and end addresses of where the elements in the buckets should be written. If a full block can be written by a single processor, then the processor writes it out with a single I/O. If the data to be written out into one block is straddled across multiple processors, the processors combine the data into a single block before writing it out as described in Section 2. There are at most $2(d-1)$ blocks in each processor that are not full, thus, line 7 takes at most $O(N/PB + d \log B)$ I/Os. Finally, line 9 takes at most $O(\lceil N/PB \rceil + \lceil d/B \rceil)$ I/Os. The theorem follows. □

3.2 Selection

Let A be an unordered set of size N and let k be an integer, such that $1 \leq k \leq N$. The *selection problem* is to find item $e \in A$ which is larger than exactly $k-1$ elements of A.

Algorithm 2 provides the PEM solution to the selection problem. A few points worth mentioning. In Lines 2 and 7, PRAM_SORT is an optimal $O(\log N)$ PRAM sorting algorithm, e.g. Cole's mergesort [6]. In line 5, SELECT is a single processor cache optimal selection algorithm. And finally, in line 8, PEM_PARTITION is a special case of Algorithm 1 which partitions input A around a single pivot and returns the size of the set of elements which are no larger than the pivot.

THEOREM 3. *If the input set A is located in contiguous memory, the selection problem can be solved in the PEM model with $O(N/PB + \log PB \cdot \log(N/P))$ I/O complexity.*

PROOF. (*sketch*) Let's analyze the I/O complexity of Algorithm 1. Each recursive invocation of PEM_SELECT partitions the input around the median of medians, which eliminates at least a quarter of all elements. The PRAM_SORT is invoked on at most P elements for a total $O(\log P)$ I/Os. By Theorem 2, the I/O complexity of PEM_PARTITION is $O(N/PB + \log P + \log B)$. Thus, the recurrence for the I/O complexity of Algorithm 2 is

$$T(N,P) = \begin{cases} T(\frac{3}{4}N, P) + O\left(\lceil \frac{N}{PB} \rceil + \log PB\right) & \text{if } N > P \\ O(\log P) & \text{if } N \leq P \end{cases}$$

Solving the recurrence yields the theorem. □

Algorithm 3 PEM_DIST_SORT(A[1:N], P)

1: **if** $P = 1$ **then**
2: Sort A using cache-optimal single-processor sorting algorithm
3: Flush the cache of the processor and return
4: **else**
{Sample $\frac{4N}{\sqrt{d}}$ elements}
5: **for each** processor i **in parallel do**
6: **if** $M < |S_i|$ **then**
7: $d = M/B$; Load S_i in M-sized pages and sort each page individually
8: **else**
9: $d = |S_i|$; Load and sort S_i as a single page
10: **end if**
11: Pick every $\sqrt{d}/4$'th element from each sorted memory page and pack into contiguous vector R^i of samples.
12: **end for**
13: **in parallel do:** combine vectors of samples $R^1, \dots R^P$ into a single contiguous vector $\mathcal{R}$ and make $\sqrt{d}$ copies of it: $\mathcal{R}_1, \dots, \mathcal{R}_{\sqrt{d}}$.
14: **for** $j = 1$ to $\sqrt{d}$ **in parallel do** {Find $\sqrt{d}$ pivots}
{Using different sets of $P/\sqrt{d}$ processors}
15: $\mathcal{M}[j] = \text{PEM_SELECT}(\mathcal{R}_j, P/\sqrt{d}, j \cdot 4N/d)$
16: **end for**
17: Pack $\sqrt{d}$ pivots $\mathcal{M}[j]$ into contiguous array $\mathcal{M}$
{Partition A around $\sqrt{d}$ pivots}
18: $\mathcal{B} = \text{PEM_MULTI_PARTITION}(A[1:N], \mathcal{M}, \sqrt{d}, P)$
19: **for** $j = 1$ to $\sqrt{d}+1$ **in parallel do**
20: recursively call PEM_DIST_SORT on bucket j of size $\mathcal{B}[j]$ using a set of $O\left(\left\lceil \frac{\mathcal{B}[j]}{N/P} \right\rceil\right)$ processors responsible for elements of bucket j
21: **end for**
22: **end if**

4. SORTING

In this section we present two sorting algorithms for the PEM model: a distribution sort and a mergesort. Both algorithms have optimal I/O complexity for a reasonable number of processors, as well as optimal (linear) space complexity. While the distribution sort is a much simpler algorithm, the mergesort boasts optimal work complexity speedup and scales better as the number of processors approaches N.

4.1 Distribution Sort

Combining the results of the previous section we can construct a solution for the distribution sort.In the distribution sort, the input is partitioned into d disjoint sets, or *buckets*, of approximately even sizes, each bucket is sorted recursively and concatenated to form a completely sorted set.

