Early Experiences in Implementing the Buffer Tree[1]

David Hutchinson

Anil Maheshwari

Jörg-Rüdiger Sack

Radu Velicescu

School of Computer Science
Carleton University
Ottawa, Ont., Canada K1S 5B6
e-mail {hutchins,maheshwa,sack,velicesc}@scs.carleton.ca

ABSTRACT

Computer processing speeds are increasing rapidly due to the evolution of faster chips, parallel processing of data, and more efficient software. Users today have access to an unprecedented amount of high quality, high resolution data through various technologies. This is resulting in a growing demand for higher performance input and output mechanisms in order to pass huge data sets from the *external memory (EM)*, or disk system, through the relatively small main memory of the computer and back again. In recent years, research into external memory algorithms has been growing to keep pace with the demand for innovation in this area.

EM algorithms for individual problems have been developed but few general purpose EM tools have been designed. A fundamental tool is the buffer tree, an external version of the (a,b)-tree. It can be used to satisfy a number of EM requirements such as sorting, priority queues, range searching, etc. in a straightforward and I/O-optimal manner.

In this paper we describe an implementation of a buffer tree. We describe benchmarking tests which lead to an experimental determination of certain parameter values different from those originally suggested in the design of the data structure. We describe implementations of two algorithms based on the buffer tree: an external memory treesort, and an external memory priority queue. Our initial experiments with buffer tree sort for large problem sizes indicate that this algorithm easily outperforms similar algorithms based on internal memory techniques. With some tuning of the buffer tree parameters we are able to obtain performance consistent with theoretical predictions for the range of problem sizes tested. We include comparisons with TPIE Merge Sort.

We conclude that (a) the buffer tree as a generic data structure appears to perform well in theory and practice, and (b) measuring I/O efficiency experimentally is an important topic that merits further attention.

1. Introduction

The Input/Output bandwidth between fast internal memory and slower secondary storage is the bottleneck in many large-scale applications, such as multimedia, GIS, land information systems, seismic databases, satellite imagery, digital libraries, real-time applications and virtual reality. A typical disk drive is a factor of $10^5 - 10^6$ slower in performing a random access than is the main memory of a computer system. Present methodologies for addressing the performance issues involving secondary storage can be classified as follows [14]:

[1] Research supported by the Natural Sciences and Engineering Research Council of Canada.

- increasing storage device parallelism, which improves the bandwidth between secondary memory and main memory,
- exploiting locality of reference via organization of the data and processing sequence,
- overlapping I/O with computation, e.g. using prefetching.

Some of the earliest work in external memory algorithms was done by Floyd [13] and Hong and Kong [16] who studied matrix operations and fast Fourier transforms. Lower bounds for a number of problems related to sorting were presented by Aggarwal and Vitter [1]. The classical I/O model was introduced by Vitter and Shriver [28]. The uniprocessor, single disk version of this model represents an EM computer system as a processor, some fixed amount of internal memory, and a disk. It is described by the following parameters:

N is the number of elements in the problem instance,

M is the number of elements that can fit in the internal memory,

B is the number of elements per block,

where $M < N$ and $1 \leq B \leq \frac{M}{2}$.

An *Input/Output operation (I/O)* is the process of reading or writing a block of B contiguous data elements to or from the disk. The I/O complexity of an algorithm is defined as the total number of I/Os that an algorithm performs. It is assumed in this model that the internal computation is free. Several other I/O models have been proposed, see e.g. [28, 12, 20]. The theoretical framework of the algorithms in this paper is based on the Parallel Disk I/O Model (PDM) proposed by Vitter and Shriver [28].

Permutations and sorting have been very widely studied in the context of this model, see [2, 1, 9, 20, 27]. Algorithms for problems in computational geometry [15, 4, 3, 8, 23], graph theory [7, 19, 3], and GIS [4] have been presented. A number of general paradigms for designing external memory algorithms have been proposed. These include *simulation* [7, 12, 15, 22], *merging* [5, 20], *distribution* [21, 15, 7], and *data structuring* [2].

