Alcom-FT Technical Report Series
ALCOMFT-TR-02-136

Funnel Heap - A Cache Oblivious Priority Queue

Gerth Stølting Brodal[*,†] Rolf Fagerberg[*]

Abstract

The cache oblivious model of computation is a two-level memory model with the assumption that the parameters of the model are unknown to the algorithms. A consequence of this assumption is that an algorithm efficient in the cache oblivious model is automatically efficient in a multi-level memory model. Arge et al. recently presented the first optimal cache oblivious priority queue, and demonstrated the importance of this result by providing the first cache oblivious algorithms for graph problems. Their structure uses cache oblivious sorting and selection as subroutines. In this paper, we devise an alternative optimal cache oblivious priority queue based only on binary merging. We also show that our structure can be made adaptive to different usage profiles.

Keywords: Cache oblivious algorithms, priority queues, memory hierarchies, merging

[*]BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf}@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[†]Supported by the Carlsberg Foundation (contract number ANS-0257/20).

1 Introduction

External memory models are formal models for analyzing the impact of the memory access patterns of algorithms in the presence of several levels of memory and caches on modern computer architectures. The cache oblivious model, recently introduced by Frigo et al. [10], is based on the I/O model of Aggarwal and Vitter [1], which has been the most widely used model for developing external memory algorithms; see the survey by Vitter [11]. Both models assume a two-level memory hierarchy where the lower level has size M and data is transfered between the two levels in blocks of B elements. The difference is that in the I/O model the algorithms are aware of B and M, whereas in the cache oblivious model these parameters are unknown to the algorithms and I/Os are handled automatically by an optimal off-line cache replacement strategy.

Frigo et al. [10] showed that an efficient algorithm in the cache oblivious model is automatically efficient on an each level of a multi-level memory model. They also presented optimal cache oblivious algorithms for matrix transposition, FFT, and sorting. Cache oblivious search trees which match the search cost of the standard (cache aware) B-trees [3] were presented in [4, 5, 6, 8]. Cache oblivious algorithms for computational geometry problems were developed in [4, 7]. The first cache oblivious priority queue was recently developed by Arge et al. [2], and gave rise to several cache oblivious graph algorithms [2]. Their structure uses existing cache oblivious sorting and selection algorithms as subroutines.

In this paper, we present an alternative optimal cache oblivious priority queue, Funnel Heap, based only on binary merging. Essentially, our structure is a single heap-ordered tree with binary mergers in the nodes and buffers on the edges. It was inspired by the cache oblivious merge-sort algorithm Funnelsort presented in [10] and simplified in [7]. As for the priority queue of Arge et al., our data structure supports the operations Insert and DeleteMin using amortized $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os per operation, under the so-called tall cache assumption $M \geq B^2$. Here, N is the total number of elements inserted.

For a slightly different algorithm we give a refined analysis, showing that the priority queue adapts to different profiles of usage. More precisely, we show that the ith insertion uses amortized $O(\frac{1}{B}\log_{M/B}\frac{N_i}{B})$ I/Os, with N_i defined in one of the following three ways: (a) the number of elements present in the priority queue when performing the ith insert operation, (b) if the ith inserted element is removed by a DeleteMin operation prior to the jth insertion then $N_i = j - i$, or (c) N_i is the maximum rank that the ith inserted element has during its lifetime in the priority queue, where rank denotes the number of smaller elements present in the priority queue. DeleteMin is amortized for free since the work is charged to the insertions. These results extends the line of research taken in [9], where (a) and (c) are called size profile and max depth profile, respectively.

This paper is organized as follows. In Section 2 we introduce the concept of mergers and in Section 3 we describe our priority queue. Section 4 gives the analysis of the presented data structure. Finally, Section 5 gives the analysis based on different profiles of usage.

2 Mergers

Our data structure is based on *binary mergers*. A binary merger takes as input two sorted streams of elements and delivers as output the sorted stream formed by merging of these. One merge step moves an element from the head of one of the input streams to the tail of the output stream. The heads of the input streams and the tail of the output stream reside in *buffers* holding a limited number of elements. A buffer is simply an array of elements, plus fields storing the capacity of the buffer and pointers to the first and last elements in the buffer.

