Implementation and Experimental Evaluation of the Cache-oblivious Buffer Heap

CS379H Thesis
Lingling Tong
Advisor: Vijaya Ramachandran

May 5, 2006

1 Introduction

During the last decade CPU speed has increased, but speedup in memory access has not kept pace. Computer memory is divided into hierarchies with varying levels. A trade-off exists between small and fast memory versus large and slow memory. To simplify I/O analysis, memory can be thought of as divided into two levels where computation on data in main memory is fast and going to external memory is slow. With the emergence of applications that perform on large sets of data, the need for I/O efficient algorithms has risen. As an example, the online travel aid MapQuest provides directions for getting from destination A to destination B. Data can be imagined as a graph, with vertices representing cities and connecting streets representing edges. Dijkstra's shortest path algorithm, which uses the priority queue data structure augmented with the decrease-key operation, can be used to compute the shortest path between A and B. Because of the large data size and the growing gap between CPU speed and memory access, efficient algorithms need to consider the non-trivial costs of I/O. This thesis takes one data structure, a cache-oblivious priority queue-buffer heap-and examines its practical running time. The goal is to see whether buffer heap is a viable alternative to binary heap when working with large datasets.

2 Background

2.1 Memory hierarchy

We begin with a brief background on memory hierarchies. Memory hierarchies were invented to provide faster memory access by dividing the flat memory structure into levels of varying size. A small, fast but expensive level is placed near

the CPU, and larger, slower, cheaper levels are arranged farther away. Different architectures handle the divisions differently. A cache is any small, fast memory level that acts as a buffer for larger, slower memory levels. Caches are designed to take advantage of locality of reference. Temporal locality refers to the observation that programs tend to reference the same data multiple times in short period of time. Spatial locality refers to the trend that if a program references data at a give memory address, it is likely to reference data in adjacent addresses as well. Thus, data is transferred between memory levels in blocks. When the data is not found in a memory level, it is considered a cache-miss, and the next larger, slower memory level would be checked. Cache misses are expensive. There are many policies on replacing data in the cache. We assume an optimal cache replacement policy.

2.2 External memory algorithms

To simplify the memory hierarchy for algorithm analysis, we use the two level I/O model [1]. In this model, the memory hierarchy is abstracted into a fast internal memory (cache) and an arbitrarily large external memory (main memory). N is the number of elements being processed, M is the size of the internal memory, and B is the block size. We assume $1 \leq B \leq M < N$ for external-memory problems. Because the time it takes to access memory on disk far outweighs the number of instructions executed, these algorithms are evaluated based on I/O efficiency. The complexity is measured in number of data transfers between cache and main memory. Some known bounds for analyzing I/O are: $scan(N)$ takes $O(\frac{N}{B})$ since N contiguous elements incur $\lceil \frac{N}{B} \rceil + 1$ cache-misses; $sort(N)$ takes $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os [1].

2.3 Cache-aware and cache-oblivious examples

Research in external memory graph algorithms and data structures has been divided into algorithms that are cache-aware and algorithms that are cache-oblivious. Cache-aware external memory algorithms achieve optimality by fine-tuning hardware parameters. As an example, say we want to matrix multiply two $n \times n$ matrices A and B and store it in matrix C. A cache-aware algorithm works by dividing each matrix into submatrices of size $s \times s$, where s is the tuning variable. If a submatrix fits into internal memory, any standard $O(s^3)$ algorithm could be used to update C. The drawbacks of cache-aware algorithms are that they are machine dependent and complicated to tune optimally, especially when dealing with multiple memory levels. Cache-oblivious algorithms, on the other hand, are system independent. They need no prior knowledge of hardware parameters. To turn the cache-aware matrix multiply example into a cache-oblivious version, a divide-and-conquer method can be used. The difference is to keep subdividing the matrix until it reaches a constant size that is small enough to fit into

internal memory of any architecture. Cache-oblivious algorithms also free the programmer from any responsibility for data movement between memory levels [2].

