Cache-Oblivious Priority Queue and Graph Algorithm Applications

Lars Arge[*]
Dept. of Computer Science
Duke University
large@cs.duke.edu

Michael A. Bender[†]
Dept. of Computer Science
SUNY Stony Brook
bender@cs.sunysb.edu

Erik D. Demaine[‡]
Lab. for Computer Science
MIT
edemaine@mit.edu

Bryan Holland-Minkley[‡]
Dept. of Computer Science
Duke University
bhm@cs.duke.edu

J. Ian Munro[§]
Dept. of Computer Science
University of Waterloo
imunro@uwaterloo.ca

ABSTRACT

In this paper we develop an optimal cache-oblivious priority queue data structure, supporting insertion, deletion, and deletemin operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers, where M and B are the memory and block transfer sizes of any two consecutive levels of a multilevel memory hierarchy. In a cache-oblivious data structure, M and B are not used in the description of the structure. The bounds match the bounds of several previously developed external-memory (cache-aware) priority queue data structures, which all rely crucially on knowledge about M and B. Priority queues are a critical component in many of the best known external-memory graph algorithms, and using our cache-oblivious priority queue we develop several cache-oblivious graph algorithms.

[*]Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, and ITR grant EIA–0112849.

[†]Supported in part by the National Science Foundation through ITR grant EIA–0112849 and by HRL Laboratories and Sandia National Laboratories.

[‡]Supported in part by the National Science Foundation through ITR grant EIA–0112849.

[§]Supported in part by the Natural Science and Engineering Council through grant RGPIN 8237-97 and the Canada Research Chair in Algorithm Design.

STOC'02, May 19-21, 2002, Montreal, Quebec, Canada.

1. INTRODUCTION

As the memory systems of modern computers become more complex, it is increasingly important to design algorithms that are sensitive to the structure of memory. One of the essential features of modern memory systems is that they are made up of a hierarchy of several levels of cache, main memory, and disk. While traditional theoretical computational models have assumed a "flat" memory with uniform access time, the access times of different levels of memory can vary by several orders of magnitude in current machines. For example, level-two cache is often around 100 times faster than main memory, while main memory is around 1,000,000 times faster than disks. In order to amortize the large access time of memory levels far away from the processor, memory systems often transfer data between memory levels in large blocks. Thus it is becoming increasingly important to obtain high data locality in memory access patterns.

The standard approach to obtaining good locality is to design algorithms parameterized by several aspects of the memory hierarchy, such as the size of each memory level, and the speed and block sizes of memory transfers between levels. Unfortunately, this parameterization often leads to complex algorithms that are tuned to particular architectures. As a result these algorithms are inflexible and not portable. To avoid the complexity of having too many parameters, a lot of research has been done on simpler *two-level* memory models. Recently, a promising new line of research has aimed at developing memory-hierarchy-sensitive algorithms that avoid any memory-specific parameterization whatsoever. It has been shown that if such a so-called *cache-oblivious* algorithm works optimally on a two-level hierarchy then it works optimally on *all* levels of a multilevel memory hierarchy—cache-oblivious algorithms automatically tune to arbitrary memory architectures. It seems surprising that data locality can be achieved without using the parameters describing the structure of the memory hierarchy, but nevertheless it is possible. Cache-oblivious algorithms have been developed for fundamental problems such as sorting and searching.

In this paper we develop an optimal cache-oblivious priority queue. Previously no such structure was known since known memory hierarchy efficient priority queues all rely crucially on parameters describing the hierarchy. We use the priority queue to develop several cache-oblivious graph algorithms. These are the first such algorithms and the complexity of most of our algorithms match the complexity of the best known two-level cache-aware algorithms.

1.1 Background and previous results

Most algorithmic work has been done in the *Random Access Machine* (RAM) model of computation, which models a "flat" memory with uniform access time. Recently however, some attention has turned to the development of theoretical models for modern complicated hierarchical memory systems—see e.g. [2, 3, 4, 6, 33, 37, 38]. Developing models that are both simple and realistic is a challenging task since a memory hierarchy is described by many parameters. In order to avoid the complications of multilevel memory models, a body of work has focused on two-level memory hierarchies. Most of this work has been done in the context of problems involving massive datasets, because the extremely long access times of disks compared to other levels of the hierarchy means that I/O between main memory and disk is often the bottleneck in such problems.

1.1.1 *I/O model.*

In the two-level *I/O model* (or *external-memory model*) introduced by Aggarwal and Vitter [5], the memory hierarchy consists of an internal memory of size M, and an arbitrarily large external memory partitioned into blocks of size B. The efficiency of an algorithm in this model (a so-called *I/O* or *external-memory* algorithm) is measured in terms of the number of block transfers it performs between these two levels (here called *memory transfers*). The simplicity of the I/O model has resulted in the development of a large number of external memory algorithms and techniques. See e.g. [10, 37] for recent surveys.

The number of memory transfers needed to read N contiguous items from disk is $scan(N) = \Theta(\frac{N}{B})$ (the *linear* or *scanning* bound). Aggarwal and Vitter [5] showed that $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers are necessary and sufficient to sort N elements. In this paper, we use $sort(N)$ to denote $\frac{N}{B}\log_{M/B}\frac{N}{B}$ (the *sorting* bound). The number of memory transfers needed to search for an element among a set of N elements is $\Omega(\log_B N)$ (the *searching* bound) and this bound is matched by the B-tree, which also supports updates in $O(\log_B N)$ memory transfers [12, 23, 27, 26]. An important consequence of these bounds is that, unlike in the RAM model, one cannot sort optimally with a search tree—inserting N elements in a B-tree takes $O(N\log_B N)$ memory transfers which is a factor of $(B\log_B N)/(\log_{M/B}\frac{N}{B})$ from optimal. Finally, permuting N elements according to a given permutation takes $\Theta(\min\{N, sort(N)\})$ memory transfers and for all practical values of N, M and B this is $\Theta(sort(N))$ [5]. This represents another fundamental difference between the RAM and I/O model, since N elements can be permuted in $O(N)$ time in internal memory.