The PEM distribution sort is presented in Algorithm 3. It proceeds as follows. First, the algorithm selects $\sqrt{d} = \min\left\{\left\lceil\sqrt{N/P}\right\rceil, \left\lceil\sqrt{M/B}\right\rceil\right\}$ pivots of approximately evenly-spaced rank (lines 5–17). Then, the algorithm partitions the original set around these pivots into $\sqrt{d}+1$ buckets (line 18). Finally, each bucket is recursively sorted in parallel utilizing only processors which are responsible for the elements of that bucket (lines 19–21). In the base case of the recursion, when the bucket size is at most N/P, a single processor sorts it using any of the cache-optimal single processor algorithms.

To find the pivots with optimal parallel I/O complexity, we adapt the approach of Aggarwal and Vitter [1]. In particular, we sample $4N/\sqrt{d}$ items, on which we run the selection algorithm $\sqrt{d}$ times to find $\sqrt{d}$ pivots, which are spaced evenly among the sample items. As the following lemma shows, these pivots provide good enough partitions to achieve logarithmic depth of the recursion.

LEMMA 1. *The size of each partition after a single iteration of PEM_DIST_SORT is at most* $\frac{3}{2}\frac{N}{\sqrt{d}}$.

PROOF. The proof by Aggarwal and Vitter [1] treats each loaded memory page independently, thus, the proof doesn't change if they are loaded and sampled by different processors. The same proof applies to the case when $M < |S_i| = N/P$. □

COROLLARY 1. *The depth of the recursion is* $O(\log_d P)$

Now we can bound the I/O complexity of Algorithm 3.

THEOREM 4. *The I/O complexity of Algorithm 3 is*

$$O\left(\left\lceil \frac{N}{PB} \right\rceil \left(\log_d P + \log_{M/B} \frac{N}{PB}\right) + f(N,P,d) \cdot \log_d P\right)$$

where
$f(N,P,d) = O\left(\log \frac{PB}{\sqrt{d}} \log \frac{N}{P} + \lceil \frac{\sqrt{d}}{B} \rceil \log P + \sqrt{d} \log B\right)$.

PROOF. In the base case, the bucket size is $O(N/P)$, so using any cache-optimal single processor sorting algorithm will take $O\left(\frac{N}{PB} \log_{M/B} \frac{N}{PB}\right)$ I/Os to sort it.

Lines 5 through 12 can be accomplished using simple scans with each processor reading at most $O(N/P)$ items for a total $O(N/PB)$ I/Os. To combine the vectors of samples into a single contiguous vector in line 13, the processors compute prefix sums on the sizes of vectors R^i to determine the addresses of items in the destination vector $\mathcal{R}$. Once the addresses are known, writing out of vector $\mathcal{R}$ can be accomplished by scanning the items in parallel in $O(|\mathcal{R}|/PB + \log B)$ I/Os. Since $|\mathcal{R}| = 4N/\sqrt{d}$, the creation of $\sqrt{d}$ copies of $\mathcal{R}$ takes $O(\lceil N/PB \rceil)$ I/Os. Thus, the total I/O complexity of line 13 is $O(N/PB + \log P + \log B)$. By theorem 3, each PEM_SELECT call takes

$$\begin{aligned} & O\left(\frac{N}{\sqrt{d}} / \left(\frac{P}{\sqrt{d}} B\right) + \log \frac{P}{\sqrt{d}} B \cdot \log \frac{N}{\sqrt{d}} / \frac{P}{\sqrt{d}}\right) \\ = \quad & O\left(\frac{N}{PB} + \log \frac{P}{\sqrt{d}} B \cdot \log \frac{N}{P}\right) \text{ I/Os.} \end{aligned}$$

Line 17 can be accomplished trivially in $O(\sqrt{d})$ I/Os. By Theorem 2, partitioning in line 18 is accomplished in

$$O\left(\frac{N}{PB} + \left\lceil \sqrt{d}/B \right\rceil \log P + \sqrt{d} \log B\right) \text{ I/Os.}$$

Finally, the recursion depth is $O(\log_d P)$. Combining the terms yields the theorem. □

THEOREM 5. *The total memory required by the PEM distribution sorting algorithm is* $O(N)$.

PROOF. The $\sqrt{d}$ copies of the sample array $\mathcal{R}$ of size $4N/\sqrt{d}$ each contribute only a linear increase of memory. All the rest of operations of the algorithm are accomplished using the input memory. □

LEMMA 2. *If* $P \leq \frac{N}{B}$ *and* $M < B^c$ *for some constant* $c > 1$, *then* $\log_{\frac{N}{P}} P = O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$.

PROOF. $\log_{\frac{N}{P}} P \leq \log_B \frac{N}{B} = (c-1) \log_{B^c/B} \frac{N}{B} \leq (c-1) \log_{\frac{M}{B}} \frac{N}{B} = O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$ □

THEOREM 6. *If the number of processors* P *is such that* $f(N,P,d) = O(\lceil N/PB \rceil)$ *and* $M < B^c$ *for some constant* $c > 1$, *then the I/O complexity of the distribution sort is*

$$O\left(\frac{N}{PB} \log_{M/B} \frac{N}{B}\right)$$

.