2.4 Priority queues

This thesis focuses on a specific data structure, the priority queue. A priority queue is a data structure that manages a set S of elements with associated key values. The basic operations are:

- *insert*(x, k)-adds element x with key value k into S
- *delete-min*()-returns the element with the smallest key and removes that element from S (in the case of min-heaps)
- *minimum*()-returns the element with the smallest key in S

Arge et al.'s cache-oblivious priority queue works by distributing elements into levels of decreasing size [3]. The amortized time is $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os, which is optimal (follows from the $sort(N)$ bound). The Funnel heap introduced by Brodal and Fagerberg is patterned after funnelsort and works using a binary merger [4]. It also has an amortized cost of $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$. Priority queues that support two additional operations are useful in many graph algorithms. An example application would be Dijkstra's algorithm for finding shortest paths. The two additional operations are:

- *delete*(x)-removes element x from S
- *decrease-key*(x, k)-sets element x with associated key value k_x to $min(k_x, k)$

The cache-aware Tournament tree supports the augmented priority queue with an amortized cost of $O(\frac{1}{B}\log_2\frac{N}{B})$ per operation [5]. The Buffer Heap (BH) introduced by Chowdhury and Ramachandran [6] is the first priority queue that has support for *decrease-key* and is cache-oblivious. It has an I/O bound of $O(\frac{1}{B}\log_2\frac{N}{B})$, a factor of $\frac{1}{B}$ improvement over the traditional binary heap. It is not known whether the $O(\frac{1}{B}\log_2\frac{N}{B})$ bound is optimal for priority queues with *decrease-key*s. A second version of buffer heap, the Slim Buffer Heap (SBH) [7], also by Chowdhury and Ramachandran, has an amortized bound of $O(\frac{1}{\lambda} + \frac{1}{B}\log_2\frac{N}{\lambda})$ where λ is a parameter, $1 \leq \lambda \leq M$, and λ refers to the portion of main memory where the data structure is stored. Access to those λ elements costs no I/O. The data structure is free to use as much of main memory for temporary calculations as it wants. The motivating factor for using SBH is in the case of shared memory space. The main difference between BH and SBH is that the U buffer in SBH is divided into levels, and those levels are bounded. What is implemented is a combination of both BH and SBH without the λ. From here forward, "buffer heap" refers to this combination. The amortized bound is the same, $O(\frac{1}{B}\log_2\frac{N}{B})$.

3 Buffer heap

3.1 Overview

The buffer heap data structure maintains the heap state by keeping track of operations stored in two buffers. The U buffer is the update buffer, and the B buffer is the element buffer where the elements of the data structure are stored. Each buffer is divided into a total of r levels where $r = 1 + \lceil \log N \rceil$, and N is the maximum number of elements. The buffers are arranged such that the top of each buffer is denoted B_0 and U_0 and the bottom as B_{r-1} and U_{r-1}. All new operations enter at the top of the buffer, U_0. The buffer heap takes the lazy approach and operates by applying updates as a batch process. Each *decrease-key* and *delete* is allowed to accumulate in the U buffer and applied to the elements in the B buffer all at once when a *delete-min* is called or the data structure periodically restructures itself. The data structure does as much useful work on data currently in the cache as possible before eviction. The invariants of the buffer heap are:

1. Each B_i contains at most 2^i elements.
2. Each U_i contains at most 2^i updates.
3. For $0 \leq i < r - 1$, the key of every element in B_i is no larger than the key of any element in B_{i+1}.
4. For $0 \leq i < r - 1$, for each element x in B_i, all updates applicable to x that are not yet applied reside in U_0, U_1, ..., U_i.
5. Elements in each B_i are kept sorted in ascending order by element ID.
6. Updates in each U_i are divided into (a constant number of) segments with updates in each segment sorted in ascending order by element ID and time stamp.

Pseudocode can be found in the appendix.

3.2 Operations

The buffer heap supports the operations *decrease-key*(x, k_x), *delete*(x), and *delete-min*. The function *decrease-key*(x, k_x) serves the dual purpose of decreasing the key associated with x to $min(k_x, key)$ if x is found in the data structure, and inserting the pair (x, k_x) into the data structure if x is not found.

3.2.1 Decrease-key/Delete

Each *decrease-key* and *delete* is first added to U_0 and marked with the current timestamp. At this point, invariant (2) may be violated. The data structure uses two internal functions. *Fix-U*(i, B') maintains invariant (2). Starting at level i, if U_i contains more than 2^i elements, *Fix-U*(i, B') applies the updates in U_i on B_i according to its operation and timestamp and collects the results in B'. *Apply-Updates*(i) handles the movement of elements between U_i and B_i. It is then possible for B_i to contain more than 2^i elements due to multiple *decrease-key*s turning into *insert*s. Then, the 2^i smallest elements by key value are kept in B_i (sorted by element ID) and the rest of the elements are moved to U_{i+1} as internal *Sink* operations. Operations in U_i that were not applied to some element in B_i are also copied as *Sink* operations into U_{i+1}. At the end of *Apply-Updates(i)*, U_i is empty. This process is repeated for each i until either invariant (2) is satisfied or $i = r$. Finally, the elements collected in B' are redistributed to the shallowest levels in the B buffer while maintaining invariants (1, 3, 5).