Recently there has been an increasing interest in implementation and experimental research work targeted to I/O efficient computation. Research work in this area includes:
(i) The TPIE (Transparent Parallel I/O Environment) project of Vengroff and Vitter [26, 24] which aims to collect implementations of existing algorithms within a common framework, and to make development of new implementations easier.
(ii) Experiments by Chiang [6] with four algorithms for the orthogonal segment intersection problem.
(iii) Cormen et al. [11, 10] have reported on a number of implementation issues and results relating to I/O efficient algorithms, including FFT computations using parallel processors, and FFT, permutations, and sorting using the Parallel Disk Model.

Motivated by the goal of constructing I/O efficient versions of commonly used internal memory data structures, Arge [2, 3] proposed the data structuring paradigm, and in particular the *Buffer Tree*. A buffer tree is an external memory search tree. It supports operations such as insert, delete, search, deletemin, and it enables the transformation of a class of internal-memory algorithms to external memory algorithms by exchanging the data structures used. A large number of external memory algorithms have been proposed [3, 2] using the buffer tree data structure, including sorting, priority queues, range trees, segment trees, and time forward processing. These in turn are subroutines for many external memory graph algorithms, such as expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning trees, ear decomposition. There are a number of major advantages of the buffer tree approach. It applies to a large class of problems whose solutions use search trees as the underlying data structure. This enables the use of many normal internal memory algorithms, and "hides" the I/O specific parts of the technique in the data structures. Several techniques based on the buffer tree, e.g. time forward processing [3], are simpler than competitive EM techniques, and are of the same I/O complexity, or better, with respect to their counterparts.

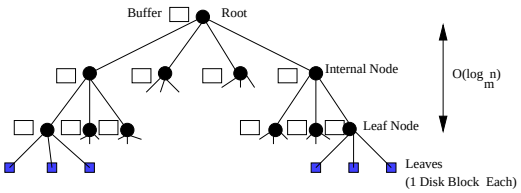

Figure 1: The buffer tree

In this paper, we describe the issues arising out of our implementation of the buffer tree. We present an implementation of the buffer tree, and show the flexibility and generality of the structure by implementing EM sorting and an EM priority queue. To test the efficiency of our implementation we used sorting as an example. For data sets larger than the available main memory, our implementation of buffer tree sort outperforms internal memory sort (e.g. qsort) by a large and increasing margin. We use an EM merge sort algorithm from TPIE to provide comparative performance results for the larger problem sizes. We observe that certain parameters suggested in [2, 3] may not provide the best results in practice. By tuning these parameters we obtained improved results while maintaining the same asymptotic worst case I/O complexity. The buffer tree is a conceptually simple data structure, and it turned out that implementation of these applications based on the buffer tree was straightforward. Therefore we can support the claim that the buffer tree is a generic EM data structure that performs well in theory and practice.

While the buffer tree gives us an I/O-optimal sort, our timing studies of the implementation indicate that its performance is sensitive to some nonlinearities in the environment or algorithm. Experimental results show that these non-linearities are reduced by an optimal choice of parameters.

2. The Buffer Tree

In this section we describe the buffer tree data structure of Arge [2, 3], with two update operations, namely insert and delete. Subsequently, we will discuss how we can perform sorting and maintain a priority queue using a buffer tree.

Let N be the total number of update operations, M be the size of the internal memory and B be the block size and set $m = M/B$ and $n = N/B$. The buffer tree is an (a, b)-tree [17], where $a = m/4$ and $b = m$, augmented with a buffer in each node of size $\Theta(m)$ blocks. Each node (with the exception of the root) has a fan out (number of children) between $m/4$ and m. Each node also contains partitioning elements, or "splitters" which delimit the range of keys that will be routed to each child. The number of splitters is one less than the fanout of the node. The height of the buffer tree is $O(\log_m n)$ (see Figure 1). Since the buffer tree is an extension of the (a, b)-tree, the computational complexity analyses of the various (a, b)-tree operations still apply. The buffers are used to defer operations, to allow their execution in a "lazy manner", thus achieving the necessary blocking for performing operations efficiently in external memory. A buffer is *full* if it has more than $m/2$ blocks.

