Measuring and Improving the Performance of Cache-efficient Priority Queues in Dijkstra's Algorithm

Mo Chen
Thesis Supervisor : Dr. Ramachandran

August 9, 2007

Abstract

The priority queue is an useful data structure in computation. There currently exist many implementations of this data structure, including some that are cache-aware and some cache-oblivious. In this study, we compare the performance of several implementations of priority queues in Dijkstra's Single Source Shortest Path algorithm. We compare high performance heaps, such as the 4ary Aligned Heap, Fast Binary Heap and Sequence Heap against in-house heap implementations, namely Buffer Heap and Auxiliary Buffer Heap, and against well-known implementations such as the textbook Binary Heap. We focus our analysis on the benefit of supporting decrease-key operations within the priority queue. Results indicate that graph density affect the relative performance of the different priority queues, and that using the *Decrease-Key* operation incurs unexpected performance hits. Furthermore, we will propose a parallel version of Buffer Heap which remains cache-oblivious and scales to a high level of parallelism.

1 Overview

This thesis focuses on two related topics regarding performance of priority queues and SSSP algorithms on cached memory. The first part includes empirical experiments on Dijkstra's algorithm using existing implementations of priority queues to investigate the benefit of supporting *Decrease-Key* instructions in priority queues. The second part includes a description and theoretical analysis of a parallel cache-oblivious priority queue based on Buffer Heap.

In the first part, we conduct empirical experiments on several categories of priority queues. Fast Binary Heap, Sequence Heap, 4-ary Aligned Heap are performance-optimized heaps for which source code was readily available [4]. Buffer Heap and Auxiliary Buffer Heap are cache-oblivious heaps which were implemented in-house[3]. As a point of comparison, we also include the DIMACS solver, a reference design SSSP solver for the 9th DIMACS Implementation Challenge [6]. Finally, we have included the textbook implementation of a binary heap, and a variation which does not support decrease-key.

Our main measure of performance of a priority queue was the speed at which Dijkstra's SSSP algorithm ran using said priority queue. More specifically, we

were interested in the speed of execution in a memory hierarchy of a typical modern processor. For the purpose of isolating reasons behind performance differences, we tested the priority queues in-core as well as out-of core, and on the simulator Valgrind. For in-core tests, we used an Intel Xeon 3.06 GHz system with 512KB of L2 cache (uruk-5,6). For out-of-core tests, we used the STXXL library, which allowed us to restrict the size of memory used in execution, and specify its block size.

In the second part of this work, we explore the possibility of creating an efficient priority queue which can exploit parallel processing and cached memory hierarchies. Cached memory is widely prevalent on modern processors, and in recent years, we have seen a move toward multi-core processors. Our hope is that such a priority queue would be more efficient on future systems compared to traditional priority queue implementations.

1.1 Ground Work

The empirical measurements are done as a continuation of the work done by Lan Roche on his thesis [1]. His experiments used graphs of a fixed density (8x), and varying number of vertices. He found that in such graphs, the general trend was that sequence heap performed best, followed by Auxiliary Buffer Heap, 4-ary Aligned Heap, Fast Binary Heap, Buffer Heap, and Binary Heap, in order of decreasing speed.

We found it interesting that overall, priority queues without support for the decrease-key operation ran Dijkstra's algorithm faster than those with support for decrease-key. This is a surprising result since previous works often overlooked the possibility of running Dijkstra's algorithm without using *Decrease-Key*. However, in Roche's work, the priority queues which do not support *Decrease-Key* were also highly optimized, and had no counterpart with support for *Decrease-Key* to compare with. The only exception to this was in the case of Auxiliary Buffer Heap, since it was a simplified version of Buffer Heap but without support for *Decrease-Key* and *Delete*. In our current work, we designed and ran experiments to investigate further whether Dijkstra's algorithm should or should not be used with *Decrease-Key*.

Buffer Heap was introduced by Chowdhury and Ramachandran [3]. The implementations of these heaps are done by Tong [8] and Chowdhury.

Fast Binary Heap, 4-ary Aligned Heap, and Sequence Heap were developed by Sanders, who also coded implementations [4]. We obtained the source code electronically. For in-core tests, we used them as-is, tweaking parameters when called for. Some changes were necessary to make them compatible with STXXL. Roche [1] had already made these changes for his experimental work.

2 Background Information

2.1 Priority Queues

A *priority queue* is a collection of elements each with a numerical *priority*, also known as its *key*. Priority queues support *insert*, *extract-min* (or *delete-min*) operations. An insert operation adds one element and its key into the priority

queue. A call to extract-min deletes the element with the lowest key from the queue, and returns the element with its key.

Optionally, a priority queue may support *delete* and *decrease-key* operation. The decrease-key operation takes as its parameters an element reference, and a new key. The result is that if the element is present in the priority queue, its current key is replaced with the new key. To implement delete and decrease-key operations efficiently, a priority queue must be able to access specific elements in constant time. Usually this is done by keeping a table of element pointers. It is easy to see that this adds a large overhead to the data structure. The reason for including these operations will be discussed in 2.3.

2.2 Dijkstra's Algorithm and Variations

Dijkstra's algorithm is a greedy algorithm. Given a non-negatively weighted graph and a source vertex, computes the shortest paths from the source to every vertex. In other words, it is a Single-Source-Shortest-Path (SSSP) algorithm. The algorithm repeatedly looks for the vertex currently known to be closest to the source, and applies *edge-relaxation* its adjacent vertices. A priority queue is used to keep a list of distances to each vertex. This way, we can find the closest vertex with an extract-min operation. Similarly, edge relaxation involves a call to decrease-key.

One possible change to Dijkstra's algorithm is to eliminate the need to support decrease-key in the accompanying priority queue. We can do this by changing the edge relaxation process. Instead of calling decrease-key, we can insert a new key with the new distance into the heap. Since the new distance is shorter, it is sure to be extracted before the old distance. This allows us to use a priority queue that does not support the Decrease-Key operation, which means it does not have to carry the aforementioned overhead of element pointers.

Another possible change to improve cache efficiency is to use two priority queues. Using a second, auxiliary priority queue prevents having to check whether each vertex has settled [5], and thus incurring a block transfer. Here, we use Buffer Heap and Auxiliary Buffer Heap as the two priority queues.

For the sake of clarity, we will use Dijkstra-dec, Dijkstra-nodec, and Dijkstra-ext to indicate the version of Dijkstra's algorithm being used, whether it be with *Decrease-Key*, without, or using dual priority queues.