PROOF. If $f(N,P,d) = O(\lceil N/PB \rceil)$, then the I/O complexity of the distribution sort from Theorem 4 reduces to

$$O\left(\frac{N}{PB}\left[\log_d P + \log_{M/B} \frac{N}{PB}\right]\right) \qquad (1)$$

- $d = M/B$: Equation (1) reduces to $O\left(\frac{N}{PB}\left(\log_{\frac{M}{B}} P + \log_{\frac{M}{B}} \frac{N}{PB}\right)\right) = O\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right)$
- $d = N/P < M$: Equation (1) is bounded by $O\left(\frac{N}{PB}\left(\log_{\frac{N}{P}} P + \log_{\frac{M}{B}} \frac{M}{B}\right)\right) = O\left(\frac{N}{PB} \log_{\frac{N}{P}} P\right)$. And since $1 \leq f(N,P,d) = O(\lceil N/PB \rceil)$ and $M = B^{O(1)}$, the theorem follows from Lemma 2.

□

This bound, as we'll see in Section 5, is optimal to within a constant factor. Note, that the assumption that $M = B^{O(1)}$ accurately represents current hardware trends for caches.

Given particular hardware with the parameters M and B, the requirement $f(N,P,d) = O(N/PB)$ provides a simple equation to evaluate the upper bound on the processors to be used in the distribution sort to achieve the optimal I/O complexity. However, the lack of a simple closed form for the upper bound on P as a function of N, M, and B doesn't provide much insight and intuition for the upper bound on P. In the following section we will present a mergesort algorithm which uses up to N/B^2 processors. This bound on P, as we'll see in Section 5, is within $B/\log B$ of the maximum.

4.2 Mergesort

For simplicity of exposition, throughout this section we utilize at most $P \leq \frac{N}{B^2}$ processors. The algorithm is correct for larger P, but the I/O bounds are not guaranteed. Our sorting procedure is a pipelined d-way mergesort algorithm similar to the sorting algorithm of Goodrich [14], which itself is a BSP adaptation of previous parallel sorting algorithms [6, 15].

A d-way mergesort partitions the input into d subsets, sorts each subset recursively and then merges them. To achieve optimal I/O complexity and parallel speedup, the sorted subsets are sampled and these sample sets are merged first. Each level of the recursion is performed in multiple rounds with each round producing progressively finer samples until eventually a list of samples is the whole sorted subset of the corresponding level of recursion. The samples retain information about the relative order of the other elements of the set through *rankings*. These rankings allow for a quick merge of future finer samples at higher levels of recursion. Each round is pipelined up the recursion tree to maximize parallelism.

4.2.1 Details of the PEM Mergesort algorithm

We set $d = \max\left\{2, \min\left\{\left\lceil\sqrt{\frac{N}{P}}\right\rceil, \left\lceil\frac{M}{B}\right\rceil\right\}\right\}$ and let T be a d-way rooted, complete, balanced tree of depth $O(\log_d P)$. Each leaf is associated with a separate processor and a subset of items of size at most $\lceil\frac{N}{P}\rceil$.

For each node v of T, define $U(v)$ to be a sorted array of items associated with the leaves of the subtree rooted at v. Then the goal of this sorting algorithm is to construct $U(root(T))$. This construction is conducted in a bottom-up, pipelined fashion in stages, where in each stage t we construct a subset $U_t(v) \subseteq U(v)$ of items at each *active* node. A node is said to be *full* in stage t if $U_t(v) = U(v)$. A node is said to be *active* in stage t if $U_t(v) \neq \emptyset$ and the node was not full in stage $t-3$.

Initially, for each leaf v of T, we construct $U(v)$ using an optimal external memory sorting algorithm, e.g., that of Aggarwal and Vitter [1]. This sets each leaf node to be initially full and active while all the internal nodes are empty.

Array B is called a *k-sample* of an array A if B consists of every k-th element of A. Note that if array A is sorted, then so is B. For every active node v of T we define a sample array $L_t(v)$ as follows:

- If v is not full, then $L_t(v)$ is a d^2-sample of $U_t(v)$.
- If v is full, then $L_t(v)$ is a d-sample of $U_t(v)$.

We define $U_t(v) = \bigcup_{i=1}^{d} L_{t-1}(w_i)$, where $\{w_1, w_2, \ldots w_d\}$ are the children of node v.

So far, all the definitions are equivalent to definitions of Goodrich [14], so we reuse the following bounds.

LEMMA 3 ([14]). *If at most k elements of $U_t(v)$ are in an interval $[a,b]$, then at most $dk + 2d^2$ elements of $U_{t+1}(v)$ are in $[a,b]$.*

COROLLARY 2 ([14]). *If at most k elements of $L_t(v)$ are in an interval $[a,b]$, then at most $d(k+1)+2$ elements of $L_{t+1}(v)$ are in $[a,b]$.*

LEMMA 4 ([14]). *The total size of all the $U_t(v)$ and $L_t(v)$ arrays stored at non-full nodes v of T is $O(N/d)$.*

For three items a, b and c, we say that a and c *straddle* [6] b if $a \leq b < c$. Let X and Y be two sorted arrays and let two adjacent items a and c in Y be straddling b, an item in X. Define the *rank* of b in Y, to be the rank of a in Y.[3] Array X is said to be *ranked* [6] into Y, denoted $X \to Y$, if for each element $b \in X$ we know b's rank in Y. If $X \to Y$ and $Y \to X$, we say X and Y are *cross-ranked*, denoted $X \leftrightarrow Y$. We maintain the following induction invariants at the end of each stage t at each node v of T.

