A Parallel Offline Data Structure for Searching

Cory Fraser
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
cfraser3@connect.carleton.ca

December 10, 2012

Abstract

Searching has always been a prominent problem in Computer Science. Many data structures exist to help with this problem each having its own strengths and weaknesses. A Parallel Buffer Tree delays answering queries on the tree in order to execute all operations in bulk. A Parallel Buffer Tree can process any N sequence of insert, remove, and find operations in $O(sort_P(N))$ parallel I/Os. $sort_P(N)$ represents the number of parallel I/Os it takes to sort N elements with P processors. Its I/O complexity is better than that of an equivalent parallel B-tree because of its batching nature.

This report provides experimental results for the parallel buffer tree. It shows that parallism speedup is high and that performance can be competitive with the C++ standard set. The report also shows that in order for the parallel buffer tree to be practical, a good solution to keeping track of, and maintaining find operation results must be devised. The method used in this report's implementation requires an object shared by all nodes of the tree which must be synchronized upon access.

1 Introduction

Searching has always been a prominent problem in Computer Science. Data structures to assist with searching, such as a binary search trees and B-Trees, have been created and continually improved over time. In recent years much work has been done on creating concurrent and parallel versions of these data structures that can take advantage of the multi-core nature of today's computers. One approach to speed up searching is to batch up multiple operations to perform on a data structure and perform them all at once with optimizations. A particular example of this sort of structure is described in a paper published in 2012, A Parallel Buffer Tree [6].

In the I/O memory model, a Buffer Tree can reduce the number of I/O operations required as compared to the popular B-Tree data structure. A Buffer Tree accomplishes this by removing the restriction that queries on the structure have to be answered immediately. Without this restriction, the tree is able to buffer operations and later perform them all once enough have been built up.

This project provides experiments to help verify some of the claims of the parallel buffer tree data structure described in [6]. An implementation of the basic version of this structure has been created which has been used to measure its performance in a variety conditions and inputs. The parallelism speedup of the parallel buffer tree is measured using

the implementation. Also, the tree's performance is compared against the C++ standard set to give a measure of its practicality.

In Section 2, we will review the relevant literature. In particular, sequential buffer trees, other external memory data structures and various parallel algorithms will be looked at. Section 3 will present the details of the parallel buffer tree algorithm, major design decisions used in the implementation, and then the experimental results obtained using the implementation. Section 4 concludes the paper with a summary of the results and possible future improvements.

2 Literature Review

The papers described in this review all fall into one of three categories. The first category will deal with papers that describe Buffer Trees and their history throughout the history of Computer Science. The second category will discuss other popular data structures that are used to reduce I/O operations in the I/O memory model. The last section will show where work on concurrent data structures and parallel algorithms is at.

2.1 Buffer Trees

Buffer Trees were first introduced in 1995 along with a few different variations of them [1]. Most search data structures made for the I/O model, would require $O(\log_B N)$ I/Os. Buffer Trees require only $O((\log_{M/B} N)/B)$ where M is the number of elements that fit in main memory and B is the number of elements per block [1, 3]. The Buffer Tree accomplishes this by removing the restriction that queries issued to the tree need to be answered immediately. In other words, the Buffer Tree is an offline data structure and may delay answering queries.

A Buffer Tree can be used to solve many different types of problems such as: sorting, list ranking and range searches, which can be used to solve geometry problems [1, 3]. The basic buffer tree supports insert, delete and find operations. A Priority Queue variant of the Buffer Tree exists which provides a deletemin operation. A Segment Tree variant also exists which can be used to solve stabbing queries. It should be noted that all these variants are external memory data structures and thus minimize I/O operations.

Shortly after Buffer Trees were introduced, a paper titled, Early Experiences in Implementing the Buffer Tree, implemented some of the Buffer Tree variants and recorded experimental results [4]. This paper used Buffer Trees to perform sorting and compared its performance to the Quicksort algorithm provided by C++. The Buffer Tree performed slightly slower than the quick sort for small input sizes, but as the input size increased, the Buffer Tree began to outperform quicksort. Many experiments with different block sizes were performed and it was found that a lower block size performed better. The experimental results showed that as input size increased, time increased only slightly above linear. Overall, this paper showed that the sequential Buffer Tree was quite viable for practical use which indicates that the parallel version may also be practically viable.

