Research Report:On the performance of LEDA-SM

A. Crauser, K. Mehlhorn, E. Althaus,
K. Brengel, T. Buchheit, J. Keller,
H. Krone, O. Lambert, R. Schulte,
S. Thiel, M. Westphal and R. Wirth

MPI–I–98–1-028 November 1998

Author's Address

Max-Planck Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken

Abstract

We report on the performance of a library prototype for external memory algorithms and data structures called LEDA-SM, where SM is an acronym for secondary memory. Our library is based on LEDA and intended to complement it for large data. We present performance results of our external memory library prototype and compare these results with corresponding results of LEDAs in-core algorithms in virtual memory. The results show that even if only a small main memory is used for the external memory algorithms, they always outperform their in-core counterpart. Furthermore we compare different implementations of external memory data structures and algorithms.

Contents

List of Figures

1 Introduction

During the last years, many software libraries have been developed to support efficient data structures and algorithms for *in-core* computation. As the data to be processed has increased dramatically, most of these libraries are used in a *virtual memory* setting, allowing their data structures to use more memory than physically available. Most of the implemented algorithms and data structures were designed for a theoretical RAM–model with unlimited memory. It has been observed by many researchers that most of these algorithms perform very badly when used in an *external memory* setting. This led to the design of algorithms and data structures for external memory [CGG$^+$95, Arg96b, UY91]. Various theoretical results have been obtained in the last years starting with classical sorting and searching problems [AV88]. Very recently a few researchers also considered practical implementations. One of the first external memory libraries was TPIE [VV95]. TPIE consists of several so called *external programming paradigms* like scanning, sorting and merging. The main drawback of TPIE is that there exists no support to internal memory libraries [HMSV97]. Whenever internal data structures or algorithms are needed they must be explicitly implemented. As stated, TPIE was developed to support programmers in developing external memory algorithms. Other authors independently developed implementations for special data structures [Chi95, HMSV97]. These special implementations are often used to analyze the performance of new external data structures and algorithms. They are often efficient but lack a basic concept for external memory management. At the moment there is no library that provides both, a collection of external data structures and algorithms and a connection to a highly efficient library for in-core algorithms and data structures.

LEDA-SM was designed to close that gap. LEDA-SM is a prototype library that supports I/O–efficient data structures that can be used in many applications. To circumvent the rewriting of efficient algorithms and data structures for in-core problems, LEDA-SM is designed as an extension of the LEDA-library [MN95] and hence requires to be used together with LEDA. LEDA-SM provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. LEDA-SM gives a precise and readable specification for each of the data types and algorithms. The specifications are short and abstract so as to hide all details of the implementation. The LEDA-SM project was started in March 1997 and is still under development. In this report we will focus on experimental results that we obtained during the development of LEDA–SM so far. The report is organized as follows. We will start with a short description of the theoretical I/O–model underlying external memory algorithms. In Section 1.2 we describe how LEDA-SM manages external memory. In Section 2 we give some benchmarks and experimental results and show that the performance of LEDA-SM is good from a practical point of view.

1.1 The Theoretical I/O–Model

Modern computers are equipped with several memory hierarchies starting with registers of the CPU, caches, main memory and last but not least secondary memory (magnetic disks or CD-ROMS) and tertiary memory (tapes). The access time to the different memory hierarchies differs a lot. The biggest gap is between main memory and secondary memory where it is typical that a disk is 10^5 to 10^6 slower than the main memory [HP90]. Unfortunately, the majority of the known algorithms ignore this fact and use disks in the same way as if all data were fitted into internal memory, so that they suffer from the so called *I/O bottleneck*: They spend most of the time in moving data to/from the disk. As disks are mechanical devices the access time is dominated by moving the disk arm. Therefore it is convenient that one should transfer a block of data instead of a single item. Algorithms for external memory should be aware of these features. The computational model that reflects most of these features is the theoretical I/O-model of [AV88]. This model abstracts a computer to consist of a *two-level memory*: a fast and small internal memory of size M, and a slow and infinitely large external memory, called *disk*. Data between internal memory and the disk are transfered in blocks of size B, this transfer is called *I/O*. The performance of algorithms in this model is evaluated by measuring: (a) the number of I/O operations performed, (b) the internal running time (CPU time), and (c) the number of occupied blocks in the external memory (also called *disk pages*). [VS94] refined this model by also investigating in the so called *D-disk* model, where the external memory is realized by D disks that can operate independently from each other. From now on, we will use this model to study the complexity of external memory algorithms.

1.2 The LEDA-SM library prototype

One of the central questions for an external memory library is the realization of external storage which can consist of D disks, each storing a fixed number of blocks [1] of size B bytes. Indeed there are three ways to model the external memory. The first is to use low level disk access to the connected disks. The advantage of this realization is fast disk access without any additional software layers that slow down the disk access. The next two possibilities both realize the external storage by files of the operating system's file system. One way is to choose a file for each data structure or for parts of it, the other is to use one single big file for each disk to be modeled. Let us briefly recall some basic facts about the underlying file system. The Unix operating system together with the C- and C++–libraries offer a variety of prospects for file accesses. The basis for file access is realized by the standard I/O system calls. All standard I/O–libraries (stdio.h and iostream.h) use these low level system calls. However the operating system also imposes some restrictions. For example the number of open files is limited by the operating system and the maximum size of a file is limited. Some limits can be removed others are operating system specific, for example the file size is often restricted to 2 Gbytes because only 32 bit seek pointers are supported by the operating system. The restriction of the number of open file descriptors is also useful because each open file descriptor will use main memory in the operating system. Let us first discuss the method where we use a file for each data structure or parts of it. This method will open many files and therefore this could easily reach the limit of allowed open files. Furthermore, by using many files, we are not sure that they will be contiguous on the disk; the files can also be fragmented. If one later wants to switch to low-level disk accesses instead of file accesses, the whole library must be rewritten because disks in their low level view do not provide the file layer. The second possibility is the following: Each disk is modeled with a *single file* and it is divided into logical blocks of a fixed size B (disk pages). The size of this file is fixed, thus modeling the fact that real disk space is bounded. If we use an empty disk and allocate the file at the beginning, it is quite sure that the file is contiguous. However we are limited by the maximum allowed file size which will limit the size of the abstract disk. The advantage of this realization is that it is quite easy to switch to low level disk access without changing the overall design. All we have to do is to map the file that models our abstract disk to the raw disk and to rewrite the methods for block access.

We have chosen the following external memory realization for the LEDA-SM library prototype. As described above, we model a disk by one single file of the file system. This currently limits the abstract disk size to 2 Gbytes (Solaris 2.5.1) and will be unlimited under Solaris 2.6. Since LEDA-SM is using the file system, its external memory data structures and algorithms can explicitly take advantage of *I/O-buffering* and *read-ahead* strategy of the file system at no additional implementation effort.[2]

[1] In reality, disk space is not unlimited.

[2] As the file system is allowed to buffer disk pages, some of the disk requests can

LEDA-SM consists of a kernel that realizes the secondary memory access and the management of secondary memory. The components of the kernel visible to the user are the external memory manager (*ext_mem_mgr*), blocks (*block*), block identifiers (*B_ID*), user identifiers (*U_ID*), and the name server (*name_server*).

Recall that there are D disks and that the number of blocks that can reside on the d-th disk, $0 \leq d < D$, is $max\_blocks[d]$. The blocks on any disk are numbered consecutively starting at zero. A *block identifier* is a pair (d,num) of integers. A block identifier is called *valid* if $0 \leq d < D$ and $0 \leq num < max\_blocks[d]$ and it is called *active* if it is valid and the block denoted by it was written to. The class *B_ID* realizes block identifiers. Observe that block identifiers refer to physical objects, namely, to regions of storage on disk. In the remainder of this section there is the need to distinguish between blocks as physical objects (= a region of storage on disk) and blocks as logical objects (=a bit pattern of a particular size). We will use the word *disk block* for the physical object and reserve the word *block* for the logical object. The disk blocks are managed by the *external memory manager* (class *ext_memory_manager*). There is only one instance of this class; it is defined in *ext_memory_manager.h*. The external memory manager can be asked to allocate disk blocks, to free disk blocks, and to transfer blocks between main memory and external memory. The allocation of a disk block is either on a disk chosen by the user or on a disk chosen by the system (if no disk is specified in the allocation request). The return value of an allocation request is a block identifier which can later be used in read- and write-operations.

An allocated disk block is always owned by a particular user. Only the owner of a disk block can write the disk block. A user is identified by a *user identification* (= an integer) of class *U_ID* and user identifiers are managed by class *name_server*. We use user identifiers for memory protection. Every instance of a data structure is a different user of the kernel and hence data structures are protected against one-another.