Binary mergers may be combined to *binary merge trees* by letting the output buffer of one merger be an input buffer of another—in other words, binary merge trees are binary trees with mergers at the nodes and buffers at the edges. The leaves of the tree contains the streams to be merged.

```
Procedure Fill(v)
    while v's output buffer is not full
        if left input buffer empty
            Fill(left child of v)
        if right input buffer empty
            Fill(right child of v)
        perform one merge step
```

Figure 1: The merging algorithm

An *invocation* of a merger is a recursive procedure which performs merge steps until its output buffer is full (or both input streams are exhausted).

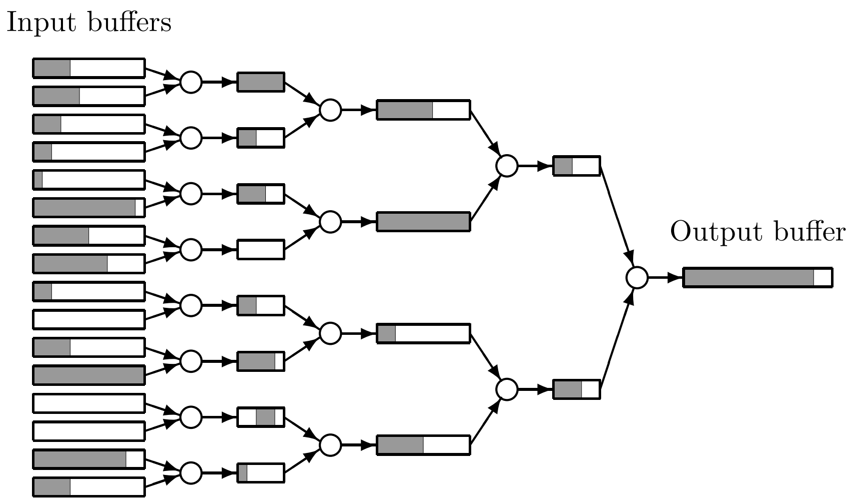

Figure 2: A 16-merger consisting of 15 binary mergers. Shaded regions are the occupied parts of the buffers.

If during the invocation an input buffer gets empty (but the corresponding stream is not exhausted), the input buffer is recursively filled by an invocation of the merger having this buffer as its output buffer. If both input streams of a merger get exhausted, the corresponding output stream is marked as exhausted. The procedure (except for the issue of exhaustion) is shown in Figure 1 as the procedure $\text{Fill}(v)$. A single invocation $\text{Fill}(r)$ on the root r of the merge tree will produce a stream which is the merge of the streams at the leaves of the tree.

One particular merge tree is the k-merger. In this paper, $k = 2^i$ for some positive integer i. A k-merger is a perfect binary tree of $k-1$ binary mergers with appropriate sized buffers on the edges, k input streams, and an output buffer at the root of size k^3. A 16-merger is illustrated in Figure 2.

The sizes of the buffers are defined recursively: Let the *top tree* be the subtree consisting of all nodes of depth at most $\lceil i/2 \rceil$, and let the subtrees rooted by nodes at depth $\lceil i/2 \rceil + 1$ be the *bottom trees*. The edges between nodes at depth $\lceil i/2 \rceil$ and depth $\lceil i/2 \rceil + 1$ have associated buffers of size $\lceil k^{3/2} \rceil$, and the sizes of the remaining buffers is defined by recursion on the top tree and the bottom trees.