In the same way that the sequential Buffer tree works to minimize I/O operations in the I/O model, the parallel Buffer tree also works to minimize I/Os but in the Parallel External Memory (PEM) model instead [6]. In the PEM model, the shared memory between processors is considered the slow memory and each processor's local cache is considered the fast memory. Thus the number of I/Os from the shared memory to local cache is what is being minimized.

2.2 External Memory Data Structures

Since Buffer Trees focus on reducing the number of I/O operations required, it is a good idea to look into some other data structures that focus on this as well. The most popular of these data structures is probably the B-Tree. A B-Tree is a tree where each node stores multiple elements within an array. This means that all the elements within a particular node are all close together within memory and so only a single I/O operation is required to load all these elements. Insert, delete, and find operations in a B-Tree can be answered in $O(\log_B N)$ I/Os [2].

B-Trees also have many variants such as: weight-balances B-Trees, persistent B-trees, and string B-trees. String B-trees are interesting because they must deal with a problem created when trying to store elements in fixed sized arrays. Since strings can be of variable length a data structure called a blind trie is used in conjuntion with the B-tree. Each node of the B-tree stores the suffix of inserted strings and contain a blind trie which can be used to match on the rest of the query string.

Another external memory data structures is the R-tree. An R-tree is similar to a B-tree however there is no I/O guarantee of $O(\log_B N)$ for delete operations [2]. R-trees constructed by repeated insertion have a problem that resulting tree shape may not be optimal for querying. This problem is not easily solved making R-trees not as popular as B-trees.

2.3 Parallel/Concurrent Algorithms

When looking at parallel data structures, it is helpful to know what other parallel/concurrent data structures and algorithms exist as well as the problems they face. The parallel Buffer tree makes extensive use of parallel sorting to sort its operation buffers. The paper, Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors, provides algorithms for parallel sorting and prefix search all of which aim to reduce the number of I/O operations when considering the Parallel External Memory (PEM) model [5].

Two sorting algorithms, Distribution Sort and Mergesort are shown both of which have the an optimal I/O bound of $O(\frac{N}{PB}\log_{M/B}\frac{N}{B})$ where P is the number of processors used. The Distribution sort is a simpler algorithm but the mergesort has optimal work complexity speedup and scales better as P approaches N [5]. The mergesort makes use of recursion which works well for multicore usage.

As for recent concurrent data structures, current work appears to be aimed at reducing the degree of locking required when many operations are performed on a structure at the same time. The article, Data structures in the Multicore age, describes algorithms for creating both lock based, and lock-free stacks [7]. Other parallel data structures are also focusing on reducing or eliminating locking but all these other structures have the restriction that queries must return immediately. The parallel Buffer tree appears to be unique among these other structures in this regard.

3 Project Report

This section of the report will now describe in detail what work was done and why. Major design decisions for accomplishing the goals of this report will be addressed as well as the consequences of them. Once all the details are clear, the experimental results obtained will be shown and analyzed.

3.1 Parallel Buffer Trees

There are two main variations of Parallel Buffer Trees that are described in [6]. There is the basic Parallel Buffer Tree which supports Insert, Find, and Remove operations. The other more advanced variation supports another operation which is range querying. This section will now take a more detailed look at both of these variations.

3.1.1 Basic Version

The basic Parallel Buffer Tree which supports Insert, Find and Remove operations can process any N sequence of these operations in $O(sort_P(N))$ parallel I/Os. $sort_P(N)$ represents the number of parallel I/Os it takes to sort N elements with P processors.

The parallel buffer tree is an (a,b)-tree where $a = f/4$, $b = f$ and $f >= PB$. B represents the number of elements stored within a node in the tree also known as the block size. In (a,b)-trees, all leaves are the same distance from the root, and internal nodes store routing elements which determine to which child an element should be sent to. In addition, internal nodes also need to store an operation buffer of capacity $\theta(fB)$. Each operation consists of three data fields: the type of operation (insert, remove, find), the value the operation affects, and the time that the operation was submitted.

There are two types of internal nodes which the parallel buffer tree algorithm often makes use of. A fringe node is an internal node whose children are all leaves of the tree. A non-fringe node is an internal node whose children are not leaves. This differentiation is important when it comes to emptying the operation buffer of an internal node.

Since the I/O complexity of the parallel buffer tree is based on sorting, it is not difficult to see that parallel sorting is an important part of the data structure. In fact, every time an internal node's operation buffer fills up, a parallel sort is required to sort these operations based on the value they affect and secondarily by the time the operation was submitted. This sorting serves two purposes, one being that all operations affecting a certain value become adjacent to each other meaning that: find operations can possibly be answered, duplicates can be filtered out, and matching insert/remove operations can cancel out. The other purpose is that the operation buffer can now be distributed to the children nodes much more easily since all operations for a specific child are adjacent to each other.

