Buffer Heap Implementation & Evaluation

Hatem Nassrat

CSCI 6104
Instructor: N.Zeh
Dalhousie University
Computer Science

Table of Contents

Introduction

In the study of computer science, researchers and students are always searching for ways to create faster more efficient applications to solve their problems. The design and analysis of algorithms is a current and growing science that aims to find the best ways of solving problems within given constraints. The study began by researching the fastest least computation intensive methods of solving mathematically challenging problems that face the computer theorist on a daily basis. This not only lead to many computation efficient algorithms, but arguably pushed the computer industry in creating faster, and smaller machines of mystifying capabilities when comparing to the technology that was available in years prior. This great shift in technology opened up a new type of computation that was only suitable for extreme machined.

Computation saving algorithms, while still significant, is drifting away from research topics and gave way to the study of algorithms that deal with massive datasets. Today, using standard machines, problems that required supercomputers such as those that appear in graph theory and combinatorics, can be solved. Algorithms being devised under this new field of research target , deal with large data that cannot be held in the internal memory of a standard machine, but on their disks. Such algorithms deal mainly with saving disk accesses rather focusing on computation. At the birth of this research area, internal memory computation was marked free, and the calls getting data from disks were analyzed. However, more recent research showed that in some circumstances, heavy computation between disk accesses may lead to a slow application.

Researchers in this area agreed that to be able to efficacize* solutions to primary problems, one should should use efficient data structures within their algorithm. One of the most fundamental data structures in the field of graph theory and I/O intensive computations in general is the priority queue. There exists many types of priority queues that exist in the I/O intensive realm, each of which has its own advantages and disadvantages. However, many of them do not have means of updating the values of its elements or preempting them once inserted into the structure. This study aims to describe and evaluate an implementation of such a data structure known as the Buffered Heap Priority Queue.

Cache Aware / Cache Oblivious Algorithms

In modern day computer architecture, computation machines have many levels of internal memory each of which is located at a certain distance from the processing unit and have an associated average speed of data delivery. Since the depth of the memory hierarchy changes from machine to machine, it becomes harder to study and devise algorithms that are efficient when dealing with data that is stored on the lowest level of any such hierarchy.

In 1988 Argwal and Vitter introduced what they call the I/O Model[1], in order to simplify and capture different machine architectures when analyzing their external memory efficiency. This model, divided into two levels, conceptually a fast internal memory level of size M and a larger but slower external memory divided into blocks of size B. In this model the complexity of an algorithm is modeled by the number of transfers of blocks of size B between the two levels. This generalization of the current computer architecture successfully captures the running time when data transfer between memory levels dominates the computation cost of an algorithm. Using this I/O model, algorithm designers began designing I/O efficient algorithms based on the arbitrary values of the external memory block size B and the internal memory capacity M. Algorithms that rely on such knowledge are now known as Cache Aware Algorithms, as they perform differently as the data is available within the different levels of

* Make efficient. (Not sure if it is a real word)

cache.

More recently came the introduction of the Cache Oblivious Model [3]. As the name implies, algorithms designed using this model are oblivious to the different levels of cache and how the data moves between those levels. However, using the assumption that computation is essentially upper bounded by data transfers, using this model researchers where able to design algorithms that were able to reduce in size in order to be efficient without knowledge of the surrounding architecture. Such algorithms exploit divide and conquer techniques without knowledge of memory sizes or layouts to compete with similar cache-aware versions. The priority queue mentioned earlier has been designed under both paradigms.

Buffer Heap

This structure was first mentioned as the bucket heap in [4] to help solve the single source shortest path algorithm (SSSP) by using it as the priority queue used by the SSSP algorithm. It is also mentioned in [5] with an experimental evaluation in [6]. The buffer heap is designed to function as a priority queue with maximum functionality. Such a priority queue supports the following Operations:

- Insert (element x,key k)
  This is performed as a combined Update/Insert operation where the correct call to the buffer heap would be to Update(x,k)
- Delete-Min()
  Deletes the element with minimum key from the PQ
- Delete(element x)
  Deletes element x, if exists, from the priority queue. Otherwise the delete is ignored.
- Update(element x,key k)
  Updates an element x with key k if k is less than the current key of x in the priority queue, if the current key of x is lower than k the update is ignored. If the element does not exist in the queue, the Update inserts the element into the priority queue.