For any update operation, a *request element* is created, consisting of the record to be inserted or deleted, a flag denoting the type of the operation, and an automatically generated time stamp. Such request elements are collected in the internal memory until a block of B requests has been formed. The request elements, as a block, are inserted into the buffer of the root using one I/O. If the buffer of the root contains less than $m/2$ blocks there is nothing else to be done in this step. Otherwise the buffer is emptied by a *buffer-emptying* process.

The buffer-emptying process at an internal node requires $O(m)$ I/Os, since we load $m/2$ blocks into the internal memory and distribute the elements among the $\Theta(m)$ children of that node. A buffer-emptying process at a leaf may require rebalancing the underlying (a, b)-tree. An (a, b)-tree is rebalanced by performing a series of "splits" in the case of an insertion or a series of "fuse" and "share" operations in the case of a delete [17]. Before performing a rebalance operation, we ensure

that the buffers for the corresponding nodes are empty. This is achieved by first doing the buffer-emptying process at the node involved. The deletion of a block may involve the initiation of several buffer-emptying processes. By using dummy blocks during the deletion process, a buffer emptying process can be protected from interference by other processes. (See [2] for details.)

The analysis (i.e., the I/O complexity) of operations on the buffer tree is obtained by adapting the amortization arguments for (a, b) trees [17]. Each update element, on insertion into the root buffer, is given $O(\frac{\log_m n}{B})$ credits. Each block in the buffer of node v holds O(the height of the tree rooted at v) credits. For an internal node, its buffer is emptied only if it gets full and moreover this requires $O(m)$ I/O's. Therefore, ignoring the cost of rebalancing, the total cost of all buffer emptying on internal nodes is bounded by $O(n \log_m n)$ I/Os. The total number of rebalance operations required in an (a, b)-tree, where $b > 2a$, over K update operations on an initially empty (a,b)-tree, is bounded by $K/(b/2 - a)$. Therefore, for N update operations, on an $(m/4, m)$-tree, the total number of rebalance operations is bounded by $O(n/m)$. Moreover, each rebalance operation may require a buffer-emptying process as well as updating the partitioning elements, and therefore may require up to $O(m)$ I/Os. Thus the total cost of rebalancing is $O(n)$ I/Os. The cost of emptying leaf nodes is bounded by the sorting operation. We summarize.

Theorem 1. *(Arge [2, 3]) The total cost of an arbitrary sequence of N intermixed insert and delete operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations.*

2.1. Sorting

A buffer tree can be used to sort N items as follows. First insert N items into the buffer tree followed by an *empty/write operation*. This is accomplished by performing a buffer-emptying process on every node starting at the root, followed by reporting the elements in all the leaves in the sorted order. This can be done within the complexity of computing the buffer tree data-structure.

Corollary 1. *(Arge [3]) N elements can be sorted in $O(n \log_m n)$ I/O operations using the buffer tree.*

The PDM compares competing EM algorithms according to the asymptotic number of I/O operations they require to solve a given problem of size N. By this model, the buffer tree sorting algorithm [3] is optimal, as the number of I/O operations matches the lower bound $\Omega(n \log_m n)$ for the sorting problem [1].[2] In practice, however, other factors can also affect the running time. Cormen and Hirschl [10] observe that many PDM applications are not I/O bound, which suggests that CPU time is an important factor to be considered. A model which includes both I/O and CPU time is presented in [12].

2.2. Priority Queues

A dynamic search tree can be used as a priority queue, since in general, the leftmost leaf of the search tree contains the smallest element. We can use the buffer tree for maintaining a priority queue in external memory by permitting the update operation described previously for insertion into the priority queue and adding a *deletemin* operation. It is not necessarily true that the smallest element is in the leftmost leaf in the buffer tree, as it could be in the buffer of any node on the path from the root to the leftmost leaf. In order to extract the minimum element, i.e., execute the deletemin operation, a buffer-emptying process must first be performed on all nodes on the path from the root to the leftmost leaf. After the buffer emptying the leftmost leaf consists of the B smallest elements, and the children of the leftmost node in the buffer tree consists of at least the $\frac{m}{4}B$ smallest elements. These elements can be kept in the internal memory, and at least $\frac{m}{4}B$ deletemins can be answered without doing any additional I/O. In order to obtain correct results for future deletemins, any new insertion/deletion must be checked first with these elements in the internal memory. This realization of a priority queue does not support the changing of priorities on elements already in the queue.