The two main internal operations for a parallel buffer tree are non-fringe node buffer emptying and fringe node buffer emptying. These operations trigger whenever a node's operation buffer fills up and are responsible for updating the structure of the tree. A node's operation buffer is considered full when it reaches a size of fB. To empty a non-fringe node's buffer, first sort the operations in the buffer, then filter out duplicate operations, answer find queries, and cancel out matching insert/remove operations. With the remaining operations, distribute them to the node's children using the routing elements. The final step of the non-fringe node buffer emptying process is to recursively empty the buffers of any children nodes which have now become full.

When a fringe node's operation buffer fills up, the first step is to take all the data elements stored within this node's children, convert them into insert operations with a $-\infty$ timestamp and add them to the operation buffer. With this done, sort the operations based on the values they affect and secondarily by their timestamp. At this point, all remove operations can be cancelled out, all find queries can be answered, and duplicate insert operations can be removed. With the remaining insert operations, depending on how many are left, one of two main cases should be performed. If the number of operations remaining

is $O <= fB$ then all the operations can be made into the new leaves of this fringe node. If the number of operations remaining is $O > fB$ then the operations should be split into groups of size $fB/2$ elements. The first group becomes the children of this fringe node. Each remaining group becomes a sibling of this fringe node. Since creating new siblings means that this fringe node's parent could have more than the maximum number of children f the tree may need to be rebalanced.

Rebalancing a node consists of splitting the node into two nodes. Each node has half the original node's children. If the node that needs to be split is the root of the tree, then a new root needs to be created. Also, since this node splitting adds another child to the split node's parent, that parent node may also need rebalancing so the parent node should be recursively checked.

3.1.2 Range Query Extension

The basic parallel buffer tree described in 3.1.1 can be extended to support the additional operation RangeQuery. The RangeQuery operation takes two values as parameters and will report all elements that fall within the range created by these values. The I/O complexity of this extended tree becomes $O(sort_P(N) + K/PB)$ where K is the size of the output generated by the N operations. All the properties described in the basic parallel buffer tree algorithm still apply here. A new restriction must be placed on the operations sent to the tree however, which is that they are well-formed. This means that every insert operation must insert an element that doesn't already exist and that all remove operations must remove an element that is already in the tree.

Each node must keep track of another piece of information in this variation. The range of values that is stored within the node and all its children. For the root node, its range is $(-\infty, +\infty)$. Using these ranges, a range query in a node's operation buffer can be propagated down to all children whose range contains at least some of the range query.

In order for range queries to work correctly, a time order representation relating remove, range, and insert operations needs to be created. This representation requires that all remove operations have an older timestamp than range operations, and that range operations have an older timestamp than insert operations. The representation works only if the well formed restriction stated earlier is met. The operations stored in a node's buffer is still sorted as normal, but subsequences of that buffer will sometimes also be sorted using this time order representation.

Applying the time order representation when emptying a node's operation buffer is a complicated process and is not the focus of this report. Thus, no further detail for range queries will be looked at here. The full details can be found in [6].

3.2 Implementation Design Decisions

This section will now detail the major design decisions made when creating an implementation of the Parallel Buffer Tree. The first major decision is that only the basic variation of the parallel buffer tree has been implemented. This means that the Insert, Remove, and Find operations are supported.

To measure parallelism of the algorithm, the parallel buffer tree results are to be compared against the results of a parallel buffer tree with reduced worker threads. This method works well to measure parallelism because in the sequential buffer tree, it's I/O complexity is also given by the sort algorithm used. A parallel merge sort is used in this implementation

of the parallel buffer tree, and a regular merge sort has optimal complexity so reducing the worker count to one effectively produces a sequential merge sort.

To measure the performance of a parallel buffer tree against the equivalent sequential data structure, a C++ std::set data structure will be used. The C++ set is very highly optimized and also provides the same operations: insert, remove and find. Both the parallel buffer tree and set maintain the ordering of elements stored within it allowing for iterating in sorted order. If the parallel buffer tree results can even come close to the std::set's results, then it is likely that with performance tuning the buffer tree could match or beat the set for some cases.