The parameterized type *block*<*E*> is used to store logical blocks in internal memory. An instance B of type *block*<*E*> can store one logical block and gives a typed view of logical blocks. A logical block is viewed as two arrays: an array of links to disk blocks and an array of variables of type E. A link is of type *block identifier*. The number *num_of_bids* of links is fixed when the block is created. The number of variables of type E is denoted by *blk_sz* and is calculated at the time of creation. *blk_sz* is dependent from the maximal size *EXT_BLK_SZ* of *block*<*E*> and from the size of data type E. Both arrays are indexed starting at zero.

Every block has an associated user identifier and an associated block

immediately be satisfied by the I/O–buffer. This notably speeds up the real performances of the external-memory algorithms.

identifier. The user identifier designates the owner of the block and the block identifier designates the disk block to which the logical block is bound. The block identifier may be invalid and the user may be unspecified (*NO_USER*). Objects of type *B_ID* and *U_ID* are managed by kernel data structures. At any time the kernel keeps track of allocated and free disk blocks. All the mechanisms described so far build an interface class that we call *external memory manager*. It consists of

- An interface for allocating and deallocating blocks
- An interface for allocating and deallocating user ids
- An interface for block transfer management

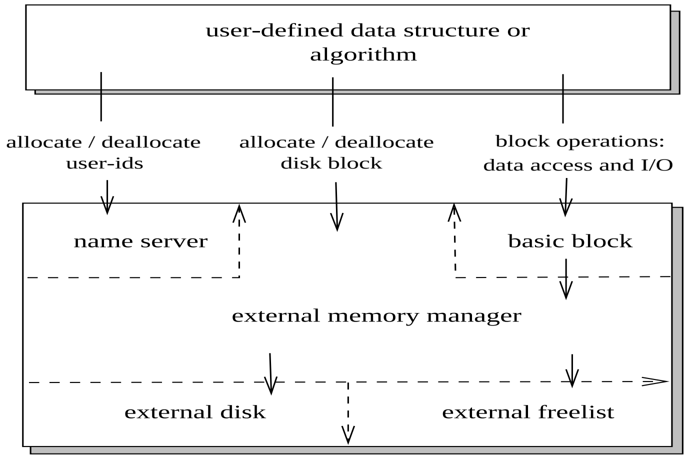

Figure 1.1: *Architecture of the external memory manager*

We call these parts interfaces because, we only specify the functionality of these parts, it is possible to use several different implementations for each interface. All these interfaces use the concept of blocks. This concept is unchangeable and builds the basis for the whole library. The block transfer and the allocation and deallocation of block identifiers can be handled in different ways. We can use up to four different methods for block transfer, namely standard I/O, system call I/O, memory mapped I/O and aio.[3] Besides the aio-library, these are standard methods for file access and are therefore portable to different machines. System dependent information i.e. the block size, the name of the disk(s), etc. is defined in a special configuration file called *.config_leda-sm* which is read when the program starts. All supported external data structures are implemented by using these interfaces and the concept of blocks. These data structures are the high level

[3]see also UNIX manual "man aioread".

access interface of the library, any I/O calls are completely hidden in the implementation. The memory manager itself and the data structures are implemented in C++ as a set of template classes and functions. A special feature of LEDA-SM's data structures is that during their constructions it is possible to specify (and therefore control) the maximum amount of internal memory that they are allowed to use. As most of the modern machines do not distinguish between physical main memory and swap space, this feature must be used carefully. It is even possible to specify more memory for the external data structures than physically available. In this case the machine will use swap space. To circumvent this problem, one should carefully choose the main memory settings for the external data structures. One way to verify this settings is to monitor the virtual memory behavior by using designated tools [4].

[4]On machines running the Solaris operating system use vmstat.

2 Low-level benchmarks for LEDA-SM

We tested LEDA-SM on a Ultra SPARC-1/143 with a single 9 Gbytes Seagate Elite 9 "Fast SCSI-2 Wide" disk (see Appendix A for technical details). Our tests consist of low-level disk tests with LEDA-SM and performance tests for the implemented data structures. In detail, we want to analyze the speed of the LEDA-SM interface that connects the library to the disk interface and we want to test our secondary memory data structures and algorithms against their equivalents of the LEDA library in a virtual memory setting. We measure the following three numbers:

1. **transfer rate**
   By the *transfer rate* we understand the amount of data per second that can be transfered by class block from the internal memory to the disk or vice versa. The transfer rate is used to estimate the effectiveness of LEDA-SM's disk access in comparison to the transfer rate, given by the manufacturer of the disk. The transfer rate is measured either by using a monitoring tool of the operating system [1] or by dividing the total amount of transfered data of our test program by its real running time. However the second way only leads to a reasonable approximation of the transfer rate if the I/O calls are the dominating part of the test program. There are applications where the internal time which is used to work on the data is bigger than the time to load and store the data (i.e. the algorithm is CPU-bound). In this case, the transfer rate, if calculated from the total running time leads to wrong interpretations of the efficiency.

2. **exact I/O bounds**
   Many of our implemented algorithms perform the same number of I/Os expressed in big Oh-notation. Therefore, to compare these algorithms, we are interested in giving exact I/O bounds. For the space

[1] On machines running the Solaris operating system used iostat

consumption, the constants are more important because in realistic applications, the available disk space is limited.

3. **real running times**
What we really want to investigate is whether or when external memory algorithms are faster than "internal memory algorithms in virtual memory". We do not measure user time and system time because even the sum of both does not give us an exact behavior for our programs. Our programs heavily perform I/Os, the time for raw data transfer between disk and kernel memory of the operating system is not measured in neither system time nor user time. Therefore the sum of user time and system time is only a bad approximation, and instead we choose the wall clock time which is the time that passes between the start of the process and its end.

During our tests we restricted the amount of used internal memory to $\approx$ 4 Mbytes. As many applications only use internal blocks and LEDA data structures we simply calculated the necessary size. We did not count the space for local variables or small local arrays. At the moment it is not possible to calculate the main memory usage of LEDA data structures by special functions. Therefore we use monitoring tools like *top* and *vmstat* to verify that our main memory settings do not lead to heavy paging on the machine. An automatic calculation of the space consumption of LEDA data types and algorithms will be integrated into later versions of the LEDA library.

2.1 Scanning

Scanning is the process of sequentially reading or writing data on the disk. If N bytes are accessed, this requires exactly $\lceil N/B \rceil$ I/Os. The scan test is used to calculate the transfer rate that can be achieved by the LEDA-SM library. It further illustrates the difference between sequential block accesses and random block accesses. We used LEDA-SM's low level disk block access routines that are provided by class block. We set the block size to 8 kbytes which is exactly the page size of the file system. The tests consisted of reading and writing either (a) consecutive blocks on disk or (b) random positions on the disk using different file I/O implementations. We first consider writing consecutive blocks.

I/Os of the file system can be done in a buffered way. If the I/Os are buffered, the data to be written is copied to a memory region of the operating system and the process can immediately proceed with its computation. After the operating system has collected some requests (usually after a fixed time interval), they are send to the disk asynchronously. The advantage of this method is that the requests can be reordered by the operating system so that consecutive disk locations can be accessed. This would result in a higher transfer rate. Another method for doing file I/O is memory mapped I/O. Memory mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes of the file are read. The advantage of mmap is, that the operating system is doing the I/Os directly on the the buffer in memory, there is no need to copy the data to a memory region of the operating system.

We now perform the following tests. We compare LEDA-SM's block transfer mechanism to a standard file access without LEDA-SM. The block transfer mechanism uses different file I/O mechanisms, namely stdio and syscall (both buffered) and mmap. The standard file access method is stdio. The tests of Figure 2.1 consist of performing consecutive write operations using either LEDA-SM or a standard file access. We see that mmap is a little bit slower than the other access methods. Although memory mapped I/O (mmap) directly writes to disk without buffering the data, it is slower than the other access methods. Stdio and syscall buffer the requests in kernel memory and don't write directly to disk. Writing is done asynchronously by the operating system when the buffers are flushed. By this, sequential writes profit from the buffering process as much larger data blocks are really flushed to disk. Memory mapped I/O (mmap) achieves an average transfer rate of approximately 3.01 Mbytes per second. Standard I/O (stdio) achieves an transfer rate of 4.97 Mbytes, and all the other methods including writing normal files without library overhead (file) achieve approximately the same transfer rate. This situation does not change if we perform consecutive read operations. Reads are a little bit faster than writes (see Figure 2.2). This comes from the fact that disks use read-ahead of further data to service the following consecutive requests from their caches. LEDA-SM's block transfer using memory mapped I/O is slower because of the following reasons. When we establish the mapping, we exactly map one block of the file. We then perform the read or write access and then destroy the mapping. The overhead, introduced by the two system calls to establish and destroy the mapping leads to this slowdown. It is not possible to map the whole file because its size can be greater than the available internal memory.

