Experimental Study of High Performance Priority Queues

David Lan Roche
Supervising Professor: Vijaya Ramachandran

May 4, 2007

Abstract

The priority queue is a very important and widely used data structure in computer science, with a variety of applications including Dijkstra's Single Source Shortest Path algorithm on sparse graph types. This study presents the experimental results of a variety of priority queues. The focus of the experiments is to measure the speed and performance of highly specialized priority queues in out-of-core and memory intensive situations. The priority queues are run in-core on small input sizes as well as out-of-core using large input sizes and restricted memory. The experiments compare a variety of well-known priority queue implementations such as Binary Heap with highly specialized implementations, such as 4-ary Aligned Heap, Chowdhury and Ramachandran's Auxiliary Buffer Heap, and Fast Binary Heap. The experiments include Cache-Aware as well as Cache-Oblivious priority queues. The results indicate that the high-performance priority queues easily outperform traditional implementations. Also, overall the Auxiliary Buffer Heap has the best performance among the priority queues considered in most in-core and out-of-core situations.

1 Introduction

Due to revolutionary advances in hard disk drive technology over the past ten years, hard drives are larger, faster, and cheaper than ever before. As a result, institutions and corporations are storing massive amounts of information. Major institutions store petabytes of information in data warehouses at low costs. Therefore, scientists are striving to solve problems on massive datasets that reside primarily on hard disks. At the same time, CPU speeds are still many times faster than the speeds of hard disks. Delivering data to the CPU has developed into one of the major bottlenecks in computer hardware performance. Consequently, computer hardware employs a memory hierarchy that places small, fast memory close to the CPU, while larger, slower disk drives are placed further away. Most memory hierarchies contain multiple levels, where data is transferred between levels when needed. Since new hard disk technology is providing the infrastructure for huge datasets, it is crucial that algorithms applied to solve common problems use the memory hierarchy efficiently. On large datasets, the performance of the algorithm can be dominated by accesses to the memory hierarchy. This research examines the priority queue data structure, which is commonly applied to the Single Source Shortest Path problem and Dijkstra's SSSP algorithm on sparse graphs.

This study presents the experimental performance of a variety of high-performance priority queues, including the 4-ary Aligned Heap [6], Chowdhury and Ramachandran's Auxiliary Buffer Heap [2], and Fast Binary Heap [6]. The experiments include well-known priority queues, such as Binary Heap [3], as well as high-performance implementations, such as the Sequence Heap [6]. The goal of the experiments is to measure the performance of the priority queues in situations that are memory intensive and require the priority queue to execute out-of-core. To accomplish this goal, large sequences of priority queue operations are generated using Dijkstra's Single Source Shortest Paths algorithm on sparse graph types. The experiments use the sequences as input where the amount of cache memory available to the priority queues is restricted.

The results of the experiments indicate that the Sequence Heap [6], Auxiliary Buffer Heap [2], and 4-ary Aligned Heap [6] are the best performing

priority queue in the in-core experiments. Auxiliary Buffer Heap, Fast Binary Heap [6], and 4-ary Aligned Heap have the best overall performance in out-of-core experiments among the priority queues included. These three priority queues performed approximately 100 times faster than traditional priority queues such as the Binary Heap. Also, the Auxiliary Buffer Heap has slightly better performance out-of-core than the Fast Binary Heap and 4-ary Aligned Heap.

Before displaying the results, background information is presented over the cache-oblivious model, Dijkstra's algorithm, and the priority queue data structure. Then, we introduce the different priority queues used in the experiments as well as STXXL, the tool used to measure out-of-core performance. In-core and out-of-core experimental results are then presented and analyzed.

Finally, a detailed analysis of Sleator and Tarjan's Splay Heap [8] is presented with emphasis on the challenges of creating efficient decrease-key and delete operations. Since the Splay Tree, which is also self-adjusting, made a huge impact on the binary tree, it will be interesting to see if the Splay Heap, which has a number of open questions, can make an impact on the priority queue.

2 Background

2.1 Memory Hierarchy

Modern computer architecture employs a memory hierarchy in order to deliver data to the CPU in an efficient manner. A typical memory hierarchy has multiple levels where the level closest to the CPU is the smallest and fastest, and each progressive level moving away from the CPU becomes larger, but slower. A typical memory hierarchy has registers closest to the CPU, an on-die Level 1 cache, an on-die Level 2 cache, a Level 3 cache, main memory (RAM), and finally a hard disk. While most memory hierarchies follow this structure, it is important to note that exact sizes and speeds of each level in the memory hierarchy vary significantly between different architectures.

When the CPU requests a piece of data, the architecture searches for it in the memory hierarchy. If the data is not present in the highest parts of the memory hierarchy, a cache miss occurs and the data must be fetched from deeper within the hierarchy. When data is requested from one level of a memory hierarchy to another level, an entire section of data, known as a block, is transferred between the levels. Therefore, levels within the memory hierarchy are comprised of blocks of data. The size of a block is dependent on the architecture. Moving data between levels of the memory hierarchy is very costly compared to the speed of the CPU.

For the purpose of analyzing the priority queues, we assume a two-level memory hierarchy with a fast but limited-size cache and a slow but infinite-size disk. The out-of-core experiments in the study also use a tool that simulates a two-level structure for the memory hierarchy. Using the two-level model, one can predict and measure how many times an algorithm incurs a block transfer. The number of block transfers, or I/O operations, an algorithm uses is a measure of I/O efficiency, or how well the algorithm takes advantage of locality within a given memory hierarchy.

2.2 Cache-Oblivious versus Cache-Aware Algorithms

To reduce the number of cache misses and associated transfers between cache levels, algorithms strive to take advantage of spatial and temporal locality. Spatial locality is the concept that a piece of data has a higher chance of being accessed if a piece of data near it has been accessed. Spatial locality dictates that data that will be accessed consecutively should be stored in the same block within the memory hierarchy. Temporal locality is the idea that data is likely to be accessed if it has been accessed in the recent past. Temporal locality implies that recently requested blocks have higher probability of being requested again, and so should be kept higher in the hierarchy. Performance of an algorithm or application can be reduced significantly if the CPU must stall in order to wait for data to arrive from the memory hierarchy. How efficient an algorithm is with regards to the memory can be measured using the two-level memory hierarchy.

When developing algorithms that are targeted to be I/O efficient, there

are two major models: the Cache-Aware and Cache-Oblivious models. In the cache-aware model, the algorithm possesses information about the memory hierarchy. For example, an algorithm could know the block size, number of blocks in a particular cache, and replacement strategy that the memory hierarchy utilizes. Using the information about the memory hierarchy to maximize locality strategies, cache-aware algorithms possess the strength of being highly optimized for that particular cache structure. However, the cache-aware algorithm may only be optimized for a certain part of a multi-level memory hierarchy, and not for the overall structure. Also, the cache-aware algorithms are not portable due to varying memory attributes of different computer architectures.

Conversely, the cache-oblivious model does not allow the algorithms to have information regarding a specific memory hierarchy. The algorithms are designed to be efficient on any size cache, with any block size. The cache-oblivious design principle makes highly optimized algorithms difficult to achieve, but allows for algorithm design that is more flexible over different architectures. In addition, cache-oblivious algorithms can be I/O efficient for all levels of a multi-level memory hierarchy, instead of specialized for just one level [2]. It is important to note that the performance expected from a cache-oblivious algorithm assumes an optimal cache replacement strategy [2]. The experiments in the study include both cache-aware and cache-oblivious priority queues.

2.3 Priority Queue

The priority queue is an extremely important data structure that has a variety of applications including graph algorithms, operating system schedulers, and router software. A priority queue is an abstract data structure that holds a set of elements and each element has a corresponding key value. Priority queues are built to support finding and removing the element in the set with either the minimum or maximum key value. For all of the experiments, only the min-priority queue is considered. The min-priority queue is required to support the following operations [3]:

Delete-Min(): This operation returns the element with the smallest key value and removes this entry from the priority queue. The key value is commonly returned in addition to the element.