This implementation of the parallel buffer tree has made two configurable parameters available. These parameters are block size and maximum branch size. It could be the case certain values for the parameters perform better than others. Thus, different values for these parameters will be used to try and find improved performance results.

Part of the buffer tree's batching nature means that it will not answer queries right away. This means that some way to retrieve these results at a later time needs to be devised. This implementation creates a query ID for every find operation submitted, stores the ID within an operation's type field and returns the ID to the submitter. The submitter can then use this ID to poll for the results of the query to see if it has been calculated yet.

The next two sub-sections will describe two pieces of software that were used in the implementation to help with certain aspects of it.

3.2.1 Cilk++

The compiler used to compile the code of this implementation was the Intel Cilk Plus implemented in the GCC compiler. Cilk provides mechanisms to make taking advantage of parallelism much easier. Recursive algorithms are especially easy to parallelize. Since the parallel merge sort is recursive, it makes sense to use Cilk for this implementation.

There are some non-evident drawbacks to using Intel Cilk Plus. A major one being that the newest free version of it is no longer supported by Intel which means that features of it start to break down as newer versions of operating systems and programming libraries become available. One such example of this encountered while implementing the parallel buffer tree is that the cilkview tool that comes with Cilk Plus doesn't work with a Linux kernel of version 3.6.

Another problem that was continually encountered while creating an implementation was that mixing Cilk Plus with the C++ libraries installed on a current Linux operating system did not always provide the expected results. For example, Posix mutexes used for thread synchronization did not perform locking as expected. Also, when freeing memory allocated to the heap space via the C++ delete or free keywords, the program process would have a small chance to crash instantly printing out a Cilk error message. This implementation was able to get around the delete and free problem by simply not releasing allocated memory and allowing memory leaks to build up. This works for this report since results could be gathered while running the program for only short lengths of time but if a parallel buffer tree were ever to be used in practice, these memory leaks would absolutely need to be fixed.

Other than the problems listed above, Cilk made it much easier to parallelize pieces of the algorithm. If an implementation of Cilk were ever part of an official GCC release then one wouldn't have to worry about these problems.

3.2.2 Boost Library

Boost is a platform independent C++ library which provides many common features that the standard template library (STL) does not. Boost aims to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Those who are familiar with some of the features of the Java programming languages Application Programming Interface (API) will notice that much of what is provided by Boost is very similar to what is provided by Java.

One feature of Boost that was used extensively by the implementation was its thread synchronization. The thread synchronization provided by Boost is very similar to Javas style of synchronization. A mutex object can be used to make certain critical sections of code only allow for one thread to execute upon it at any given time. The mutex object can be used in the same way the synchronized keyword in Java is used.

It can be seen that the use of Boost in this project provided significant benefits, all of which are platform independent. If in the future, running the algorithm on a Windows operating system becomes a requirement, changes to the Boost parts of the code will be minimal if any at all. Also, since Boost follows common programming practices, any experienced programmer who hasnt used Boost before should be able to recognize familiar patterns and quickly figure everything out.

3.3 Experimental Results

This section will now show and discuss the results of three different types of measurements. The first set of results will show the parallelism speedup gained by using the parallel buffer tree over the sequential version. The second set of results compares the parallel buffer tree's performance to the performance of the standard C++ set. The last set of results attempts to find optimal values for the maximum branches and block size parameters.

All the following results were collected on an Intel Quad core 2.4GHz CPU running the Fedora 16 Linux operating system. The system is configured with 12 GB of random access memory (RAM). All time measurements were taken multiple times and averaged to minimize outlier results.

3.3.1 Parallism Speedup

The parallism results obtained can be seen in Figure 1. What is shown is the run times for each type of operation with a four worker Parallel Buffer Tree and a two worker parallel buffer tree. This section will now look at each operation type's results individually.

When looking at the insert operation times it looks like the parallism speed up is almost close to two. To get a better idea, the actual measured values can be used. For an input size of 10000, the two core insert took 315ms and the four core insert took 221 which comes out to a speedup of 1.43. This isn't very good but if a higher input size like 1600000 is looked at instead the speedup comes to $9798/5354 = 1.83$ which is much better.