Induction Invariants:

1. $L_t(v) \to L_{t-1}(v)$.
2. If v is not full, then $U_t(v) \to L_{t-1}(w_i)$ for each child w_i of v in T.
3. $L_t(v) \to U_t(v)$.

[3]In the exposition below, we assume that all arrays involved in ranking are augmented with dummy nodes $-\infty$ and $+\infty$ as the first and last elements of the of the arrays.

At each stage t, we maintain only arrays $U_t(v)$, $U_{t-1}(v)$, $L_t(v)$, $L_{t-1}(v)$ and $S_{t-1}(v)$ for each active node v of T. The array $S_{t-1}(v)$ is an auxiliary array used during computation of each stage. Each entry $S_{t-1}(v)[i]$ is of size $O(1)$ associated with the item $L_{t-1}(v)[i]$.

We will show that we can compute stage $t+1$ with each processor reading and writing at most $O\left(\frac{N}{PB}\right)$ blocks. To accomplish this we need to maintain the following load-balancing invariant at each node v of T. Let $r = \frac{N}{Pd}$. (Note that $r \geq d$ because $r = \frac{N}{Pd} \geq \frac{N}{P} \cdot \frac{1}{\sqrt{N/P}} = \sqrt{\frac{N}{P}} \geq d$.)

Load-balancing Invariant:

- If array A is not full, then A is partitioned into contiguous segments of size r each, with a different processor responsible for each segment.
- If array A is full, then A is partitioned into contiguous segments of size N/P each, with a different processor responsible for each segment.

Since, by Lemma 4, the total size of all non-full arrays is at most $O(N/d)$, each processor is responsible for $O(N/P)$ items.

LEMMA 5. *If $P \leq \frac{N}{B^2}$ then $d \leq \frac{N}{PB}$.*

PROOF. Note that $P \leq \frac{N}{B^2} \Rightarrow \sqrt{\frac{N}{P}} \geq B$. Then $\frac{N}{PB} = \frac{\sqrt{\frac{N}{P}}\sqrt{\frac{N}{P}}}{B} \geq \frac{d\sqrt{\frac{N}{P}}}{B} \geq d$. □

Computation of Stage $t+1$:

1. Consider a set B_k of items from array $L_t(w_i)$ which are straddled by the k-th and $(k+1)$-th items of the array $L_{t-1}(w_i)$. The processors responsible for the elements of $L_t(w_i)$ can determine the members of B_k using Induction Invariant 1 for each B_k by a simple linear scan of the items of $L_t(w_i)$. The same processors can then calculate the ranges of indices of elements of $L_t(w_i)$ that comprises the set B_k. These ranges are then merged and written by the processors responsible for B_k into $S_{t-1}(w_i)[k]$, i.e., the auxiliary space associated with item $L_{t-1}(w_i)[k]$.
2. For each element $a \in U_t(v)$, using Induction Invariant 2, the processor p responsible for a, can determine a's rank in $L_{t-1}(w_i)$ for each child w_i. Let it be k_i. Then, the processor p can read the item $S_{t-1}(w_i)[k_i]$ of the auxiliary array to get the indices of the elements of $L_t(w_i)$ which are straddled by $L_{t-1}(w_i)[k_i]$ and $L_{t-1}(w_i)[k_i+1]$. Reading the locations of $L_t(w_i)$ indicated by those indices will provide p with the actual values of the items of $U_{t+1}(v)$.
3. Each processor responsible for a contiguous segment $[e,f)$ of $U_t(v)$ reads the items of $L_t(w_i)$ from the previous step and merges them using a simple d-way merge to construct a sublist of $U_{t+1}(v)$ of size $O\left(\frac{N}{P}\right)$. Note, that the rank of an item $g \in L_t(w_i)$ in $U_{t+1}(v)$ is the sum of the ranks of g in $L_t(w_i)$ for all children w_i. Since each processor reads at least one item from each of its children (even if it falls outside of the $[e,f)$ range), we can compute g's rank in $L_t(w_j)$ for all the siblings w_j of w_i during the merge procedure and,

therefore, accurately compute the rank of each item g in $U_{t+1}(v)$. This automatically gives us Induction Invariant 2. Concurrently with the merge procedure, as $U_{t+1}(v)$ grows, the processor reads the segment of $U_t(v)$ that it is responsible for and computes $U_t(v) \to U_{t+1}(v)$. At the same time, using Induction Invariant 3, it also computes $U_{t+1}(v) \to U_t(v)$, resulting in $U_t(v) \leftrightarrow U_{t+1}(v)$. The processor also determines the sample $L_{t+1}(v)$ and ranks it into $U_{t+1}(v)$, giving us Induction Invariant 3. Induction Invariant 1 is achieved by using the cross-rankings of $U_t(v)$ and $U_{t+1}(v)$ to rank $L_{t+1}(v)$ into $L_t(v)$.