1.1.2 *Cache-oblivious model.*

The main disadvantage of a two-level memory model is that the programmer must focus efforts on a particular level of the hierarchy, resulting in programs that are less flexible to different-scale problems and that do not adapt well when the dominating level changes. Nevertheless, despite the disadvantages of two-level models, the I/O model has been successful because it is convenient for the algorithm designer to consider only two levels of the hierarchy.

Very recently, a new model that combines the simplicity of the two-level models with the realism of more complicated hierarchical models was introduced by Frigo et al. [25]. The idea in the *cache-oblivious model* is to design and analyze algorithms in the I/O model but without using the parameters M and B in the algorithm description. It is assumed that $M > B^2$ (the *tall cache* assumption) and that when an algorithm accesses an element that is not stored in main memory, the relevant block is automatically fetched with a *memory transfer*. If the main memory is full, the *ideal* block in main memory is elected for replacement based on the future characteristics of the algorithm, that is, an *optimal* paging strategy is assumed. While this model may seem unrealistic, Frigo et al. [25] showed that it can be simulated by essentially any memory system with only a small constant-factor overhead. The main advantage of the cache-oblivious model is that it allows us to reason about a simple two-level memory model, but prove results about an unknown, multilevel memory model. Since an analysis of an algorithm in the two-level model holds for any block and main memory size, it holds for *any* level of the memory hierarchy. As a concequence, if the algorithm is optimal in the two-level model, it is optimal on *all* levels of the memory hierarchy.

Frigo et al. [25] developed optimal cache-oblivious algorithms for matrix multiplication, matrix transposition, Fast Fourier Transform, and sorting. Optimal cache-oblivious algorithms have also been found for LU decomposition [15, 35]. Bender et al. [13] and subsequently Brodal et al. [16], Bender et al. [14], and Rahman et al. [32] developed cache-oblivious B-trees with a search cost of $O(\log_B N)$ matching the standard (cache-aware) B-tree. The practical efficiency of the developed algorithms have been investigated in [32, 16, 14].

1.1.3 *Priority queues.*

A priority queue maintains a set of elements each with a priority (or key) under the operations *insert*, *delete*, and *deletemin*, where a deletemin operation finds and deletes the minimum key element in the queue. The heap data structure is a standard implementation of a priority queue and a balanced search tree can of course also easily be used to implement a priority queue. In the I/O model a priority queue based on a B-tree would support all operations in $O(\log_B N)$ memory transfers. The standard heap can also be easily modified (to have fanout B) so that all operations are supported in the same bound (see e.g. [29]). The existence of a cache-oblivious B-tree immediately implies the existence of a $O(\log_B N)$ cache-oblivious priority queue.

As discussed, the use of an $O(\log_B N)$ search tree—or priority queue—to sort N elements results in algorithms that are a factor of $(B\log_B N)/(\log_{M/B}(N/B))$ from optimal. To sort optimally we need a data structure supporting the relevant operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ memory transfers. Note that for reasonable values of N, M, and B, this term is less than 1 and we can therefore only obtain this bound in an amortized sense. To obtain such a bound, Arge developed the *buffer tree technique* [8]. The main idea in this technique is to perform operations in a lazy (or batched)

Problem	Our cache-oblivious result	Previous best cache-aware result	
Priority queue	$O(\frac{1}{B}\log_{M/B}\frac{N}{B})$	$O(\frac{1}{B}\log_{M/B}\frac{N}{B})$	[8]
List ranking	$O(sort(V))$	$O(sort(V))$	[19, 8]
Tree algorithms	$O(sort(V))$	$O(sort(V))$	[19]
Directed BFS and DFS	$O((V+E/B)\log_2 V + sort(E))$	$O((V+E/B)\log_2 V + sort(E))$	[18]
		$O(V+\frac{EV}{BM})$	[19]
Undirected BFS	$O(V+sort(E))$	$O(V+sort(E))$	[30]
Minimal spanning forest	$O(sort(E)\cdot\log_2\log_2 V)$	$O(sort(E)\cdot\log_2\log_2\frac{VB}{E})$	[11]
	$O(V+sort(E))$	$O(v+sort(E))$	[11]

Figure 1: Summary of our results (Priority queue bounds are amortized).

manner using main memory sized buffers attached to the nodes of the data structure. Arge [8] showed how to use the buffer tree technique on a B-tree in order to obtain a priority queue supporting all operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers. However, this structure seems hard to make cache-oblivious since it (apart from the memory sized buffers) involves periodically finding the $\Theta(M)$ smallest key elements in the structure and storing them in internal memory. I/O-efficient priority queues have also been obtained by using the buffer technique on heap structures [24, 28]. Similar to the structure by Arge, the heap structure by Fadel et al. [24] seems hard to make cache-oblivious because it requires collecting $\Theta(M)$ insertions and periodically performing them all on the structure at the same time. The structure by Kumar and Schwabe [28] avoids this by using the buffer technique on a tournament tree data structure. However, their structure still requires memory sized buffers. Apart from this they only obtained $O(\frac{1}{B}\log_2 N)$ bounds, mainly because their structure was designed to also support an update operation. Finally, Brodal and Katajainen [17] developed a priority queue structure based on a M/B-way merging scheme, which also seems hard to make cache-oblivious.