Find-Min(): This operation returns the element with the smallest key value. The key value is commonly returned in addition to the element.

Insert(x,k): This method creates an entry with **k** as the input key value and an associated element **x** and inserts this entry into the priority queue.

In addition to these required operations, a priority queue can optionally support the following operations:

Decrease-Key(x,k): This operation will replace the key value of the entry pointed to by **x** with the value **k** inside the priority queue. The new key value **k** is assumed to be smaller than the key value currently associated with the entry **x** in the priority queue.

Delete(x): This method removes the entry in the priority queue referred to by **x**.

Many common graph algorithms employ these optional operations, including Dijkstra's algorithm. Therefore, the experimental study includes priority queue implementations that support decrease-key and delete as well as implementations that do not include these optional operations.

2.4 Dijkstra's Single Source Shortest Path Algorithm

Although there are many important applications of the priority queue data structure, the task examined in this study is the Single Source Shortest Path problem on a graph containing edges with nonnegative weights. The problem

states that given a graph and a source vertex **s** within that graph, the solution must give the weight of the shortest path, called **d**, through the graph from the source vertex **s** to each vertex **v** in the graph [3].

Dijkstra's algorithm is a very well-known, well-studied algorithm to solve this problem. In Dijkstra's algorithm, the vertices of the graph and their associated shortest paths are stored in a priority queue. The study uses two versions of Dijkstra's algorithm, one where the priority queue employs the decrease-key operation and one where the priority queue uses only delete-min and insert operations. In the version of Dijkstra's algorithm employing decrease-key, called DIJKSTA-DEC [1], vertices are stored in the priority queue with their shortest path from **s** seen so far, **d**, as the key value. If a shorter path is found, the key value **d** is updated using decrease-key until that vertex is removed using a delete-min operation.

In the version of Dijkstra's algorithm that does not use decrease-key, called DIJKSTRA-NODEC [1], vertices are still stored in the priority queue with the shortest path from **s**, i.e. **d**, as the key value. Instead of using the decrease-key operation to update the shortest path for a node, a brand new entry is inserted into the priority queue. When a shorter path from **s** to a vertex **v** is found at a later time, another entry for same vertex **v** is inserted into the priority queue. This strategy results in the priority queue containing multiple entries for the same vertex, but with different key values representing **d**. When the delete-min operation occurs, DIJKSTRA-NODEC [1] must maintain a record of which nodes have already finalized their shortest path from **s** and discard any unnecessary entries leaving the priority queue.

It is important to note that when the graph is sparse, the performance of the priority queue dominates the overall performance of Dijkstra's algorithm [3]. Thus, the faster the amortized running time of the priority queue implementation, the faster the overall performance. Furthermore, the performance of Dijkstra's algorithm on large datasets will be determined by the memory efficiency of the priority queue used.

3 Priority Queues Used in Experimental Study

A number of different priority queues are part of the experimental study. In general, the study contains traditional priority queues, such as the Binary Heap, as well as high-performance priority queues such as the 4-ary Aligned Heap. Each priority queue implementation was included in the study for a reason and has amortized running times and I/O bounds that will help predict their performance. The study introduces the Buffer Heap and Auxiliary Buffer Heap, traditional implementations, as well as high-performance implementations.

3.1 Traditional Priority Queues

3.1.1 Binary Heap

The Binary Heap is a very well-understood implementation of a priority queue. The Binary Heap is intuitive and easily implemented. The Binary Heap supports insert, delete-min, and decrease-key in $\mathcal{O}(\log N)$ worst-case time [3]. The Binary Heap is the benchmark that will be used to measure the performance of the other, more optimized priority queues.

3.1.2 Fibonacci Heap

The Fibonacci Heap is a classic implementation of a priority queue that is commonly used in Dijkstra's algorithm using decrease keys. The Fibonacci Heap supports the delete-min and delete operations in amortized $\mathcal{O}(\log N)$ time while the insert and decrease-key operation executes in amortized $\mathcal{O}(1)$ time [3]. The structure of the Fibonacci Heap can be described as a grouping of unordered min-heap trees that are linked together at the roots. The development of the Fibonacci Heap was motivated by the desire to reduce the asymptotic running time of decrease-key while maintaining the amortized $\mathcal{O}(\log N)$ bound on delete-min because the decrease-key operation is executed many more times than the delete-min operation in most graph types [3]. Because Fibonacci heap was designed with Dijkstra's algorithm in

mind, it is an excellent benchmark in the experiments, especially ones using DIJKSTRA-DEC.

3.1.3 Two Pass Pairing Heap

The Two Pass Pairing Heap is an easier to implement, higher-performance response to the Fibonacci Heap. The structure of the Pairing Heap is similar to the Binomial Heap but contains self-adjusting operations similar to the Splay Heap. The Pairing Heap supports insert and decrease-key operations in $\mathcal{O}(1)$ worst-case time [5]. This implementation also supports delete-min and delete operations in $\mathcal{O}(\log N)$ amortized time. The delete-min operation is designed to move down the heap pairing potential new minimum elements and then move back up the heap connecting these pairs until the new Pairing Heap is formed [5]. Given that the Pairing Heap is a response to the Fibonacci Heap and can also support the decrease-key operation in amortized $\mathcal{O}(1)$ time, the Pairing Heap should be included in experiments with DIJKSTRA-DEC.

3.1.4 Splay Heap

The Splay Heap is a priority queue implementation developed by Sleator and Tarjan [8]. The Splay Heap is described as self-adjusting and is designed to be efficient over a sequence of priority queue operations even though any individual Splay Heap operation could be expensive. The Splay heap does not maintain global properties, and instead performs local operations on nodes to be efficient. The Splay Heap supports the insert and delete-min operations in $\mathcal{O}(\log N)$ amortized running time [8]. The Splay Heap supports a meld operation that merges two Splay Heaps into one large heap, and most operations depend heavily on the meld operation. The meld combines the two right paths of the input trees into the left path of the result heap, while maintaining heap order [7]. The Splay Heap was chosen due to its focus on efficiency in an amortized sense, the open questions available, and its ease of implementation. Later in the study, the decrease-key and delete operations of the Splay Heap are analyzed.

3.2 Buffer Heap and Auxiliary Buffer Heap

The Buffer Heap and the Auxiliary Buffer Heap are cache-oblivious priority queue implementations [2]. The Buffer Heap fully supports insert, delete-min, delete, and decrease-key operations. The amortized running time of insert, delete-min, delete, and decrease-key operations is $\mathcal{O}(\log N)$ [2]. The Buffer Heap is very cache efficient and can support these operations with $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$ amortized number of block transfers where B is the block size, M is the size of the cache, and N is the maximum number of elements in the queue [2]. The structure of the Buffer Heap is based on two buffers, an update (U) buffer and an element (B) buffer that is split into different levels. Operations enter the U buffers and are not applied until a delete-min is executed, where all of the operations are then processed. The deeper levels contain updates and elements that are not likely to be referenced soon, and are stored on disk [2].

The Auxiliary Buffer Heap is similar to the Buffer Heap in structure, yet the Auxiliary Buffer Heap only supports the insert and delete-min operations [2]. The amortized and block transfer bounds are the same as the buffer heap, but the implementation of the Auxiliary Buffer Heap is more efficient because it does not support the decrease-key operation. Stripping away the unnecessary operations allows the Auxiliary Buffer Heap to become very high performance while remaining cache efficient.

3.3 High-Performance Priority Queue Software

3.3.1 4-ary Aligned Heap

The 4-ary Aligned Heap is a priority queue implementation that was designed by LaMarca and Ladner [6]. The implementation used in the experimental study is an optimized version of the 4-ary Aligned Heap from Peter Sanders [6]. The 4-ary Aligned Heap is an array-based priority queue implementation where data is aligned to the cache blocks. This requires that the algorithm know the size of the blocks on the architecture that the implementation will execute on, making this a cache-aware algorithm. The 4-ary Aligned Heap

is not affected by the size of the cache, but by the size of the block. The cache-aware property ensures that any access to an entry in the priority queue can incur at most only one block transfer. The 4-ary Aligned Heap's insert and delete-min operations both run in worst-case $\mathcal{O}(\log_4 N)$ time [6] with $\mathcal{O}(\log_4 N)$ amortized number of block transfers, but the 4-ary Aligned Heap does not support decrease-key. This priority queue will be excellent competition to memory efficient implementations such as the cache-oblivious Auxiliary Buffer Heap.