4. Flush the data remaining in the cache to main memory and rebalance the responsibilities of each processor to satisfy the load-balancing invariant.

LEMMA 6. *The I/O complexity of computing stage* $(t+1)$ *is* $O(\frac{N}{PB})$.

PROOF. Let's analyze the I/O complexity of each step in the algorithm

Step 1: By Corollary 2, the size of B_k is at most $3d+2$. And since $r \geq d$, at most a constant number of processors are responsible for the items in each set B_k. Thus, the range of ranks of the items of B_k in $L_t(w_i)$ can be computed using a constant number of I/Os by the processors responsible for B_k. Each processor reads and writes at most r items and since all arrays are contiguous and sorted, the total I/O complexity of this step is $O(r/B) = O(N/PB)$.

Step 2: Since the arrays are sorted, the contiguous segments of array $U_t(v)$ correspond to the contiguous segments of arrays $L_{t-1}(w_i)$ and $L_t(w_i)$. Therefore, the I/O access can be accomplished in blocks. By Lemma 3 and the Load-balancing Invariant, the total number of items read by each processor is at most $O(dr + 2d^2 + dB)$ (the last dB term comes from the fact that each processor reads at least one block of data from lists $L_{t-1}(w_i)$ and $L_t(w_i)$ of each of its d children). To calculate the I/O complexity of this step, let's assume that a processor reads r_i items for each child i, i.e. $\sum_{i=1}^{d} r_i \leq O(dr + 2d^2 + dB)$. Then the I/O complexity of this step is

$$\begin{aligned} \sum_{i=1}^{d} \left\lceil \frac{r_i}{B} \right\rceil &\leq \sum_{i=1}^{d} \left(\frac{r_i}{B} + 1 \right) \leq \frac{O(dr + 2d^2 + dB)}{B} + d \\ &\leq O\left(\frac{N}{PB} + d \right) = O\left(\frac{N}{PB} \right). \end{aligned} \quad (2)$$

The third inequality comes from the fact that $d = \min\left\{ \sqrt{\frac{N}{P}}, \frac{M}{B} \right\}$

Step 3: Reading of the contiguous segment of $U_t(v)$ of size r by each processor is the only additional I/O access of this step. Since the segment is contiguous, the additional I/O complexity of this step is $O(r/B) = O(N/PB)$.

Step 4: Rebalancing step can be accomplished in $O(1)$ I/Os, and since each processor reads in at most N/P items in the previous steps, the flushing of the cache and, therefore, this step, can be accomplished in $O(N/PB)$ I/Os.

Since each of the four steps takes $O\left(\frac{N}{PB}\right)$ I/Os, the total I/O complexity of computing stage $t+1$ is $O\left(\frac{N}{PB}\right)$. □

COROLLARY 3. *If* $P \leq N/B^2$, *the I/O complexity of the PEM mergesort algorithm is* $O\left(\frac{N}{PB}\left(\log_{\frac{M}{B}} \frac{N}{PB} + \log_d P\right)\right)$, *where* $d = \min\left\{ \left\lceil \sqrt{\frac{N}{P}} \right\rceil, \left\lceil \frac{M}{B} \right\rceil \right\}$.

PROOF. Our definition of $L_t(v)$ implies that if a node v becomes full in stage t, then v's parent becomes full in stage $t+3$. Therefore, our algorithm has $O(\log_d P)$ stages each of which requires $O\left(\frac{N}{PB}\right)$ I/O accesses. Together with the initial step of sorting the items at the leaves of the tree T, the total I/O complexity of our algorithm is

$$Q(N, P; M, B) = O\left(\frac{N}{PB} \left(\log_{\frac{M}{B}} \frac{N}{PB} + \log_d P \right) \right).$$

□

COROLLARY 4. *If* $P \leq N/B^2$ *and* $M = B^{O(1)}$, *the I/O complexity of the PEM mergesort algorithm is*

$$O\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B} \right)$$

.

PROOF. Consider the two cases:

- $d = M/B < \sqrt{N/P}$:
$Q(N, P; M, B) = O\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{PB}\right) + O\left(\frac{N}{PB} \log_{\frac{M}{B}} P\right)$
$= O\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right)$.
- $d = \sqrt{N/P} < M/B$:
$Q(N, P; M, B) \leq O\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{M^2}{B^3}\right) + O\left(\frac{N}{PB} \log_{\frac{N}{P}} P\right)$
$\leq O(1) + O\left(\frac{N}{PB} \log_{M/B} \frac{N}{B}\right)$

The final inequality follows from the fact that the algorithm utilizes at most $P \leq N/B^2 \leq N/B$ processors and, therefore, Lemma 2 applies. □

This bound, as we'll see in Section 5, is optimal to within a constant factor. In addition, the computational complexity of each processor is within a constant factor of optimal:

LEMMA 7. *The parallel computational complexity of the PEM mergesort algorithm is* $O(\frac{N}{P} \log N)$.