To achieve I/O efficiency in the cache oblivious model, the layout of k-merger is also defined recursively. The entire k-merger is laid out in memory in contiguous locations, first the top tree, then the middle buffers, and finally the bottom trees, and this layout is applied recursively within the top and

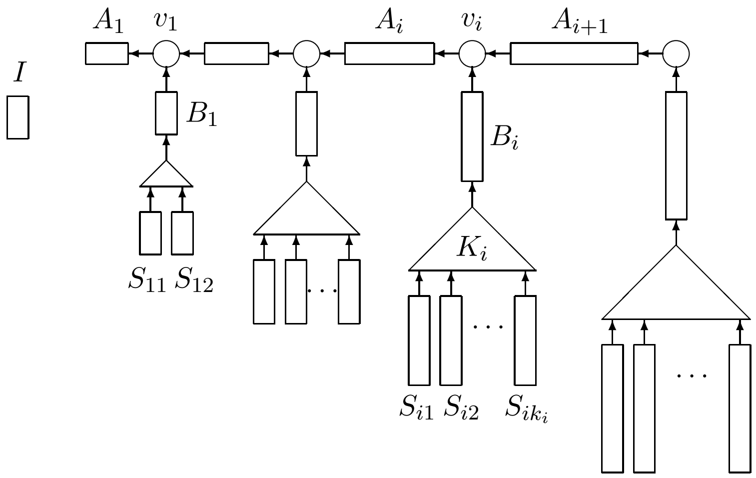

Figure 3: The priority queue based on binary mergers.

each of the bottom trees.

The k-merger structure was defined by Frigo et al. [10] for use in their cache oblivious mergesort algorithm Funnelsort. The algorithm described above for invoking a k-merger appeared in [7], and is a simplification of the original one. For both algorithms, the following lemma holds [7, 10].

Lemma 1 *The invocation of the root of a k-merger uses $O(k+\frac{k^3}{B}\log_{M/B} k^3)$ I/Os. The space required for a k-merger is $O(k^2)$, not counting the space for the input and output streams.*

3 The priority queue

In this section, we present a priority queue based solely on binary mergers. Our data structure consists of a sequence of k-mergers of double-exponentially increasing k, linked together in a list by binary mergers and buffers as depicted in Figure 3. In the figure, circles denote binary mergers, rectangles denote buffers, and triangles denote k-mergers. Note that the entire structure constitutes a single binary merge tree.

More precisely, let k_i and s_i be values defined inductively as follows,

$$\begin{aligned} (k_1, s_1) &= (2, 8)\ , \\ s_{i+1} &= s_i(k_i+1)\ , \\ k_{i+1} &= \lceil\lceil {s_{i+1}}^{1/3} \rceil\rceil\ , \end{aligned} \tag{1}$$

where $[\lceil x \rceil]$ denotes the smallest power of two above x, i.e. $[\lceil x \rceil] = 2^{\lceil \log x \rceil}$.

Link i in the linked list consists of a binary merger v_i, two buffers A_i and B_i, and a k_i-merger K_i with k_i input buffers $S_{i1}, \ldots, S_{ik_i}$. The output buffer of v_i is A_i, and its input buffers are B_i and the buffer A_{i+1} from the next link. The output buffer of K_i is B_i. The size of both A_i and B_i is k_i^3, and the size of each S_{ij} is s_i. Link i has an associated counter c_i, which is an integer between one and $k_i + 1$ (inclusive), initially of value one. It will be an invariant that $S_{ic_i}, \ldots, S_{ik_i}$ are empty.

Additionally, the structure contains one insertion buffer I of size s_1. All buffers contain a (possibly empty) sorted sequence of elements. The structure is laid out in memory in the order I, link 1, link 2, ..., and within link i the layout order is A_i, B_i, K_i, S_{i1}, S_{i2}, ..., S_{ik_i}.

The linked list of buffers and mergers constitute one binary tree T with root v_1 and with sorted sequences of elements on the edges. We maintain the invariant that this tree is heap-ordered, i.e. when traversing any path towards the root, elements will be passed in decreasing order. Note that the invocation of a binary merger maintains this invariant. The invariant implies that if buffer A_1 is non-empty, the minimum element in the queue will be in A_1 or in I.

To perform a DELETEMIN operation, we first check whether buffer A_1 is empty, and if so invoke FILL(v_1). We then compare the smallest element in A_1 to the smallest element (if any) in I, remove the smaller of these from its buffer, and return it.