By buffering and read-ahead of modern file systems, consecutive file operations are easy to optimize for nearly every modern file system. Random I/Os behave differently. Now, buffering and read-ahead don't play an important role because we do not target consecutive disk locations. Thus more seeks on the disk occur and should lead to a slowdown. As the operating system is buffering requests, when performing linear I/Os we only see a few seeks on the disk and many track to track seeks because the data will be stored consecutively. If we use a random access pattern nearly every positioning will be a normal seek. If we look at the specification of the Elite 9 disk, the ratio average seek time over track to track seek is 7. If we look at the average transfer rate, we see that it decreases dramatically. For standard I/O and random writes, the transfer rate dropped to 0.88 Mbytes per second (a slowdown of a factor 5.6). Memory mapped I/O is faster than the other methods if we are performing only a small number of random writes, and is getting slower for a larger number of operations. Random read operations behave the same way as random write operations with the exception that memory mapped I/O is slower than all other operations.

One goal was to estimate the overhead for file access that we introduced by using a class library. For consecutive reads and writes we see that the overhead compared to standard file access without library (file) is negligible. For random reads, `file` is 1.28 times faster than stdio and for random writes it is 1.11 times faster. Thus we conclude that the overhead introduced by our library is small.

The experiments show that the theoretical model is just an approximation of reality. In theory there is no difference between n random block I/Os and n consecutive block I/Os whereas in practice there can be a speed factor of 5 or more. We conclude that it is important to design algorithms that perform mostly consecutive I/Os.

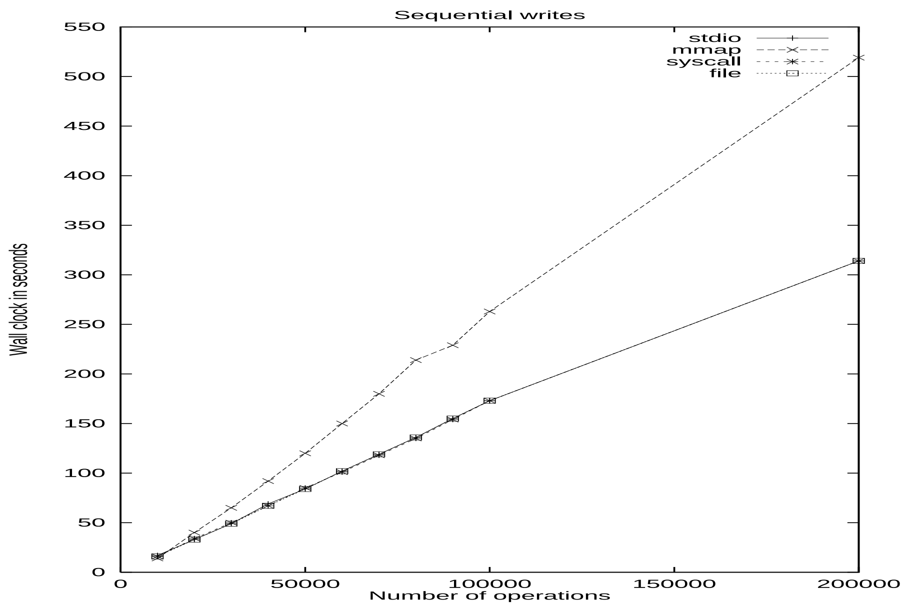

Figure 2.1: *Sequential write performance of LEDA-SM using file access methods stdio, mmap, syscall compared to direct file access by standard I/O without LEDA-SM (file)*

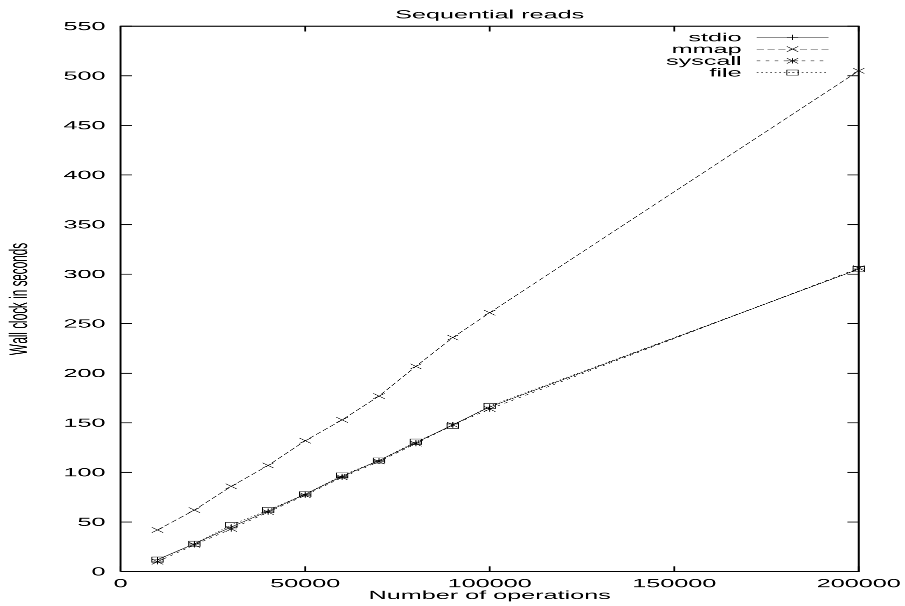

Figure 2.2: *Sequential read performance of LEDA-SM using file access methods stdio, mmap, syscall compared to direct file access by standard I/O without LEDA-SM (file)*

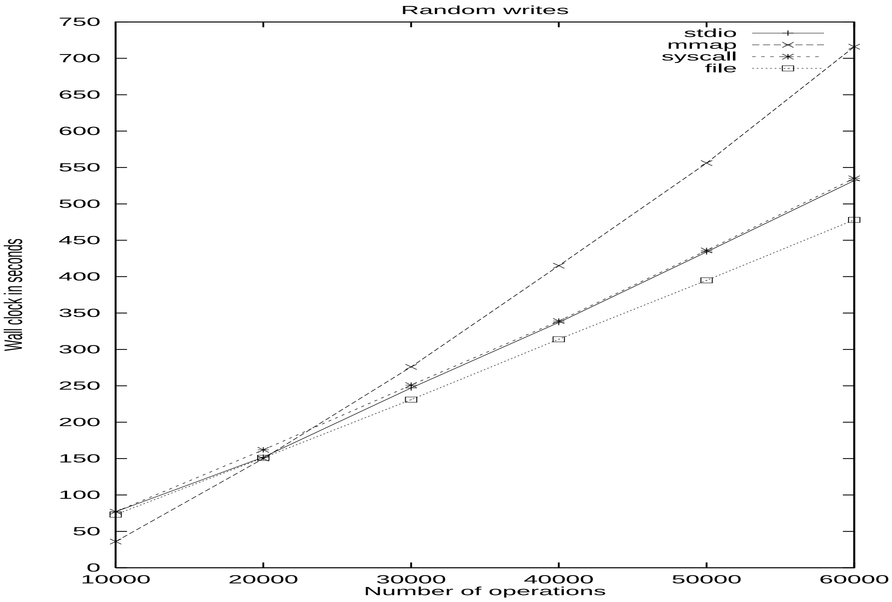

Figure 2.3: *Random write performance of LEDA-SM using file access methods stdio, mmap, syscall compared to direct file access by standard I/O without LEDA-SM (file)*