1.1.4 *I/O-efficient graph algorithms.*

The superlinear lower bound on permutation in the I/O model has some consequences for the I/O-complexity of graph algorithms, because the solution of almost any graph problem involves somehow permuting the V vertices or E edges of the graph. Thus $\Omega(\min\{V, sort(V)\})$ is in general a lower bound on the number of memory transfers needed to solve most graph problems. Refer to [9, 19, 30]. As mentioned, this bound is $\Theta(sort(V))$ in all practical cases. Still, even though a large number of I/O-efficient graph algorithms have been developed (see e.g. [37] and references therein), not many algorithms match this bound. Below we review the results most relevant to our work.

Like for PRAM algorithms, list ranking—the problem of ranking the elements in a linked lists stored unordered on disk—is the most fundamental I/O graph problem. Using PRAM techniques, Chiang et al. [19] developed the first I/O-efficient list ranking algorithm. Using an I/O-efficient priority queue, Arge [8] showed how to solve the problem in $O(sort(V))$ memory transfers. The list ranking algorithm and PRAM techniques can be used in the development of $O(sort(V))$ algorithms for most problems on trees, such as computing an Euler Tour, Breadth-First-Search (BFS), Depth-First-Search (DFS), and computing a centroid decomposition [19]. The best known DFS and BFS algorithms for general directed graphs use $O(V+\frac{EV}{BM})$ [19] or $O((V+E/B)\log_2 V + sort(E))$ [18] memory transfers. For undirected graphs, an improved $O(V+sort(E))$ BFS algorithm has been developed [30]. The best known algorithms for computing the connected components and the minimal spanning forest of a general undirected graph both use $O(sort(E)\cdot\log_2\log_2(\frac{VB}{E}))$ or $O(V+sort(E))$ memory transfers [30, 11]. Both these algorithms are improvements of algorithms developed in [1, 19, 28].

1.2 Our results

The main result of this paper is an optimal cache-oblivious priority queue. Our structure supports insert, delete, and deletemin operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log N)$ amortized computation time—it is described in Section 2. As discussed, even though several efficient priority queues were previously known for the I/O model, none of them can readily be made cache-oblivious. One key difficulty in designing a cache-oblivious priority queue is that in order to be optimal the structure has to adapt to arbitrary values of *both* B and M, in contrast to existing cache-oblivious data structures, which only adapt to B [13, 14, 16]. We overcome this difficulty by a novel combination of several new ideas with ideas used in previous recursively defined cache-oblivious algorithms and data structures [25, 13], the buffer technique of Arge [8, 28], and the M/B-way merging scheme utilized by Brodal and Katajainen [17]. We believe that our ideas will be useful in the development of other cache-oblivious data structures, in particular structures that adapt to both B and M.

In the second part of the paper, Section 3, we use the priority queue to develop several cache-oblivious graph algorithms. Previously, no such algorithms were known. We first show how to solve the list ranking problem in $O(sort(V))$ memory transfers. Using this result we develop $O(sort(V))$ algorithms for fundamental problems on trees such as the Euler Tour, BFS, and DFS problems. Using these algorithms and the techniques used in them many other problems on trees can be solved efficiently. The complexity of all of these algorithms matches the complexity of the best known cache-aware algorithms. Next we consider DFS and BFS on general graphs. Using our priority queue and a modified version of a data structure used in the $O((V+E/B)\log V + sort(E))$ DFS and BFS algorithms for directed graphs [18], we make these algorithms cache-oblivious. We also discuss how the best known $O(V + sort(E))$ BFS algorithm for undirected graphs [30] can be made cache-oblivious. Finally, we develop two cache-oblivious algorithms for computing a minimal spanning forest (MSF), and thus also for computing connected components, of an undirected graph using $O(sort(E) \cdot \log_2 \log_2 V)$ and $O(V + sort(E))$ memory transfers, respectively. The two algorithms can be combined to compute the MSF in $O(sort(E) \cdot \log_2 \log_2 \frac{V}{V'} + V')$ memory transfers for any V' independent of B and M. Figure 1 summarizes our results. We believe our priority queue and the developed algorithms can be used in the development of many other cache-oblivious algorithms.

2. PRIORITY QUEUE

In this section we describe our new optimal cache-oblivious priority queue. In Section 2.1 we define the data structure and in Section 2.2 we describe the supported operations.

2.1 Structure

2.1.1 *Levels.*

Our priority queue data structure consists of $\Theta(\log \log N)$ *levels* whose sizes vary from N to a constant size c. The size of a level corresponds (asymptotically) to the number of elements that can be stored within it. The i'th level from above has size $N^{(2/3)^{i-1}}$ and for convenience we refer to the levels by their size. Thus the levels from largest to smallest are level N, level $N^{2/3}$, level $N^{4/9}$, ..., level $X^{9/4}$, level $X^{3/2}$, level X, level $X^{2/3}$, level $X^{4/9}$, ..., level $c^{9/4}$, level $c^{3/2}$, and level c. Intuitively, smaller levels store elements with smaller keys or elements that were more recently inserted. In particular, the minimum key element and the most recently inserted element are always in the smallest (lowest) level c. Both insertions and deletions are initially performed on the smallest level and may propagate up through the levels.

2.1.2 *Buffers.*

Elements are stored in a level in a number of *buffers*, which are also used to transfer elements between levels. Level X consists of one *up buffer* u^X that can store up to X elements, and at most $X^{1/3}$ *down buffers* $d_1^X, \ldots, d_{X^{1/3}}^X$ each containing between $\frac{1}{2}X^{2/3}$ and $2X^{2/3}$ elements. Thus the maximum capacity of level X is $3X$. Refer to Figure 2. Note that the size of a down buffer at one level matches the size (up to a constant factor) of the up buffer one level down.