The basic data structure used within the buffer heap is essentially a set of Element and Update Buffers, each of which has an associated level and size. These buffers are organized in levels of increasing size where each level holds 2^i elements and 2^i updates. Leading to a total of $r = \log_2 N$, where N is the number of elements or operations (depending on which is greater at any point in time). This structure is a lazy update structure where operations are accumulated and batch processed. This is achieved by generally inserting operations into the first (top) level buffer of the structure and pushing them down when appropriate. The operations are applied to the element buffers at update buffer overflow or upon receiving a delete-min signal. A few Invariants are associated with the buffer heap in order to maintain correctness of its operations and are as follows:

1. Each Element buffer B_i contains at most 2^i elements
2. Each Element buffer U_i contains at most 2^i operations
3. The key of every element in B_i must be no greater than any element in B_k where $k>i$ and $0 <= i,k < r-1$
4. For any element in B_i operations associated with that element that are yet to be applied must reside in the update buffers of level i or a level $< i$.
5. Elements in each B_i and Operations in each U_i are kept sorted by (applicable) element id.

Through these invariants one can claim the correctness of the implementation of the priority queue operations that are described bellow.

Buffer Heap Implementation

The implementation of this structure was written in C++ due to its memory allocation capabilities, small overhead, and availability of extensible external memory libraries. To implement the buffer heap, decisions where made intending to optimize the structure of the buffer heap in order to increase efficiency. The buffer heap was constructed as a single dynamically re-sizable vector holding all the levels of the priority queue. This vector was constructed using the STXXL library [2] for C++ to increase I/O effeciency. The levels were laid out in order of decreasing size, thus keeping the “top” level buffer at the end of the vector. Thus allowing the addition of new operations to the end of the buffer. Both elements and update buffers were merged into a single buffer at each level, maintaining the invariants mentioned above. The interleaving of both elements and operations allows for a single sequential scan of each level in order to apply the operations to the elements of that level. Such a structure seemed promising at first, however, in concurrence with other modifications, proved to be troublesome to maintain. The following descriptions would use the assumptions stated here to describe the internal operations of the proposed buffer heap.

Operations

Update/Delete

Such operations are added to the top level update buffer U_0 of the buffer heap marked with the current time stamp[*]. Doing so may cause the update buffer to overflow by having size greater than 2^0. This is fixed by a call to an internal buffer heap function that applies the updates in level_0 . Update operations that have a key less than the greatest key within the buffer insert elements within the buffer, and push a delete signal to the next level in the structure. Otherwise they are considered inapplicable and pushed to the next level along with inapplicable delete operations. The application of operations may cause the element buffer when updates turn into to overflow when updates are mutated into inserts and thus a selection algorithm (details to follow) is called to select the smallest 2^0 elements. The rest of the elements are pushed along side the non-applicable updates to the next level of the structure. This operation may cause overflows within the next levels and thus this procedure is repeated until the invariants are regained.

Delete-Min

When receiving a delete min signal it is not inserted into the operation buffers as it is to be processed immediately to make sure it returns the correct result. To do so, the delete-min calls the internal apply-updates operation on level starting with the top level. If there are elements to return the procedure is stopped and the elements are returned. Otherwise the procedure is called on the following levels in order until this is satisfied. If there are no elements in the buffer heap this would cause the delete-min to return with an error.

I/O Bound

Following from the bound proofs described in [4][5], the cache aware and cache oblivious buffer heaps supports the operations described above in am $O(\frac{1}{B}\log_2\frac{N}{B})$ amortized. Due to the nature of performing updates within the implementation described here, this bound may be slightly greater by a factor of $\log_2 N$ leading to $O(\frac{1}{B}\log_2^2\frac{N}{B})$ for processing delete-min

[*] Time stamping can be avoided with correct implementation. Details to follow

operations when N is small. The disruption of the amortized bound occurs due to the implementation of the internal apply-updates procedure. Due to some restrictions described bellow, levels are sorted before and after a call to apply updates.

Selection (Handling Overflows)

The selection algorithm plays a major role in the performance of the priority queue as it is responsible for handling overflows in each level of the queue. [6] showed experimental evidence of significant performance degradation when using the randomized selection algorithm for this feature. It has bee shown that the sample selection algorithm can outperform the randomized version by a factor of two in speed. In the current implementation of the buffer heap, the C++ standard nth_element function available in the STL was used to perform this selection. The more recent versions of the STL such as the one being used for this implementation, implement a variant of the sample selection algorithm and are proved to observe good performance. A study of experimental and current selection algorithms was performed and presented in the Copenhagen STL conference of 2002, where it has been shown that the std::nth_element procedure of C++ outperforms many other algorithms when using integer keys on vectors [7].