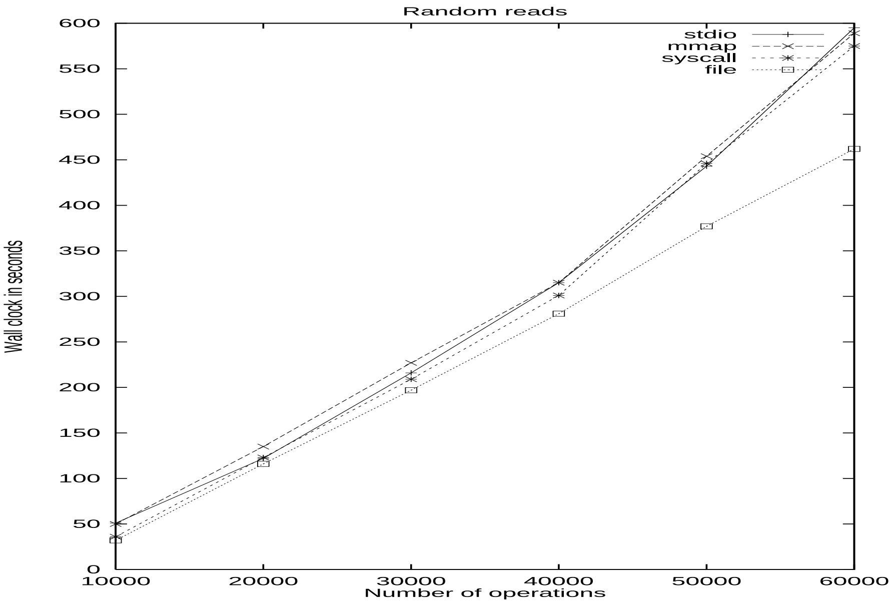

Figure 2.4: *Random read performance of LEDA-SM using file access methods stdio, mmap, syscall compared to direct file access by standard I/O without LEDA-SM (file)*

2.1.1 Choosing the Block size

The block size is not a fixed parameter in the LEDA–SM library. It is possible to specify the block size. For most of the tests we used a block size of 8 kbytes because this is exactly the page size of the operating system and the underlying file system. By using the same amount of block transfer, we are fair if we compare our algorithms to "internal memory algorithms in virtual memory". If we increase the block size, we can increase the transfer rate for consecutive operations and for random operations [Gro98](see Appendix A.1. However we remark that choosing the block size according to the fastest achievable transfer rate is not always suitable. Some applications cannot fill the blocks, therefore one should choose a smaller block size to avoid wasting disk space. Additionally, if we look at the $\log_{M/B}$ merge degree, of external multiway mergesort, the merge degree decreases if we increase B.

2.1.2 Performance of File–System I/O

Although every database system is avoiding file–system based I/O, it is not a bad choice for computing in external memory. However we should keep in mind that file–systems are tuned for specific kinds of I/O–requests. All the buffering mechanisms are tuned for normal user requests which are typically small in their requested size and in number. However the buffering gives us a good potential for speeding up consecutive operations. Furthermore asynchronous write is exactly the kind of disk access that one wants to have. This allows to overlap computation and I/O in some manner. However we should be careful with random operations and also with data structures that have in some kind a random access pattern to disk locations (either read or write). In this case we will always experience the pure limit of disk mechanics. If we use computers with larger main memory and don't allocate all the available physical memory, the file system I/O can profit a lot from large I/O buffers as modern operating systems will use as much available internal memory for buffering of requests. In some way we followed this approach by using only a very small fraction of the available main memory for the data structures (4 Mbytes during our tests).

2.2 Sorting in LEDA-SM

The classical algorithm for sorting in external memory is multiway-mergesort [CLR90]. The input to be sorted is divided into runs of length M. These runs are sorted in internal memory and later merged together. This leads to a sorting algorithm which performs optimally $\mathcal{O}(n/B \cdot \log_{M/B} n/B)$ I/Os and $\mathcal{O}(n \log n)$ operations. The space needed to sort n items of size x bytes is $2nx$ bytes on disk. We implemented multiway mergesort as follows. Internal sorting is realized by LEDA quicksort. The presorted runs are merged using an internal LEDA priority queue. The runs are creation by reading blockwise M items from the input, sorting them internally using LEDA quicksort and then writing them back, thus the run creation phase needs $2 \cdot n/B$ I/Os. The data transfer rate that is achieved during the creation of the presorted runs is very high because the I/O–requests target consecutive locations of the disk. The merging process is invoked in $\log_{M/B} n/B$ rounds. We always try to merge as many runs as possible. During this process we cannot expect consecutive I/O requests. This will immediately decrease the data transfer rate and the total running time. Mergesort is implemented as a low level template routine that can directly be used in many other data structures. A comparison function can be used for the data types to be sorted. We give a running example. We sort data type `int` (see Figure 2.5). A main memory of 4 Mbytes was used, the block size was 16 kbytes.

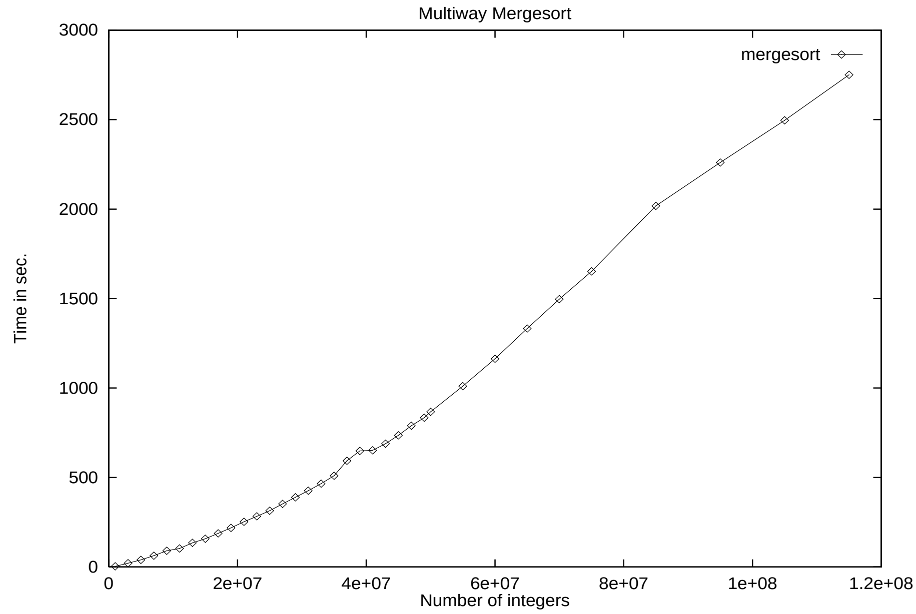

Figure 2.5: *Running time of LEDA-SM's multiway-mergesort implementation*

Our goal was to investigate whether and when external sorting is faster than an "internal memory sorting algorithm that operates in virtual memory". Therefore we compared the external sorting algorithm with the LEDA sorting algorithm that implements quicksort (see Figure 2.6). It is known from [AEH84] that quicksort is the best choice for sorting in virtual memory. Our quicksort implementation was allowed to use all the available main memory of size 64 Mbytes (M_q) and as much swap space as needed; whereas our multiway-mergesort implementation was restricted to use only approximately 4 Mbytes (M_e) of internal memory.

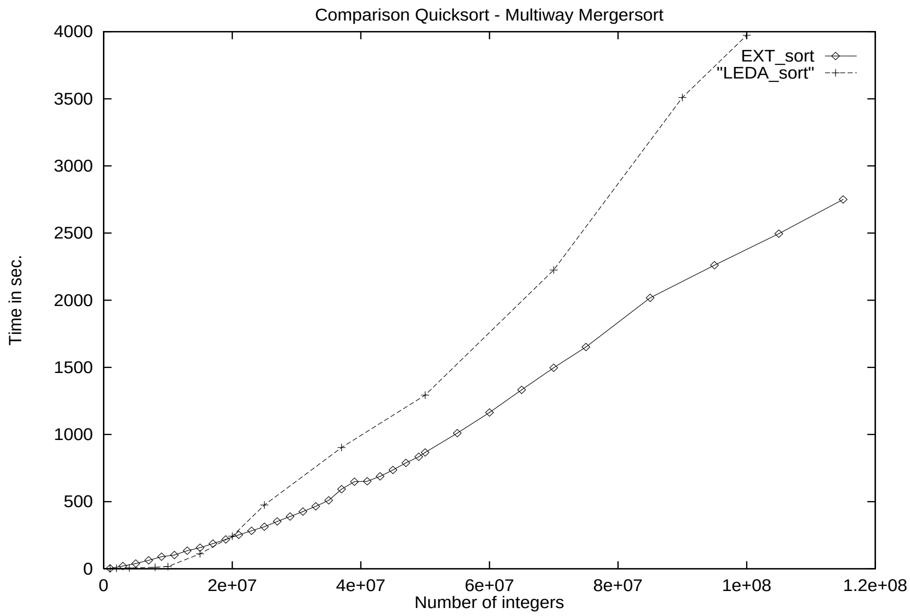

Figure 2.6: *Run-Time comparison between LEDA's quicksort implementation in virtual memory and LEDA-SM's multiway-mergesort implementation*