3.2.2 Delete-min

The case of *delete-min()* is different. A *delete-min()* is not inserted into U_0. Instead, it calls *Apply-Updates(i)* until some element has been collected into B'. Because of the invariants, we know that B_0 contains the minimum element, and it is returned and removed from the B buffer.

Finally, the data structure is optionally periodically reconstructed when the number of operations in the U buffer becomes twice the number of elements in the B buffer. Reconstruction calls *Apply-Updates(i)* from $i = 0$ to $i = r - 1$. And, the collected elements are also redistributed to the shallowest levels in B maintaining invariants (1, 3, 5). Reconstruction guarantees that the buffer heap uses linear space in the size of the elements.

Correctness follows from maintenance of the invariants.

3.2.3 I/O bound

We use the potential method to calculate amortized I/O bound. Amortized analysis provides the bound for cost per operation. The potential function uses a potential Φ defined on a data structure D such that $\Phi(D_i) = \Phi(D_0)$ where D_i is the data structure after ith operation and D_0 is initial data structure. Then, the amortized cost is the actual cost plus the change in potential.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{1}$$

Observations A few observations will be helpful for the analysis for buffer heap. The general movement of elements follows a U shape where an operation starts in U_0, moves down the U buffer, crosses over to the B buffer, and moves

up the B buffer. Since the height of each buffer is r, the total levels traversed is at most $2r$. The elements in each B_i will traverse at most $i+1$ levels before being removed from the data structure. Finally, each *decrease-key* is treated as two operations. The application of *Apply-Updates(i)* could result in an overflow of B_i, in which case the excess elements each generate a new internal *sink* operation and are moved to U_{i+1}.

Lemma: The amortized cost of operations on the buffer heap is $O(\frac{1}{B}\log_2 \frac{N}{B})$.

The potential function is defined as follows:

$$\Phi(H) = \frac{2}{B}\sum_{i=0}^{r-1}\{(2r-i)u_i + (i+1)b_i\} + \frac{2\Delta_0 r}{B} \tag{2}$$

Here u_i refers to the number of operations in U_i, and b_i refers to the number of elements in B_i. The Δ_0 represents the number of new operations since the last reconstruction. The $\frac{1}{B}$ results from the sequential scan of X elements costing $\Theta(\frac{X}{B})$ I/O. The $(2r-i)u_i$ accounts for movement of elements in the U buffer; the $(i+1)b_i$ accounts for movement of elements in the B buffer. The $\Delta_0 r$ provides credit for movement of new operations.

Reconstruct Reconstruction occurs whenever $\Delta_0 = \frac{N_e}{2}$, where N_e is number of elements in the data structure after the last reconstruction. This means that b_i is between $\frac{N_e}{2}$ and $\frac{3N_e}{2}$. The worst case scenario is that all Δ_0 updates travel r levels before being applied for an actual cost of $O(1+\frac{\Delta_0 r}{B})$. After reconstruction, all u_i are emptied and each update adds at most 1 element. The potential after reconstruction is $\frac{2}{B}\sum_{i=0}^{r-1}\{(r+1)u_i + (i+1)b_i\}$. This gives a potential change of at least $\frac{\Delta_0}{B}$, for an amortized cost of at most 0.

Decrease-key/Delete The change in potential is $2\times\frac{6r}{B}$ for *decrease-key* and $\frac{6r}{B}$ for *delete*. We show that the cost for accessing each level via calls to *Fix-U* pays for itself. We assume the first $\log B - 1$ levels incur no I/O. The cost for accessing a level higher than $\log B - 1$ has also been paid. Let j be highest level i accessed by *Apply-Updates*. If U_j is full then, the potential drop from the movement of the 2^j updates to U_{j+1} is sufficient to pay for scanning and updating those elements. Redistribution using a linear selection algorithm also takes $O(\frac{2^j}{B})$ I/Os since the 2^j term dominates in choosing 2^k, $0 \le k \le j$ smallest keys for redistribution. If U_j is not full then the movement of each update to U_{j+1} pays for its scan. Then, the amortized cost becomes $O(\frac{1}{B}\log_2 N)$ for both *decrease-key* and *delete*. And, because the first $\log B$ levels have no I/O cost, it becomes $O(\frac{1}{B}\log\frac{N}{B})$.