To perform an INSERT operation, the new element is inserted in I while rearranging the contents of the buffer to maintain sorted order. If the number of elements in I is now seven or less, we stop. If the number of elements in I becomes eight, we perform the following *sweep* operation, where i is the lowest index for which $c_i \leq k_i$. The sweep operation will move the content of links $1, \ldots, i-1$ to the destination buffer S_{ic_i}. It traverses the path p in T from A_1 to S_{ic_i} and for each buffer on this path records how many elements it currently contains. It forms a sorted stream σ_1 of the elements in the buffers on the part of p from A_i to S_{ic_i} by traversing that part of the path. It forms a sorted stream σ_2 of all elements in links $1, \ldots, i-1$ and in buffer I by marking A_i as exhausted and calling DELETEMIN repeatedly. It then merges σ_1 and σ_2 into a single stream σ, inserts the front elements of σ in the buffers on the path p during a downwards traversal in such a way that all these buffers contain the same numbers of elements as before the insertion, and inserts the remaining part of σ in S_{ic_i}. Finally, it resets c_ℓ to one for $\ell = 1, 2, \ldots, i-1$ and increments c_i.

4 Analysis

4.1 Correctness

Correctness of DeleteMin is immediate by the heap-order invariant. For Insert we must show that the invariant is maintained and that S_{ic_i} does not overflow during a sweep ending in link i.

After an Insert, the new contents in the buffers on the path p are the smallest elements in σ, with a distribution exactly as the old contents. Hence, an element on this path can only be smaller than the element occupying the same location before the operation. It follows that heap-order is maintained.

Call B_ℓ, K_ℓ, and $S_{\ell 1}, \ldots, S_{\ell k_\ell}$ the *lower part* of link ℓ. The lower part of link ℓ is emptied each time c_ℓ is reset, so it never contains more than the number of elements inserted into $S_{\ell 1}, S_{\ell 2}, \ldots, S_{\ell k_\ell}$ since last time c_ℓ was reset. If follows by induction on i that the number of elements inserted into S_{ic_i} during a sweep ending in link i is at most $s_1 + \sum_{j=1}^{i-1} k_j s_j = s_i$.

4.2 Complexity

Most of the work performed is the movement of elements upwards in the tree T during the invocations of the binary mergers in T. In the analysis, we account for the I/Os incurred during the filling of an output buffer of a merger by charging them evenly to the elements filled into the buffer. The exception is when an A_i or B_i buffer is not filled completely due to exhaustion of the corresponding input buffers, where we account for the I/Os by other means.

We claim that the number of I/Os charged to an element during its ascent in T from an input stream of K_i to the buffer A_1 is $O(\frac{1}{B} \log_{M/B} s_i)$, if we identify elements residing in buffers on the path p before a sweep with those residing at the same positions in these buffers after the sweep.

To prove the claim, we assume that the maximal number of small links are kept in cache always—the optimal cache replacement strategy of the cache oblivious model can only incur fewer I/Os. More precisely, let Δ_i be the space occupied by links 1 to i. From (1) we have ${s_i}^{1/3} \leq k_i < 2{s_i}^{1/3}$, so the $\Theta(s_i k_i)$ space usage of $S_{i1}, \ldots, S_{ik_i}$ is $\Theta({k_i}^4)$, which dominates the space usage of link i. Also from (1) we have ${s_i}^{4/3} < s_{i+1} < 3{s_i}^{4/3}$, so s_i and k_i grows doubly-exponentially with i. Hence, Δ_i is dominated by the space usage of link i, implying $\Delta_i = \Theta({k_i}^4)$. We let i_M be the largest i for which $\Delta_i \leq M$ and assume that links 1 to i_M are kept in cache always.