We maintain three invariants about the relationships between the elements in buffers of various levels:

INVARIANT 1. *At level X, elements are sorted* **among** *the down buffers, that is, elements in d_i^X have smaller keys than elements in d_{i+1}^X, but the elements within d_i^X are unordered.*

The element with largest key in each down buffer d_i^X is called a *pivot element*. Pivot elements simply mark the boundaries between the ranges of the keys of elements in down buffers.

INVARIANT 2. *At level X, the elements in the down buffers have smaller keys than the elements in the up buffer.*

INVARIANT 3. *The elements in the down buffers at level X have smaller keys than the elements in the down buffers at the next higher level $X^{3/2}$.*

The three invariants ensure that the keys of the elements in the down buffers get larger as we go from smaller to larger levels of the structure. Furthermore, an order exists between the buffers on one level: keys of elements in up buffers are larger than keys of elements in down buffers; therefore, down buffers are drawn below up buffers on Figure 2. However, the keys of the elements in an up buffer are unordered relative to the keys of the elements in down buffers one level up. Intuitively, up buffers store elements that are "on their way up", that is, they have yet to be resolved as belonging to a particular down buffer in the next level (or higher levels). Analogously, down buffers store elements that are "on their way down"—these elements are partitioned into several clusters so that we can quickly find the cluster of smallest key elements of size roughly equal to the next level down. In particular, the element with overall smallest key is in the first down buffer at level c.

2.1.3 *Layout.*

We store the priority queue in a linear array as follows. The levels are stored consecutively from smallest to largest. Thus each level occupies a single region of memory. Level X reserves space for exactly $3X$ elements, X for the up buffer and $2X^{2/3}$ for each possible down buffer. The up buffer is stored first, followed by the down buffers stored in an arbitrary order but linked together to form an ordered linked list. Thus $3\sum_{i=0}^{\log_{3/2}\log_c N} N^{(2/3)^i} = O(N)$ is an upper bound on the total size of the array.

LEMMA 1. *The cache-oblivious priority queue uses $O(N)$ space.*

2.2 Operations

To implement the priority queue operations we will use two general operations, *push* and *pull*. Push inserts X elements into level $X^{3/2}$, and pull removes the X elements with smallest keys from level $X^{3/2}$ and returns them in sorted order. This way, deletemin corresponds to pulling an element from the smallest level, and insert corresponds to pushing an element into the smallest level. More generally, whenever an up buffer in level X overflows we push the X elements in the buffer one level up, and whenever the down buffers in level X becomes too empty we pull X elements from one level up.

2.2.1 *Push.*

To insert X elements into level $X^{3/2}$, we first sort the X elements cache-obliviously using $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers [25]. Next we distribute the elements in the sorted list into the $X^{1/2}$ down buffers of level $X^{3/2}$ by scanning through the list and simultaneously visiting the down buffers in (linked) order. More precisely, we append elements to the end of the current down buffer $d_i^{X^{3/2}}$, and advance to the next down buffer $d_{i+1}^{X^{3/2}}$ as soon as we encounter an element with larger key than the pivot of $d_i^{X^{3/2}}$. Elements with keys larger than the pivot of the last down buffer are inserted in the up buffer $u^{X^{3/2}}$. Scanning through X elements take $O(1+X/B)$ memory transfers. Even though we do not scan through every down buffer, we perform at least one memory access for each of the $X^{1/2}$ buffers. Thus the total cost of distributing the X elements is $O(X/B + X^{1/2})$ memory transfers.

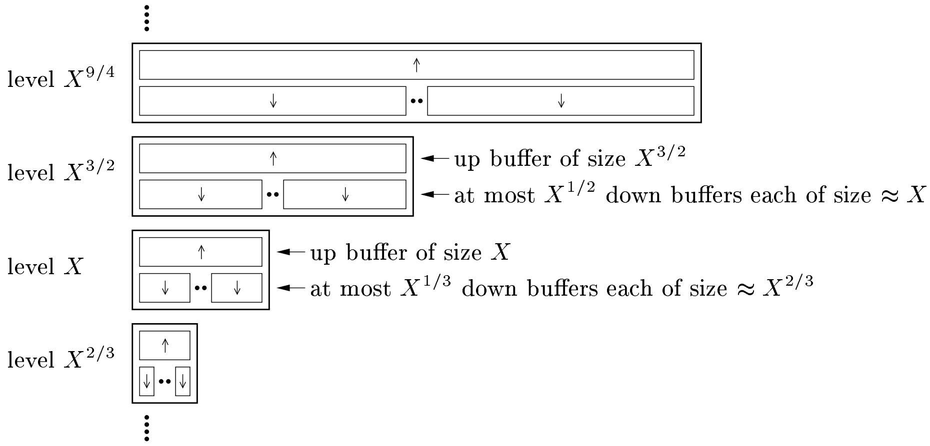

Figure 2: Levels $X^{2/3}$, X, $X^{3/2}$, and $X^{9/4}$ of the priority queue data structure.

During the distribution a down buffer may run full, that is, contain $2X$ elements. In this case, we split the buffer into two down buffers each containing X elements. This split can be performed in $O(1 + X/B)$ memory transfers by first finding the median of the elements in the buffer in $O(1 + X/B)$ transfers [25], and then partitioning the elements into the two new buffers in a simple scan. (Here we are exploiting that the down buffers can be stored out-of-order). Because X elements must have been inserted since the last time the buffer split, the amortized splitting cost per element is $O(1/X + 1/B)$. If the level already had the maximum number $X^{1/2}$ of down buffers before the split, we remove the last down buffer $d^X_{X^{1/3}}$ by inserting the less than $2X$ elements in $d^X_{X^{1/3}}$ into the up buffer. In total, the amortized number of memory transfers used on splitting buffers while distributing the X elements is $O(1+X/B)$. The whole process maintains Invariant 1-3.