Theorem 2. *(Arge [3]) The total cost of an arbitrary sequence of N insert, delete and deletemin operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations.*

[2] For a single disk. n is the number of disk blocks in the problem. m is the number of disk blocks that fit into the memory size M.

3. Implementation Issues

3.1. Implementation of the Buffer Tree

3.1.1. The (a,b)-Tree

The buffer tree is an (a,b)-tree with buffers added to each tree node. One source of code for an (a,b)-tree is LEDA [18]. It quickly became clear, however, that this code was designed specifically for internal memory usage, as nodes were linked by many pointers to support a wide range of higher level operations. Converting the various pointers of the (a,b)-tree implementation to external memory representations turned out to be time consuming and ineffective for a data structure that was required to be I/O efficient. Too many I/O operations were required to update a single field in a child or parent of the node in memory to give attractive EM performance.

3.1.2. The Buffers

Each node of the buffer tree has an associated buffer, which may contain between 0 and $m/2$ blocks (between 0 and $kM/2$ bytes) of data[3] which have not yet been inserted into the (a,b)-tree part of the buffer tree. The fanout of an internal node is at most m. Therefore, for a buffer tree consisting of ℓ levels, there may exist up to $m^{\ell-1}$ buffers, each $kM/2$ bytes in size.

Our implementation currently models each buffer as a Unix file. Due to restrictions on the number of Unix files that could be open at a time, each buffer file is closed after use and reopened when necessary. The time required by file open and close operations, as measured in our tests, was small. However, preliminary experiments suggest that this scheme may limit the effectiveness of asynchronous I/O, since the file *close* operation must wait for any outstanding I/O to complete. In this paper we report primarily on our experiences using synchronous I/O.

3.1.3. Compatibility, Usability and Accessibility

We wanted our implementation to be compatible with the use of LEDA [18] and with TPIE [26, 25]. The large number of algorithms and data structures available in LEDA forms an attractive context for implementation of internal memory algorithms, which are often components of a larger external memory system. For example, the external memory priority queue uses an internal memory priority queue as a component. The collection of efficient external memory techniques provided by TPIE forms an attractive workbench for building and testing new EM implementations. We expect to use TPIE services in a later version of our implementation and perhaps offer it as an addition to the library. We chose C++ as our implementation language to preserve potential compatibility with these code libraries. In addition, we adopted the automated documentation tools from LEDA.

3.2. Implementation of an EM Sort Using the Buffer Tree

A buffer tree can be made to sort a data set simply by inserting the data into the tree, and then force-emptying the buffers. Our implementation allows the leaves to be read sequentially from left to right, thus the implementation of sorting was straightforward.

3.3. Implementation of an EM Priority Queue

The buffer tree can be modified to construct an I/O-optimal priority queue with *insert* and *deletemin* operations in EM [3]. Our implementation is obtained as follows:

- The leftmost leaf node of the buffer tree, together with its associated $\frac{m}{4}$ to m leaf blocks are kept in internal memory, instead of on disk. The (a,b)-tree nodes on the path from the root to the leftmost leaf are also kept in memory. For typical values of M, N, B, a, and b these (a,b)-tree nodes make a negligible impact on internal memory consumption.
- The data records in memory are organized using an appropriate internal memory priority queue. In our initial implementation, this is a conventional heap, originally obtained from LEDA.

[3] This may increase temporarily during a buffer emptying process.

- Any requests that would normally be inserted into the root buffer of the buffer tree are first compared to the leftmost splitters of the (a,b)-tree nodes on the path from the root to the leftmost leaf. As argued above, this gives a small constant number of comparisons in practice. If a request would be routed to the cached data blocks, it is inserted directly into the internal heap. Otherwise, it is inserted into the buffer tree in a "normal" fashion.