Quicksort is much faster as long as we stay in internal memory. The reasons for this is that multiway mergesort must first load the data from the disk before the sorting is invoked, and afterwards it must write it back. Multiway mergesort is getting faster if we exceed the main memory. At an input size of $n = 1.25 \cdot M_q$ ($M_q = 64$ Mbytes), multiway mergesort is faster than quicksort. In the range between M_q and $1.25 \cdot M_q$, quicksort is still faster. There are several reasons for this. First of all, quicksort is allowed to use more internal memory than multiway mergesort. Therefore, paging only occurs in the first recursion of quicksort and it is likely that all the other recursions will run in internal memory. The second reason is that multiway mergesort must load its data from disk while the data of quicksort remains in virtual memory. This behavior changes if we allow multiway mergesort to use more internal memory. We analyze in more detail the I/O complexity for our specific values of $M_e = 4$ Mbytes and $B = 16$ kbytes. The logarithmic I/O term ($\log_{M_e/B} n/B$) of the I/O–complexity is always between one and two. Therefore one merging step is sufficient to sort the data. Indeed sorting 1 million integers does not need a merge phase whereas sorting 2 million integers and all the following test sizes need one merging phase. The sharp bend at the input size of 85 millions `ints` in the Figure 2.6 occurs because we change the internal heap realization of the merge steps. During our tests we realized, that internal computation is critical for the performance. It turned out that the originally used LEDA Fibonacci-heap was too slow for smaller inputs. Therefore we used a simple heap structure based on arrays for smaller input size and change to Fibonacci heaps later. As a result we see that for large n, the multiway-mergesort is 2 times faster than quicksort. We also see that the gap between the two curves representing the running times is increasing. The time consuming part of sorting is the run-creation phase. Indeed, sorting is not I/O-bound as it first seems. Data can be loaded at a speed of approximately 4 Mbytes/sec. 4 Mbytes amounts to 10^6 ints, and sorting 10^6 ints takes approximately 1 second. Therefore half of the time is used by internal computation. During our tests, only one merging phase was necessary and merging is typically faster than the run-creation phase. This came from the fact that our disk space was restricted to 2 Gbytes and the setting of M and B allowed to merge the runs in one phase. We note that even for large inputs, we don't expect a merge-recursion that will be greater than 3. For example with $M = 64$ Mbytes and $B = 16$ kbytes, it is possible to merge and input of 256 Gbytes in one round. Thus, the total running time is dominated by internal sorting which should always be a candidate for improvement. If we sort user-defined data types, the comparison is done by using a user-defined compare function instead of smaller-equal operators. By this, the comparison introduces an overhead that is not negligible. Especially, if the user-defined data type is simple (for example a tuple consisting of two ints), the function call overhead for the comparison is tremendous. Therefore it is absolutely necessary to inline the comparison code [2]. This inlining leads to an overall speedup of two for the external sorting code.

[2]This is done in LEDA-version 3.7.2 by using template-sorting code together with inlining of comparison functions.

3 B-Trees

B–Trees [BM72] are widely used in database systems for searching. B-Trees are balanced search trees with a node fanout of $O(B)$. For standard search–tree *online* operations like insert, delete and search, the structure achieves $\mathcal{O}(\log_B n/B)$ I/Os for a B–tree consisting of n elements. We have chosen a B^* implementation in which the information is stored in the leaf nodes and internal nodes only store separator keys and the pointers to their children.. The implementation uses template code. To avoid parent pointers in internal nodes that one needs for rebalancing B-trees , an external stack is used to cache the path from the root of the tree to the leaft where the inserted element should be placed. The leaves are doubly linked to supply fast access to successors and predecessors. By caching the most frequently used disk pages for both the leaves as well for the path from the root to the leaves, it is possible to save some I/Os.

We compare B-Trees directly to internal memory search trees, e.g. 2-4-trees of LEDA. B-Trees use the pagers to reduce I/Os. We perform the inserts and delete_mins seperately that means we first insert all elements by calling n times insert and then perform n delete_min operations. Figure 3.1 shows that 2-4-trees are outperformed by B-Trees. If we perform more than $2 * 10^6$ insert operations on a LEDA 2-4-trees the system starts to swap a lot. At the point of $2 * 10^6$ insert operations B-Trees are faster than 2-4-trees. Figure 3.2 shows the performance of the operation delete_min. We immediately see, that delete_min is faster than insert. Although both operations run in $O(\log_B n/B)$ I/Os, during delete_min we don't need to search the smallest element. Instead we simply store a pointer to the smallest element so that we can immediately access the correponding leaf disk page.

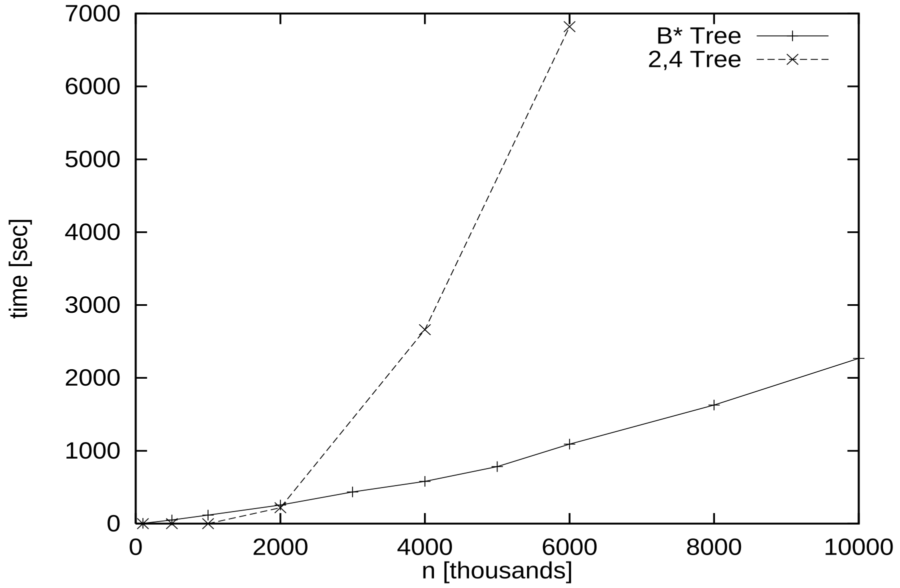

Figure 3.1: *Performance of insert of B-Trees compared to insert of 2-4-trees.*

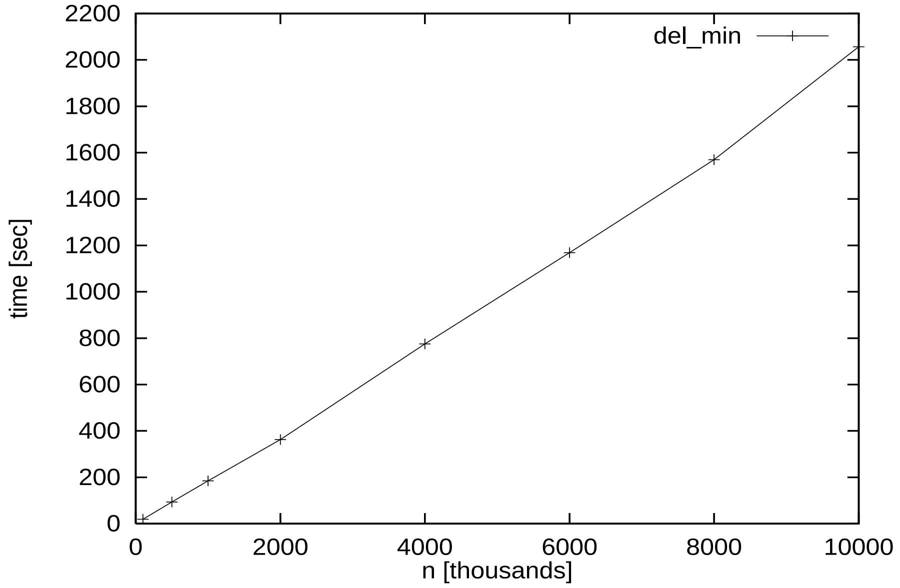

Figure 3.2: *Performance of delmin for B-Trees.*

4 External Priority Queues

Priority queues can be used for a variety of algorithms. In internal memory, there exist many efficient implementations of priority queues. A good survey for the efficiency of priority queue implementations is given in [CGS97]. However many of these data structures behave very badly when used in an external memory setting. This comes from the fact that the data structures are often accessed in a random manner through pointers. A typical priority queue operation that behaves like this is *decrease priority*. Therefore most of the known external memory priority queues do not support this operation. We will first introduce a fast and simple external memory priority queue without a decrease_priority operation. This queue is based on internal radix heaps [AMOT90].