3.3.2 Fast Binary Heap

The Fast Binary Heap obtained from Peter Sanders is a highly optimized version of the Binary Heap [6]. This Fast Binary Heap has amortized running time of $\mathcal{O}(\log N)$ for the insert and delete-min operations [6]. The Fast Binary Heap has $\mathcal{O}(\log N)$ amortized number of block transfers for both operations. The optimizations reduce the number of conditional branches and memory accesses that the delete-min operation executes. These techniques greatly reduce cache misses and improve the performance of individual delete-min operations. This highly optimized Fast Binary Heap will be another high-performance priority queue in the study. The Fast Binary Heap does not support decrease-key or delete operations, further increasing its performance [6].

3.3.3 Sequence Heap

The Sequence Heap is a highly-optimized cache-aware priority queue implementation developed by Peter Sanders [6]. Sequence Heap is designed to use **k**-way merging techniques to implement a priority queue. The Sequence Heap maintains an insertion buffer of the most recently inserted elements that has to be smaller than the cache being optimized for. When the insertion buffer is too full, the buffer is sorted and sent to external memory. When needed, the sorted sequences combine using **k**-way merging and the smallest key values are sent to a deletion buffer for quick delete-min operations [6]. The value **k** is changed depending on the architecture, making this implementation cache-aware. Therefore, the Sequence Heap can be very I/O

efficient because of the insertion buffer and a correctly chosen **k** value for the architecture. The **k** value must be less than the size of the fast memory available to the architecture. Sanders states that **k**=128 is a good value for most current architectures and is used in the experimental studies [6]. The Sequence Heap also does not support decrease-key.

4 STXXL

In order to measure the performance of the priority queues in out-of-core, memory intensive situations, the study needs a tool that can handle large datasets and provide the ability to restrict the size of cache memory available to the priority queues. The STXXL interface [4] is the tool that allows the experiments to test the out-of-core performance of the priority queues. STXXL allows for the use of extremely large datasets, provides more efficient I/O operations, and provides cache and memory statistics along with timing results [4]. STXXL is an extension of the C++ Standard Template Library and interfaces with C++ in a similar fashion as STL. STXXL generates a virtual two-level memory structure with a limited-size, but quick cache and an unlimited-sized, but slow hard disk [4]. The cache and the hard disk are partitioned into fixed sized blocks. The cache is fully associative and the cache utilizes a Least Recently Used (LRU) paging strategy [4]. The user can specify the size of the blocks as well as the capacity of cache by passing parameters into the STXXL container classes.

4.1 STXXL Interface

STXXL is accessed through a variety of container classes representing common data structures. The out-of-core implementations in the experimental study are created using the STXXL container Vector. The STXXL Vector functions exactly like the STL Vector and very similar to a primitive array data structure. The STXXL vector has a flat structure, is indexed using the [] operator like an array, and can be resized dynamically [4].

However, STXXL has a major restriction; only primitive data types can

be stored in an STXXL Vector. The STXXL Vector container cannot store objects, structures, or pointers of any kind. Therefore, the priority queue implementations are forced to be constructed without these programmer friendly constructs. This caused some very interesting implementation challenges for the Buffer Heap, 4-ary Aligned Heap, and Fast Binary Heap.

4.2 STXXL Implementation Challenges

All of the priority queue implementations used in the out-of-core experiments have an STXXL vector as the data structure that holds the priority queue entries. The STXXL vector allows the size of the priority queue to become very large while the amount of memory available to the priority queue can be limited. However, the restriction that only primitive data types are allowed in the STXXL containers created a tough implementation challenge for the Buffer Heap [9], 4-ary Aligned Heap [6], and Fast Binary Heap [6].

Converting the Buffer Heap to be compatible with the STXXL Vector container class was a complex task. The Buffer Heap already had a relatively complex implementation. To represent the entries in the Buffer Heap, a C++ struct was used [9]. The fields inside the struct, which included such information as key value, time stamp, and element id, varied depending on if the entry was in update buffer or element buffer. The core data structure of the entire buffer heap was a large array of pointers to these structs representing entries. The update and element buffer entries were interleaved throughout the core array by level. Thus, the array would start with the first level update buffer, followed by the first level element buffer, and then move to the next level throughout the array, continuing the interleaving technique [9]. Therefore, indexing into the proper entry required finding the correct buffer type, correct buffer level, correct entry in the buffer, and finally the field needed. The struct that represented an entry in the heap needed to be eliminated in order to support STXXL.

The entry struct was emulated by manually performing the actions of a C++ struct within all the methods of the Buffer Heap. Therefore, the core array was changed to an array of primitives. The idea is that the fields originally stored in the entry struct are now stored directly in the core STXXL

array. Since all of the data is stored in one core array, this STXXL array is extremely large. Global offsets are available for each field of the old entry struct. The implementation challenge came in how to properly index a desired piece of data in the heap. Since each entry struct has a fixed size, called REC-SIZE, entries of the Buffer Heap now have to be aligned on a multiple of REC-SIZE. When traversing or indexing into the array, all original methods that moved by increments of one have to now move in increments of REC-SIZE. This change implies that any method that accessed the core array in the original design had to be changed to fit this new backbone of the Buffer Heap. Since the implementation was complex, the data structure referenced the primary array in a number of different ways, which made converting the Buffer Heap to STXXL particularly challenging.

The 4-ary Aligned Heap [6] and Fast Binary Heap [6] were also difficult to convert into STXXL versions. The core data structure of 4-ary Aligned Heap and Fast Binary Heap is already an array, so these priority queues were easier to convert to STXXL than the Buffer Heap. However, the loop unrolling techniques used in the implementations were not compatible. These optimizations rely heavily on macros and generic programming techniques that STXXL does not support.

5 Experimental Results

5.1 Input Graphs and Hardware

The inputs to the experiments consist of two major types. The first input type is a graph representation that is given to the two versions of Dijkstra's algorithm using different priority queue implementations. $\mathcal{G}_{n,m}$ graphs are the most common graph type used as input into Dijkstra's algorithm. A $\mathcal{G}_{n,m}$ graph in a graph type where N = the number of vertices and M = the number of edges in the graph. The edges of the graph are randomly placed between the different vertices with a non-negative edge weight. The graphs are generated so that the graph is one large connected component. The experiments also include graph types such as d-ary and grid graphs.

In addition, the priority queues are run individually on sequences of priority queue operations. The sequences are generated by running the desired version of Dijkstra's algorithm and printing out the operations and what their inputs are. The sequence is then read in at a later time and the priority queues are run independently. This removes any overhead that might be incurred by the Dijkstra's algorithm implementations. The priority queue implementations execute on sequences that both contain and omit the decrease-key operation.

The implementations that comprise the experiments come from a variety of sources. I developed the implementations of Splay Heap, both with and without decrease-key. The STXXL compatible versions of the Buffer Heap [2], part of the Auxiliary Buffer Heap [2], 4-ary Aligned Heap [6], and Fast Binary Heap [6] were also developed by me. Lingling Tong is responsible for in-core version of the Buffer Heap [9]. Rezaul Chowdhury developed the Binary Heap, Fibonacci Heap, and Pairing Heap implementations used in the experiments [1]. The in-core implementations of the Sequence Heap [6], 4-ary Aligned Heap [6], and Fast Binary Heap [6] were obtained from publicly available code by Peter Sanders. The DIMACS solver was obtained from the DIMACS challenge website.