- The balance of the underlying (a,b)-tree is maintained in a normal fashion. If the leftmost leaf node underflows, sharing or fusing with siblings will occur. If it overflows, splitting may occur as it would in an (a,b)-tree.

3.4. Testing Platform and Parameters

We performed most of our development and experimental work on a network of sixteen 166 MHz Pentiums, each with 32 MB of internal memory and a pair of 2 GB hard disks used exclusively for data storage. A central 1 GB hard disk is available via NFS for program storage. The processors each run the Linux operating system. Although we did not use it except for NFS access to the central program storage, the processors are interconnected by a fast ethernet switch. We found that our experimental timings were reproducible between processors, and so we were able to run multiple timing tests independently, simultaneously, and reliably on this platform.

We chose $k = 16$ bytes as the record size for our tests. A record (sometimes we will call it an element) consisted of four integer (4 byte) fields: a key, an associated "data field", a timestamp, and an operation type (insert/delete/query) field. We chose $kB = 4096$ bytes, since this was the system page size. We chose $kM = 500KB$ (m=125), which is small, but allowed us to choose manageable problem sizes (from a disk space point of view) yet still apply stress to our algorithms.

3.5. Testing

In order to obtain meaningful performance results, we attempted to control the following factors:

Contention with other processes or users of the machine for machine resources such as memory, CPU and disks: We perform the testing on dedicated machines, and so the results were not affected by other users. The Unix operating system spontaneously initiates various system tasks to perform routine maintenance and monitoring functions, thus it is not easy to avoid contending with these tasks for system resources. Smaller test runs may vary significantly due to these effects. However, on the larger test runs, the influence of system tasks on the run time can generally be ignored.

Virtual memory effects such as the 'transparent' behaviour of the operating system to swap parts of the program image between main memory and secondary storage: The swapping of portions of a task to disk to make room for another activity can occur without warning or notification. In our tests we attempted to minimize the likelihood of this occuring by choosing kM, the problem size in bytes, to be much smaller than the physical memory size. For instance, on our Linux machines with 32MB of physical memory, we performed the majority of our tests using $kM = 500KB$. Another tactic is to use the Linux *mlockall* service call to lock the application into memory. This seems to work reasonably well in some cases, and does allow the application to use up to 50% of the physical memory without fear of being affected by virtual memory effects. However, the requesting program must be running with "root" privileges for the request to be honoured.

3.6. Test Results

3.6.1. Comparison to Quicksort

We found that the buffer tree easily outperformed the "built-in" internal memory quicksort technique. A simple quicksort program was written using the built-in C function "qsort". Figure 2 shows the results for a range of problem sizes. For larger input sets, the (recursive) quicksort program ran out of stack space on our system, but by that time the internal sort was slowing due to virtual memory effects and the buffer tree was already outperforming it.

3.6.2. Tuning the Buffer Tree

We discovered that the value of b relative to m is important to the performance of buffer tree sort on random data. For buffers of size $\frac{m}{2}$ and $(a, b) = (\frac{m}{4}, m)$ as suggested in [3], we obtain (partial)

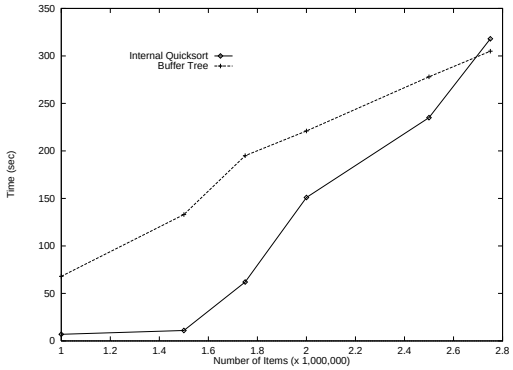

Figure 2: Timings for Internal Quicksort and Buffer Tree Sort on Random Inputs: For problem sizes larger than 2.8 million items the internal quicksort failed due to lack of internal memory.