2.3 The Theoretical and Practical Benefit of Supporting Decrease-Key operations

The theoretical time bound for Dijkstra's algorithm with *Decrease-Key* using a priority queue which supports *Insert*, *Delete-Min*, and *Decrease-Key* in $O(\log n)$ time, which was typical, on a graph with m edges and n vertices, is $O((m+n)\log n)$. Using a priority queue such as Fibonacci Heap which supports *Decrease-Key* operations in constant time, this time bound could be decreased to $O(m + n\log n)$. In practice, however, Fibonacci Heap is a complex data structure and has slow running times.

In our case, even though all the heap operations have the same worst case asymptotic time, in practice, *Decrease-Key* is typically much faster compared to *Insert* and *Delete-Min* operations, since a small change in a key value would only cause a small disturbance to the heap structure, and not require much

rearrangement to preserve the heap property. In addition, to do edge relaxation without decrease-key, we would need to insert a new element into the heap for every call that would otherwise be a decrease-key. On non-sparse graphs, this should noticeably increase the size of our heap. These reasons lead us to hypothesize that taking the additional overhead of supporting decrease-key should be worthwhile as long as the graph is not sparse.

2.4 Cache-efficient and Cache-oblivious Algorithms

Traditionally, an algorithm's speed is measured in the number of computational steps per operation. In modern computer systems, due to the disparity between processor speeds and memory access speeds, we can no longer assume that each computational step will take the same amount of time. It is important to note that even RAM accesses are thousands of times slower than arithmetic computations within the processor. The trend has been that memory access times continue to lag further and further behind processor speeds. The challenge this presents to an algorithm designer is that now we must not only reduce the number of computation steps, but also the number of memory accesses. Fortunately, we can expect some small amount of fast cache memory to use in this goal.

In general, a cache efficient algorithm is an algorithm that uses smaller but faster cache memory to substantially reduce the number of accesses to larger but slower memory. Note that this doesn't only refer to using processor cache for RAM accesses, but also using RAM for disk accesses, and other similar uses.

The notion of cache-obliviousness means that an algorithm does not know about the size and block size of the cache, nor its layout, be it distributed or shared. This concept was first presented by Frigo et al. [2] We will assume that there exists a scheduler that is able to assign concurrent tasks to processors in a way that will optimize cache hits. The scheduler is allowed to know about the size and layout of the cache to achieve this goal.

3 Priority Queues being Tested

We will give a brief description of each of the priority queues we ran tests on.

3.1 Priority Queues Developed In-House

These priority queues were all developed under Dr. Ramachandran's research group.

3.1.1 Binary Heap

The plain binary heap was implemented based on the well known textbook implementation. For simplicity, we did not add any additional constructs and only performed basic, local optimizations. The heap is stored as a list of pointers in a std::vector object. A separate std::vector stores keys, values, and back-pointers to the first vector. This additional level of indirection allows us to implement decrease-key operations, as the second vector is never rearranged. Deleted keys form holes for new elements to fill. A decrease-key operation

modifies the key in the second vector, and causes the first vector to be possibly rearranged, which would trigger pointer updates.

3.1.2 Modified Binary Heap

Removing the need to support decrease-key operations means that only one vector is necessary to store the key/value pairs. Furthermore, no pointer or back pointers need to be stored. We applied these changes to the binary heap, and called it Modified Binary Heap. The hope for the Mod-BH is that its greater simplicity would lead to better performance. In addition, since it directly mirrors Binary Heap, we can use it to benchmark the performance benefits of having a decrease-key operation. The Modified Binary Heap was developed specifically to address the primary issue in the experimental section of this work, which was to investigate the usefulness of decrease-key operations in priority queues.

3.1.3 Buffer Heap

The *Buffer Heap* is an efficient cache-oblivious priority queue. It supports Extract-min, Delete, and Decrease-Key operations in $O(\frac{1}{B}\log_2 \frac{N}{B})$ amortized block transfers [3]. To achieve, this, it uses buffers of increasing size to hold queue elements, and an update buffer for each level of element buffer. Since each buffer is in the form of a stack, maintenance and updates can be done in a cache-oblivious fashion.

3.1.4 Auxiliary Buffer Heap

Auxiliary Buffer Heap is a modification based on the Buffer Heap which does not support the decrease-key operation. This heap can be used on its own as part of the Dijkstra-nodec algorithm, or as a component in the dual buffer heap algorithm, which was its original purpose.

3.2 High Performance Priority Queue Software

Other people developed these priority queues. We obtained the source code and used them for comparison.

3.2.1 Sequence Heap

Sequence heap was developed by Sanders [4] as a fast cache-efficient priority queue. It is cache-aware. It uses a small insert heap to store recently inserted elements until they need to be sorted and added to the main structure. The main structure is a collection of k-mergers on sorted sequences. The k-merger operator cache-efficiently merges these sequences into group buffers, which are then merged into a deletion buffer, all on demand. In experiments, the sequence heap is quite efficient, beating out all other heaps presented here on in-core tests.

3.2.2 Fast Binary Heap

Fast Binary Heap is an optimized implementation of the Binary Heap. It includes an insertion buffer, and uses the *bottom up heuristic* for delete-min. This

strategy lifts up elements on the top levels when a root is deleted. This strategy reduces the number of comparisons needed to perform a delete-min operation.

3.2.3 4-ary Aligned Heap

The 4-ary Aligned Heap is similar to Fast Binary Heap in that both use the bottom up heuristic. In addition, 4-ary Aligned Heap is cache aware. WIth knowledge of the cache line size, it aligns data to minimize cache misses.

3.2.4 DIMACS Solver

The DIMACS solver is one of the reference implementations used to judge the 9th DIMACS Implementation Challenge on SSSP algorithms [6]. It is worth noting that the DIMACS solver is a standalone SSSP solver, and not simply a priority queue. In addition, it is only capable of handling integer-valued edge weights.

4 Graph Experiments

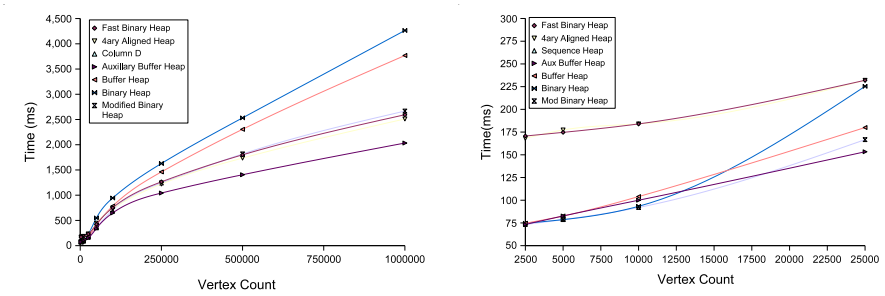
As previously mentioned, previous experiments showed that priority queues which did not support *Decrease-Key* were generally faster, when used on at least some graphs. To expand on that, we first determine precisely which graphs favored *nodec* priority queues.

