LITERATURE REVIEW: Parallel Offline Data Structures for Searching

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

October 16, 2012

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 will attempt to verify the claims of the parallel buffer tree data structure described in [6]. An implementation of the basic version of this structure will be created which can be used to measure its performance in a variety conditions and inputs. If time permits, the extension to the buffer tree that enables range searching could be implemented as well.

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.

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.