Delete-min The potential change is $\frac{2r}{B}$. The amortized cost of *delete-min* is the same as that for *decrease-key/delete*. The call to *Apply-Updates(i)* empties

out all U_j elements $j < i$ up to U_i, which pays for the scanning and movement of those elements. And, the U_i overflow case handled by *Fix-U* has also been shown to be self-paid. The amortized cost is then $O(\frac{2}{B}\{\log N - \log B\}) = O(\frac{1}{B}\log\frac{N}{B})$.

4 Implementation

4.1 Buffer heap data structure

The language of choice is C++ for its object-oriented style and template datatypes. The representation of each element/update is a struct that contains the x value, key value, and an unsigned integer state field. The state field is broken up into a timestamp, level number, and operation code.

Conceptually, the buffer heap is divided into two buffers. For implementation purposes, it is more efficient to use one buffer, as we want to maximize spatial locality. The natural representation for the buffer would be the stack due to the pushing and popping from one end. However, we are dealing with two buffers interleaved into one and require the simultaneous scan of both U_i and B_i. Thus, our basic data structure is a one-dimensional, dynamically resizable vector. The vector doubles in size whenever it becomes full and shrinks during reconstruction. During reconstruction we know r, the size of the data structure, so it becomes efficient to resize our vector at that time to minimize wasted space. The arrangement of the U and B segments are as follows. If the vector grows from 0 to N, then the buffers are laid out as B_{r-1}, U_{r-1}, ..., B_0, U_0 in contiguous segments, with a size marker between each segment. This arrangement makes it efficient to add a new operation to the head of U_0.

Two internal operations are utilized to maintain correctness: *sink-i* and *delete-i*. *Sink-i* marks updates that originated with a *decrease-key* applied to some level i and became evicted due to B_i overflow. A *delete-i* operation is added to U_{i+1} whenever a decrease-key operation inserts a new element into B_i. These operations help maintain a correct state.

4.2 Selection

The part of the pseudocode that required optimization is the linear selection algorithm for handling B_i overflows and redistribution of the elements. Because of the requirement that scanning both B_i and U_i must be done in linear time, the B_i data must be kept stored in x-value order. However, overflows and redistribution are handled based on key value ordering. These two competing factors contribute to the bottleneck of the buffer heap.

The first algorithm examined for doing linear selection is *randomized-select*. *Randomized-select* works by randomly choosing a pivot element in the array, partitioning the elements around that pivot, and returning the pivot if the index

of the pivot is i. If the pivot is not at index i, then it recursively calls itself on the left subsection or the right subsection. Because *randomized-select* is dependent on randomly choosing the pivot, its run-time could be as bad as $\Theta(n^2)$ if we want to find the smallest element and it always partitions around the largest. But, our input data is mostly randomized, and *randomized-select* runs in $O(n)$ on average.

The second linear select algorithm is *sample-select*. *Sample-select* works by taking a sampling sized $n^{\frac{3}{4}}$ of an input array A of size n, call the resulting array S. It sorts the elements of S and finds the elements x and y with ranks $i \times \frac{s}{n} - \sqrt{n}$ and $i \times \frac{s}{n} + \sqrt{n}$ where s is size of sampled array. It then scans the original array to find the ranks of x and y in A, call them L and R. With high probability, $L < i < R$. Then a scan is made through the original array A and all elements z, $x \leq z \leq y$, are collected in a new array Q. After sorting the elements of Q, the element with rank i is in $Q[i - L]$. Even though there are two calls to sort routines, *sample-select* runs in $O(n)$. The two calls to sort are made on arrays of size at most $n^{\frac{3}{4}}$ so the dominating term is the linear scan of A.