Implementation Issues

Due to the nature of the problem of having large datasets, inserting and deleting elements at random positions of the vector are not feasible. Therefore the decision was made to layout the levels of the buffer heap in order of decreasing size as was described earlier. This allowed for new elements to be pushed at the end of the vector, which in reality appends the operation to the end of the top level buffer.

A second decision, which proved to be the most drastic, was to perform all the operations in place as to reduce the overhead of copying out each level and copying them back in, which lead to the following. When elements or operations are deleted they are marked as deleted and moved to the end of the level. Occasionally when the level is too large due to deletes hanging around they are pushed to the level bellow. When an update operation get processed prior to finding its element, then the delete signal is contained in one of the deleted containers found at the end of the level and pushed to the next level. Pushing elements and operations to the next level is done by labeling those elements / operations / deleted-containers with the value of the next level and a sort operation handles their movement as to append to the end of the target level.

This implementation not only proved to be costly but could essentially break on certain inputs. The breaking of the structure occurs when an update is processed at level *i* prior to an element being removed from level *i*. The cost associated with this implementation is with respect to the necessary calls to sort that move deleted/pushed operations from the middle of a level's buffer to the end of the level.

These problems could have been avoided with the current large vector structure as follows. When applying updates each level is copied out to a temporary vector, and the operations are applied to that vector. After application of all the updates, the elements belonging to that level are copied back to the levels buffer and a new marker is set for the end of that level, increasing the size of the level that follows. Objects marked with a push are then merged with elements of the following level thus maintaining the invariants. This idea is yet to be implemented, and is not detailed in the experimental analysis section bellow.

Tuning Parameters

There are many parameters that can affect the performance of the buffer heap due to the external memory nature of the data structure. These parameters allow for fine tuning of the buffer heap in order to reduce the number of I/Os and place a limit on the memory requirements of the buffer heap.

Internal Memory Consumption

As the buffer heap currently stands it has a few parameters that limit its memory consumption. The first of these parameters is one that fores the sorter to stay within a memory limit. The second, is a set of parameters that allow fine tuning of the main structure (vector) within the priority queue. The internal memory consumption of the vector is limited to the multiplication of, the number of blocks_per_page, the pages_in_cache, and the block_size_in_bytes. Added to these two segments there exists a few internal variables and a small internal memory vector that get allocated and de-allocated, adding a maximum of 2MB of internal memory.

Other buffer heap parameters

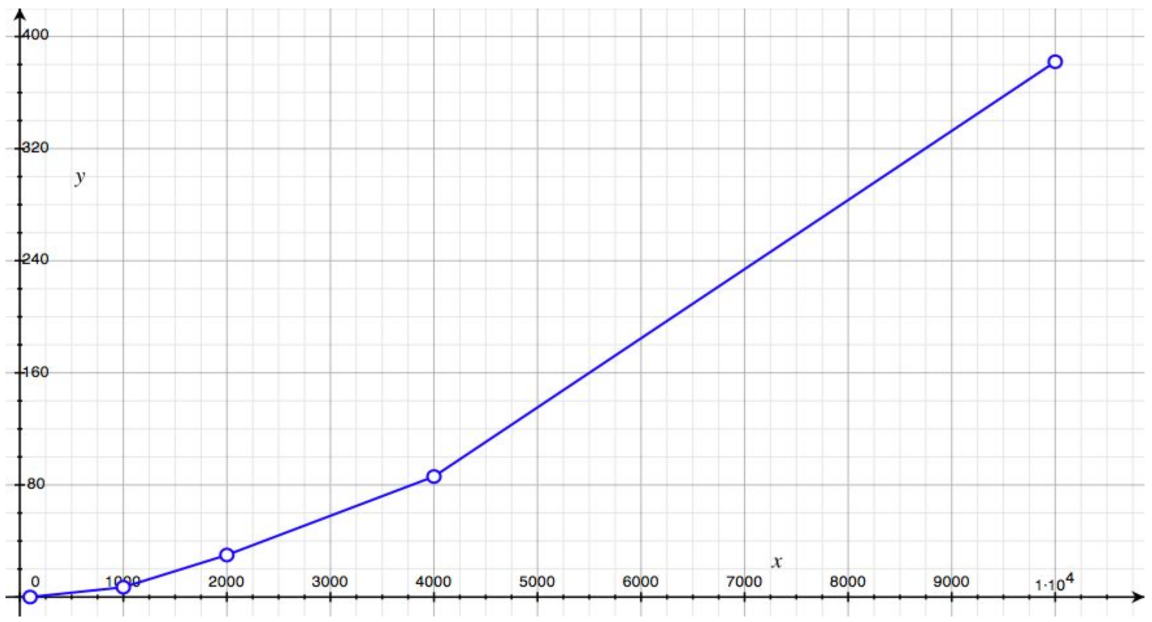
The major parameters that can affect the performance of the buffer heap are the base size of the buffers, which is 2 in the above description. The buffer heap implemented is malleable such that this value can be supplied by the user. Also as described the buffer heap begins at level *0*, making the first level capable of only holding a single element and levels increase exponentially from there. The capability of starting the buffer heap at a level required by the user has been incorporated into the data structure.