Consider the ascent of an element from K_i to B_i for $i > i_M$. By Lemma 1, each invocation of K_i incurs $O(k_i + \frac{k_i^3}{B} \log_{M/B} k_i^3)$ I/Os. From $M < \Delta_{i_M+1}$ and the above discussion, we have $M = O(k_i^4)$. The tall cache assumption $B^2 \leq M$ gives $B = O(k_i^2)$, which implies $k_i = O(k_i^3/B)$. As we are not counting invocations of K_i where B_i is not filled completely, i.e. where K_i is exhausted, it follows that each element is charged $O(\frac{1}{B} \log_{M/B} k_i^3) = O(\frac{1}{B} \log_{M/B} s_i)$ I/Os to ascend through K_i and into B_i. The element can also be charged during insertion into A_j for $j = i_M, \ldots, i$. The filling of A_j incurs $O(1 + |A_j|/B)$ I/Os. From $B = O(k_{i_M+1}{}^2) = O(k_{i_M}{}^{8/3})$ and $|A_j| = k_j{}^3$, we see that the last term dominates. Therefore an element is charged $O(1/B)$ per buffer A_j, as we only charge when the buffer is filled completely. From $M = O(k_{i_M+1}{}^4) = O(s_{i_M}{}^{16/9})$, we by the doubly-exponentially growth of the s_j values have $i - i_M = O(\log \log_M s_i) = O(\log_M s_i) = O(\log_{M/B} s_i)$, where the last equality follows from the tall cache assumption. Hence, the ascent through K_i dominates over insertions into the A_j's, and the claim is proved.

To prove the complexity stated in the introduction, we note that by induction on i, at least s_i insertions take place between each sweep ending in link i. A sweep ending in link i inserts at most s_i elements in S_{ic_i}. We let the last s_i insertions preceeding the sweep pay for the I/Os charged to these elements during their later ascent through T. By the claim above, this cost is $O(\frac{1}{B} \log_{M/B} s_i)$ I/Os per insertion. We also let these insertions pay for the I/Os incurred by the sweep during the formation and placement of streams σ_1, σ_2, and σ, and for I/Os incurred by filling buffers which become exhausted. We claim that these can be covered without altering the $O(\frac{1}{B} \log_{M/B} s_i)$ cost per insertion.

The claim is proved as follows. The formation of σ_1 is done by a traversal of the path p. By the specified layout of the data structure (including the layout of k-mergers), this traversal is part of a linear scan of the part of memory between A_1 and the end of K_i. Such a scan takes $O((\Delta_{i-1} + |A_i| + |B_i| + |K_i|)/B) = O(k_i^3/B) = O(s_i/B)$ I/Os. The formation of σ_2 has already been accouted for by charging ascending elements. The merge of σ_1 and σ_2 into σ and the placement of σ are not more costly than a traversal of p and S_{ic_i}, and hence also incur $O(s_i/B)$ I/Os. To account for the I/Os incurred when filling buffers which become exhausted, we note that B_i, and therefore also A_i, can only become exhausted once between each sweep ending in link i. From $|A_i| = |B_i| = k_i{}^3 = \Theta(s_i)$ it follows that charging each sweep ending in link i an additional cost of $O(\frac{s_i}{B} \log_{M/B} s_i)$ will cover all such fillings, and the claim is proved.

In summary, charging the last s_i insertions preceeding a sweep ending in link i a cost of $O(\frac{1}{B}\log_{M/B} s_i)$ I/Os each will cover all I/Os incurred by the data structure. Given a sequence of operation on an initial empty priority queue, let $i_{\max}$ be the largest i for which a sweep ending in link i takes place. We have $s_{i_{\max}} \leq N$, where N is the number of insertions in the sequence. An insertion can be charged by at most one sweep ending in link i for $i = 1, \ldots, i_{\max}$, so by the doubly-exponentially growth of s_i, the number of I/Os charged to an insertion is

$$O\left(\sum_{k=0}^{\infty} \frac{1}{B} \log_{M/B} N^{(3/4)^k}\right) = O\left(\frac{1}{B}\log_{M/B} N\right) \ .$$

5 Profile Adaptive Performance

To make the performance dependent on N_ℓ, we make the following changes to our priority queue. Let r_i denote the number of elements residing in the lower part of link i. The value of r_i is stored at v_i and will only need to be updated when removing an element from B_i and when a sweep operation creates a new S_{ij} list (in the later case $r_1, \ldots, r_{i-1}$ are reset to zero).