A summary of the timing findings is given in Figure 1 and a plot is given in Figure 2.

n	*rand-select*	*samp-select*	*samp-mod*
100000	0.0304	0.0168	0.0160
500000	0.1248	0.0784	0.0592
1000000	0.1720	0.1160	0.0784
5000000	1.4073	0.5768	0.5280
10000000	2.5202	1.1601	1.0537
50000000	12.7392	5.4507	4.2379

Figure 1: Timing data for rand-select and sample-select

All testing was done on a random sample of size n, searching for the element with rank i, also randomly chosen, and averaged over 5 runs. *Rand-select* and *samp-select* were coded according to the above descriptions, with no modifications. Because the bound for *rand-select* is only linear in the expected case and the bound for *samp-select* is $O(n)$ with high probability in n, *samp-select* does better. *Samp-mod* contains additional improvements to the basic *sample-select* idea.

The first observation is that sorting the sampled array S and finding x and y with ranks l and r can be done in linear time with calls to *samp-select*. The next changes are based on finding x and y and collecting $x \leq z \leq y$ at the same time. This feat is accomplished using a 3-way partitioning algorithm by Bentley and McIlroy's [8]. After partitioning, if $L \leq i \leq R$, then we know our element with rank i is in $A[L..R]$. And, we make a recursive call on $A[L..R]$. If $L > i$ we recurse on $A[0..L-1]$. If $i > R$, we recurse on $A[R+1..n-1]$. The basecase is handled with a call to *rand-select*. Based on the timings, *samp-mod* was chosen

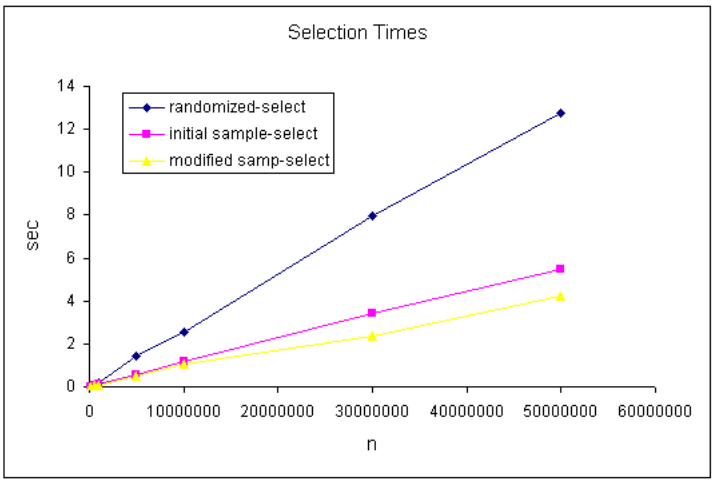

Figure 2: Plot of selection data

for incorporation into buffer heap.

4.3 Incorporation into buffer heap

Implementation from the pseudocode is straightforward.

The two functions open to variations are merge and redistribution. For merge, in the general case, each level U_i can have multiple segments from the movement of elements and updates from B_{i-1} and U_{i-1} respectively. Before *Apply-Updates* is called, the different segments of U_i should be merged. Merging happens two segments at a time and uses extra space equal to the size of one of the merging segments. For redistribution, *sample-select* finds the 2^ith largest element and partitions the elements around that element. Partitioning does not guarantee any ordering of the first 2^i elements. Because those elements need to be placed back into the B buffer in x value order, a system sort is called. Repeated calls to sort are expensive.

The improved version of buffer heap uses a merge during the collection of elements for redistribution. Using merge does not disrupt the amortized bound since merging takes linear time in the length of the segment, and the dominating factor is the length of the longest segment, which is paid for by the movement of elements of that segment to lower B buffers. Then, when the elements are ready for redistribution, no call to sort is necessary, as the previous merge steps have already placed the elements in x value order. We also allowed the capability of turning off reconstruction and beginning the buffer heap on a level other than $i = 0$, i.e., for some j, the first j levels of B and U are not used, and operations are first inserted into U_j.

5 Evaluation

The binary heap data structure was chosen for comparison as it has worst case bound of $O(\log_2 N)$. The hope is that the buffer heap would provide a rough $\frac{1}{B}$ improvement to the binary heap. Once we deemed the buffer heap to be correct, we used operations generated from Dijkstra's shortest path on a random graph as input. All test data fit into internal memory.