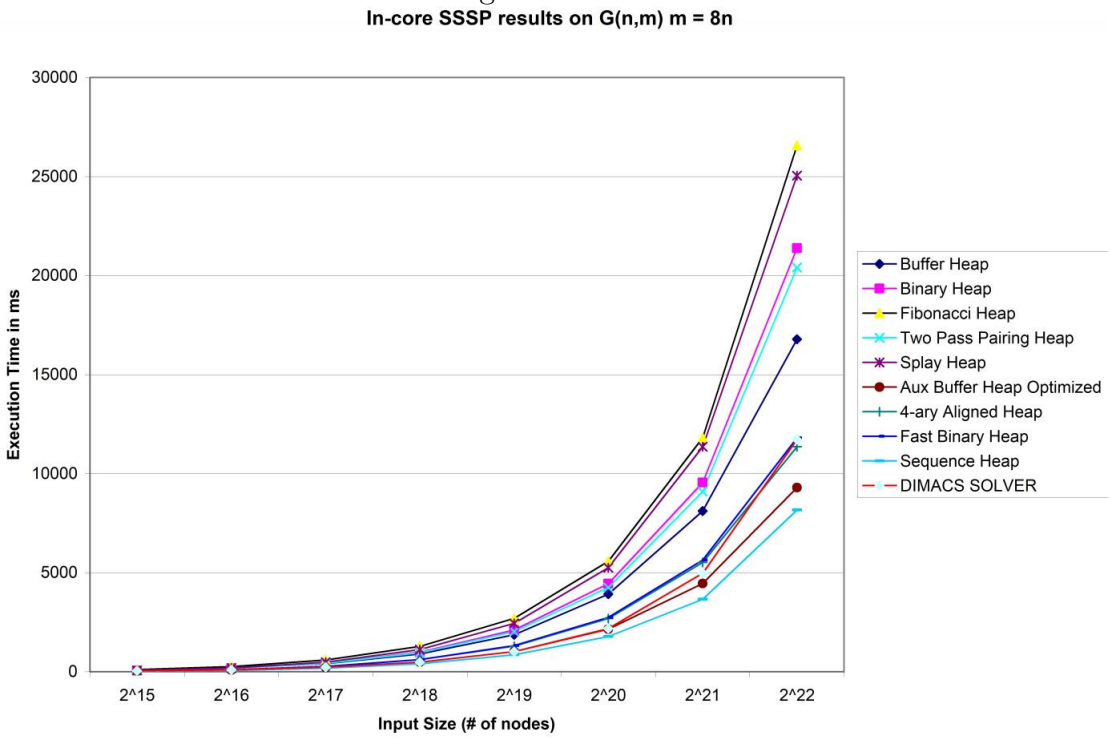
The computer architecture that the experiments run on is a dual processor 3.06GHz Intel Xeon shared memory machine with 4GB of RAM. Each processor's memory hierarchy contains an 8KB Level 1 data cache and an on-die 512KB Level 2 cache with a block size of 64 bytes. This memory hierarchy will be used during the in-core experiments while the out-of-core experiments will use the two-level memory hierarchy of STXXL. In both cases, the cache-aware algorithms, namely Sequence Heap and 4-ary Aligned Heap, are provided with the size of the block or the size of the cache, whichever is necessary for the algorithm. In addition, a 73.5 GB hard drive supports the virtual disk necessary for the STXXL experiments.

5.2 In-Core Results

The experiments run in this section contain in-core versions of the priority queue implementations being tested. Thus, the in-core implementations were

not modified for STXXL in any way. It is expected that the I/O efficiency of the priority queues will not have as great an impact when the input size is small. The cache-aware algorithms are given the block size of 64 bytes from the Intel Xeon L2 cache. Figure 1 shows the in-core execution times of various priority queue implementations being used in Dijkstra's algorithm on a $\mathcal{G}_{n,m}$ graph where m = 8n. This in-core experiment contains an Auxiliary Buffer

Figure 1:

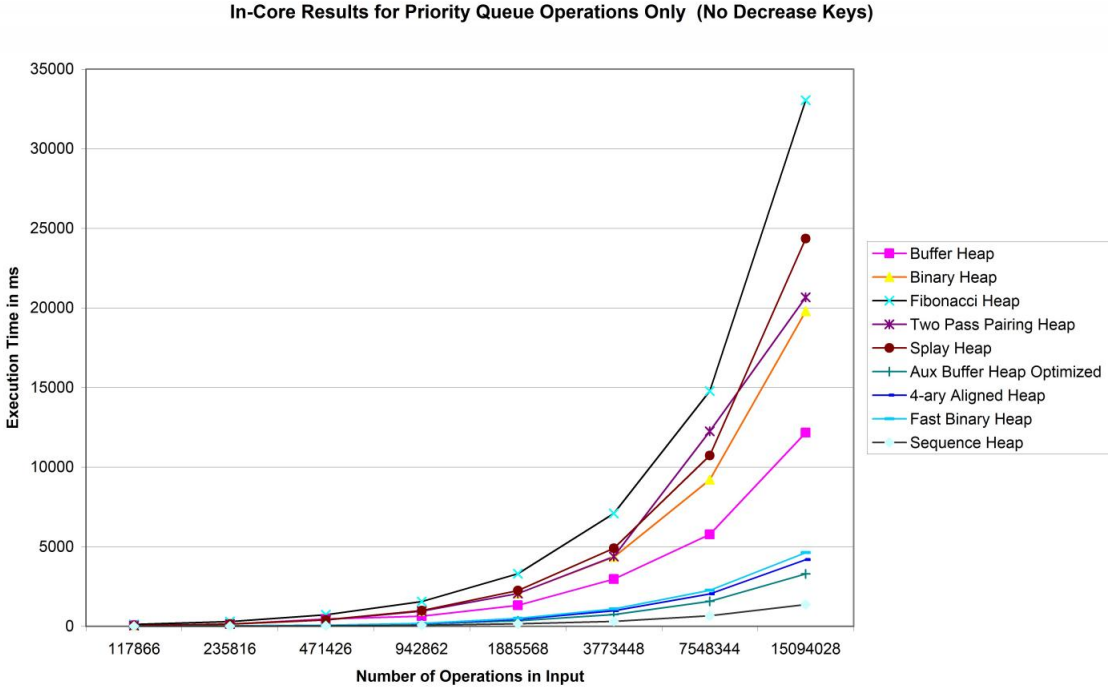


Heap harboring optimizations that do not exist in the out-of-core version, but are similar to optimizations in 4-ary aligned heap and sequence heap. A good benchmark for in-core performance in this experiment is the DIMACS challenge solver, which is touted to be very high performance on the integer data type.

The Sequence Heap [6] is the highest performing priority queue implementation in-core followed by the Auxiliary Buffer Heap [2]. Both of these priority queues perform faster than the DIMACS challenge solver. The three slowest

priority queues include the traditional designs: Fibonacci Heap, Splay Heap [8], and Binary Heap. This implies the traditional priority queue implementations are significantly slower than newer designs like the cache-oblivious Auxiliary Buffer Heap and the cache-aware Sequence Heap. Using the DIMACS challenge solver as a metric for performance, the 4-ary Aligned Heap, Fast Binary Heap, Sequence Heap, and Auxiliary Buffer Heap can be considered high-performance priority queues. One factor that improves the performance of these 4 priority queues is they do not support decrease-key. To help distinguish between which priority queues have higher performance in-core, the priority implementations were run on a sequence of priority queue operations only (Figure 2).

Figure 2:



Removing the potential overhead from Dijkstra's algorithm clearly groups the priority queue implementation into two distinct groups. The highest performing designs, Sequence Heap, Auxiliary Buffer Heap, 4-ary Aligned Heap, and Fast Binary Heap (in that order), perform on average 7.25 times faster than the slower designs, Fibonacci Heap, Splay Heap, Two Pass Pairing

Heap, and Binary Heap. This reinforces the concept that modern algorithm designs must consider the memory hierarchy in order to be efficient in practice for sufficiently large input sizes. The Sequence Heap is the fastest, being twice as fast as its nearest competitor, the Auxiliary Buffer Heap.