4.1 R–Heaps as external priority queues

We describe a simple and very fast priority queue data structure based on internal two-level radix heaps [AMOT90] (shortly *R–heaps*). Let C be a positive integer. We make two assumptions:

- All priority labels of the elements currently in the R–heap are non-negative integers in the range $[min, min+C]$, where min is the priority value of the last element deleted from the heap (*zero* otherwise).
- The queue is *monotone* so that the priority values of the deleted elements are *nondecreasing*.

Let us now consider an arbitrary positive integer r (also called *radix*) and choose the parameter h so that $r^h > C$ (that is, we set $h = \lceil \log_r C \rceil$). Let k be an arbitrary element and let $k_h k_{h-1} \dots k_0$ be its representation in base r (denoted by $(k)_r$). Similarly, let $m_h m_{h-1} \dots m_0$ be the representation of min in base r (denoted by $(min)_r$). According to the assumptions above, we have that if an element k belongs to the queue then $k - min \ < r^h$. Consequently $k_h = m_h$ or $k_h = m_h + 1$.

We are ready now to describe our external R–heap which consists of three parts:

- h arrays of size r each. An array entry is a linear list of blocks called a *bucket* (In a simple way a bucket can be seen as an external stack). Let $\mathcal{B}(i,j)$ denote the bucket associated with the j–th entry of the i–th array, for $0 \leq i < h, 0 \leq j < r$. For each bucket we maintain the first block (disk page) in main memory. This constrains r to be such that $h \cdot r \cdot B \leq M$.
- a bucket N containing all elements k with $k_h = (m_h + 1) mod\ b$.
- an internal memory priority queue $\mathcal{Q}$ containing all indices of non-empty buckets. The indices are ordered lexicographically, i.e., $(i,j) < (i',j')$ if either $i < i'$ or $i = i'$ and $j < j'$. $\mathcal{Q}$ will never store more than $h \cdot r$ indices.

We now discuss how the operations *Insert* and *Delete_min* are implemented.

Insert: In order to insert a new element k in the external R–heap, we first compute the least significant $h+1$ digits of its representation in base r, thus taking $O(h)$ time and no I/Os. Then we insert k into the bucket $\mathcal{B}(i,j)$ such that $i = \max\{l m_l \neq k_l, 0 \leq l \leq h\}$ and $j = k_i$. Clearly bucket $\mathcal{B}(i,j)$ may be currently empty, and in this case we also insert the bucket index (i,j) into $\mathcal{Q}$.

Pseudo-code for insert(x) (x=$< d, i >$):

```
insert(x)
  compute i = max{l  m_l ≠ d_l, 0 ≤ l ≤ h} and j = d_i
  if ( B(i,j) is empty) Q.insert( (i,j) )
  insert x into B(i,j)
```

Lemma 1 *An* Insert *takes amortized* $O(1/B)$ *I/Os and* $O(h+\log(hr))$ *CPU time.*

Proof: This follows immediately from the discussion above. The constant in the I/O term is one. ∎

Delete_min: If the bucket $\mathcal{B}(0, m_h)$ is non-empty we just delete an arbitrary element from this bucket. This takes amortized $O(1/B)$ I/Os and $O(\log(hr))$ time. Otherwise we use the internal priority queue $\mathcal{Q}$ to find the first non-empty bucket. The idea is to perform a *Delete_min* on $\mathcal{Q}$. This is either a bucket $\mathcal{B}(i,j)$, for some i and j, or the bucket N (here we set $i = h$). In both the two cases, we scan the non-empty bucket and determine the new minimum element *min*. It is crucial to observe that all elements

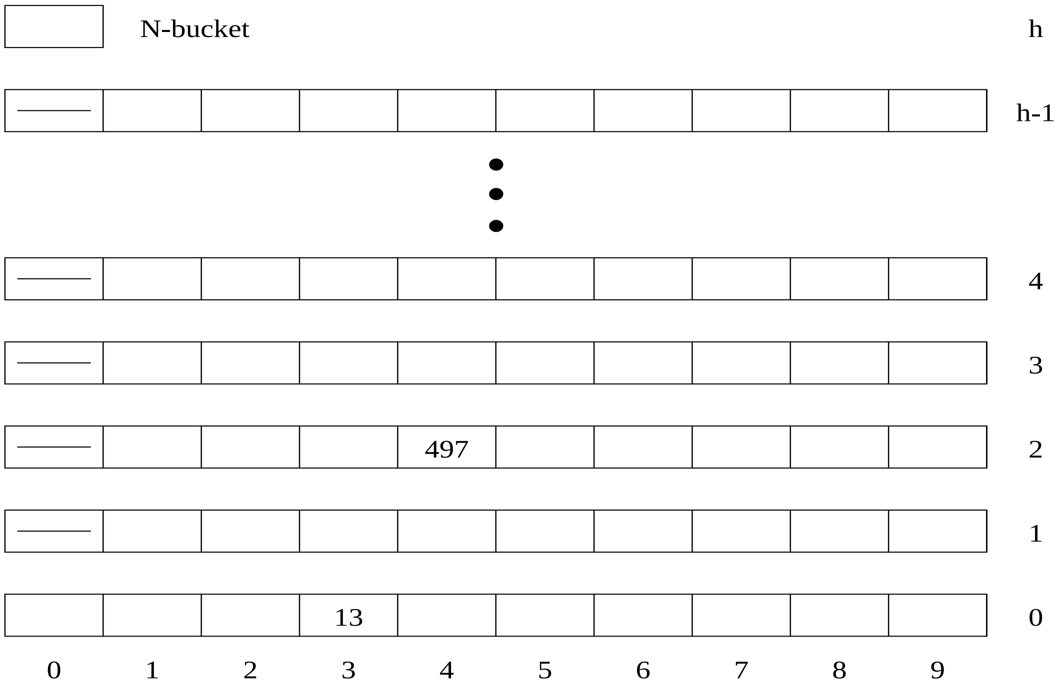

Figure 4.1: *Radix heap example with $r = 10$: 13 is the current minimum, 497 differs in the second digit and is therefore stored in $B(2,4)$.*

e in the examined bucket agree with the new minimum min in the digits $e_l = min_l$, for all $l \geq i$, according to the way we allocated the keys into the various buckets and that the new minimum min has to be stored in bucket $B(0, m_h)$. Since the minimum has changed we have to reorganize the elements in the current bucket (i,j).. The new bucket for each element in $\mathcal{B}(i,j)$ is determined according to the rule exploited for the *Insert*: we store e into $\mathcal{B}(i',j')$ where $i' = \max\{rm_r \neq k_r, 0 \leq r \leq h\}$ and $j' = k_{i'}$. Observe that all these elements are moved to a bucket with *smaller first index*, since these elements disagreed with the old minimum in their i-th digit and agree with the new minimum in their i-th digit. We update $\mathcal{Q}$ accordingly and return any element stored in the bucket $\mathcal{B}(0, m_h)$.

Pseudo-code for x = delete_min():

```
delete_min()
(i,j) = Q.find_min();
if ((i,j) == (0,m_h)
  delete an arbitrary element from B(0,m_h)
  if(B(0,m_h) is empty) Q.delete((0,m_h)) fi
fi
```

```
else
  determine the minimum element v =< d,i > in B(i,j)
  compute (d)_r
  set min = d
  insert v into B(0,d_h) = B(0,m_h)
  Q.insert((0,d_h))
  for_all remaining elements e of B(i,j) except v
     insert(e)
  delete an arbitrary element from B(0,m_h)
  if(B(0,m_h) is empty) Q.delete((0,m_h)) fi
```

Lemma 2 *A* Delete_min *takes $O(h/B)$ amortized I/Os and $(h \log(rh))$ amortized CPU time.*

Proof: Each element can be redistributed at most h times, namely once for each level. This amounts to 2 scans per redistribution, if we avoid the scan for finding the minimum element in a bucket. This implies the I/O-bound. For the CPU time we observe that time $O(\log(hr))$ is required to find the first non-empty bucket (by using $\mathcal{Q}$) and that the same amount of time pays for moving a single element. Since each element is moved at most h times, the bound follows. ∎

It remains to determine the appropriate values of r and h that allow the R–heap data structure to work correctly. The only constraint we imposed on these parameters was that $r \cdot h \cdot B \leq M$, which allowed to stuff one block per bucket into internal memory. Since $h = \log_r C$, it suffices to choose the maximum value of r such that the constraint above holds, that is (where $m = M/B$):

$$r = M/(hB) = (m/\log C)\log r \approx (m/\log C)\log(m/\log C).$$

We have therefore proved the following result:

Theorem 1 *Let $r = (m/\log C)\log(m/\log C)$ and let $h = \log_r C$. An* Insert *into an external R–heap takes amortized $O(1/B)$ I/Os and $O(h + \log(hr))$ CPU time and a* Delete_min *takes $O(h/B)$ amortized I/Os and $O(h\log(rh))$ amortized CPU time.*

As far as the external space consumption is concerned, we observe that only *one* disk page can be non-full into each bucket (by looking at a bucket as a stack). But this page does not reside on the disk, so that there are no partially filled disk pages. We can therefore conclude that:

Lemma 3 *An external R–heap storing N elements occupies $\Theta(N/B)$ disk pages.*

Our implementation of radix heaps is designed in such a way that for a given C it calculates automatically the best values for r and h. The values are computed according to the condition that h should be chosen minimal.