If the up buffer runs full during the above process, that is, contains more than $X^{3/2}$ elements, then all of these elements are recursively *pushed* into the next level up. Note that after such a recursive push, $X^{3/2}$ elements have to be inserted (pushed) into the up buffer of level $X^{3/2}$ before another recursive push is needed. Ignoring the cost of this recursion for the moment we have:

LEMMA 2. *A push of X elements from level X up to level $X^{3/2}$ can be performed in $O(X^{1/2} + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers amortized while maintaining Invariants 1–3.*

2.2.2 *Pull.*

To delete the X smallest keys elements from level $X^{3/2}$, first assume that the down buffers contain at least $\frac{3}{2}X$ elements. In this case the first three down buffers $d_1^{X^{3/2}}$, $d_2^{X^{3/2}}$, and $d_3^{X^{3/2}}$ together contain the smallest between $\frac{3}{2}X$ and $6X$ elements (Invariants 1 and 2). We find and remove the X smallest elements simply by sorting these elements using $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. The remaining between $X/2$ and $5X$ elements are left in one, two, or three down buffers of size between $X/2$ and $2X$. These buffers can easily be constructed in $O(1+X/B)$ transfers, and thus this procedure uses $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers in total. It is easy to see that it maintains Invariants 1–3.

In the case where the down buffers contain fewer than $\frac{3}{2}X$ elements, we first *pull* the $X^{3/2}$ elements with smallest keys from the next level up. Because these elements do not necessarily have smaller keys than the, say U, elements in the up buffer $u^{X^{3/2}}$, we first sort this up buffer and merge the two sorted lists. Then we insert the U elements with largest keys into the up buffer, and distribute the remaining between $X^{3/2}$ and $X^{3/2} + \frac{3}{2}X$ elements into $X^{1/2}$ down buffers of size between X and $X + \frac{3}{2}X^{1/2}$ each (such that the $O(1/X + 1/B)$ amortized down buffer split bound is maintained). It is easy to see that this procedure maintains the three invariants. Afterwards, we can find the X minimal key elements as above. Note that after a recursive pull, $X^{3/2}$ elements have to be deleted (pulled) from the down buffers of level $X^{3/2}$ before another recursive pull is needed. Note also that a pull on level $X^{3/2}$ does not affect the number of elements in the up buffer $u^{X^{3/2}}$. Because we fill up the down buffers after a recursive pull using one sort and one scan of $X^{3/2}$ element, this cost is dominated by the cost of the recursive pull operation on the next level up. Ignoring these costs for the moment we have:

LEMMA 3. *A pull of X elements from level $X^{3/2}$ down to level X can be performed in $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers amortized while maintaining Invariants 1–3.*

2.2.3 *Total cost.*

To analyze the amortized cost of an insert or deletemin, we consider the total number of memory transfers used to perform push and pull operations during $N/2$ operations. After every $N/2$ operations we rebuild the structure such that all up buffers are empty and such that level X has $X^{1/3}$ down buffers of size $X^{2/3}$ (except for level N, which has $\Theta(N^{1/3})$ down buffers of size $N^{2/3}$). This way the largest level of the priority queue is always of size $\Theta(N)$, and after the rebuilding X elements have to be pushed into or pulled from level X before recursive push or pulls are needed. We can easily perform the rebuilding in a sorting step using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers, or $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ transfers per operation.

We charge a push of X elements from level X up to level $X^{3/2}$ to level X. Since X elements have to be inserted in the up buffer u^X of level X between such pushes, and as elements can only be inserted in u^X when elements are inserted (pushed) into level X (pulls from level X and from level $X^{3/2}$ into level X do not affect the number of elements in u^X), $O(N/X)$ pushes are charged to level X during the $N/2$ operations. Similarly, we charge a pull of X elements from level $X^{3/2}$ down to level X to level X. Since between such pulls (at least) X elements have to be deleted from the down buffers of level X by pulls on X (push of elements into level X only increase the number of deletions needed and push of elements from level X to level $X^{3/2}$ does not affect the number of elements in down buffers of level X), $O(N/X)$ pulls are charged to level X during the $N/2$ operations.

By Lemma 2 and 3 we know that a push or pull charged to level X uses $O(X^{1/2} + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory accesses. This cost is reduced to $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ when considering a sequence of push and pulls: first consider a push or pull of $X \geq B^2$ elements into or from level $X^{3/2} \geq B^3$. In this case we trivially have that $O(X^{1/2} + \frac{X}{B}\log_{M/B}\frac{X}{B}) = O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. If $B \leq X \leq B^2$, the $X^{1/2}$ term in the push bound can dominate and we have to analyze the cost of a push more carefully. In this case we are working on level $X^{3/2}$ where $B^{3/2} \leq X^{3/2} \leq B^3$. There is a constant number of such levels. Recall that the $X^{1/2}$ cost was from distributing X sorted elements into the less than $X^{1/2}$ down buffers of level $X^{3/2}$. More precisely, a block of each buffer may have to be loaded and written back without transferring a full block of elements into the buffer. However, because $X^{1/2} \leq B$ and $M = \Omega(B^2)$ (the tall-cache assumption), a block for each of the buffers can fit into main memory. In fact, a block of each of the buffers of each of the constant number of levels between $B^{3/2}$ and B^3, as well as all levels less than $B^{3/2}$, fit in memory. Consequently, if a fraction of the main memory is used to keep a partially filled block of each buffer of these levels in memory at all times, and only full block are written to disk, the $X^{1/2}$ term would be eliminated. The optimal paging strategy will do at least as good as this strategy and thus eliminate the $X^{1/2}$ term. (LRU and FIFO strategies achieve the same effect within a constant factor of the value of M [25].) Finally, push and pull operations of $X < B$ elements have no transfer cost at all since the optimal paging strategy can keep all levels less than $B^{3/2}$ in main memory at all times.