Experiment and results

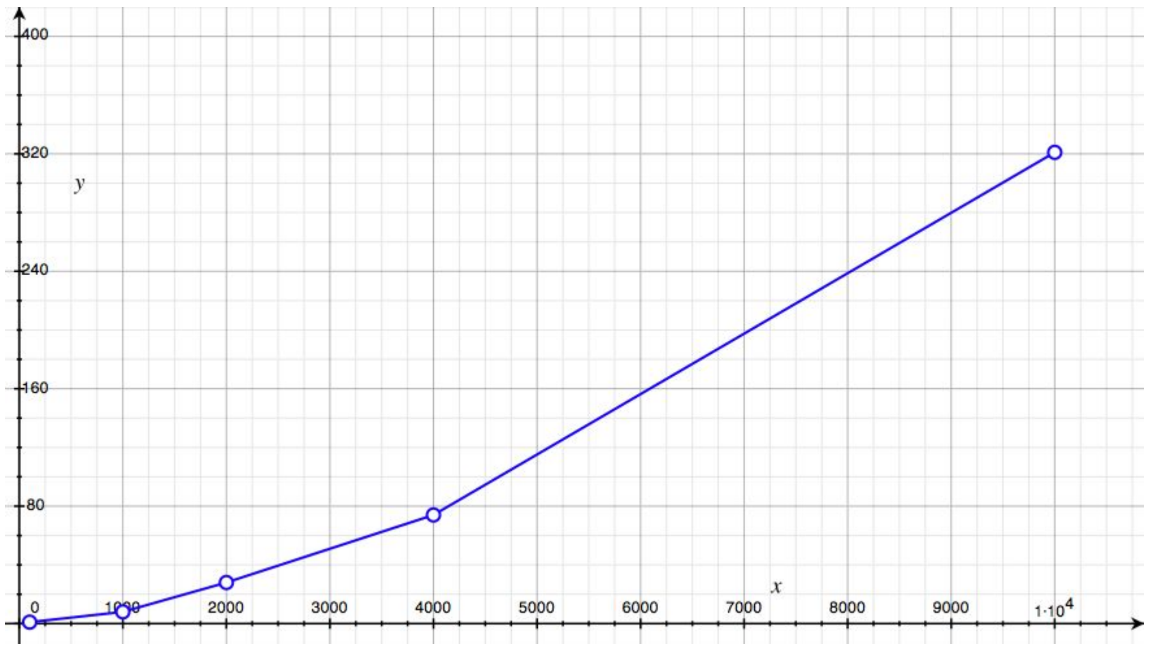
The library used in the buffer heap implementation was the STXXL library implemented for C++. Tuning of the parameters that control the reading and writing of the stxxl vector to and from memory showed some issues with the direct implementation of this library. The recommended and default block size for use with this library is two megabytes. Leading to a recommended internal memory consumption of around 64 megabytes. In terms of implementation issues it seems that modification of any blocks causes a direct writing of the block to external memory even if the block need not be moved out of memory. Initially, using the default parameters, the buffer heap implementation was completing (amortized) a single operation every 2-3 seconds when given 100 elements. Changing the block size to 4MB lead to an amortized operation every 5 seconds and when testing with 8MB blocks on the same input it took more than 9seconds per operation. However, after seeing these results, the buffer heap was recompiled using smaller values in order to determine the appropriate block size. Using an 4 kilobyte block size the buffer heap completed the 100 operations in less than 1 second. Assuming the libraries inefficiencies with multiple dumping to the hard disk, it becomes apparent that the block size should be relative to the average level size, therefore the bigger the input the larger the block size to apply to it.