PROOF. The total internal computation time of each processor (together with the initial sorting of the items at the leaves of T) is

$$\begin{aligned} T(N, P) &= O\left(\frac{N}{P} \log \frac{N}{P} \right) + O\left(\frac{N}{P} \log d \cdot \log_d P \right) \\ &= O\left(\frac{N}{P} \log N \right). \end{aligned}$$

□

LEMMA 8. *The total memory required by the mergesort algorithm is* $O(N)$.

PROOF. The total size of all the arrays at full nodes of the tree is $O(N)$. The total size of all the arrays at the non-full nodes, by Lemma 4, is $O(N/d)$. The rankings of Induction Invariant 2 require $O(d)$ memory per item in non-full lists, while other rankings require $O(1)$ memory per item. Thus, the total memory required by our algorithm is $O(N)$. □

This establishes the following result:

Theorem 7. *Given a set S of N items stored contiguously in memory, one can sort S in CREW PEM model using $p \leq \frac{N}{B^2}$ processors each having a private cache of size $M = B^{O(1)}$ in $O\left(\frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B}\right)$ parallel I/O complexity, $O\left(\frac{N}{P}\log N\right)$ internal computational complexity per processor and $O(N)$ total memory.*

5. BOUNDS FOR SORTING ALGORITHMS IN THE PEM MODEL

First we prove the lower bound for the I/O complexity of sorting in the deterministic CREW PEM model.

Theorem 8. *The number of I/O rounds required to sort N numbers in the CREW PEM model is at least*

$$\Omega\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B}\right\} + \log\frac{N}{B}\right)$$

Proof. The proof is presented in Appendix A. □

Theorems 8 implies that our mergesort algorithm from previous section is asymptotically optimal if it utilizes up to N/B^2 processors. Now we prove that using more than $\frac{N}{B\log B}$ processors does not improve the I/O complexity of *any* sorting algorithm in the CREW PEM model, i.e. our mergesort algorithm is factor $B/\log B$ within optimal processor utilization.

The trivial upper bound of N/B for processor utilization follows from the following observation. Each processor needs to read at least one block, and multiple processors reading the same block doesn't improve the I/O complexity of any algorithm. This trivial bound can be improved by studying the *packing problem*:

Packing problem: *Data D of size $n' < N$ is scattered randomly across main memory of size N. The goal is to collects D into contiguous memory.*

Theorem 9. *$P \leq \frac{N}{B\log B}$ is the upper bound on optimal processor utilization to solve the packing problem in the CREW PEM model.*

Proof. Any algorithm has to scan every location of the memory to locate all items of D. With P processors, scanning requires $\Omega(N/PB)$ I/O transfers. Gathering the data scattered across different blocks into contiguous location requires $\Omega(\log B)$ I/Os. Combining both bounds we conclude that to solve the packing problem any algorithm requires $\Omega(N/PB + \log B)$ I/O transfers. The theorem follows. □

The packing problem is reducible to sorting, thus, the I/O complexity of any sorting algorithm does not improve by using more than $\frac{N}{B\log B}$ processors. Therefore, our mergesort algorithm, which uses at most N/B^2 processors, is factor $B/\log B$ within the optimal processor utilization. Reducing the gap remains an open problem.

6. DISCUSSION

In this paper we presented the Parallel External Memory (PEM) model which combines the parallelism of the PRAM model with the I/O efficiency of the external memory model. We presented several algorithms for various fundamental combinatorial problems. Our algorithms build on lessons learned from extensive research in parallel and single-processor I/O-efficient algorithms to provide solutions which are I/O-efficient and scale well with the number of processors.

Having only two levels of hierarchy in the PEM model provides a major limitation for the model. With the increase in the number of processors, the access to the main memory becomes a bottleneck in physical implementation of the model. Thus, to maintain the scalability of physical implementation, the PEM model needs to be extended to support a multi-level hierarchy such that at any particular level, only small-sized subsets of caches are sharing the cache of the higher level. The result would be a tree-like memory hierarchy with a small branching factor. A step in this direction has been taken recently with analysis of a three-level hierarchy with a mix of private and shared caches [3]. However, it remains to be seen if this can be extended to an arbitrary size of the hierarchy and if efficient PRAM-style parallel algorithms can be designed for such a model.

Acknowledgment

We would like to thank Jeff Vitter and Won Chun for several helpful discussions. We would also like to thank anonymous reviewers for helpful comments on the earlier versions of the paper.

7. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM Sympos. Parallel Algorithms Architect.*, pages 228–237, New York, NY, USA, 2005. ACM.

[3] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. 19th ACM-SIAM Sympos. Discrete Algorithms*, 2008.

[4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proc. 19th ACM Sympos. on Parallel Algorithms Architect.*, pages 105–115, New York, NY, USA, 2007. ACM.