4.1 Graphs of Varying Density and Types

Roche's results used a constant density of 8. Since *Decrease-Key* operations are performed more often on denser graphs, we will first look at the effect of varying graph density. To achieve this end, we first use random $\mathcal{G}_{n,m}$ graphs with a constant edge count, and varying number of vertices. This keeps the relative sizes of the graphs similar. Secondly, we use sparse non-$\mathcal{G}_{n,m}$ graphs to see if the effect of *Decrease-Key* is consistent with different graph types.

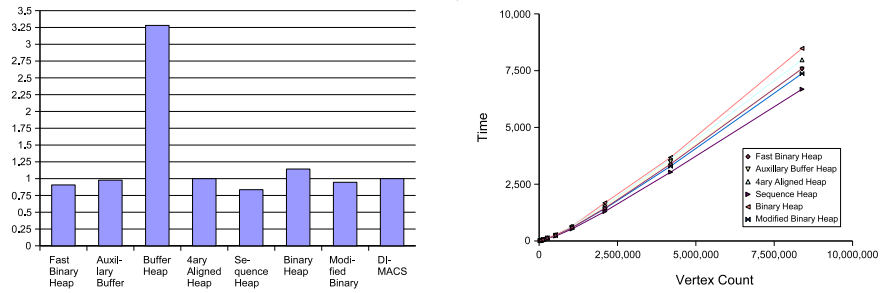
Random Graphs These graphs were random $\mathcal{G}_{n,m}$ graphs with randomized edge weights. Theoretically, SSSP on a dense graph should benefit from having the *Decrease- Key* operation than on a sparse graph, since the size of the heap is kept smaller using *Decrease-Key*. The results, shown on figure 1 are consistent with this assessment. Despite this, it was still slower to use Dijkstra-dec as opposed to Dijkstra-nodec, as indicated by the modified Binary Heap and Auxiliary Buffer Heap both beating or tieing their *Decrease-Key*-capable counterparts on all density values. On very dense graphs, the differences were quite small.

Roadmap Graphs and Grid Graphs Roadmaps are sparse, since intersections are rarely more than four way. Our results on roadmaps are shown on 2(a). As we would expect, priority queues that do not support decrease-key performed better. What is surprising is that cache-efficient priority queues such as Auxiliary Buffer Heap and 4ary Aligned Heap were slower than the Fast Binary Heap and Modified Binary Heap, which are not cache-efficient.



(a) (b)

Figure 1: $\mathcal{G}_{n,m}$ graphs, 8 million edges, variable vertex count



(a) (b)

Figure 2: (a) Roadmap graphs, time relative to DIMACS solver (b) Grid graphs

Grid graphs have a fixed structure that resembles a two dimensional square grid. These sparse graphs were also populated with random edge weights. On figure 2(b) we see that unlike the trends on roadmaps, Auxiliary Buffer Heap is slightly faster than Modified Binary Heap. However, the overall trend that supporting *Decrease-Key* hurts performance still holds.

4.2 Analysis of Graph Experiment Results

4.2.1 Binary Heap and Modified Binary Heap

In our experiments on graphs, we saw two trends between Binary Heap and Modified Binary Heap. Firstly, that the modified Binary Heap was faster in general compared to the original. Secondly, that the original Binary Heap started to catch up in performance when the edge density became larger. Our hypothesis for why this was happening is that *Insert* and *Delete-Min* operations are faster on the modified Binary Heap if the heaps were the same size, but since Binary Heap has a lower element count when the graph is dense, the increased number of levels on the modified Binary Heap made operations slower in comparison.

To find out the relation between time spent on each operation type, we ran the profiler Callgrind. See figure 3 for sample results. Callgrind records the number of instruction fetches, cache misses (in various levels/sections of cache). Using this information, and arbitrary weights on each event, we can construct an estimated cycle count per call, $t = (I_r + 10L_1 + 100L_2)/c$, where I_r is the number

of instruction fetches, L_1 is the number of L1 misses, and L_2 is the number of L2 misses. While this estimate is simplistic compared to the real world performance of modern processors, it is nontheless a reasonable approximation. Using this data and the number of calls to each subroutine (not shown), we can compute the number of cycles spent on each call to each operation. This is shown in figure 4.

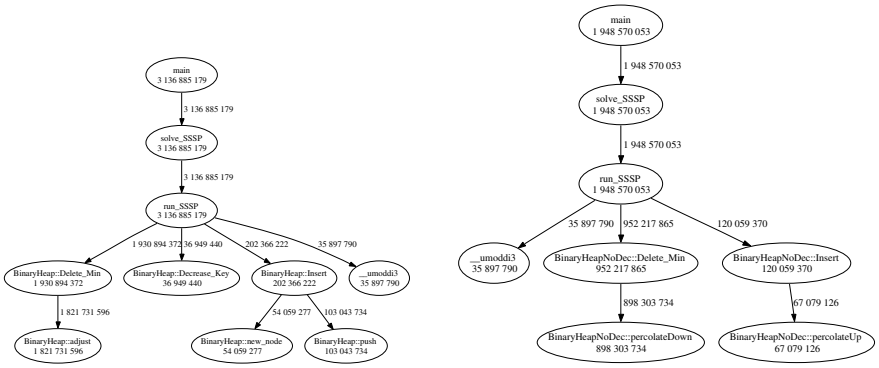


Figure 3: Callgraph results on Binary Heap and modified Binary Heap on $\mathcal{G}_{2500,400}$. Number shown are approximate cycle counts in all calls to each subroutine.

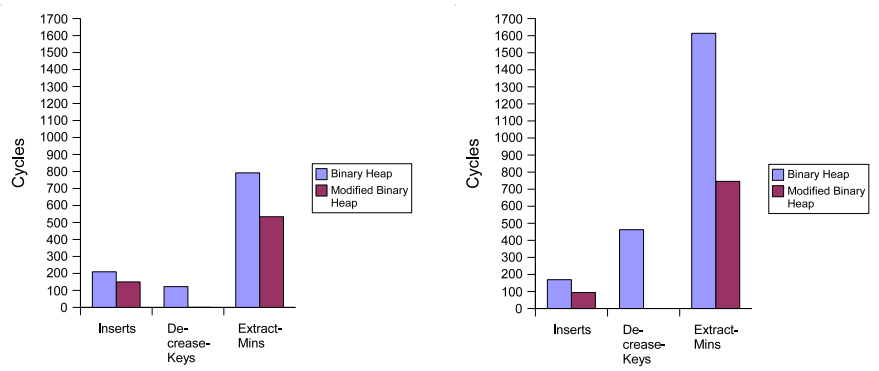


Figure 4: Approximate average number of cycles spent in each call to priority queue operations on $\mathcal{G}_{2500,400}$ and $\mathcal{G}_{250000,4}$