After research with different block sizes, different start levels, and base level sizes for the priority queue it has been seen that this heap performs best with blocks of size 8KB, Base Level size of 6 and skipping the first 3 levels, making the first level of size 6^3 = 216 elements. With these parameters the buffer heaps internal memory consumption is around 2MB, with most of it reserved for sorting (buffer heap usage is around 270KB). These parameters will be used to study the buffer heap's performance on different dataset sizes and types. The following graphs illustrate the results. (X axis is the number of operations, Y axis is the time consumed is seconds.

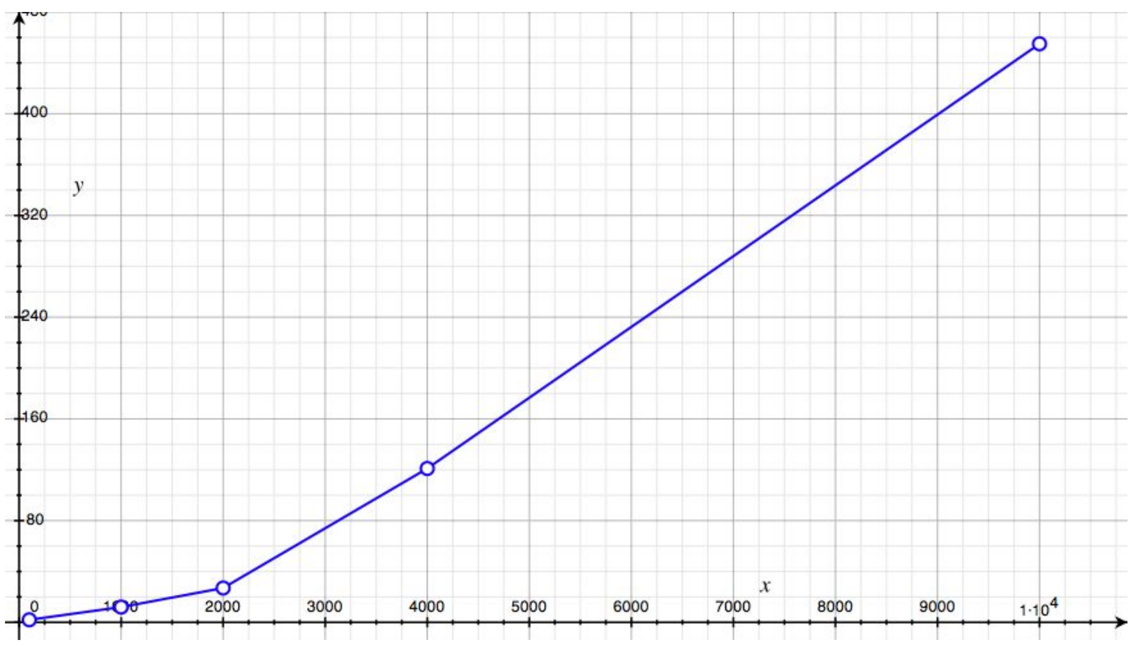
Linear



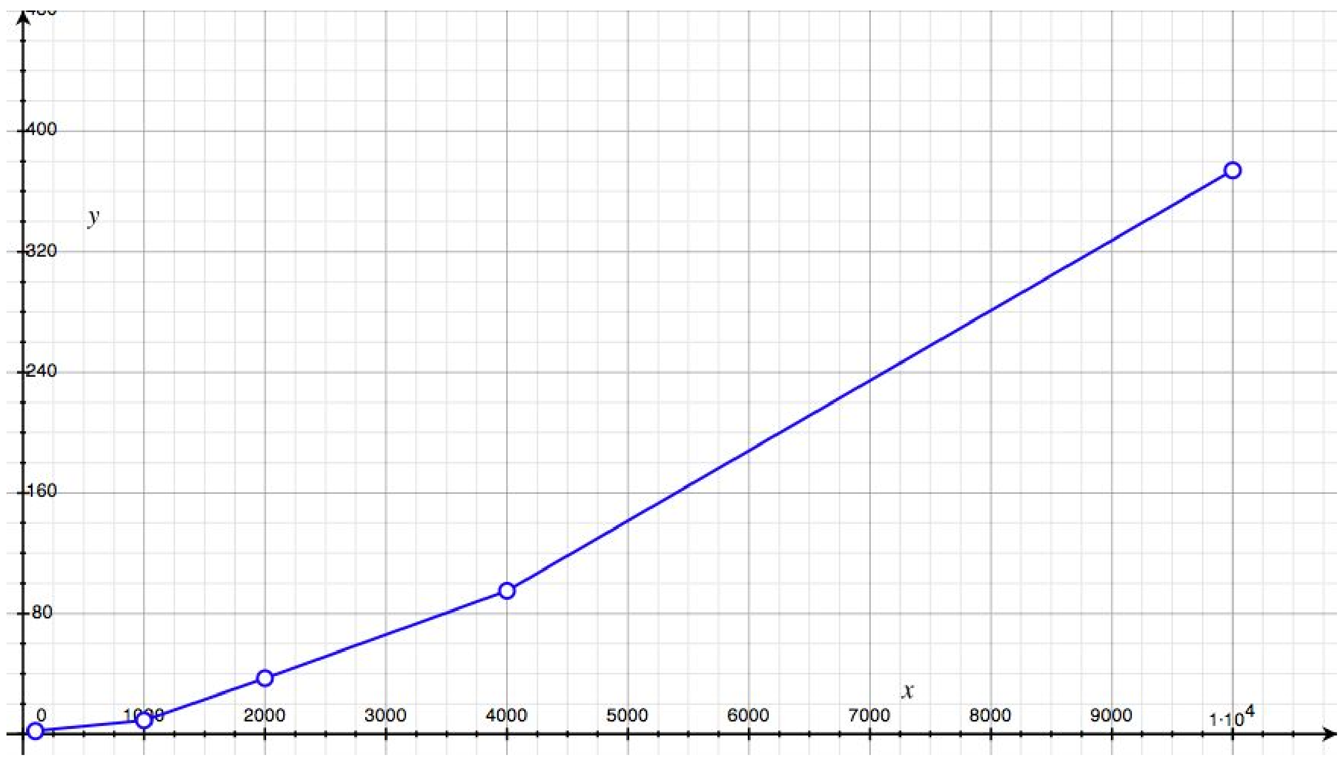
Local Update



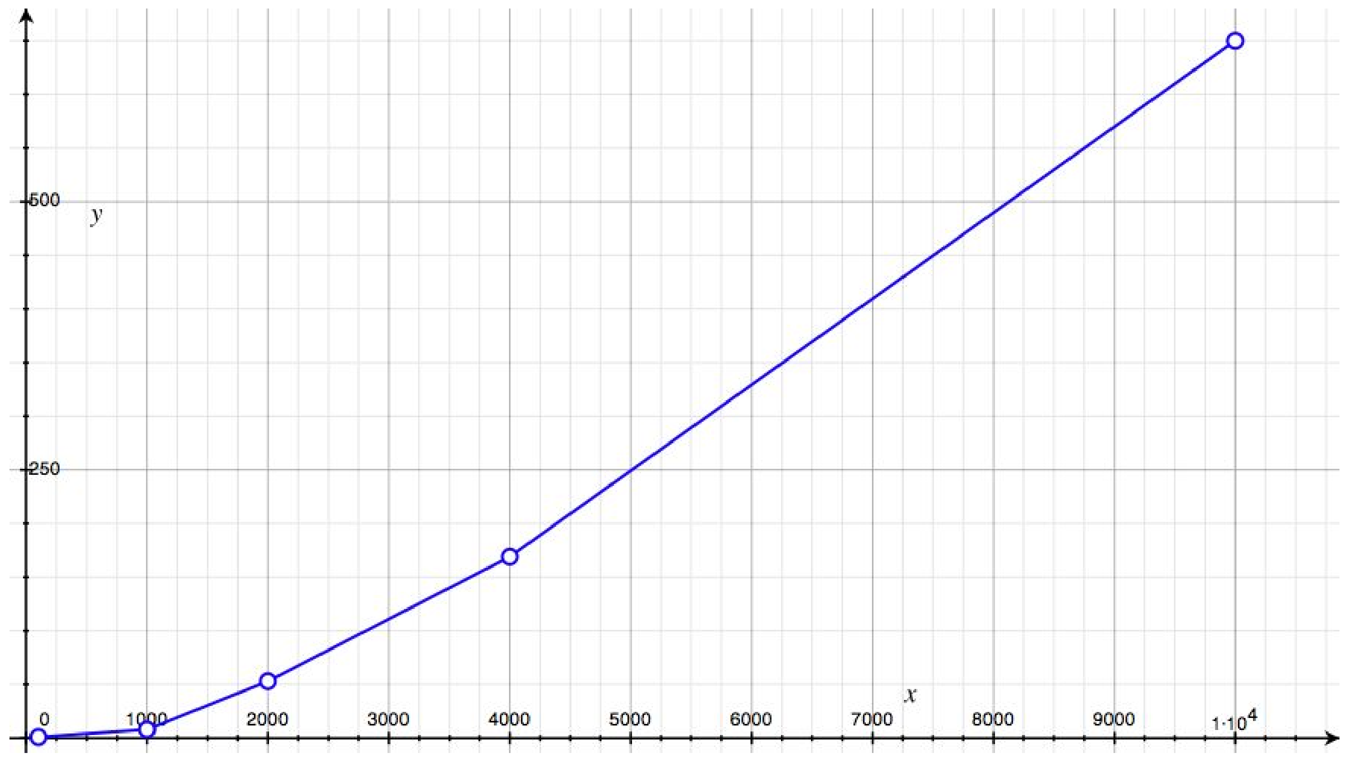
Random



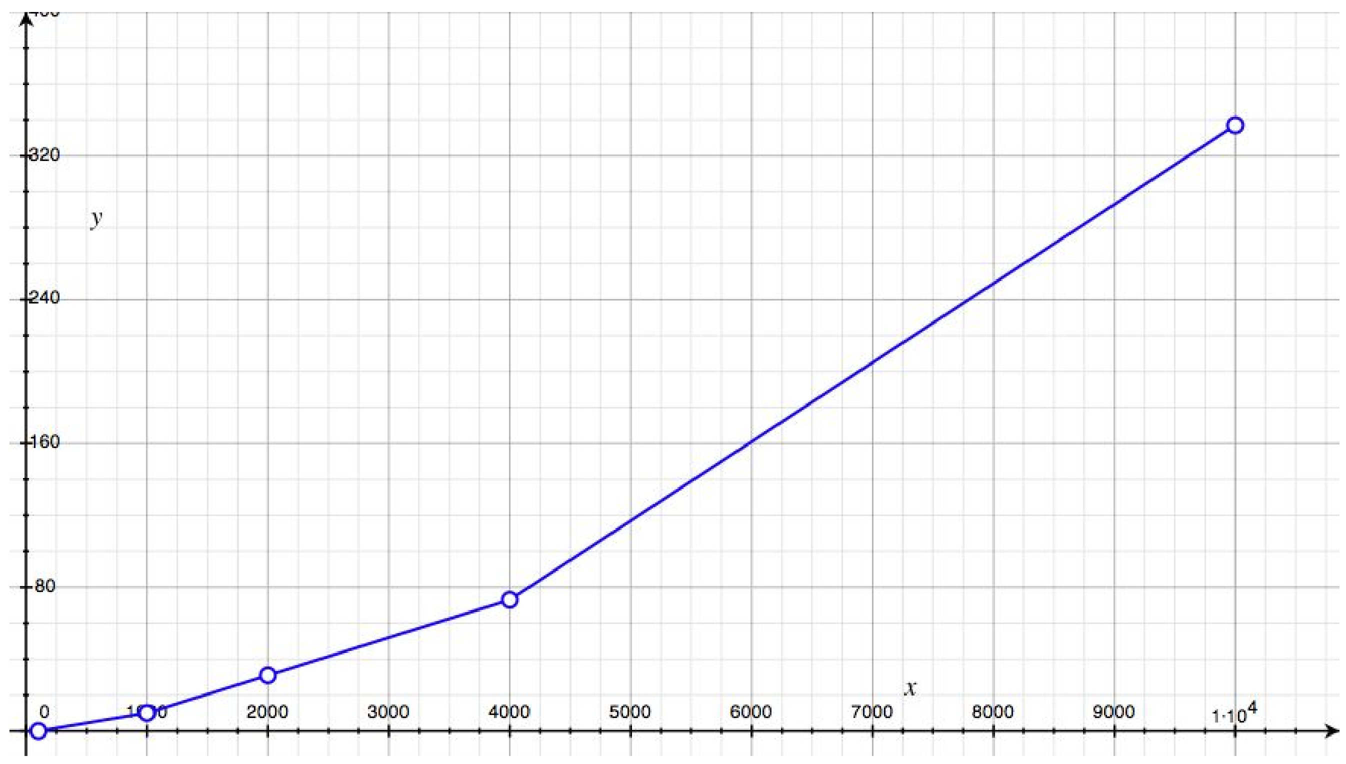
Reverse



Update1



Update2



Conclusions

Input Analysis

The results show that the priority queue performed at a sub par rate. This is largely due to the multiple calls to sort done by this buffer heap. As explained earlier this can and should be changed in order to get a reasonable external memory priority queue. Also the fact that some updates may not push a delete call down the tree is alarming as that degrades the correctness of the buffer heap. This is a relevant issue for the update1, update2 and random input types.