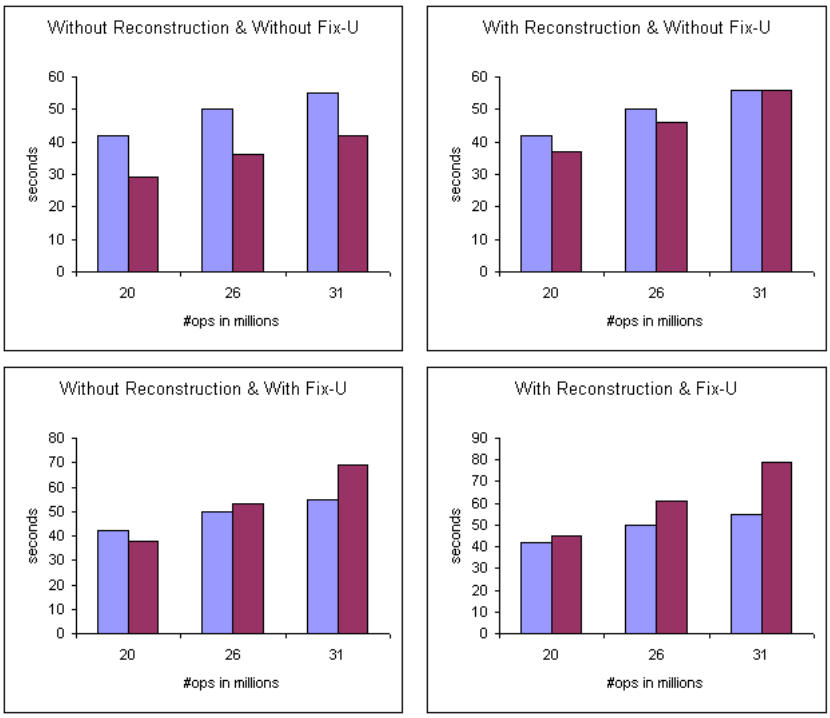

Figure 3: Comparison of binary heap and buffer heap with and without reconstruction and fix-u. The left (blue) bar is binary heap. The right (red) bar is buffer heap.

The above values were averages from 3 runs on Uruk-2, a 3.0 GHz Intel Xeon machine with 4GB RAM, 8KB L1 cache and 512KB L2 cache. A similar trend was seen on leaflock, a dedicated Sun machine. The test data was the sequence of operations performed during Dijkstra's algorithm on a randomly generated graph. In all four figures, the number of vertices was 10 million.

Buffer heap does better with reconstruction and fix-u turned off. For the purposes of shortest paths, turning those routines off is reasonable, as the space requirement is known ahead of time. As the number of operations is increased buffer heap slows down faster than binary heap. This trend can be accounted for by the extra cost of traversing down the levels in buffer heap for *decrease-key*s. Without reconstruction and fix-u, buffer heap has a reduction in time of about 25% over binary heap. For smaller sets of vertices, that percentage is lower,

and in many cases, due to the high cost of overhead, buffer heap runs slower. For significantly large set of vertices and external memory access, we expect the percentage to be higher.

Preliminary results using STXXL show the expected increase in performance. STXXL is a C++ STL extension library that allows simulation on restricted memory sizes. This way we can force external memory access for both buffer heap and binary heap. The data was provided courtesy of Lan Roche. Buffer heap does extremely well when the memory size is restricted to less than 4N bytes where N is the number of vertices. Buffer heap showed an increase by a factor of 115.

6 Conclusion

The design of buffer heap was motivated by the desire to improve runtime of finding the shortest path on graphs with a large number of vertices and edges. The amortized cost of buffer heap as compared to binary heap showed an increase by a factor of $\frac{1}{B}$. Preliminary runtime results show a lower than expected increase. The reason is two-fold. One, the overhead of maintaining the buffer heap state is high. Comparison between turning reconstruction and fix-u on and turning those routines off shows that turning reconstruction and fix-u off gives an improved runtime. Two, perhaps the data size is not large enough and going to external memory will provide the difference. Results from STXXL provide evidence for significant improvement of buffer heap compared with binary heap when the memory size is restricted less than 4N bytes, where N is the number of vertices.

References

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

[2] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pp. 285-297, 1999.

[3] L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, and J.I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of ACM Symposium on Theory of Computing*, pp. 268-276, May 2002.

[4] G.S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation, LNCS 2518*, pp. 219-228, Nov. 2002.

[5] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pp. 169-177, 1996.

[6] R.A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the 16th Annual Symposium on Parallelism in Algorithms and Architecture (SPAA)*. pp. 245-54, 2004.