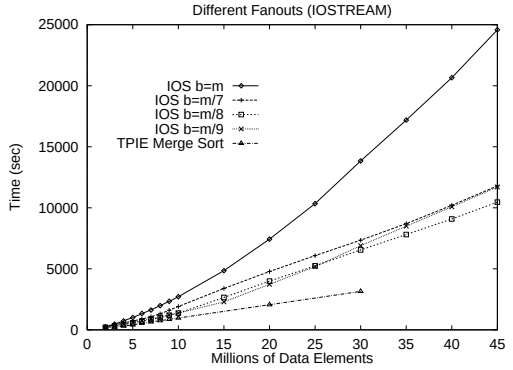

Figure 3: Timings for Buffer Tree Sort and TPIE Merge Sort: TMS ran out of disk space at about 35 million elements because it keeps its original data file. BTS does not require that all of the input data be available before it begins, and does not require a separate file for the original data.

block sizes of approximately $\frac{B}{2}$ keys pushed to the next level for each of (perhaps) m children of a node whenever the parent's buffer is emptied. Reducing the fanout, while maintaining the buffer size increases the expected number of elements in each block. We found that $(a, b) = (\frac{m}{32}, \frac{m}{8})$ gave the best performance in our tests. Smaller or larger values of b resulted in longer run times. Figure 3 shows running time curves for Buffer Tree Sort (BTS) and for TPIE Merge Sort (TMS). Results for BTS are shown for several values of b, where $a = b/4$ in all cases. Both TMS and BTS are running with synchronous I/O and single buffering. The TPIE MMB stream option is used. The Buffer Tree is using the 'iostream' access method. BTS performance is best for about $b = \frac{m}{8}$ and gets worse if b differs much from this value. BTS with $b = m/8$ has running times that change nearly linearly with the problem size for the problem sizes shown.

We caution that our experiments focussed on finding support for the predictions of asymptotic behaviour of Buffer Tree Sort. The actual running times of BTS may be improved by further tuning, and the performance of TPIE Merge Sort may improve with other choices of parameters and options.

We found the increased speed with smaller b intriguing, and so we counted the number of *block pushes* performed by the algorithm. A block push occurs when data is pushed to a child buffer after the parent's buffer becomes full. It may consist of a partial block, a full block, or more than a block

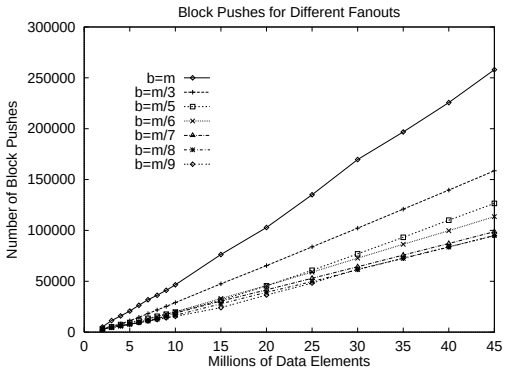

Figure 4: Number of Block Pushes for Buffer Tree Sort on Random Inputs

of request elements. Reducing the fanout increases the expected size of the data in a block push, and therefore may reduce the number required, and the number of I/O operations as a result. Figure 4 shows the relationship between several fanout values b and the number of block pushes over a range of input sizes. The reduction in block pushes seems to be the major reason for the improvement in running time between $b = m$ and $b = \frac{m}{8}$.

3.6.3. Non-linearities in the Running Time

We observed that contrary to predictions of the I/O model, for random input data our buffer tree sort implementation tended to have non-linear run times as the problem size increased. (Actually, we expect the running time to increase more than linearly by a logarithmic factor. However, since the base of this logarithm is large, the predicted increase in running time is close to linear for the range of problem sizes considered.)

Figure 5 shows a graph of problem size versus runtime for random input data and $b = m$. Also shown in this graph are curves for the various activities of the buffer tree, i.e., a breakdown of where this time is spent. The total running time appears to be increasing superlinearly with the problem size. *Total Running Time* is the sum of running time for *Insertions* plus *Force Empty All Buffers*. Running time for *Insertions* is composed of the sum of *Insertion: Empty Internal Buffers* plus *Insertion: Empty Leaf Buffers*. Both of these seem to be more than linear with the problem size. However, referring to Figure 4, the number of block pushes is not increasing superlinearly.