We discovered that we were only partially correct in our hypothesis. It is true that modified Binary Heap spent less time per operation. It is also true that time cost per operation becamse closer as the graph became denser, but there was another factor that played into the closing of the speed gap. Calling *Decrease-Key* once in Dijkstra-dec to relax an edge was significantly faster than calling both *Insert* and *Delete-Min* once each in Dijkstra-nodec, regardless of the density of the graph. On dense graphs, the distribution of calls shifted more towards Decrease-Key, as more edges were visited along several paths. We can

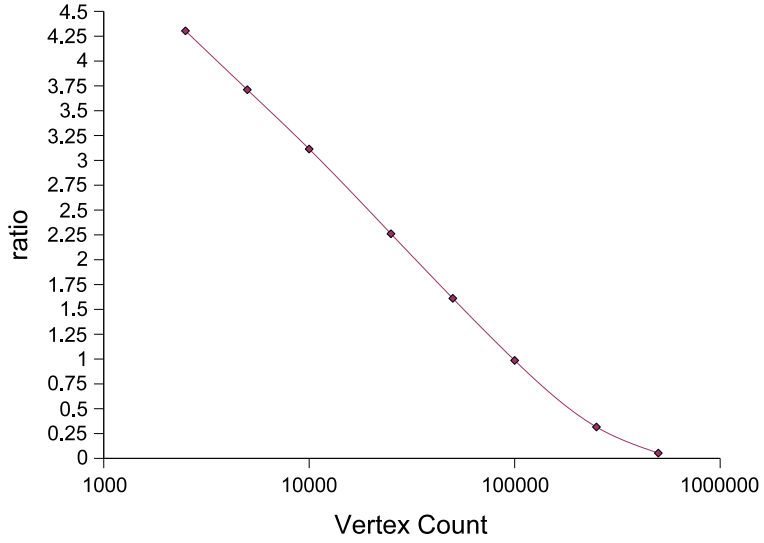


Figure 5: Ratio of *Decrease-Key* operations to *Insert* or *Delete-Min* operations, $\mathcal{G}_{n,m}$ graph with 1 million edges

see this trend on figure 5.

4.2.2 Out-of-Core Analysis

To analyze the cache-efficiency of supporting *Decrease-Key* operations, we ran our graph tests under STXXL, and constrained the amount of cache to 256KB, in 64 4KB blocks, and measured the number of block transfers incurred in executing Dijkstra's algorithm. The results are shown in figure 6. We see that the Binary Heap, lacking any sophisticated performance optimizations, is quite cache-inefficient. The cache-aware 4-ary Alighed Heap, and the simpler Fast Binary Heap and modified Binary Heap, form a class of their own for sparse graphs. At higher densities, modified Binary Heap moves closer to the cache-oblivious Buffer Heap and Auxiliary Buffer Heap. We suspect this is because modified Binary Heap has the smallest memory signature due to its simple nature, therefore fits more easily into cache, despite it not being optimized for cache efficiency.

Although Binary Heap has many more cache misses than its modified cousin, Buffer Heap and Auxiliary Buffer Heap are quite close in cache efficiency. Therefore, we don't believe that supporting *Decrease-Key* operations makes a priority queue inherently more or less cache efficient.

Dijkstra-ext, used on Buffer Heap and Auxiliary Buffer Heap, is more cache-efficient on sparse graphs than any of the other priority queues. However, its running time in-core is quite slow due to its additional complexity.

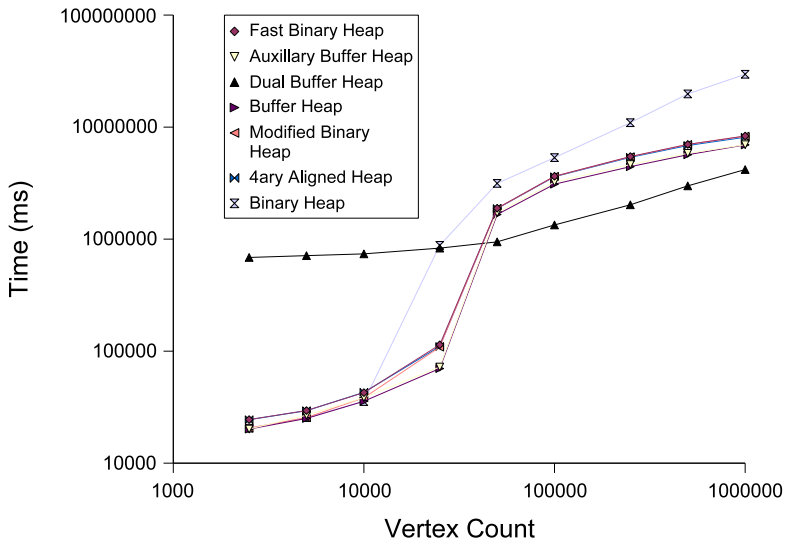


Figure 6: Block transfers under under STXXL, constrained to 64x4KB cache, $\mathcal{G}_{n,m}$ graph with 2 million edges

4.3 Concluding the First Part

In light of these discoveries, our conclusion is that using *Decrease-Key* in Dijkstra's algorithm slows down execution time, because building in support for *Decrease-Key* incurs overheads in space and time usage. On sparse graphs, these overheads results in dramatically slower running times. The benefits using *Decrease-Key* to solve dense graphs do not fully offset the overhead, and results in the same or slower running times, and increased priority queue complexity.

5 Parallelizing Buffer Heap

5.1 Introduction

The *Buffer Heap* is an efficient cache-oblivious priority queue. It supports Delete-Min, Delete, and Decrease-Key operations in $O(\frac{1}{B}\log_2 \frac{N}{B})$ amortized block transfers [3]. As a general priority queue, it is useful in a variety of applications. One possible application of Buffer Heap is in executing Dijkstra's Algorithm.

The purpose of this work is to propose an implementation of Buffer Heap which can run on a p-processor system. The goal is for the algorithm to remain cache-oblivious, with the same or nearly the same amortized I/O bounds, and at the same time achieve a near-p-fold decrease in running time with a wide range of values for p.

To achieve this goal, we will focus on parallelizing the Delete-Min operation of Buffer heap. The Delete-Min operation requires several merges, sorts, and selection steps. These are what will ultimately determine the running time of

the Delete-Min operation, which in turn takes up almost most of the running time of the algorithm.