Altogether, each of the $O(N/X)$ push and pull operations charged to level X uses $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. Thus the total amortized transfer cost of an insert or deletemin operation in the sequence of $N/2$ such operations is $O(\sum_{i=0}^{\infty}\frac{1}{B}\log_{M/B}(N^{(2/3)^i}/B)) = O(\frac{1}{B}\log_{M/B}\frac{N}{B})$.

We note that a delete operation can be supported in the same bound using ideas from [8, 28]. Basically, to delete an element, we insert a special "delete element" with the relevant key into the priority queue, and the deletion actually occurs when this element arrives in the same buffer as the element to be deleted. Note that the delete operation requires that we know the key of the element to be deleted. Details will appear in the full paper where we will also analyze the (RAM) computation time of each of the operations.

THEOREM 1. *A set of N elements can be maintained in a linear-space cache-oblivious priority queue data structure supporting each insert, deletemin, and delete operation in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $O(\log_2 N)$ amortized computation time.*

3. GRAPH ALGORITHMS

In this section we discuss how our cache-oblivious priority-queue can be used to develop several cache-oblivious graph algorithms. We first consider the simple list ranking problem and algorithms on trees, and then we go on and consider BFS, DFS and minimal spanning tree algorithms.

3.1 List ranking

In the list ranking problem we are given a linked list of V nodes, each with a pointer (edge) to the next node in the list, stored as an unordered sequence. The goal is to determine the *rank* of each node v, that is, the number of edges from v to the end of the list.

Based on ideas from efficient PRAM algorithms [7, 21], Chiang et al. [19] designed an $O(sort(V))$ I/O model list ranking algorithm. The main idea in this algorithm is to find an *independent set* of $\Theta(V)$ nodes (nodes without edges to each other), contract the edges incident to the nodes in this set ("bridge out" the nodes in the independent set), recursively rank the remaining list, and finally reintegrate the contracted nodes in the list (compute their rank). Except for the computation of the independent set, the non-recursive steps of this algorithm can easily be performed using only a constant number of scans and sorts of the unordered list. Thus they can also be performed cache-obliviously in $O(sort(V))$ memory accesses. Details will appear in the full paper. Chiang et al. [19] gave several $O(sort(V))$ algorithms for computing an independent set of size V/c for some constant $c > 0$, resulting in a $T(V) = O(sort(V)) + T(V/c) = O(sort(V))$ list ranking algorithm.

The independent set algorithms of Chiang et al. [19] are based on 3-coloring algorithms. Every node is colored with one of three colors such that adjacent nodes have different colors. The independent set then consists of the set of nodes with the most popular color. Their algorithms required B not being too large or M/B not being too small. These constraints were later removed by Arge [8] and Kumar and Schwabe [28] using I/O-efficient priority queues. The main idea in the resulting 3-coloring algorithm is as follows. An edge (v, w) is called a *forward* edge if v appears before w in the (unordered) sequence of nodes—otherwise it is called a *backward* edge. First the list is split into two sets consisting of forward running segments (forward lists) and backward running segments (backward lists). Each node is included in at least one of these sets and nodes at the head or tail of a segment (nodes at which there is a reversal of the direction) will be in both sets. This step can easily be performed cache-obliviously in a scan of the nodes of the list. Next the nodes in the forward lists are colored red or blue by coloring the head nodes red and the other nodes alternatingly red and blue. The coloring is performed using a priority queue. In a single scan the head vertices are identified, colored, and inserted in a priority queue with keys equal to their position in the unordered list. Next the minimal key node v is repeatedly extracted and v's successor is colored the opposite color of v and inserted in the queue. Since only forward list are colored, the nodes will be colored (accessed) in the order they are stored on disk, that is, in a single scan of the list. Apart from this scan, a total of $O(V)$ operations are performed on the priority queue. Consequently, using our cache-oblivious priority queue, the coloring can be performed cache-obliviously in $O(sort(V))$ memory accesses. After coloring the forward lists red or blue, the nodes in the backward lists are colored green and blue in a similar way, with the head nodes being colored green. In total, every node is colored with one color, except for the heads/tails which have two colors. By coloring a head/tail node red unless it was initially colored blue and green, in which case it is colored green, a 3-coloring is obtained [19].

THEOREM 2. *The list ranking problem on a V node list can be solved cache-obliviously in $O(sort(V))$ memory accesses.*

3.2 Algorithms on trees

Many efficient PRAM algorithms on undirected trees uses Euler Tour techniques [34, 36]. An Euler of a graph is a path that traverses each edge exactly once and forms a cycle. Not every graph has an Euler Tour but a tree where each undirected edge has been replaced with two directed edges does. Using ideas utilized in efficient PRAM algorithms, an Euler Tour of such a modified tree can easily be computed using a scan of the graph followed by a list ranking step. Thus it can be computed cache-obliviously in $O(sort(V))$ memory accesses. Using an Euler Tour, list ranking, and sorting, the BFS and DFS numberings of the nodes of a tree can also easily be computed cache-obliviously in $O(sort(V))$ memory accesses [34, 36, 19]. Another example of a problem that can be solved using these primitives is centroid decomposition. Details will appear in the full paper.