The priority queue performed best on the Local Update operations, with this type of input, the elements are not drastically updated and this is beneficial in a number of ways. The elements of the priority queue do not have to move far up the structure, as they are only slightly modified. Also, the slight change in priority helps since the updates are not split into an insert and delete operation, since it is reasonable to suggest that the update would reach the applicable element's level prior to it needing to insert itself into a previous level.

Other inputs that were relatively fast, were the linear, reverse and update2 input types. The linear was expected to outperform the other input types due to its nature of adding things in sequence, and therefore not requiring to evacuate elements out of there levels. However, the reverse input performed better than the linear input. This was unexpected as elements during this process are expected to be pushed down the structure more often during insertion.

The random input did not have any special formatting to give it an edge over the other input types and caused a significantly slower performance in the buffer heap. However, this type of input should be expected of real world applications and should be a concern for optimization. It is also not obvious on how this can be done as things are done in a fairly sequential manner.

The worst performing input, out of the possible inputs was the update1 operations. This is expected as updates tend to split into two operations and therefore the size of this type of input can be generalized to have a value around a third greater (i.e. an input of this type of size N can be as great as (4/3)*N).

Considerations

There are many issues that were discussed within this paper that cause this implementation to perform slower than needed. To gain performance there are a few issues that need to be resolved. Merging instead of sorting is an essential segment of the buffer heap algorithm and I/O bound analysis and as mentioned earlier is needed in order to increase efficiency.

Mentioned earlier was the need to optimize the priority queue in order to increase performance given random data. A way of doing so, which leads to further optimizations of the other input types, is by dividing up the priority queue into a constant number of smaller priority queues. A good hash function can be used to decide where operations and elements go. A delete-min would call delete-mins on all priority queues in order to determine the minimum element. This decision seems to lead to the same amortized bound and may perform better than the traditional one priority queue structure.

A major way in reducing the I/Os performed by this implementation would be to profile the application given certain parameters and aim to optimize the parameters in order to make the algorithm run faster. This is however only applicable when using this priority queue in a real large scale demanding application. Fine tuning the parameters may ensure better performance; However, may not show a drastic improvement. This last comment was never the less proved false under the default parameters given by STXXL.