To analyze the running time performance of the parallel algorithm, we will use the CREW (Concurrent Read Exclusive Write) PRAM model as introduced in Chapter 1 of Jájá's book [7]. The chapter defines the notion of *running time*, $T(n)$, and *work* done, $W(n)$. Running time, $T(n)$, also known as *parallel time*, or *critical path length*, is the minimum time a given algorithm takes to execute, given unlimited number of processors. Work, $W(n)$ is the number of operations needed to complete the computation, regardless of number of processors. A parallel algorithm is considered *work-optimal* when $W(n)$ is equal to the work done of a sequential algorithm. One other useful measurement is the *cost* of running an algorithm. The cost, $C(n)$ is defined to be the amount of computational resource dedicated to completing the algorithm. In other words, it is the number of processors times the amount of time until completion. It is possible for $C(n)$ to be greater than or equal to $W(n)$, but never less. It is useful to minimize cost while simultaneously minimizing asymptotic running time. This happens at $W(n) = C(n) = pT(n)$. Solving for p, we get $p = \frac{W(n)}{T(n)}$.

In order to analyze the I/O performance[1] of the algorithm, we examine it under two different caching models. The *shared cache* model assumes that there is one pool of cache that is accessible by all processors. The *distributed cache* model, on the other hand, gives each processor its own pool of cache, which is not accessible by other processors. Both models assume that the cache is arranged in blocks of size B, and that the cache replacement policy is optimal. In other words, the block that will be accessed furthest in the future will be evicted first. The total amount of cache across all processors is denoted by M. In this work, caches are assumed to be fully associative. One *block transfer* brings a contiguous block of data into the cache, replacing one of the existing blocks. Subsequent reads and writes to a block in the processor's cache, in the case of a distributed cache, or the common cache, in the case of a shared cache, are considered cache hits, and do not incur a block transfer. The caching performance of an algorithm will be measured in the number of block transfers required to complete the computation in the worst case on an input of a given length.

The notion of cache-obliviousness means that an algorithm does not know about the size and block size of the cache, nor its layout, be it distributed or shared. This concept was first presented by Frigo et al. [2] We will assume that there exists a scheduler that is able to assign concurrent tasks to processors in a way that will optimize cache hits. The scheduler is allowed to know about the size and layout of the cache to achieve this goal.

5.2 Brief Description of Buffer Heap

The Buffer Heap is composed of two sets of buffers in levels. Let r be the total number of levels. The *element buffers* are labeled B_0 to B_r. The *update buffers* are similarly labeled U_0 to U_r. The size of each B_i is capped at 2^i, and the size of each U_i is unbounded. The range of keys within each level is bounded below by a boundary value that is dynamically updated. These buffers are stored in

[1] We will use the term I/O performance, and cache-efficiency interchangeably. More specifically, the number of I/Os refers to the number of cache misses

a stack, with level 0 on top of the stack, followed by level 1, and so on.

For operations such as *Insert* and *Decrease-Key*, the operation is simply appended to U_0.

During a *Delete-Min* operation, we call *Apply-Updates*, which sorts U_0, and merges instructions and elements until at least one element is found which doesn't need to be deleted. Finally, it puts the merged elements back into the element buffers, and pushes any leftover elements into the next level's update buffer. This final step requires multiple selection steps, one for each level of the element buffer.

5.3 Parallel Subroutines

We will attack the parallelization problem by parallelizing the merge, sort, and selection subroutines, which take up most of the time and I/O. The last subroutine, scan, is omitted since it is trivial.

5.3.1 Merge

Input: Two lists, A_1, A_2, sorted by element ID. Assume that both A_1 and A_2 are of length n. The number of processors p.

Output: A sorted list A_o such that A_o has all elements that appear in A_1 or A_2.

Description The parallel merge algorithm is a modified version of an optimal $O(\log \log n)$ time parallel algorithm given by Jájá [7, p. 153]. We have modified Jájá's algorithm to exploit locality with a small cost in running time. Each step is done by p processors unless otherwise specified.

1. Divide both arrays evenly into p pieces.
2. Let $q = \langle q_1, q_2, \cdots, q_p \rangle$ be the first elements of each piece in A_1, and $r = \langle r_1, r_2, \cdots, r_p \rangle$ be the same for A_2.
3. Merge q and r using $\frac{p}{\log p}$ processors in the following fashion.
   (a) Divide q and r into segments of length $\log p$. Let the array of the first elements of each segment in q be $q' = \langle {q'}_i \rangle$, and the the first elements of each segment in r be $r' = \langle {r'}_i \rangle$.
   (b) Rank each element of q' in r' using a binary search. This determines the segment of r each element of q' rests in.
   (c) Rank each element of q' within its corresponding segment of r using a linear search. For any arbitrary q', its rank in r is computed by $Rank(x) = \alpha \log p + \beta$, where α is the rank of the first element of the segment in q' containing x, and β is the rank of x in its corresponding segment in r.
   (d) Repeat steps (b) and (c) to rank each element of r' in q. This divides the merge problem into up to $\frac{2p}{\log p}$ non overlapping subproblems of size at most $2 \log p$ each, since both q and r are divided along the same list of pivots.

(e) Compute the index to write out each merged segment within a single merged array. The index to start writing segment i is equal to the sum of the sizes of all the segments with indexes less than i. Use the logarithmic running time parallel prefix sum algorithm given by Jájá [7, p. 44] on an array of segment sizes to find these indexes.

(f) Merge each segment concurrently. Write the result back at the correct index.

4. Rank each q_i in A_2, and each r_i in A_1 using linear searches. Note that we already know which piece of B each q_i lies in, so we only have to search in that piece. This divides the problem into up to up to $2p$ non overlapping subproblems each of size at most n/p.

5. Compute the index to write out each merged piece within a single merged array. This is again equivalent to finding the prefix sums of an array C of p elements. Use the following algorithm with $\frac{p}{\log p}$ processors.

   (a) Divide C evenly into sections of size $\log p$, call it $C_1, C_2, \cdots, C_{\lceil \frac{p}{\log p} \rceil}$. Use one processor to find the array of prefix sums of C_i, call these arrays $y_{i,j}$.

   (b) Find the prefix sums of the array composed of the first elements of each C_i, call this array of prefix sums x_i.

   (c) For any given i, the prefix sum of C up to i is $s_i = x_k + y_{k,i \bmod \log p}$, where $k = \lfloor \frac{i}{\log p} \rfloor$.

6. Merge each piece concurrently. Write the result back at the correct index.

Running Time To find the worst case running time of this algorithm, we can look at the amount of time each step takes. Steps 1, 2 and 3(a) can be done in constant time using one processor per piece. Step 3(b) takes time $O(\log \frac{p}{\log p})$. 3(c) and 3(d) takes time $O(\log p)$. 3(e) takes time $O(\log \frac{p}{\log p})$. 3(f) takes $O(\log p)$ with the standard sequential merge algorithm. Step 4 takes $O(\frac{n}{p})$ time, since we have to look at at most all the elements of both arrays. Step 5(a), (b), (c) take time $O(\log \log p)$, $O(\log \frac{p}{\log p})$, and $O(\log p)$ respectively. Step 6 take $O(\frac{n}{p})$ time. Therefore, in total, the merge has a time bound of $T(n) = O(\frac{n}{p} + \log p)$.