4.1.1 Performance of Radix Heaps

When testing priority queues (i.e. Radix-Heaps and Buffer-Trees), we use the following benchmark. We first insert all elements and then delete all elements. This test performs the maximum number of I/Os for both operations. If we would mix insertions and deletions, we could not guarantee that I/Os are performed. I could be the case that most of the buckets are small and a lot of elements could still be in internal memory. Therefore we perform first all insertions and then all deletions. This allows us to calculate the average transfer rates to disk that are achieved by this operations.

Radix heaps outperform all other priority queue implementations we tested. The advantage of radix heaps are the small constants. Inserting elements requires amortized $1/B$ I/Os. This is the fastest known I/O–bound for external priority queues. The amortized number of I/Os for a delete-min are $2 \cdot (h/B)$. Figure 4.2 is an example demonstrating the performance of insert and delete_min operations of LEDA-SM's radix heaps. We used 4 Mbytes of internal memory (M) , the block size B was set to 8 kbytes and C was set to 1000. As the number of I/Os performed during n inserts is linear, we expect a linear running time for insert. The curve for delete_min has a small jump at the beginning, this jump occurs because the value of $log_r C$ changes from one to two for our settings of M, B and C. Due to the small jumps of the logarithmic term of delete_min, the curve seems to be linear. If we compute the transfer rate of insert, we achieve approximately 3 Mbytes/sec which is good compared to the maximum of 4.97 Mbytes/sec than can be achieved using consecutive I/Os. It is also obvious from the I/O bounds for the operations insert and delete_min that insert performance is independent from the value of k and delete_min should be faster with increasing k (see also Figure 4.5).

Practical Modifications The height h of the radix heap is always chosen to be minimal therefore minimizing the amortized number of I/Os for delete_min. The choice leads to a value of r that can either be a power of two or not. If r is not a power of two, we use a precomputed lookup table for the powers of r and division to compute the bucket number for insert. If r is a power of 2, we can compute the bucket numbers by bit shifts. This leads to two different variants [1]. It is obvious that the bit shift realization is always faster than the division method (see Figures 4.3 and 4.4).

[1] The implementation is not fixed, the radix heap calculates the best number for h, r in the constructor, and afterwards depending on r it chooses either the bit shift method or the division method

The Influence of the main memory size To be theoretically optimal we should give all the available memory to the radix heap. However we must be careful, there is still the internal priority queue that needs some space **and** the whole library needs some space as well. Therefore we cannot increase M to the main memory limit. Figure 4.6 shows the influence of M on the performance of insert and delete_min ($C = 1000, B = 8$ kbytes and M between 2 and 64 Mbytes). From the theoretical I/O bounds we know that insert should be independent from M and delete_min should become faster if M increases. If we look at Figure 4.6 we see the following: delete_min if getting faster if M increases, however we see a big running time increase if $M > 58$ Mbytes. This happens because the operating system starts swapping. Insert does not seem to be independent from M because there is a jump at the range of 40 Mbytes. However, this jump does not occur because of increasing M, instead our implementation has switched from bit shift to division so that insert was slowed down a bit. During insert paging heavily occurs at about 56–58 Mbytes. The best performance ratio is achieved for about 42 Mbytes. However we should not forget that this specific result also depends on C,B and n.

4.1.2 External radix heaps versa virtual memory priority queues

Another important task is to compare the external radix heaps to internal priority queue data structures. We used an internal radix heap and a Fibonacci heap and compared them to the external radix heaps. The external radix heap outperforms the internal one except in very small cases (see Figures 4.7,4.8). In detail, the performance of insert and delete_min for the internal priority queues is very fast if the total size of the data structure will fit into main memory. At the point of 1.6 million operations for the Fibonacci heap (2.8 million for the radix heap) the performance dramatically decreases. The reason is very simple. The size of a node in a Fibonacci heap is approximately 40 bytes. This means that if we perform more than 1.6 million insertions, swap space is used. For the delete_min, this is even worse. We must construct the heap from the linked list of nodes. This accounts for a lot of random accesses to pages of the virtual memory and is the reason for the poor delete_min performance.

4.1.3 Sorting integers with radix heaps

A radix heap can immediately be used if we have to sort non-negative integers in the interval $\{0, \dots, C\}$. N elements are sorted by calling N times insert and then N times delete_min. If the interval C is small, we can immediately profit from the small height of the radix heap whereas mergesort cannot profit from this expectation to the input. When sorting integers in a small range (for example between zero and 1000), the radix heap simply outperforms multiway–mergesort (see Figure 4.9). The logarithmic term is smaller than that of multiway–mergesort and the constants are better. We achieve an exact complexity of $n/B+2n*(h/B)$ I/Os whereas the complexity of mergesort is $2n/B + 2n/B \log_{M/B} n/B$ I/Os.

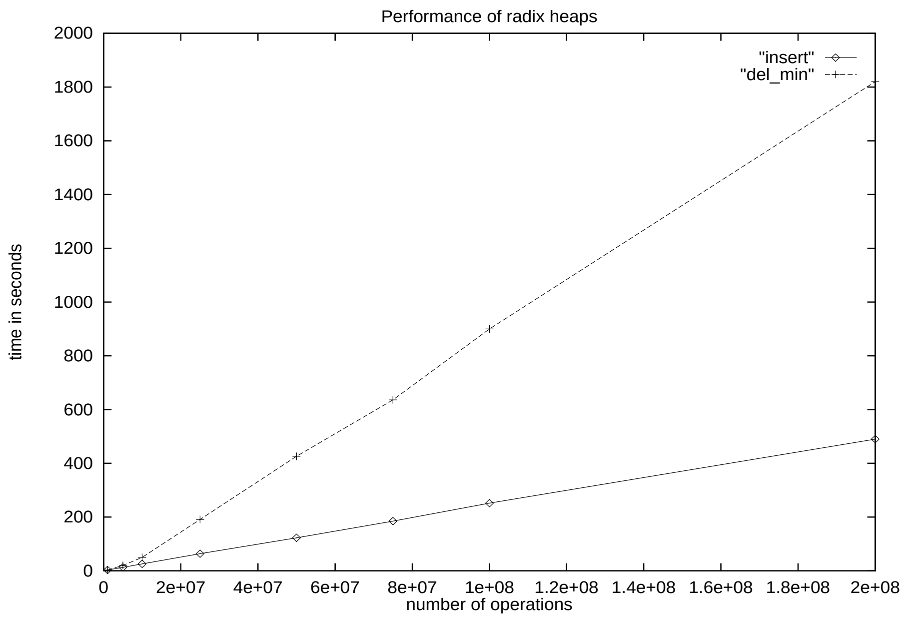

Figure 4.2: *Performance of the operations insert and delete_min for LEDA-SM's radix heap. An internal memory of size 4 Mbytes is used and C is set to 1000.*

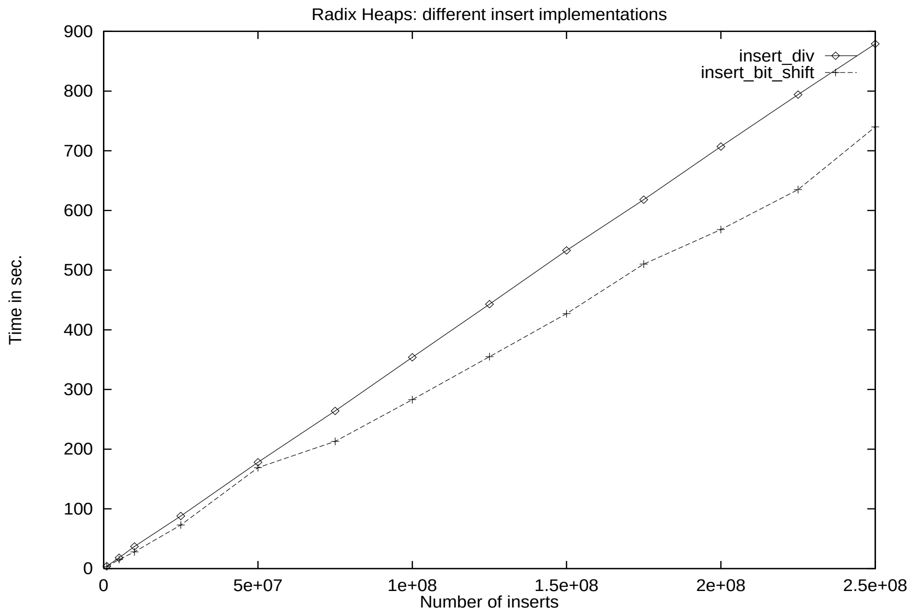

Figure 4.3: *Two insert variants for LEDA-SM's radix heaps.The bucket number is either computed by division (insert_div) or by bit shift (insert_bit_shift)*