Adjusting the parameter b in the buffer tree both reduced the number of I/O operations performed by BTS and apparently removed the non-linear behaviour in our tests. Figure 6 shows the same graph as Figure 5 for $b = m/8$. In contrast to Figure 5, the Total Running Time curve is quite linear after about 10 million elements. The component curves in the figure are equally well behaved.

While the constant represented by the slope of the running time curve is larger for BTS than for TMS, we note that BTS is an online sorting technique and therefore addresses a different situation than does TMS. (See Figure 7).

3.6.4. Experiments with Parallel Disks

We experimented briefly with storing the buffer tree on multiple disks, by striping the buffers and leaves across two disks. We obtained a multiple disk driver (the "PDM API") from Tom Cormen at Dartmouth College, and ported it from the DEC Alpha environment to Linux without much difficulty. To manage concurrent disk access, the PDM API requires a Posix threads implementation, which we obtained from Florida State University. Perhaps due to the large data cache maintained by Linux, we found that large data volumes were required before two parallel disks outperformed a single disk for a simple "write-as-quick-as-you-can" application. For buffer tree sort this would require a single block push to be very large. We tried increasing m to allow this and did see marginally better performance with two disks for moderate values of n. Unfortunately, as n grew

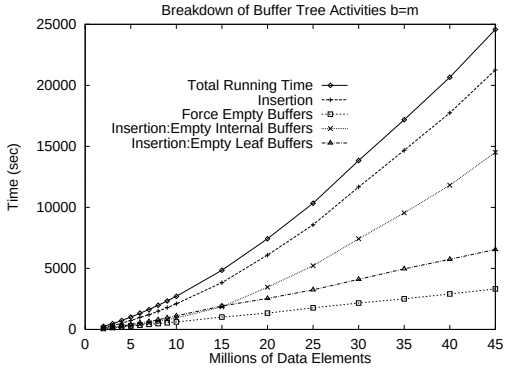

Figure 5: Timings for Buffer Tree Sort on Random Inputs. $(a, b) = (\frac{m}{4}, m)$, $m = 125$, $B = 256$.

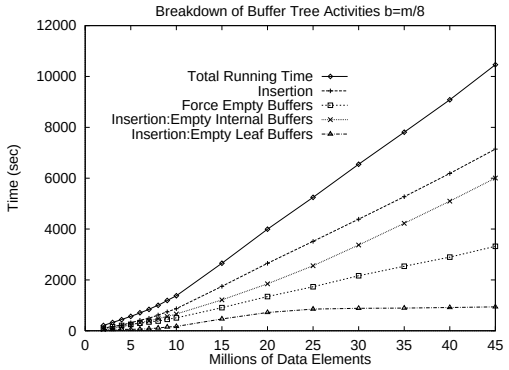

Figure 6: Timings for Buffer Tree Sort on Random Inputs. $(a, b) = (\frac{m}{32}, \frac{m}{8})$, $m = 125$, $B = 256$.

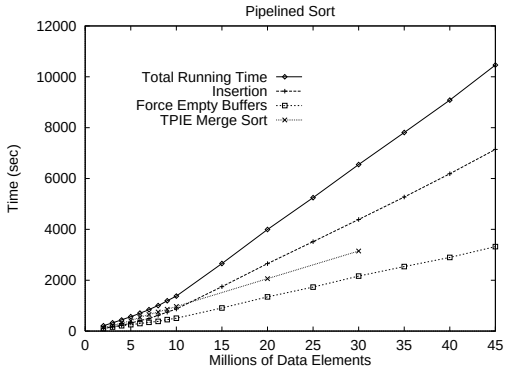

Figure 7: Pipelining sort with generation of inputs: if the generation of the inputs is sufficiently time consuming, Buffer Tree Sort can provide a speed advantage over offline methods by permitting the insertion time to be hidden by the time to generate its inputs. The time to Force Empty Buffers then may be the only time that remains "visible".