Work Since steps 1, 2, 4, 6 are done using p processors, for those steps $W(n) \leq pT(n) = O(n)$. For steps 3 and 5, $W(n) \leq pT(n) = \frac{p}{\log p} \log p = O(p)$.

This means that this algorithm has $W(n) = O(n + p)$, which is optimal for $p = O(n)$. The *cost*, however, is $C(n) = pT(n) = O(n + p \log p)$, which is only linear when $p \log p = O(n)$. Hence, when $p = \frac{n}{\log n}$, $T(n) = O(\log n)$.

This parallel algorithm requires the CREW model since steps 3(b) and 3(d) require concurrent reads.

Cache Misses Under Shared Cache Steps 1 and 2 do not require any I/O. Steps 3 and 5 work on up to 2 of arrays of size p, so under the assumption $M > 3p$, $O(\frac{p}{B})$ cache misses will bring in the entire data set and write the output. Step 4 and 5 can be done by assigning each processor to work on one

piece at a time. This needs at most $O(\frac{n}{B})$ steps to look through the entire array, granted that $M \geq 3pB$ to provide each processor with 2 blocks hold one block of each input array and one block to write the output in. In the end, we have $O(\frac{n}{B} + \frac{p}{B})$ cache misses. At the optimal speedup range where $p = O(\frac{n}{\log n})$, the number of cache misses is $O(\frac{n}{B})$.

Cache Misses Under Distributed Cache Steps 1 and 2 similarly require no I/O. Step 3(a) and 3(b) need $\frac{p}{\log p}$ processors to each perform $O(\log \frac{p}{B \log p})$ I/O, assuming that each operation is a cache miss until the binary search no longer takes strides greater than size B. Hence, a total of $O(p)$ cache misses are needed. 3(c) and 3(d) require $O(\frac{p}{\log p} + \frac{p}{B})$ block transfers. 3(e) requires $O(p)$ block transfers assuming every operation is a cache miss. 3(f) requires $O(\frac{p}{\log p} + \frac{p}{B})$ block transfers. Step 4 needs $O(\frac{n}{B} + \frac{n}{\log p})$ block transfers. Here, using a linear search simplifies the I/O bound without expanding it. Step 5(a), (b), and (c) need $O(\frac{p}{\log p} \log \log p)$, $O(\log \frac{p}{\log p})$, and $O(\frac{p}{B} + \frac{p}{\log p})$ respectively, assuming all cache misses. Finally, step 6 needs $O(\frac{n}{B} + p)$.

The total number of block transfers is then within $O(\frac{n}{B} + p)$. Our choice of limiting steps 3 and 5 to using only $\frac{p}{\log p}$ processors was to reduce the I/O bound on a distributed cache. A straightforward implementation using p processors would use $O(\frac{n}{B} + p \log p)$ block transfers.

Comparison with Sequential Algorithm A trivial sequential implementation of a cache oblivious merge would have $O(n)$ running time and $O(\frac{n}{B})$ cache misses. The parallel version of merge has a p-fold speedup compared to the sequential version when $p \log p \leq n$. The algorithm has the same number of cache misses as the sequential version when $p \leq n$ for the shared cache version, and $pB \leq n$ for the distributed cache. Both are assuming that $M \geq 3pB$.

5.3.2 Merge Sort

Input: unsorted array A with elements from an arbitrary set S. Number of processors p. A partial order r on S.

Output: sorted A, where $r(A_i, A_j) \forall i < j$.

Description

1 Divide A into p pieces. Sort each piece cache-obliviously using one processor.

2 for h = 1 to $\log p$

3 for $1 \leq j \leq n/2^h$ pardo

4 merge pieces $2j - 1$, $2j$.

Running Time Step 1 can be done in parallel, in $O(\frac{n}{p} \log \frac{n}{p})$ steps using the sequential algorithm. Step 4 can be done using the above described merge algorithm. Since $\frac{p}{2^h}$ processors are available for each merge, and each merge is of size $\frac{n}{2^{h-1}}$, each merge step takes

$$O(\frac{\frac{n}{2^{h-1}}}{\frac{p}{2^h}} + \log \frac{p}{2^h}) = O(\frac{n}{p} + \log \frac{p}{2^h})$$

time. Assuming that $p \log p \leq n$, the bound simplifies to $O(\frac{n}{p})$. Since there are $\log p$ such steps, steps 2-4 take a total of $O(\frac{n}{p} \log p)$ time. Therefore, the entire algorithm takes $O(\frac{n}{p} \log \frac{n}{p} + \frac{n}{p} \log p) = O(\frac{n}{p} \log n)$ time to complete given $p \log p \leq n$. A good estimate for p is $p < \frac{n}{\log n}$. With this value of p, the algorithm has a running time of $O(\log^2 n)$.

Work Without actually looking at the actual number of operations performed, we can conclude that $W(n) \leq pT(n) = O(\frac{np}{p} \log n) = O(n \log n)$, which is optimal for comparison based sorts.

Cache Misses Under Shared Cache In order for step 1 to cache efficient on a shared cache, the size of the cache, M, must be greater than pB^2 so that each processor has the equivalent of a *tall cache*. We can use the funnel sort algorithm, which gives an IO bound per processor of $O(\frac{n}{pB} \log_{M/pB} \frac{n}{pB})$ [2]. We have also previously determined that merging takes $O(\frac{n}{B})$ cache misses when $M \geq 3pB$, so steps 2-4 take $O(\frac{n}{B} \log p)$ cache misses. Adding these two expressions requires us to find the max of $\log_{M/pB} \frac{n}{pB}$ and $\log p$. We can use the bound $pB \leq n$, which would bound the max at $\log \frac{n}{B}$. Finally, this means that the entire algorithm incurs $O(\frac{n}{B} \log \frac{n}{B})$ cache misses when $M \geq pB^2$.

Cache Misses Under Distributed Cache Step 1 has the same number of cache misses as the shared cache algorithm, which is $O(\frac{n}{B} \log_{M/pB} \frac{n}{pB})$ among all processors. This requires a tall cache for each processor, so $M \geq pB^2$. If we assume that $pB \leq n$, Step 2-4 can be done in $O(\frac{n}{B} \log p)$ block reads. The reason we want to ensure that $pB \leq n$ is so that the one block transfer per processor per level does not become the dominating term. The total is $O(\frac{n}{B} \log \frac{n}{B})$ block reads under the assumption $pB \leq n$ and $M \geq pB^2$.