5.3 Out-of-Core Results using STXXL

The out-of-core experimental results test the cache and memory efficiency of some of the high-performance priority queues in the study. STXXL is used to measure the I/O efficiency of the different priority queue designs. Due to the resource issues of working with large datasets, the experiments were easier to execute by taking medium sized inputs and using STXXL to restrict the amount of fast memory available. Therefore, the out-of-core experiments take as input graphs of similar size as the in-core graphs, but the priority queues are restricted to a small-sized cache.

To see how the high performing priority queues in-core compare to the traditional implementations in memory intensive situations, the Binary Heap, Fibonacci Heap, and Pairing Heap [1] were tested using Dijkstra's Algorithm against the Auxiliary Buffer Heap [2] on the same input size where the cache size was less than four times the number of vertices in the input graph. This sized cache forces the majority of the priority queue to reside on the STXXL disk. Figure 3 shows how many times slower each traditional priority queue is compared to the Auxiliary Buffer Heap as well as how many times more block transfers these implementations incurred.

Figure 3:

Priority Queue	Number of Times Slower than Auxiliary Buffer Heap	Number of Times more Block Transfers Incurred than Auxiliary Buffer Heap
Binary Heap (No Decrease-Key)	115	237
Fibonacci Heap (Decrease-Key)	344	451
Two Pass Pairing Heap (Decrease-Key)	264	347

Each priority queue was run in the version of Dijkstra's algorithm stated next to their name. The slow performance of the Binary Heap could be attributed to Binary Heap supporting the optional decrease-key operation and Binary Heap ignoring the memory hierarchy. However, the Fibonacci Heap and Pairing Heap using decrease keys have low performance compared to the Auxiliary Buffer Heap due to these two implementations not considering the memory hierarchy. These results indicate that there is a major performance difference between the traditional and the newer priority queue implementations when the memory hierarchy is involved.

Next, the experiment focused on which of the high-performance priority queues, the cache-oblivious Auxiliary Buffer Heap [2], Fast Binary Heap [6], or cache-aware 4-ary Aligned Heap [6] has the best memory efficiency. The Sequence Heap was omitted from the out-of-core experiments due to the complexity of the implementation and the data structure being incompatible with STXXL. In each experiment, the cache-aware 4-ary Aligned Heap is passed the size of the block used in the STXXL experiment, usually 4 kilobytes. Figure 4 shows the block reads, block writes and I/O time of the three

Figure 4:

Number of Blocks Read from Disk

Number of Operations	3773448	7548344	15094028	30193274
4-ary Aligned Heap	31200	1577896	4068262	8818504
Fast Binary Heap	30097	1881611	4660437	10025314
Auxiliary Buffer Heap	29298	95961	273528	697080

Number of Blocks Written to Disk

Number of Operations	3773448	7548344	15094028	30193274
4-ary Aligned Heap	32838	1583220	4080957	8845942
Fast Binary Heap	33783	1888983	4675180	10054800
Auxiliary Buffer Heap	35057	107474	296564	743167

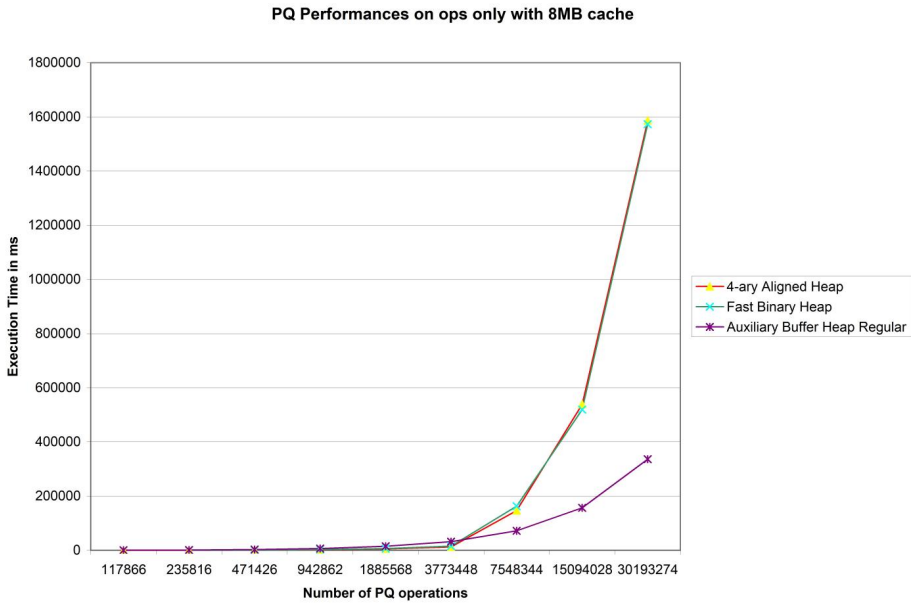
I/O Wait Time Incurred in ms

Number of Operations	3773448	7548344	15094028	30193274
4-ary Aligned Heap	1273	101458	432457	1366528
Fast Binary Heap	1800	108502	395393	1303031
Auxiliary Buffer Heap	1222	4599	14798	34102

priority queues on a sequence of delete-min and insert operations where the implementations were only allowed an 8MB cache. The input sequence was generated using DIJKSTRA-NODEC [1] on a $\mathcal{G}_{n,m}$ graph where m = 8n.

As the number of operations that a priority queue executes increases, the 4-ary Aligned Heap and Fast Binary Heap incur many more block transfers than the Auxiliary Buffer Heap. At the largest input size, the 4-ary Aligned Heap creates 12 times more block read and block writes than the Auxiliary Buffer Heap. This results in the 4-ary Aligned Heap incurring approximately 40 times more I/O wait time than the Auxiliary Buffer Heap. This is an expected result because the theoritcal bounds predict the 4-ary Aligned Heap will create $\mathcal{O}(\log N)$ block transfers and the Auxiliary Buffer Heap will create $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$. This predicts a B times improvement for the Auxiliary Buffer Heap. Since each B=4096 bytes and each block can hold 64 priority queue entries in this experiment, the number of block transfers is expected to be 64 times less for Auxiliary Buffer Heap. However, due to overhead in the implementations and the optimizations provided by Sanders [6], the speed is reduced by a factor of 5, to approximately 12 times, which is what is seen in figure 4.

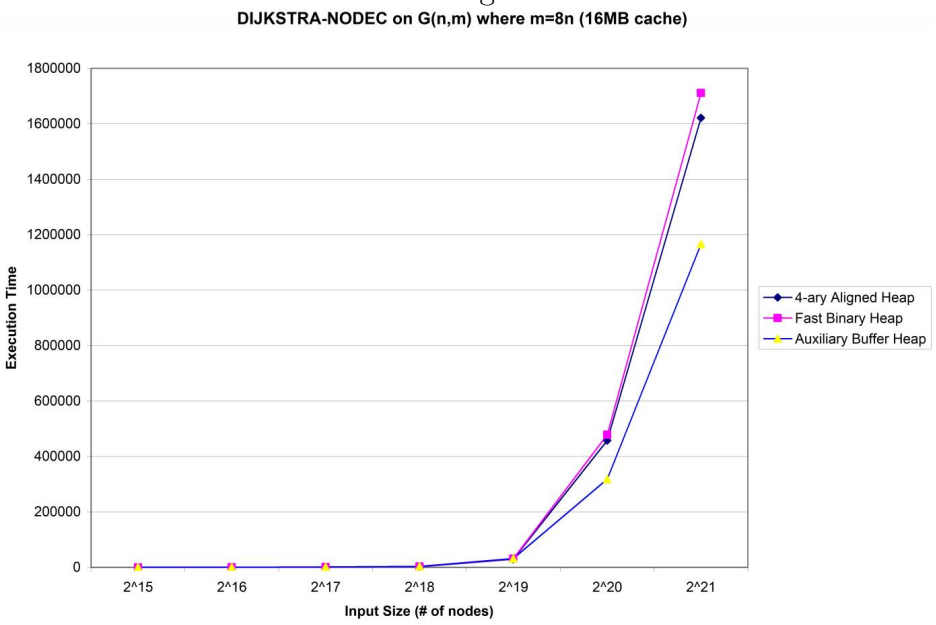
Figure 5:



The possible explanation for the 12 times decrease in block transfers resulting in 40 times better I/O wait performance is that Auxiliary Buffer Heap behaves like a stack. In general, elements are both added and removed from the top update and element buffers. The stack like structure will result in block transfers being read and written to the same tracks on the hard disk. Since the Auxiliary Buffer Heap reads and writes to the same tracks on the hard disk, this results in much less seek time. The 4-ary Aligned Heap and Fast Binary Heap, which are array based implementations, have accesses that can span the entire physical structure of the implementation, resulting in more data being stored on different tracks on the hard disk.