When looking at the remove operation times, these also look close to two. The remove operation is also the fastest operation which is unusual. For an input size of 100000, the speedup is $396/207 = 1.91$ which is very high. Oddly though for an input size of 1600000, the speedup is $7411/4329 = 1.72$ which is lower than the smaller input size's speedup. This can be explained if both results were supposed to be around 1.8 usually and were just a little off. The high speedup for a small input which goes against the insertion results can be explained by the fact that when the remove operations are executed, insertions have

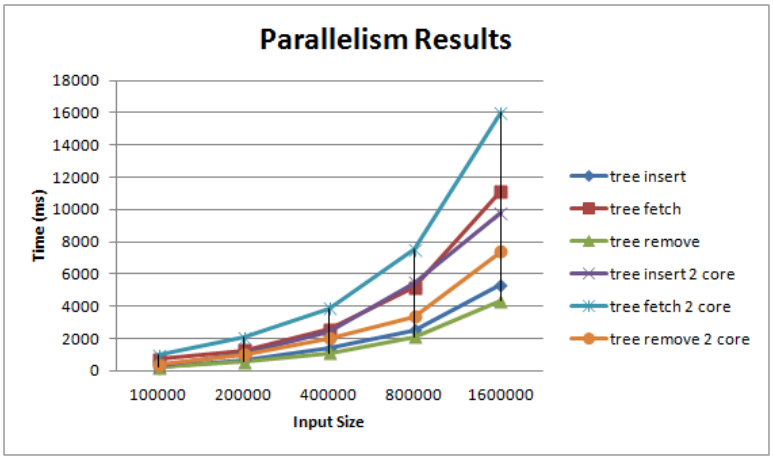

Figure 1: Measured Running Times of a 2 vs 4 core Buffer Tree

already been previously done which means the tree already has a sufficient number of nodes to provide parallelism.

The find operation results are surprising in two ways when looking at Figure 1. Find is the slowest operation and its parallelism speedup looks very low. This large slowdown can be accounted to the fact that Find operations need a way to report results once they are calculated. This requires that the tree keep track of query IDs and their associated results. Whenever a find operation can be answered, it has to be reported to an object shared by all nodes in the tree which requires synchronization. This synchronization reduces parallelism and has a significant effect on find performance.

Overall, except for the find operation, parallelism of the parallel buffer tree is very good with a decent amount of input. These results do indicate that something needs to be done with the implementation of the find operation to make this data structure practical.

3.3.2 Comparison vs Efficient Sequential Algorithm

The performance comparison of the Parallel Buffer Tree against the C++ standard set can be seen in Figure 2. It is abundantly clear that the C++ set greatly outperforms the parallel buffer tree. This alone is not enough to invalidate the parallel buffer tree however. The insert and remove operations are only about a factor of 2.5 times slower than the same operation's times for the C++ set. This time difference could be remedied with about another 10 CPU cores based on the good parallelism speed seen in Section 3.3.1. Also, since the parallel buffer tree operates in an external memory model, it can theoretically be modified to work well using hard disk space as well as RAM. This would raise its size limit to a number much higher than the C++ set.

The results of Figure 2 do show one troubling piece of information. The fetch operation for the parallel buffer tree is the slowest operation of all. In the C++ set's case, the find operation is much faster than the insert and remove operations. The reason for the poor performance of the find operation is the same as discussed in Section 3.3.1, the delayed answering of find operations requires that answers later be reported to a shared object which requires synchronization.

Overall, the insert and remove operations for the parallel buffer tree are competitive

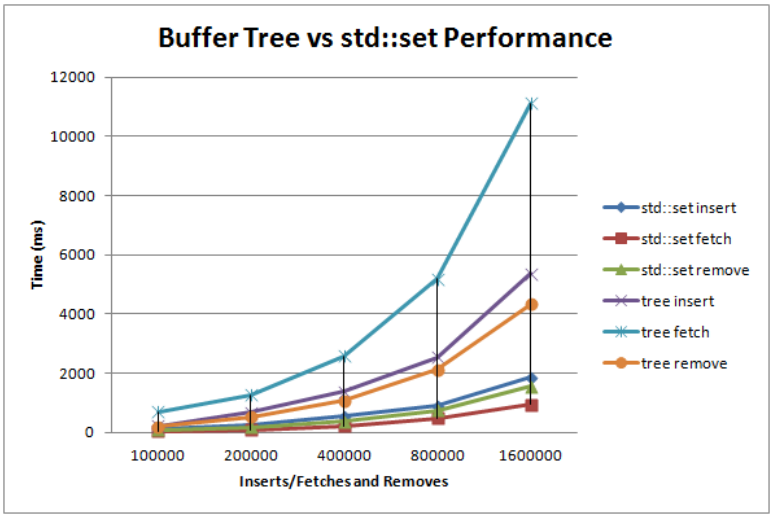

Figure 2: Measured Running Times of the std::set and the Parallel Buffer Tree

enough with the performance of the C++ set in that with some performance tuning or a few more cores the tree would surpass the set. A big problem still needs to be solved with the performance of the tree's find operations. An alternate method of reporting the results of a find query needs to be devised which doesn't require as much synchronization to make the parallel buffer tree practical.