STXXL is a generalized library for performing I/O efficient operations in external memory applications. This generality undermines performance, to get better performance out of a data structure, one would need to write the I/O specifics of the structure to suit the needs of the structure. This direction may be able to save a few more I/Os since the algorithm used to get data to and from memory is tailored to suit the needs of the data structure.

In conclusion the priority queue discussed here needs a complete rewrite in order for it to be used in practice. It may be reasonable enough in order to test parameters, but needs work in order to increase its efficiency. Reasonable decisions should also be made in order to fine tune the performance of the structure.

References

1. Alok Aggarwal, Jeffrey SV. The input/output complexity of sorting and related problems. Commun ACM 1988; 31: 1116-1127.

2. Dementiev R, Singler J, and Beckmann A. Standard template library for extra large datasets. 2007; 1.1.0: .

3. Frigo M, Leiserson CE, Prokop H, and Ramachandran S. Cache-oblivious algorithms. Foundations of Computer Science, 1999 40th Annual Symposium on 1999; 285.

4. Meyer U, Zeh N. I/O-efficient undirected shortest paths. Proceedings of the 11th Annual European Symposium on Algorithms 2003; 434-445.

5. Ramachandran V, Chowdhury R. Proceedings of the 16th Annual Symposium on 2004; 245-254.

6. Tong L. Implementation and experimental evaluation of the cache-oblivious buffer heap. 2006; .

7. Yde L. Performance engineering the nth element function. 2002; .