Figure 5 shows the execution times of the three priority queues on this same sequence of priority queue operations. The data shows that when the input size was small, the 4-ary Aligned Heap was the fastest. However, as the input size grew and the priority queue size grew, the I/O wait time began to dominate running time. In the largest inputs, the Auxiliary Buffer Heap performed approximately 4.5 times faster than the other two implementations. This experiment suggests the Auxiliary Buffer Heap is the most efficient with regards to the memory hierarchy.

Figure 6:



However, Figure 6 shows that the performance advantage of the Auxiliary Buffer Heap is not as dramatic when run on DIJKSTRA-NODEC [1] with a 16MB cache.

In this experiment, the Auxiliary Buffer Heap was only about 80 percent faster than the 4-ary Aligned Heap. However, the smaller difference in performance of DIJKSTRA-NODEC in this case could be attributed to the bookkeeping associated with DIJKSTRA-NODEC. Given other experimental results, decreasing the cache size would allow the priority queue implementations to wield more influence over the execution time.

5.4 Summary of Results

- Using the DIMACS solver as a benchmark for in-core performance of Dijkstra's Algorithm, the Sequence Heap [6], Auxiliary Buffer Heap [2], 4-ary Aligned Heap [6], and Fast Binary Heap [6] are all very high-performance. Traditional implementations performed worse than the DIMACS solver. (Figure 1)

- The raw in-core performance of Sequence Heap and other high-performance priority queues are on average 7.25 times fast than traditional implementations. (Figure 2)

- With traditional implementations running approximately 100 times (Figure 3) slower than the high-performance priority queues, the results indicate the cache-aware and cache-oblivious strategies in the high-performance priority queues contribute significantly to out-of-core performance.

- Auxiliary Buffer Heap is faster than Fast Binary Heap and 4-ary Aligned Heap in large, out-of-core input sizes. (Figure 5)

- When DIJKSTRA-NODEC is run out-of-core, the performance difference between the highly specialized priority queues shrinks due to overhead involved with DIJKSTRA-NODEC. (Figure 6)

6 Splay Heap Analysis

The Splay Tree made a huge impact with regards to the binary tree data structure. Based on the same principles as the Splay Tree, the Splay Heap is a self-adjusting priority queue by Sleator and Tarjan [7]. The Splay Heap strives to be simple to implement while being efficient over a sequence of operations. The original Splay Heap does not support the decrease-key or delete operations. While Sleator and Tarjan do propose one method to support the optional operations, no formal analysis of the optional operations is presented [7]. However, the question is open if the strategy presented is the most efficient way to implement decrease-key and delete. To start, we will examine the Splay Heap's most important operation, the meld, and prove its correctness. In addition, the strategy purposed by Sleator and Tarjan to implement the decrease-key and delete operations will be examined [7]. Also, potential strategies for improving the optional operations are discussed.

6.1 Splay Heap Structure and Meld Operation

The Splay Heap has a structure similar to the binary tree. Nodes in the heap store the element and key value and can have up to two children. There exists a pointer to the root of the tree. Similar to the Binary Heap, the Splay Heap must maintain the heap property, where a node's key value must be less than the key value of its children [7]. This structure results in the minimum element being placed at the root of the tree [7].

The original Splay Heap supports the standard delete-min and insert operations in $\mathcal{O}(\log N)$ amortized running time [8]. In addition, the Splay Heap supports a meld operation in $\mathcal{O}(\log N)$ amortized running time [8]. The meld operation takes as input the roots to two disjoint Splay Heaps and combines them into one Splay Heap. Most operations in the Splay Heap depend heavily on the meld operation [7]. The Splay Heap inserts new entries by creating a Splay Heap with one node containing the input information and then uses meld on the singleton heap and the Splay Heap. The Splay Heap deletes the minimum element by melding the two children of the root followed by eliminating the root node [7].

The meld operation is the most important operation in the Splay Heap design. The meld operation takes as input two disjoint Splay Heaps and returns a pointer to the root of the newly formed Splay Heap. Melding follows the general strategy of combining the two right paths of the input trees in heap order [7]. To reduce the amortized complexity, the children along the path are swapped to reduce the length of the right paths. The result heap has the far left path as the merger of the two right paths of the original heaps, maintaining heap order. Sleator and Tarjan prove the amortized running time of the meld operation [7] and therefore prove the amortized running time of the delete-min and insert operations [7]. However, Sleator and Tarjan do not provide a full proof of correctness for the meld operation. I provide a full proof of correctness of the meld operation in Appendix B. This exercise is important to understanding the self-adjusting strategy and the properties that the Splay Heap must maintain throughout a meld, which was helpful in thinking about ways to change the decrease-key and delete operations.

6.2 Decrease-Key and Delete

Sleator and Tarjan present a strategy for allowing the Splay Heap to support the decrease-key and delete operation. [7] The Splay Heap structure changes by adding a parent pointer to each node. Thus, the parent of node X points to the node Y such that CHILD(Y) == X. The root nodes parent is null. Now, the Splay Heap can delete a node X inside the heap by using the parent pointer to attach the result of melding the children of X to node X's parent. The node can be removed and the heap property is maintained [7]. The Splay Heap can now use the decrease-key operation by using the delete and insert operations. The strategy is to delete the node with the old key and insert a new node with the same element, but the updated key value [7]. The delete operation takes $\mathcal{O}(\log N)$ amortized running time while decrease-key takes two $\mathcal{O}(\log N)$ operations to complete. While the amortized running time of decrease-key is still $\mathcal{O}(\log N)$, executing two operations could hurt real-life performance.

Using this strategy by Sleator and Tarjan, I created an STXXL compatible implementation of Splay Heap that supports decrease-key and delete. The pseudo-code of this implementation is available in Appendix A.

Figure 7:

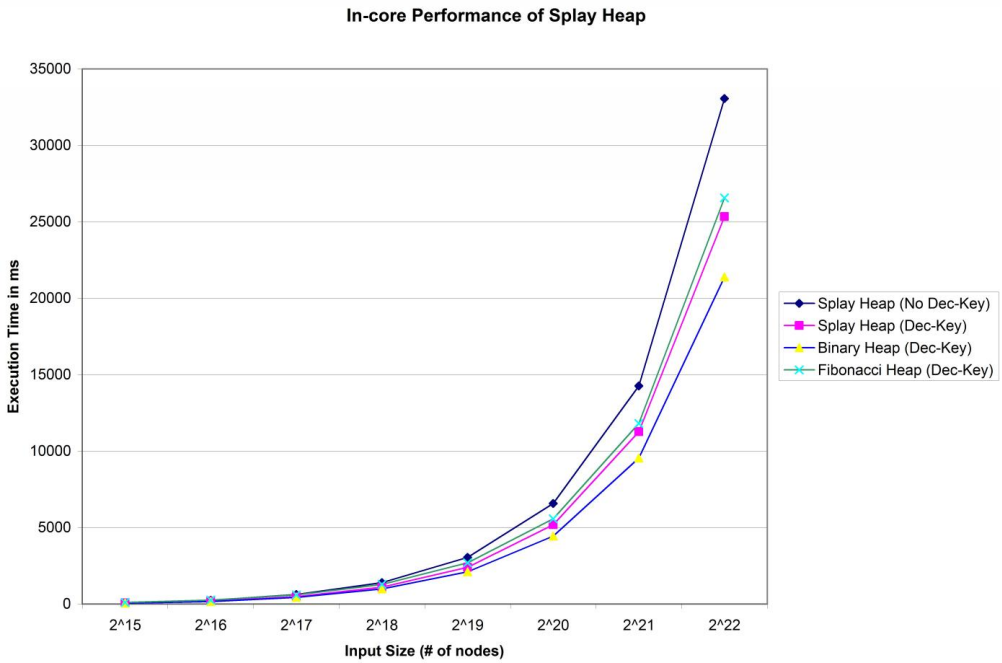


Figure 7 shows the in-core performance of the Splay Heap run on DIJKSTRA-DEC [1], Splay Heap run on DIJKSTRA-NODEC [1], Fibonacci Heap, and the Binary Heap. Fibonacci Heap and Binary Heap are both using DIJKSTRA-DEC. The addition of the decrease-key and delete operation improves performance in this experiment by 30 percent as the input size increased. However, both versions of the Splay Heap did not perform better than the Binary Heap using decrease-keys. Since Splay Heap using decrease-key performed better than Splay Heap without, there is practical application for continuing to improve the decrease-key and delete operation.

6.3 Future Work/ Improving Decrease-Key and Delete

With the Splay Heap supporting the decrease-key purposed by Sleator and Tarjan [7] showing an improvement over the Splay Heap without decrease-keys, it is apparent that adding a more efficient decrease-key or delete method could greatly improve the practical performance of the Splay Heap. It is

important that the idea of a self-adjusting data structure continue to be preserved with new decrease-key ideas. Maintaining global information about the heap may not be feasible for the Splay Heap.

Since the Splay Heap has a similar structure to the Binary Heap, one possible strategy for implementing decrease-key is creating an operation similar to Heapify for the Splay Heap [3]. This Heapify method would move the node with the new key up the heap until the heap property is restored. While this method would remove the redundant delete and insert operations, the worst-case running time for this decrease-key operation could be as bad as $\mathcal{O}(N)$ if the tree becomes a linked list [3].

Another idea to improve the decrease-key is to design an insert method specifically for the decrease-key operation. The new insert method could use the knowledge about the old key and the new key to help improve the speed of the insert. Also, using a meld to insert one node ensures that the new node will be inserted along the left path of the main heap. If the left path is extremely long, the insert could waste time moving in a top-down fashion. One idea is to take the difference between the old and new key and then make a decision to meld in a top-down fashion or move in a bottom-up direction. This strategy could be successful in a min-heap where the keys are non-negative, because then a node can guess how close it is to being the minimum.

There are ways for improving the practical performance of the decrease-key operation and these ideas should be tested in practice where applicable. The question is still open if there is a way to achieve a better amortized running time for the decrease-key operation.

7 Conclusion

The need for more memory efficient algorithms is crucial because of the increasing size of data sets in industry and the increasing speed disparity between CPUs and hard disks. This experimental study focused on the out-of-core performance of a variety of high-performance priority queues in addition to well-known priority queue implementations. The priority queues were

tested sequences of priority queue operations using Dijkstra's Algorithm solving the Single Source Shortest Paths problem. Some of the priority queues did not consider the memory hierarchy, while other implementations were cache-oblivious or cache-aware. The study used STXXL to measure the I/O efficiency of the priority queues. Converting the Buffer Heap, 4-ary Aligned Heap, and Fast Binary Heap to STXXL format was an interesting implementation challenge.

The results indicate that overall, the priority queues that do not attempt to support the optional decrease-key operation perform the fastest in both in-core and out-of-core experiments. The Sequence Heap was the fastest priority queue in-core while the Auxiliary Buffer Heap had the highest performance out-of-core followed closely by the 4-ary Aligned Heap and Fast Binary Heap. The traditional priority queues, such as Binary Heap and Fibonacci Heap, perform very poor compared to these high-performance priority queues, especially in memory intensive situations. The major difference in performance is likely due to the traditional heaps not considering the memory hierarchy in their designs.

In addition, the Splay Heap is an interesting priority queue implementation that has potential for improvement. The correctness of the crucial meld operation is presented as well as an implementation of the Splay Heap including decrease-key. While no explicit improvement to the data structure was made, the Splay Heap has the potential to improve its performance through a new decrease-key operation.

This experimental study was intended to display the strengths and weaknesses of the different priority queues available. The priority queues that kept the memory hierarchy in consideration tended to perform the best in the out-of-core experiments. Algorithm and data structure design must continue to strongly consider the impact of the memory hierarchy in order to continue to get performance that matches with theoretical predictions.

Acknowledgements

I would like to thank Dr. Ramachandran for all of her patience and guidance and Rezaul Alam Chowdhury for all of his help along the way.

References

[1] R.A. Chowdhury, V. Ramachandran, L. Tong, D. Lan Roche. An Empirical Study of Some Cache-Oblivious and Traditional Implementations of Dijkstra's SSSP Algorithm for Real-Weighted Sparse Graphs. Unpublished Manuscript. Available online at: http://www.cs.utexas.edu/users/shaikat/papers/SSSP-Experiments.pdf.

[2] R.A. Chowdhury and V. Ramachandran. Cache-Oblivious Shortest Paths in Graphs Using Buffer Heap. *Proceedings of the 16th Annual Symposium on Parallelism in Algorithms and Architecture (SPAA)* , pp. 245-254, 2004.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*, 2001.

[4] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL Data Sets. *ESA2005: 13th Annual European Symposium on Algorithms*, October 2005.

[5] M.L. Fredman, R. Sedgewick, D.D. Sleator, and R.E. Tarjan. The Pairing Heap: A New Form of Self-Adjusting Heap. pp. 111-129, 1986.

[6] P. Sanders. Fast Priority Queues for Cached Memory. *Journal of Experimental Algorithmics (JEA), Volume 5*, pp. 1-25, 2000.

[7] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Trees. *Proceedings of the fifteenth annual ACM symposium on Theory of computing STOC '83*, pp. 235-245, 1983.

[8] D.D. Sleator and R.E. Tarjan. Self-Adjusting Heaps. *SIAM J. Computing Vol. 15, No.1*, pp. 52-69, 1986.

[9] L. Tong. Implementation and Experimental Evaluation of the Cache-oblivious Buffer Heap. Undergraduate Honors Thesis, CS-TR-06-21, The University of Texas at Austin, Department of Computer Sciences. May, 2006.