The only other modification is the following change to the sweep operation of the insertion algorithm. Instead of finding the lowest index i where $c_i \leq k_i$, we find the lowest index i where either $c_i \leq k_i$ or $r_i \leq k_i s_i/2$. If $c_i \leq k_i$, the sweep operation proceeds as described Section 3, and c_i is incremented by one. Otherwise $c_i = k_i + 1$ and $r_i \leq k_i s_i/2$, in which case we will recycle one of the S_{ij} buffers. If there exists an input buffer S_{ij} which is empty, we use S_{ij} as the destination buffer for the sweep operation. If all S_{ij} are nonempty, the two input buffers S_{ij_1} and S_{ij_2} with the smallest number of elements have a total of at most s_i elements. Assume without loss of generality $\min S_{ij_1} \geq \min S_{ij_2}$. We merge the content of S_{ij_1} and S_{ij_2} into S_{ij_2}. By merging from the largest elements this merge only requires reading and writing each element one. Since $\min S_{ij_1} \geq \min S_{ij_2}$ the heap order remains satisfied. Finally we apply the sweep operation with S_{ij_1} as the destination buffer.

5.1 Analysis

The correctness follows as in Section 4. For the complexity, we as in Section 4 only have to consider the case where $i > i_M$. We note that in the modified algorithm, the additional number of I/Os required by a sweep operation is $O(k_i + s_i/B)$ I/Os, which is dominated by the $O(\frac{s_i}{B}\log_{M/B} s_i)$

I/Os required to perform the sweep operation. We will argue that the sweep operation collects $\Omega(s_i)$ elements from links $1, \ldots, i-1$ which have been inserted since the last sweep ending in link i or larger. Furthermore, for half of these elements the value N_ℓ is $\Omega(s_i)$. The claimed amortized complexity $O(\frac{1}{B} \log_{M/B} N_\ell)$ then follows as in Section 4, except that we now charge the cost of the sweep operation to these $\Omega(s_i)$ elements.

The main property of the modified algorithm is captured by the following invariant: For each i, the links $1, \ldots, i$ contain in total at most $\sum_{j=1}^{i} |A_j| = \sum_{j=1}^{i} k_j^3$ elements which have been removed from A_{i+1} by the binary merger v_i since the last sweep ending a link $i+1$ or larger. After a sweep ending at link $i+1$, we define all elements in A_j to have been removed from A_ℓ, where $1 \leq j < \ell \leq i+1$. When an element e is removed from A_{i+1} and is output to A_i, then all elements in the lower part of link i must be larger than e. All elements removed from A_{i+1} since the last sweep operation ending at link $i+1$ or larger were smaller than e. These elements must either be stored in A_i or have been removed from A_i by the merger in v_{i-1}. It follows that at most $\sum_{j=1}^{i-1} |A_j| + |A_i| - 1$ elements removed from A_{j+1} are present in links $1, \ldots, i$. Hence, the invariant remains valid after moving e from A_{i+1} to A_i. By definition, the invariant remains valid after a sweep operation.

A sweep operation ending at link i will create a list with at least $s_1 + \sum_{j=1}^{i-1} k_j s_j/2 \geq s_i/2$ elements. From the above invariant at least $t = s_i/2 - \sum_{j=1}^{i-1} k_j^3 = \Omega(s_i)$ elements must have been inserted since the last sweep operation ending at link i or larger. Finally, for each of the three definitions of N_ℓ in Section 1 we for at least $t/2$ of the elements have $N_\ell \geq t/2$:

(a) For each of the $t/2$ most recently inserted elements, at least $t/2$ elements were inserted when these elements where inserted.

(b) For each of the $t/2$ earliest inserted elements, at least $t/2$ other elements have been inserted before they themselves get deleted.

(c) The $t/2$ largest elements each have (maximum) rank at least $t/2$.

This proves the complexity stated in Section 1.

References

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

[2] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing*. ACM Press, 2002. To appear.

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

[4] M. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002. To appear.

[5] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Ann. Symp. on Foundations of Computer Science*, pages 399–409. IEEE Computer Society Press, 2000.

[6] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–39, 2002.

[7] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002. To appear.

[8] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.

[9] M. J. Fischer and M. S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.

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