[7] R.A. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pp. 735-744, Jan. 2005.

[8] J. L. Bentley and M. D. McIlroy, Engineering a sort function, *Software–Practice and Experience*, 23, 1249–1265 (1993).

7 Appendix

```
Buffer Heap

Decrease-Key(x, key)/Delete(x):
1) Insert the operation into U0 augmented with the current timestamp
2)  a) Set B' <- 0, i <- 0
    b) Fix_U(i, B')
3) Redistribute B' to shallowest levels maintaining invariants
4) Reconstruct

Delete-Min():

1) Set i <- -1
   while i < r-1 do
     a) set i <- i+1
     b) Apply_Updates(i)
     c) if Bi is non-empty then exit loop
2) a) set B' <- Bi, i <- i+1
   b) Fix_U(i, B')
3) a) Extract the element with the minimum key from B' to return
   b) Move remaining elements from B' to the shallowest possible element
      buffers maintaining invariants
4) Reconstruct

Reconstruct():
```

```
// No is number of operations since the last reconstruction
// Ne is number of elements in BH after last reconstruction
1) if No = Ne/2+1
     a) for i = 0 to r-1 do
1) Apply_Updates(i)
2) Merge the elements of Bi with B' by element id
     b) distribute the elements remaining in BH to the shallowest possible element buf

Fix-U(i, B'):

1) while i < r and Ui overflows
     a) Apply_Updates(i)
     b) Merge the elements of Bi with B' by element id
     c) set i <- i+1
2) if i < r then merge the segments of Ui

Apply-Updates(i, B'):

1) if Bi is empty and i < r-1
     a) merge segments of Ui
     b) move the contents of Ui as a new segment of Ui+1
     c) set Ui <- 0
2) else
     a) merge the segments of Ui
     b) if i = r-1 then set k<-inf else set k<-largest key of elements in Bi
     c) scan Bi and Ui simultaneously and for each operation in Ui if the
        operation is:
        1) delete(x), then remove any element (x, kx) from Bi if exists
        2) decrease-key(x, kx)/sink(x, kx), then if any element (x, kx')
           exists in Bi
             a) replace it with (x, min(kx, kx'))
             b) otherwise copy (x, kx) to Bi if kx <= k
     d) if i < r-1 then
        1) copy each decrease-key(x, kx)/sink(x, kx) in Ui with kx > k to Ui+1
        2) for each delete(x) and decrease-key(x, kx) with kx <= k in Ui copy
           a delete(x) to Ui+1
     e) if Bi overflows then
        1) if i = r-1 then r <- r+1
        2) keep the 2i elements with the smallest 2i keys in Bi and insert
           each remaining element (x, kx) into Ui+1 as sink(x, kx)
     f) set Ui <- 0
```

Selection

```
Randomized-select(A, p, r, i)
1.  if p = r
2.    then return A[p]
3.  q <- Randomized-partition(A, p, r)
4.  k <- q - p + 1
5.  if i = k
6.    then return A[q]
7.  elseif i < k
8.    then return Randomized-select(A, p, q-1, i)
9.  else return Randomized-select(A, q+1, r, i-k)

Partition(A, p, r)
1.  x <- A[r]
2.  i <- p - 1
3.  for j <- p to r - 1
4.    do if A[j] <= x
5.      then i <- i + 1
6.        exchange A[i] <-> A[j]
7.  exchange A[i+1] <-> A[r]
8.  return i + 1

Randomized-partition(A, p, r)
1.  i <- Random(p, r)
2.  exchange A[r] <-> A[i]
return Partition(A, p, r)

Sample-select(A, 0, n, i)
(1) Pick a random sample S of the elements of array A by choosing each
    element in A to be in S with probability p = 1/(n1/4) independent of the
    other elements.
(2) Sort the elements in the set S using Quicksort.
(3) Let |S| = s and let l = (s/2)-sqrt(n) and r = (s/2)+sqrt(n).
    Find the element x with rank l in S. Find the element y with rank r in S.
(4) Find the ranks of x and y in A and let these ranks be L and R,
    respectively.
(5) If L < n/2 and R > n/2 then
    (i)  Find all elements z in A such that x < z < y, and place these
         elements in a set Q.
    (ii) Sort the elements in Q using Quicksort and output the element with
         rank ((n+1)/2)-L in Q as the median element of A.
```