Appendix A

This appendix contains the pseudocode for the major operations of the Splay Heap that supports decrease-key.

```
Line 1: int SplayHeap::Insert( int x, int k ) {
Line 2:       new_index = new_node(x,k);
Line 3:       root = Meld(root, new_index);
Line 4:       root→PARENT = NIL;
Line 5:       return new_index;
Line 6: }
```

```
Line 1: SplayHeap::Do_Delete_Min(localRoot, x, k )
Line 2: {
Line 3:       if(localRoot == NIL)
Line 4:       {
Line 5:             x = NIL
Line 6:             k = INF;
Line 7:       }
Line 8:       else
Line 9:       {
Line 10:
Line 11:            x = localRoot → ID
Line 12:            k = localRoot → KEY
Line 13:           new_root = Meld(localRoot → LEFT, localRoot → RIGHT);
Line 14:            free_node(localRoot);
Line 15:            localRoot = new_root;
Line 16:      }
Line 17: }
```

```
Line 1: SplayHeap::Decrease_Key( x, k )
Line 2: {
Line 3:     if ( k < xp → KEY )
Line 4:     {
Line 5:             if(x == root)
Line 6:             {
Line 7:                     root → KEY = k
Line 8:             }
Line 9:             else
Line 10:              {
Line 11:                    int mx; int mk
Line 12:                    Delete(x,mx,mk)
Line 13:                    Insert(mx,k)
Line 14:             }
Line 15:    }
Line 16: }
```

```
Line 1: SplayHeap::Delete_Min( mx, mk )
Line 2: {
Line 3:     Do_Delete_Min(root, mx, mk)
Line 4:     root → PARENT = NIL
Line 5: }
```

```
Line 1: SplayHeap:: Delete( node, mx, mk )
Line 2: {
Line 3:     if(node == root)
Line 4:             Delete_Min(mx,mk)
Line 5:     else
Line 6:     {
Line 7:             int parent = node → PARENT
Line 8:             if(parent → LEFT == node)
Line 9:             {
Line 10:                    Do_Delete_Min(node,mx,mk)
Line 11:                    parent → LEFT = node
Line 12:            }
Line 13:            else
Line 14:            {
Line 15:                    Do_Delete_Min(node,mx,mk)
```

```
Line 16:                        parent → RIGHT = node
Line 17:                }
Line 18:                if(node != NIL)
Line 19:                        node → PARENT = parent
Line 20:        }
Line 21: }
```

```
Line 1: int SplayHeap::Meld(int heap1, int heap2) {
Line 2:     int root, leftPath, activeHeap, inactiveHeap;
Line 3:     if(heap1 == NIL)
Line 4:             return heap2;
Line 5:     else if(heap2 == NIL)
Line 6:             return heap1;
Line 7:     /* Start Initialization */
Line 8:     if(heap1→KEY > heap2→KEY ) {
Line 9:             swap(heap1,heap2);
Line 10:     }
Line 11:     activeHeap = heap1; inactiveHeap = heap2;
Line 12:     root = activeHeap;
Line 13:     leftPath = activeHeap;
Line 14:     activeHeap = root→RIGHT;
Line 15:     leftPath→RIGHT = leftPath→LEFT;
Line 16:     activeHeap→PARENT = NIL;
Line 17:
Line 18:     while( activeHeap != NIL) {
Line 19:             if( activeHeap→KEY > inactiveHeap→KEY ) {
Line 20:                     swap(activeHeap,inactiveHeap);
Line 21:             }
Line 22:             leftPath→LEFT = activeHeap;
Line 23:             activeHeap→PARENT = leftPath;
Line 24:             leftPath = activeHeap;
Line 25:             activeHeap = leftPath→RIGHT;
Line 26:             leftPath→RIGHT = leftPath→LEFT;
Line 27:             activeHeap→PARENT = NIL;
Line 28:     }
Line 29:     leftPath→LEFT = inactiveHeap;
Line 30:     inactiveHeap→PARENT = leftPath;
Line 31:     return root;
```

Line 32: }

Appendix B

To prove the correctness of meld, we will prove the correctness of the while loop used to implement meld.

Proof of Loop Invariants:

Pre-condition: Heap1 and heap2 are correct splay heaps that obey heap order and have correct parent pointers. Heap1 and heap2 are destroyed by the meld process.

Post-condition: Meld function returns the root to a new heap that is a combination of the two input heaps. The new heap has a left path that is a combination, maintaining heap order, of the two right paths of the input heaps. The left children of the nodes on the two right merge paths have become the right children of the resulting left path. All parent pointers have been correctly updated on the input elements.

Definitions:

1.) Let the notation [H, X,Y] denote the set of nodes on the path between node X and node Y in the splay heap H, where X and Y are included in the set. (if X==Y, the set is X)

2.) Let the notation [H, X,Y) denote the set of nodes on the path between node X and node Y in the splay heap H, where only X is included in the set. (If X==Y, the set is NIL)

3.) If a node X is a left child or right child of a different node, node Y, then Y is defined as the parent node of X. The parent of the root node is NIL.

textbfLoop Invariants:

1.) The paths [heap1, active/inactiveHeap) and [heap2, active/inactiveHeap)

are both right-paths of the input trees and have already been melded into the result tree.

2.) The path [root, leftPath] is the left path of the result tree and that path is comprised of nodes from the paths, [heap1, active/inactiveHeap) and [heap2, active/inactiveHeap).

3.) The key value of activeHeap is greater than the key value of leftPath.

4.) Each node on [root, leftPath] has had its original left sub tree changed to become its right sub tree.

5.) The parent pointer of leftPath is properly updated according to the above parent definition.

Meld Operation Pseudo-code

```
Line 1: int SplayHeap::Meld(int heap1, int heap2) {
Line 2: int root, leftPath, activeHeap, inactiveHeap;
/* Start Default Cases */
Line 3: if(heap1 == NIL)
Line 4:         return heap2;
Line 5: else if(heap2 == NIL)
Line 6:         return heap1;
Line 7: /* Start Initialization */
Line 8: if(heap1→KEY > heap2→KEY ) {
Line 9:         swap(heap1,heap2);
Line 10: }
Line 11: activeHeap = heap1; inactiveHeap = heap2;
Line 12: root = activeHeap;
Line 13: leftPath = activeHeap;
Line 14: activeHeap = root→RIGHT;
Line 15: leftPath→RIGHT = leftPath→LEFT;
Line 16: activeHeap→PARENT = NIL;
```

After Initialization,
Invariant 1 holds due to line 13 and 14 and inactiveHeap == heap2.
Invariant 2 holds because the only node melded is the root and leftPath == root. (Line 14)

Invariant 3 is true because activeHeap is leftPaths child. (Line 13,14) and Line 8 and 9 guarantee root is the min element.
Invariant 4 is true by Line 15.
Invariant 5 is true by Line 16 removing any stale pointer.

Line 18: while(activeHeap != NIL) {

Assuming activeHeap traces a right path, the loop will repeat as long as each right path still has nodes left to be melded.

Line 19: if(activeHeap→KEY > inactiveHeap→KEY) {
Line 20: swap(activeHeap,inactiveHeap);
Line 21: }

Lines 19-21 ensures that the smallest element between the two right paths rooted at activeHeap and inactiveHeap is the next node to be melded. (Invariant 3)

Line 22: leftPath→LEFT = activeHeap;

Line 22 melds the next node from the right path (by Invariant 1), into leftPath. This satisfies Invariant 2 because the result tree continues to meld on [root, leftPath] from [heap1, active/inactiveHeap) and [heap2, active/inactiveHeap) by invariant 1.

Line 23: activeHeap→PARENT = leftPath;

Maintains Invariant 5 because activeHeap has become the left child of a different node.

Line 24: leftPath = activeHeap;

Moving leftPath down the leftmost path makes Invariant 5 complete and maintains Invariant 2 because the next iteration will continue to meld on the left most path of the result tree.

Line 25: activeHeap = leftPath→RIGHT;

Maintains Invariant 1 because the activePath is still building [heap1, active/inactiveHeap)

or [heap2, active/inactiveHeap) by continuing to move the meld path down the heap to the right only.

Line 26: leftPath→RIGHT = leftPath→LEFT;

Maintains Invariant 4 because leftPaths Left subtree is becoming its right subtree and this is the deepest node of [root, leftPath].

Line 27: activeHeap→PARENT = NIL;
Line 28: }

Removes any stale pointers to maintain Loop invariant 5.

At the end of the loop, with invariant 1 holding throughout iterations, [heap1, active/inactiveHeap) and [heap2, active/inactiveHeap) can only have traced right-most paths of the input trees. When Invariant 2 is maintained throughout the loop, the left-most path of the result tree will be made up only of the combination of the two right-most paths of the input tree. Since invariant 3 says the smallest element on the two input paths is the next to be melded, the resulting left path must also obey heap order. Since invariant 4 made each node on [root, leftPath] change its left subtree to become its right subtree, all left children on the merge path have become right children of the result path. By updating the parent pointers of leftPath, the parent pointers are maintained throughout the entire tree.

Line 29: leftPath→LEFT = inactiveHeap;

Merges the remaining right path.
Line 30: inactiveHeap→PARENT = leftPath;
Adjust its parent pointer.
Line 31: return root;
Line 32: }