THEOREM 3. *The Euler Tour, BFS, DFS, and centroid decomposition problems on a tree can be solved cache-obliviously in $O(sort(V))$ memory accesses.*

3.3 DFS and BFS

We now consider DFS and BFS algorithms for general graphs. We first consider directed graphs and then we develop an improved algorithm for BFS on undirected graphs.

3.3.1 *Directed graphs.*

For brevity, we only discuss directed DFS. All the discussed algorithms can easily be modified to solve the BFS problem. In the RAM model, directed DFS can be solved in linear time using a stack containing vertices v that have not yet been visited but have an edge (w, v) incident to a visited vertex w. In the I/O model this algorithm may require $\Omega(E)$ memory transfers since one memory transfer may be needed to check if v have already been visited when edge (w, v) is processed. Chiang et al. [19] improved this to $O(V+\frac{E}{B}\frac{V}{M})$ by storing the stack in main memory and periodically removing all edges incident to vertices on the stack. It seems hard to make this algorithm cache-oblivious since it depends crucially on knowledge of the main memory size. Buchsbaum et al. [18] described another $O((V+\frac{E}{B})\log_2 V + sort(E))$ algorithm. This algorithm uses three basic structures: a priority queue, a stack, and a so-called *buffered repository tree*. While a stack is cache-oblivious (*push* or *pop* requires $O(1/B)$ memory accesses amortized), the buffered repository tree need to be modified in order to be cache-oblivious.

A buffered repository tree (BRT) maintains $O(E)$ elements with keys in the range $[1..V]$ under the operations *insert* and *extract*. The insert operation inserts a new element, while the extract operation reports and deletes all elements with a certain key. Our cache-oblivious version of the BRT tree consists of a static binary tree with the keys 1 through V in the leaves. Each node and leaf of the tree has a buffer associated with it. The buffer of a leaf v contains elements with key v and the buffers of the internal nodes are used to perform insertions in a batched manner. An insert is performed by inserting the new element into the root buffer. An extraction of elements with key v is performed by traversing a root-leaf path to the leaf containing v. At each node μ on this path the associated buffer is scanned, the relevant elements reported and deleted, and the remaining elements distributed among the two buffers associated with the children of μ. At the leaf v the elements in the buffer are reported and deleted. When emptying a buffer, an element with key w is distributed to the buffer of the child of μ on the path to w, and we make sure that elements inserted in a buffer at the same time are placed in consecutive memory locations. This way the buffer of a node μ can be viewed as consisting of a linked list of *buckets* of elements in consecutive memory locations, with the number of buckets being equal to the number of buffer emptyings that have been performed on the parent of μ since the last emptying of μ's buffer.

LEMMA 4. *A cache-oblivious BRT supports insert and extract operations in $O(\frac{1}{B}\log_2 V)$ and $O(\log_2 V)$ memory accesses amortized, respectively.*

PROOF. Emptying the buffer of a node (or a leaf) μ containing X elements in K buckets takes $O(1 + X/B + K)$ memory accesses. We charge the X/B-term to the inserts that inserted the X elements. Since each element is charged at most once on each level of the tree, an insert is charged $O(\frac{1}{B}\log_2 V)$ accesses in total. The K-term is charged to the extract operation that created the K buckets. Since an extract creates 2 buckets on each level of the tree, it is charged a total of $O(\log_2 V)$ memory accesses. □

Using our cache-oblivious BRT structure and cache-oblivious priority queue in the algorithm by Buchsbaum et al. [18] we can obtain a cache-oblivious DFS algorithm. A number of data structures are maintained during this algorithm: a stack S containing the vertices on the path from the root of the DFS tree to the current vertex, a priority queue $P(v)$ for each vertex v containing edges (v, w) connecting v with a possibly unvisited vertex w, as well as a global BRT structure D containing edges (v, w) incident to a vertex w that has already been visited but where (v, w) are still present in $P(v)$. The key of an edge (v, w) in D is v. Initially all edges are inserted in the $P(v)$'s. To compute the DFS tree, the vertex u on the top of the stack is repeatedly considered. All edges (u, w) corresponding to u are extracted from D and deleted from $P(u)$. If $P(u)$ is now empty all neighbors of u have been visited and it is removed from the top of the stack. Otherwise a delete is performed on $P(u)$ to obtain a vertex v to visit next. This vertex is pushed on the stack and edges (w, v) incident to it are inserted in D (since v is now visited). Note that the edges are not deleted directly from the relevant $P(w)$'s since that could cost a memory access per edge. Since the algorithm performs $O(V)$ stack operations and extract operations on D, $O(E)$ inserts on D and operations on the priority queues, and uses $O(1)$ extra memory accesses to access a priority queue every time a vertex is processed, it uses $O((V+\frac{E}{B})\log_2 V + sort(E))$ memory accesses in total. Refer to [18] for details.

3.3.2 *Undirected graphs.*

While known I/O model algorithms for directed DFS and BFS take the same number of memory transfers, an improved $O(V + sort(E))$ algorithm has been developed by Munagala and Ranade for undirected BFS [30]. The main idea in this algorithm is to visit the vertices in the graph in "layers" defined by their distance from the root of the BFS tree. Since the graph is undirected, layer $i+1$ vertices can be obtained from two lists of layer i and $i-1$ vertices using a few scan and sort steps. The resulting algorithms, which is easily seen to be cache-oblivious, uses $O(V + sort(E))$ memory accesses. Refer to [30] for details, which will also appear in the full version of this paper.