3.3.3 Effect of Branch and Block Size

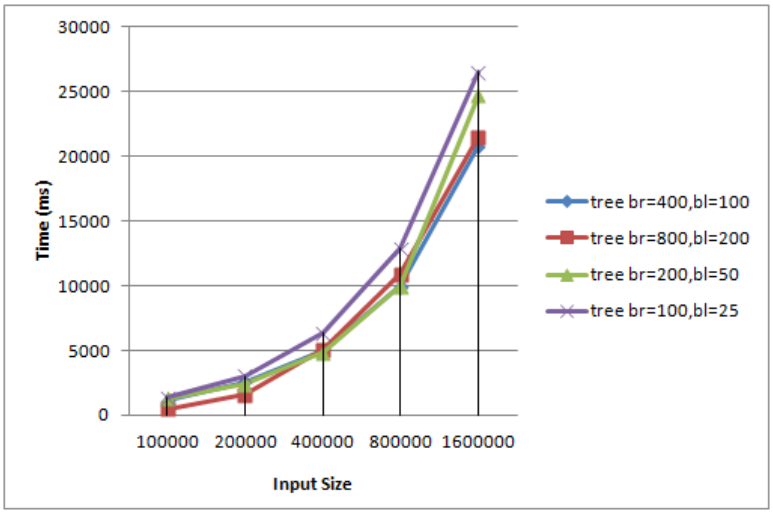

Figure 3: Measured Running Times of various branch and block sizes

The performance comparison of the Parallel Buffer Tree with varying maximum branch and block sizes can be seen in Figure 3. It can be seen that the effect of these two parameters is minimal. It can also be seen that a very low branch size and block size does perform

slightly worse than higher values for these parameters.

The two best performance configurations for the large input size was the highest branch and block sizes. The 400 branch size, 100 block size configuration performed the same as the 800 branch size, 200 block size configuration at the highest input size. What is interesting is that the largest configuration values performed faster at small inputs than the other configuration did. This can be explained because the larger the branch and block sizes, the more operations it takes to fill an operation buffer and cause the tree to actually do work. So in the case of the largest branch and block configuration, its buffer never actually flushes at the low input sizes.

Overall the effect of branch size and block size on the parallel buffer tree's performance is minimal. Too small a value yields some performance slow downs. Increasing these values beyond the optimal values doesn't appear to have any penalty which makes it easier to optimally configure.

4 Conclusion

From the results shown in this report, it can be seen that the Parallel Buffer Tree provides a good parallelism speedup over its sequential counterpart. Also, it was shown that the insert and remove operations can perform competitively against the C++ standard set given enough cores or some performance tuning.

This report also shows that a significant problem still exists which is how to keep track of and report find operation results in a performant and high parallelism way. The obtained results show that find operations perform much too slow relative to insert and remove operations with low parallelism. This problem currently prevents the parallel buffer tree from being practical.

The implementation produced along with this report is far from perfect. Since an unsupported version of Cilk was used, many problems occurred during implementation which prevented it from running optimally, such as the memory releasing crash. If Cilk is ever included in an official GCC release, it would be worth it to update the implementation to run with this newer version of Cilk and see how the results change.

A future extension to this work could be to compare the parallel buffer tree to a parallel or sequential B-tree implementation. A comparison of two external memory based data structures would provide a more equivalent comparsion than the C++ set provided. Also, the parallel buffer tree range query extension could be implemented and compared with the basic version to determine any tradeoffs of the extension.

References

[1] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 334–345, 1995.

[2] L. Arge. External memory data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 2003.

[4] J. Sack R. Velicescu D. Hutchinson, A. Maheshwari. Early experiences in implementing the buffer tree. *Citeseer*, 1997.

[5] M. Nelson L. Arge, M. Goodrich. Fundamental parallel algorithms for private-cache chip multiprocessors. *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206, 2008.

[6] N. Zeh N. Sitchinava. A parallel buffer tree. *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 214–223, 2012.

[7] N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.