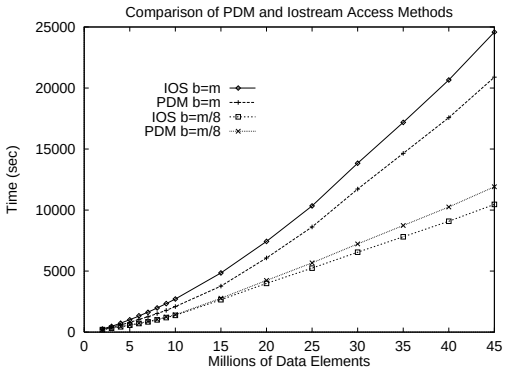

Figure 8: Running Times of PDM and Iostream I/O Access Methods: The PDM uses asynchronous I/O and this seems to give it an advantage for the larger fanouts

towards a more interesting size we began to see our performance degrade, apparently due to virtual memory effects. We concluded that we needed more real memory for this sort of experiment.

Figure 8 shows running times for a single disk under the PDM API and C++ iostream access methods. The PDM API could be expected to be slightly slower as it introduces some extra computation such as its use of threads. This seems to be true in the case of $b = \frac{m}{8}$, but its ability to overlap computation with I/O (asynchronous I/O) seems to allow it to outperform in the case $b = m$.

4. Conclusions

In this paper we describe an implementation of a buffer tree and two EM algorithms based on the buffer tree: an external memory treesort, and an external memory priority queue.

Our tests on random input sets lead to an experimental determination of parameter values different from those originally suggested in the design of the data structure.

Although the running times of our treesort implementation (BTS) with parameter $b = m$ clearly show non-linearities, $b = \frac{m}{8}$ produced a running time curve which is for practical purposes a straight line when the problem size is more than 10 million elements. The application was also heavily I/O bound. This supports the prediction of the algorithm [3] and the model [28] that the asymptotic running time is $\Theta(n \log_m n)$ I/Os and the number of I/O operations is the dominant issue in the algorithm.

The non-linear behaviour of BTS with parameter $b = m$ was manifested to various degrees in some of the other fanouts which we tried. While we expected some effect on running time as this parameter was varied, the sensitivity to non-linearity is troubling and we do not rule out implementation decisions as a possible cause. We hope that by further unit testing and performance measurements of the various components we will soon be able to explain this behaviour.

We conclude that (a) the buffer tree as a generic data structure appears to perform well in theory and practice, and (b) measuring I/O efficiency experimentally is an important topic that merits further attention.

4.1. Acknowlegements

We would like to thank Lars Arge and Jeff Vitter for their encouragement and interest in this work, Tom Cormen for providing the PDM API, and Doron Nussbaum and Darren Vengroff for helpful discussions.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[3] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, 1996.

[4] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *ESA, LNCS 979*, pages 295–310, 1995.

[5] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. In *Proc. ACM SPAA*, 1996.

[6] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[7] Y.-J. Chiang et al. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[8] Yi-Jen Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results.* PhD thesis, Brown University, August 1995.

[9] Thomas H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[10] Thomas H. Cormen and Melissa Hirschl. Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation. Technical Report PCS-TR96-293, Dartmouth College, Computer Science, Hanover, NH, September 1996.

[11] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[12] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Proc. ACM SPAA*, pages 106–115, 1997.

[13] R. W. Floyd. Permuting information in idealized two-level storage. In *Complexity of Computer Calculations*, pages 105–109, 1972. R. Miller and J. Thatcher, Eds. Plenum, New York.

[14] G. A. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4), December 1996.

[15] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *FOCS*, pages 714–723, 1993.

[16] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981.

[17] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[18] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995.

[19] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[20] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *ACM SPAA*, pages 29–39, 1991.

[21] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *ACM SPAA*, pages 120–129, 1993.

[22] J.F. Sibeyn and M. Kaufmann. Bsp-like external-memory computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, 1997.

[23] M. Smid. *Dynamic Data Structures on Multiple Storage Media.* PhD thesis, University of Amsterdam, 1989.

[24] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[25] D. E. Vengroff. *TPIE User Manual and Reference.* Duke University, 1995.

[26] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

[27] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

[28] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.