[5] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 139–149, 1995.

[6] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[7] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97, 1986.

[8] T. H. Cormen and M. T. Goodrich. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys*, 28A(4), 1996.

[9] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[10] P. de la Torre and C. P. Kruskal. A structural theory of recursively decomposable parallel processor-networks. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 570, Washington, DC, USA, 1995. IEEE Computer Society.

[11] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35(6):567–598, 2002.

[12] D. Geer. Chip Makers Turn to Multicore Processors. *IEEE Computer*, 38(5):11–13, 2005.

[13] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proc. 8th ACM Sympos. Parallel Algorithms Architect.*, pages 223–232, New York, NY, USA, 1996. ACM Press.

[14] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 2000.

[15] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, 43(2):331–361, 1996.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 714–723, 1993.

[17] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Mass., 1992.

[18] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. Elsevier/The MIT Press, Amsterdam, 1990.

[19] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 142–153, New York, NY, USA, 1993. ACM Press.

[20] G. Lowney. Why Intel is designing multi-core processors. `https://conferences.umiacs.umd.edu/paa/lowney.pdf`.

[21] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Sympos. Parallel Algorithms Architect.*, pages 120–129, 1993.

[22] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *J. ACM*, 42(4):919–933, July 1995.

[23] J. Rattner. Multi-Core to the Masses. *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 3–3, 2005.

[24] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[25] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[26] U. Vishkin. A PRAM-on-chip Vision (Invited Abstract). *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, 2000.

[27] J. Vitter. External memory algorithms. *Proceedings of the 6th Annual European Symposium on Algorithms*, pages 1–25, 1998.

[28] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *J. Parallel Distrib. Comput.*, 17:107–114, 1993.

[29] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proc. 22nd Annu. ACM Sympos. Theory Comput.*, pages 159–169, 1990.

[30] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

APPENDIX

A. LOWER BOUNDS FOR SORTING IN THE PEM MODEL

In this appendix we prove Theorem 8. The lower bounds provided in this section hold for the deterministic PEM model. Furthermore, we assume a CREW PEM model, in which only one processor may write to a particular block at one time. We derive a sorting lower bound in the PEM model by first deriving a lower bound on a simpler problem—that of permuting. In the permuting problem, the N keys to be sorted form a permutation of $\{1, 2, ..., N\}$.

THEOREM 10. *The worst-case number of I/O rounds required to permute N distinct keys is*

$$\Omega\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B}\right\}\right)$$

PROOF. Follows from the lower bound of the external memory model of Aggarwal and Vitter [1]. □

The lower bound on the complexity of permuting implies the same lower bound on the complexity of sorting. This bound is a good one for many values of the parameters. However, in some cases we can achieve a better lower bound by looking at the complexity of communicating information between different processors. We again look at a simpler problem first, that of computing the logical "or" of N boolean values.

This problem was thoroughly studied in the PRAM model of computation by Cook *et al.* [7] and in the BSP model by Goodrich [14]. They provide a lower bound of $\Omega(\log N)$ on the complexity of computing the "or" of N boolean values. As it will turn out, their lower bound in the PRAM model can be extended, with slight modifications, to the PEM model. Before stating and proving the theorem, we must introduce some additional terminology and notation, which we take from [7].

Here we model the execution of the PEM system in a slightly different manner. We allow an arbitrarily large set of processors $\Pi = \{P_1, P_2, ...\}$. At each time step, each processor is in one of infinitely many states. This effectively takes the place of the processor's local cache, allowing for

the storage of an unbounded amount of information. For each time step t and processor P_i, let q_i^t denote the state of that processor at time t. In each round, a processor will, depending upon its state, read the contents of a block of main memory, write out to a block of main memory, and change its state for the subsequent round. The only restriction on this behavior is that only one processor may write to a given block during any particular round.

The theorem in [7] applies to more than just computing the *or* function, and we maintain that generality here. In particular, consider any function f which operates on an input string $I = (x_1, x_2, ..., x_n)$ of Boolean values. Let $I(k)$ denote the input string $(x_1, x_2, ..., \overline{x_k}, ..., x_n)$, where $\overline{x_k}$ denotes the complement of Boolean value x_k. I is said to be a *critical input* for a function f if $f(I) \neq f(I(k))$ for all $k \in \{1, 2, ..., n\}$. We will show a lower bound on the time to compute any function which has a critical input. Notice that *or* has a critical input consisting of all 0's.

DEFINITION 2. *An input index i is said to* affect a processor P at time t with I *if the state of P at time t with input I differs from the state of P at t with input $I(i)$. Likewise, an input index i is said to* affect a memory block C at time t with I *if the contents of C at time t with input I differ from the contents of C at time t with input $I(i)$.*

Let $K(P, t, I)$ denote the set of input indices which affect processor P at t with I. Likewise, let $L(C, t, I)$ denote the set of input indices which affect block C at t with I.