THEOREM 4. *The DFS or BFS tree of a directed graph can be computed cache-obliviously in $O((V + \frac{E}{B})\log_2 V + sort(E))$ memory accesses. The BFS tree of an undirected graph can be computed cache-obliviously in $O(V + sort(E))$ memory accesses.*

3.4 Minimal Spanning Forest

We now consider algorithms for computing the minimal spanning forest (MSF) of an undirected weighted graph. In the I/O model, a string of algorithms have been developed for the problem, culminating in an algorithm using $O(sort(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ memory transfers developed by Arge et al. [19, 1, 28, 11]. This algorithm consists of two phases [11]. In the first phase an edge contraction algorithm inspired by PRAM MSF algorithms is used [20, 22]. After the number of vertices has been reduced to $O(E/B)$, a modified version of Prim's algorithm is then used in the second phase. Using our cache-oblivious priority queue we can modify both of the phases to work cache obliviously. However, since we cannot decide cache obliviously when the first phase has reduced the number of vertices to $O(E/B)$, we are not able to combine the two phases as effectively as in the I/O model. Below we first sketch how to make the algorithms used in the two phases cache-oblivious, and then we discuss their combination. Details will appear in the full paper.

3.4.1 *Phase 1.*

The basic algorithm based on vertex contraction proceeds in stages [20, 19, 28]. In each stage the minimum weight edge incident to each vertex is selected and output as part of the MSF, and the vertices connected by the selected edges are contracted into super-vertices (that is, the connected components of the graph of selected edges are contracted). This way the number of vertices is reduced to $O(V/2^i)$ after i stages. We can easily select the minimum weight edges cache-obliviously in $O(sort(E))$ memory accesses using a few scans and sorts. To perform the contraction, we select a *leader vertex* in each connected component of the graph of selected edges and replace every edge (u, v) in the graph with the edge $(leader(u), leader(v))$. To select the leaders, we use of the following three properties: the selected edges of a connected component form a tree, exactly one edge in each tree (the minimal weight edge in the tree) is selected twice, and the weights of the edges along a simple path from this edge to a leaf of the tree appear in increasing order. Thus in a single scan of the selected edges we can select the leaders of each component by identifying one of the vertices incident to each of the edges selected twice. We can then use our cache-oblivious tree algorithms developed in Section 3.2 (and thus our cache-oblivious priority queue) to distribute the identity of the leader to each vertex in each component in $O(sort(V))$ memory transfers. Finally, we can easily replace each edge (u, v) with $(leader(u), leader(v))$ cache-obliviously in $O(sort(E))$ memory accesses using a few scanning and sorting steps. In total we perform a stage in $O(sort(E))$ memory accesses and thus to reduce the number of vertices to $V' = V/2^i$ we use $O(sort(E) \cdot \log_2(V/V'))$ memory accesses. By grouping the stages into super-stages as in [11], we can improve this to $O(sort(E) \cdot \log_2 \log_2(V/V'))$. Thus we can obtain an $O(sort(E) \cdot \log_2 \log_2 V)$ MSF algorithm by continuing until all edges have been contracted.

3.4.2 *Phase 2.*

A standard implementation of Prim's MST algorithm [31] uses $\Omega(E)$ memory transfers. Recall that this algorithm grows the MST iteratively from a source node while maintaining a priority queue on the vertices not included in the MST. Arge et al. [11] showed how to implement it to use $O(V + sort(E))$ memory transfers by storing edges in the priority queue instead of vertices. Their algorithm can immediately be made cache-oblivious using our cache-oblivious priority queue.

3.4.3 *Combined algorithm.*

In the I/O model, an $O(sort(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ MSF algorithm can be obtained by running the phase 1 algorithm until the number of vertices have been reduced to $V' = E/B$ using $O((sort(E) \cdot \log_2 \log_2(\frac{VB}{E}))$ memory transfers and then finishing the MSF in $O(V' + sort(E)) = O(sort(E))$ memory transfers using the phase 2 algorithm. As mentioned, we cannot combine the two phases as effectively in the cache-oblivious model. In general however, we can combine the two algorithms to obtain an $O(sort(E) \cdot \log_2 \log_2(V/V') + V')$ algorithm for any V' independent of B and M.

THEOREM 5. *The minimal spanning forest of an undirected weighted graph can be computed cache-obliviously in $O(\min\{V + sort(E)), sort(E) \cdot \log_2 \log_2 V\})$ memory accesses or more generally in $O(sort(E) \cdot \log_2 \log_2(V/V') + V')$ memory accesses for any V' independent of B and M.*

4. CONCLUSIONS

In this paper we presented an optimal cache-oblivious priority queue and used it to develop efficient cache-oblivious algorithms for several graph problems. We believe the ideas utilized in the priority queue will prove useful in the development of other cache-oblivious data structures.

Many important problems still remains open in the area of cache-oblivious algorithms and data structures. In the area of graph algorithms for example, it remains open to develop a cache-oblivious MSF algorithm with complexity matching the best known cache-aware algorithm. Cache-oblivious shortest path algorithms also still have to be developed.

References

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343, 1998.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symp. on Theory of Computation*, pages 305–314, 1987.

[3] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. ACM Symp. on Theory of Computation*, pages 173–185, 1988.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 204–216, 1987.

[5] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[6] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 1994.

[7] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33:269–273, 1990.

[8] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[9] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995.

[10] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2002. (To appear).

[11] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.

[12] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[13] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 339–409, 2000.

[14] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–38, 2002.

[15] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[16] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.

[17] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.

[18] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.

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

[20] F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 1982.

[21] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

[22] R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[23] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[24] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 285–298, 1999.

[26] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[27] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[28] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[29] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, 1, 1996.

[30] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.

[31] R. C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, 36:1389–1401, 1957.

[32] N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *Proc. Workshop on Algorithm Engineering, LNCS 2141*, pages 67–78, 2001.

[33] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281, 1995.

[34] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[35] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[36] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, 1985.

[37] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[38] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.