Comparison with Sequential Algorithm The optimal cache oblivious sort algorithm has running time $O(n \log n)$ and requires $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ block transfers. [2] The algorithm is p times faster than the sequential algorithm if $p \log p \leq n$. Satisfying $M \geq pB^2$ allows for only slightly more shared or distributed cache misses than the sequential version.

5.3.3 Selection

Given an unsorted list A, and p processors. Find the ith largest element in the list.

Description We can use the sample select algorithm. [8]

1. Sample $s = n^{\frac{3}{4}}$ items from A. Call the new list S.
2. Find the largest p' such that $p' < p$ and $p \log p \leq n^{\frac{3}{4}}$. Sort S with p' processors.
3. Let $x = s[\frac{is}{n} - \sqrt{n}]$ $y = s[\frac{is}{n} + \sqrt{n}]$
4. Let $L = Rank(A, x)$, $R = Rank(A, y)$. Do this by scanning A. With high probability, $L < i < R$.

5. Let $Q = \{\text{elements of A with values between x and y}\}$.

6. Find the largest p'' such that $p' < p$ and $p \log p \leq n^{\frac{1}{2}}$. Sort Q using p'' processors.

7. Return $Q[i - L]$

Running Time Step 1 requires $\frac{n^{\frac{3}{4}}}{p}$ time. Step 2 uses $O(\frac{n^{\frac{3}{4}}}{p} \log n^{\frac{3}{4}})$ time with the parallel merge sort algorithm. The lower bound is $O(\log^2 n^{\frac{3}{4}}) = O(\log^2 n)$. Step 3 is a constant time operation. Step 4 requires $O(\frac{n^{\frac{3}{4}}}{p})$ time. It is trivial to execute step 5 in $O(n/p)$ running time down to $O(1)$ time at $p = n$. Step 6 sorts at most $2\sqrt{n}$ elements, and requires $O(\frac{\sqrt{n}}{p} \log n)$ time. Similar to step 2, this also has a lower bound of $O(\log^2 n)$. Step 7 requires constant time.

In total, this algorithm runs in $O(n/p + \log^2 n)$ time. We continue to benefit in asymptotic running time up to $p = O(\frac{n}{\log^2 n})$.

Work Without having to consider steps, we can conclude from the running time that $W(n) \leq pT(n) = O(n)$. This is optimal. The cost remains linear as long as $p = O(n)$.

Cache Misses Under Shared Cache Assuming $M \geq pB^2$.

Step 1 requires $\frac{n^{\frac{3}{4}}}{B}$ block transfers. Step 2 uses $O(\frac{n^{\frac{3}{4}}}{B} \log n^{\frac{3}{4}})$ block transfers. Step 3 requires no I/O. Step 4 requires $O(\frac{n^{\frac{3}{4}}}{B})$ reads, since we are doing scans. Step 5 requires $O(\frac{n}{B})$ reads. Step 6 takes fewer block transfers than step 2. Step 7 requires constant I/O operations.

The dominant term is $O(\frac{n}{B})$.

Cache Misses Under Distributed Cache The distributed cache model gives the same I/O bounds at each step as the shared cache model. However, there is an additional condition that $pB = O(n)$ so that step 5 does not dominate.

Comparison with Sequential Algorithm A sequential cache-oblivious version of selection has $O(n)$ running time, and $O(1 + \frac{N}{B})$ cache misses. [9] Given $pB = O(n)$, $M \geq pB^2$, and $pB = O(n)$, the this parallel version of search is p times faster than the sequential version, and incur the same number of cache misses.

5.4 Parallel Buffer Heap

Intuitively, Buffer Heap gets its cache-efficiency by using cache-friendly data structures such as stacks and arrays, and cache-efficient subroutines such as scans and sequential versions of the subroutines presented above.

Since all the parallel subroutines can achieve the same I/O bounds as the sequential version, the amortized I/O cost of buffer heap remains very close. That is, if the required constraints can be met. It is easy to see that some of the constraints would be violated if p is large and the size of the Buffer Heap is

small. Fortunately, we can artificially limit the number of processors used when executing a subroutine in order to keep the processors from contending for I/O accesses. There are three kinds of constraints in the subroutines .

1. Constraints which depend on p and n, such as $p \log p \leq n$. For these, we can simply calculate the optimal value of p.
2. Constraints do not depend on n, such as $M \geq 3pB$. For these, decreasing the value of p will not break the constraint.
3. The third kind, namely $pB \leq n$ is present for sorting. The algorithm cannot enforce this since it does not know the value of B. However, since optimal running time requires that $p < n/\log n$, and in practice, B is quite small, this is not likely to cause problems.

As a consequence of this, the Parallel Buffer Heap has the same I/O performance as its sequential counterpart on a shared cache system, but cannot guarantee the same I/O bounds when running on a distributed cache system.

5.4.1 Detailed Analysis of a Call to Delete-Min

We will first take a look at the cost of an individual *Delete-Min* operation. Since the cost depends on the current state of the heap, we will parameterize the state of the heap as follows.

Let r be the number of populated levels in the buffer heap. For $0 \leq i \leq r-1$, let b_i be the number of items in each element buffer, and let u_i be the number of operations in each update buffer. Note that *Apply-Updates* may delete all the elements at any given level, and will stop when it encounters the first element which does not get deleted. Let k be the number of levels visited by the *Apply-Updates* function. After applying updates, the elements that were not deleted from the first k levels are passed to *Redistribute-Elements*, to form full element buffers on the first l levels. For the sake of simplification, let $x = \sum_{i=0}^{k} u_i + b_i$, or the number of elements visited by Apply-Updates.

Running Time Let N_S be the number of steps required to perform a sequential *Delete-Min* operation.

$$
\begin{aligned}
N_S \leq & c_1 u_0 \log u_0 && \text{(Sorting the first level)} \\
& + c_2 \sum_{i=0}^{k} (b_i + (k-i)u_i) && \text{(Merging each level)} \\
& + c_3 \sum_{i=0}^{l} 2^i && \text{(Selection to redistribute)} \\
\leq & c(u_0 \log u_0 + kx + 2^{l+1})
\end{aligned}
$$

Here, $c, c1, c2, c3$ are arbitrary constants. Meanwhile, let N_P be to the number of steps required to perform a *Delete-Min* operation in parallel.

$$
\begin{aligned}
N_P \leq & c_1(\frac{u_0}{p}\log u_0 + \log^2 u_0) && \text{(Sorting the first level)}\\
& + c_2 \sum_{i=0}^{k}(\frac{b_i}{p} + \log b_i + \frac{(k-i)u_i}{p} + (k-i)\log u_i) && \text{(Merging each level)}\\
& + c_3 \sum_{i=0}^{l}(\frac{2^i}{p} + \log^2 2^i) && \text{(Selection to redistribute)}\\
\leq & c(\frac{u_0}{p}\log u_0 + \log^2 u_0 + \frac{kx}{p} + k^2 \log x + \frac{2^{l+1}}{p} + l^3)\\
= & c(\frac{N_S}{p} + \log^2 u_0 + k^2 \log x + l^3)
\end{aligned}
$$

Similarly, $c, c1, c2, c3$ are arbitrary constants. The critical path in the parallel algorithm costs $O(\log^2 u_0 + k^2 \log x + l^3)$ steps.