Define K_t and L_t to be sequences satisfying the following recurrence equations:

1. $K_0 = 0$
2. $L_0 = B$
3. $K_{t+1} = K_t + L_t$
4. $L_{t+1} = 3K_t + 4L_t$

LEMMA 9. *$|K(P, t, I)| \leq K_t$ and $|L(C, t, I)| \leq L_t$, for all P, C, t, and I.*

PROOF. By induction on t. When $t = 0$ (the base cases), $K(P, t, I)$ is empty, and $L(C, t, I)$ consists of B bits for any input block, and is empty for any other memory block. If, at time $t+1$, processor P reads block C with input I, then $K(P, t+1, I) \subseteq L(C, t, I) \cup K(P, t, I)$, and so $|K(P, t+1, I)| \leq |L(C, t, I)| + |K(P, t, I)|$ which by the induction hypothesis will be at most $L_t + K_t$. The most difficult point arrives in proving that $|L(C, t+1, I)| \leq L_{t+1} (= 3K_t + 4L_t)$.

There are two cases to consider. In the first case, some processor P writes into block C with input I at time $t+1$. In this case, index i affects C at $t+1$ with I only if i affects P at $t+1$ with I. Thus $|L(C, t+1, I)| \leq |K(P, t+1, I)| \leq K_t + L_t < L_{t+1}$.

The second case is that no processor writes into block C at time $t+1$ with input I. In this case, an input index can affect C at $t+1$ in one of only two ways. Recall that a block is affected by an input index if its contents are different with input I than with input $I(i)$. The first way this might happen is if index i affects C at time t. The second way is if some processor does write into block C on input $I(i)$. We say that index i *causes* processor P to write into C at $t+1$ with I if P writes into C with $I(i)$ at time $t+1$. We use $Y(C, t+1, I)$ to denote the set of indices which cause some P to write into M at $t+1$ with I. As previously noted, index i can only affect C at $t+1$ with I if i affects C at t or if i causes some processor to write into M at $t+1$ with I. Hence $L(C, t+1, I) \subseteq L(C, t, I) \cup Y(C, t+1, I)$.

We must now obtain a bound on the size of $Y = Y(C, t+1, I)$. Let $Y = \{i_1, ..., i_r\}$ with $r = |Y|$, and let $P(i_j)$ denote the processor which index i_j causes to write into C with I. Note that some of these processors might be the same for different indices in Y. The following claim provides an important relationship between the bits that cause two processors to write into C and the bits that affect the two processors.

CLAIM 1. *For all pairs $i_j, i_k \in Y$ such that $P(i_j) \neq P(i_k)$, either i_j affects $P(i_k)$ with $I(i_k)$ or i_k affects $P(i_j)$ with $I(i_j)$.*

The claim is true since if neither processor is affected then they will both write into C at time $t+1$ with input $I(i_j)(i_k)$.

Now consider A, the set of all pairs (i_j, i_k) for which i_k affects $P(i_j)$ with $I(i_j)$. We obtain upper and lower bounds on the size of A, which in turn will yield a bound on the value of $r = |Y|$. For each of the r possible values of i_j, at most K_{t+1} of the choices for i_k affect $P(i_j)$ with $I(i_j)$, since $|K(P(i_j), t+1, I(i_j))| \leq K_{t+1}$. Thus $|A| \leq rK_{t+1}$. We now seek a lower bound on the number of pairs (i_j, i_k) for which $P(i_j) \neq P(i_k)$. This will yield a lower bound on $|A|$, since we know, from the above claim, that at least one of (i_j, i_k) or (i_k, i_j) is in A. There are r choices for i_j and for a given i_j there are at least $r - K_{t+1}$ choices for i_k, since at most $|K(P(i_j), t+1, I)| \leq K_{t+1}$ indices can cause $P(i_j)$ to write into C with I. Hence, there are at least $r(r - K_{t+1})$ pairs (i_j, i_k) for which $P(i_j) \neq P(i_k)$ and thus, by our claim, $|A| \geq \frac{1}{2} r(r - K_{t+1})$. We thus have that $\frac{1}{2} r(r - K_{t+1}) \leq |A| \leq rK_{t+1}$, which directly yields that $r \leq 3K_{t+1} = 3K_t + 3L_t$. Recalling that in this second case we have $L(C, t+1, I) \subseteq L(C, t, I) \cup Y(C, t+1, I)$, we have $|L(C, t+1, I)| \leq L_t + |Y| \leq 3K_t + 4L_t = L_{t+1}$. □

THEOREM 11. *Let f be any function on N bits that has a critical input. The number of I/O rounds required to compute f is at least $\log_b(N/B)$, where $b = \frac{1}{2}(5 + \sqrt{21})$.*

PROOF. The theorem follows from Lemma 9 by noting that L_t is bounded from above by Bb^t, where $b = \frac{1}{2}(5+\sqrt{21})$. Since f has a critical input I^*, the block containing the final output, C_O, must be affected by all of the n input bits. Hence, when the algorithm terminates, $|L(C_O, t, I^*)| = n$, so t must be at least $\log_b(N/B)$.

□

Combining the two lower bounds results in Theorem 8.