Cached Performance On a sequential or shared cache system, let M_S be the number of block transfers required to perform a *Delete-Min* operation.

$$
\begin{aligned}
M_S \leq & c_1\lceil\frac{u_0}{B}\log u_0\rceil && \text{(Sorting the first level)}\\
& + c_2 \sum_{i=0}^{k}(\frac{b_i}{B} + \frac{(k-i)u_i}{B}) && \text{(Merging each level)}\\
& + c_3 \sum_{i=0}^{l}\frac{2^i}{B} && \text{(Selection to redistribute)}\\
\leq & c(\frac{u_0}{B}\log u_0 + \frac{kx}{B} + \frac{2^{l+1}}{B})
\end{aligned}
$$

$c, c1, c2, c3$ are arbitrary constants, and B is the size of a cache block. On a distributed cache, things are a little worse since we have to incur a cache miss every time we access a level. Let the number of block transfers be M_D.

$$
\begin{aligned}
M_D \leq & c_1\lceil\frac{u_0}{B}\log u_0\rceil && \text{(Sorting the first level)}\\
& + c_2 \sum_{i=0}^{k}\lceil\frac{b_i}{B} + \frac{(k-i)u_i}{B}\rceil && \text{(Merging each level)}\\
& + c_3 \sum_{i=0}^{l}\lceil\frac{2^i}{B}\rceil && \text{(Selection to redistribute)}\\
\leq & c(\frac{u_0}{B}\log u_0 + \frac{kx}{B} + \frac{2^{l+1}}{B} + k)\\
= & c(M_S + k)
\end{aligned}
$$

As always, $c, c1, c2, c3$ are arbitrary constants, and B is the size of a cache block.

5.5 Concluding the Second Part

All the work for Buffer Heap are done during *Delete-Min*. The I/O bounds are the same or very close to their sequential counterpart. In practice, the value of k and l are small, and x is large so the critical path terms should not tend to dominate.

6 Conclusions

6.1 Conclusions

In the first section of this work, we studied whether we should use *Decrease-Key* in Dijkstra's algorithm. We observed in previous works that running Dijkstra's algorithm without *Decrease-Key* was faster on certain graphs. This is interesting since Dijkstra's algorithm traditionally used only priority queues with *Decrease-Key* operations. Since priority queues could be smaller and faster if they did not support *Decrease-Key*, there was potential for improvement to Dijkstra's algorithm. We expected that smaller heap sizes and fast path relaxation in Dijkstra-dec to offset the overheads of supporting *Decrease-Key*. Our results indicated that this was not the case. We ran tests on graphs of varying types and edge densities, and in all of these tests, we found that Dijsktra-nodec consistently outperformed Dijkstra-dec. When the edge density of the graph became very high, the performance difference became small. The running time and memory usage overheads in supporting *Decrease-Key* could not be justified in any of the graphs we tested. Therefore, until there is data indicating otherwise, we recommend using Dijkstra-nodec with an efficient priority queue that does not support *Decrease-Key* over using Dijkstra-dec for solving SSSP problems on any graph type.

In the second section of this work, we proposed a parallel cache-efficient priority queue based on Chowdhury and Ramachandran's Buffer Heap [3]. This data structure achieves the same number of cache misses as the sequential version under a shared cache, and slightly more under the assumption of a distributed cache. It achieves these bounds without knowing whether the cache is shared or distributed, and without knowing the layout of the cache. With the trend of modern computer systems moving towards more parallelism, and more dependence on complex, multi-leveled cache schemes, we believe that such a cache-oblivous algorithm can find itself adapting well to various levels of cache, which may not only change in size and block size, but also change from being distributed to shared caches or vice versa. The potential gains from cache-obliviousness was not free, however. Simple sequential algorithms such as merge became much more complex in order to satisfy the varying range of criteria.

One of the biggest difficulties came from balancing between parallelism and cache-efficiency. To achieve high parallelism, an algorithm ideally divides a data set into small independent problems that can be worked on simultanously. However, to be cache-efficient, especially on a distributed cache, an algorithm should work on at least one cache block worth of data at once. This is made difficult by the cache-oblivious model, since we cannot know how large a block is. Therefore, ideally we want to work on a large continuous block of data on a single processor. This directly conflicts with our wish for fine-grained

parallelism. On a shared cache, this is easier since many processors can access one block of cache without incurring additional cache misses.

6.2 Future Work

Since the original Buffer Heap and its no-*Decrease-Key* variant, the Auxiliary Buffer Heap, perform well on Dijkstra's algorithm, one could conceivably develop a parallel cache-oblivious implementation of Dijkstra's algorithm using the parallel Buffer Heap. Additionally, due to some of the difficulties mentioned above, the parallel subroutines do not achieve the lower bound for critical path length, but merely attempt to get close to the lower bound.

References

[1] Lan Roche. Experimental Study of High Performance Priority Queues. *University of Texas Computer Sciences Undergraduate Thesis*, 2007.

[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] Rezaul Alam Chowdhury, Vijaya Ramachandran. Cache-Oblivious Shortest Paths in Graphs Using Buffer Heap. *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA 2004), Barcelona, Spain, pp. 245-254, 2004.

[4] Peter Sanders. Fast Priority Queues for Cached Memory. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation, Volume 1619 of the Lecture Notes in Computer Science*. 1999. pp. 312-327. Springer-Verlag, Berlin/Heidelberg.

[5] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th SPDP*, pp. 169-177, 1996.

[6] DIMACS 9th Implementation Challenge. http://www.dis.uniroma1.it/ challenge9/

[7] Joseph Jájá *An Introduction to Parallel Algorithms* 1992: Addison-Wesley Publishing Company.

[8] Lingling Tong. Implementation and Experimental Evaluation of the Cache-oblivious Buffer Heap. *University of Texas Computer Sciences Undergraduate Thesis*, 2006.

[9] Erik D. Demaine. Cache-Oblivious Algorithms and Data Structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets, Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark* (BRICS 2002) June 27-July 1, 2002

[10] R.M. Karp. and V. Ramachandran. Parallel Algorithms for Shared Memory Machines. In *Handbook of Theoretical Computer Science : Algorithms and Complexity* MIT Press. 1990.