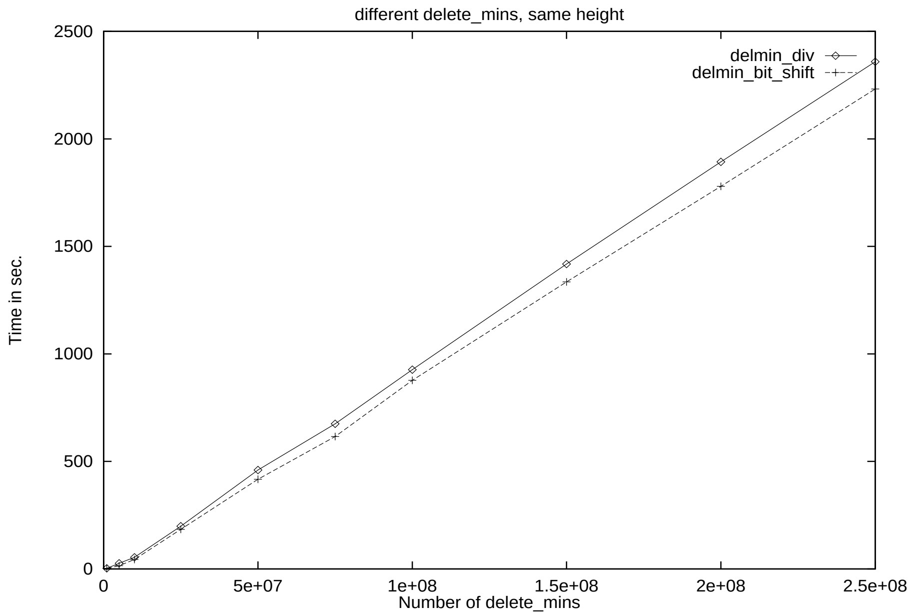

Figure 4.4: *The influence of the two radix heap insert variants on the performance of delete_min. Delmin_div is using the insert variant based on divison, delmin_bit_shift is using the insert variant based on bit shifts.*

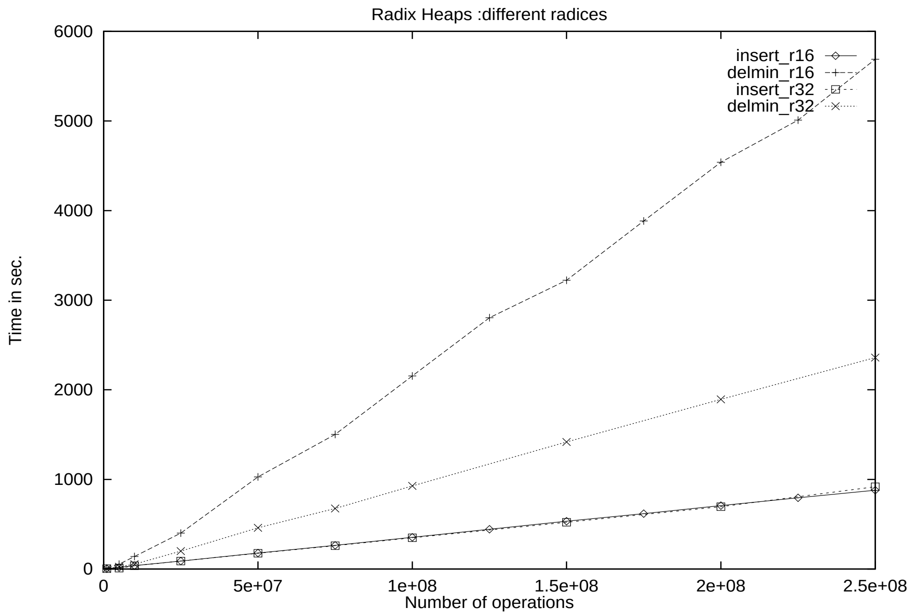

Figure 4.5: *Radix Heap performance with different radixes*

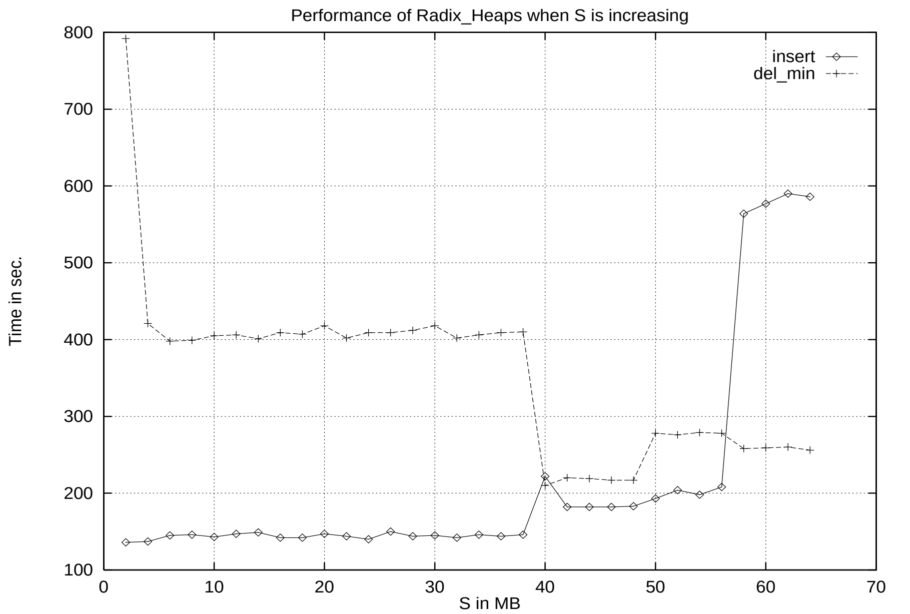

Figure 4.6: *The Influence of memory on radix heaps*

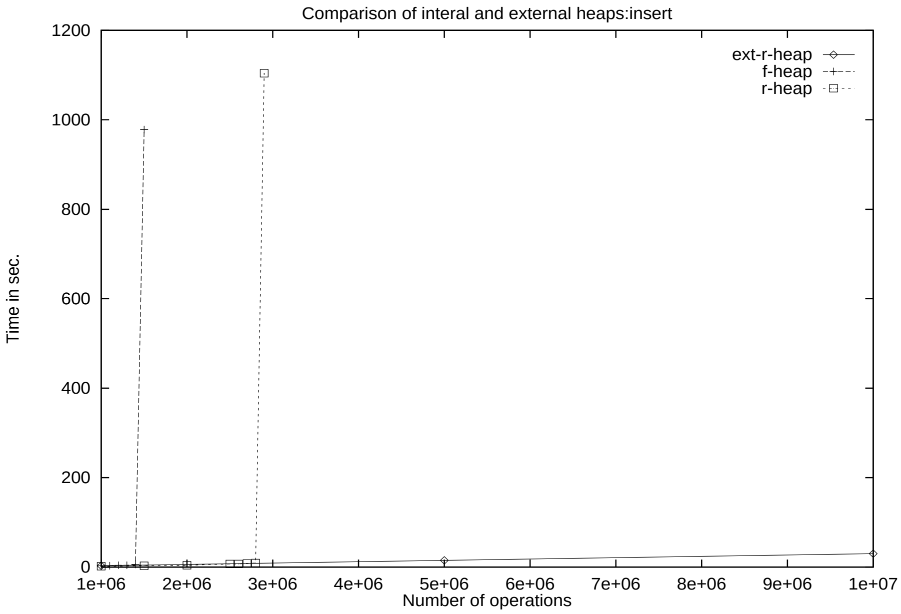

Figure 4.7: *Internal heaps versa external R–heaps:insert*

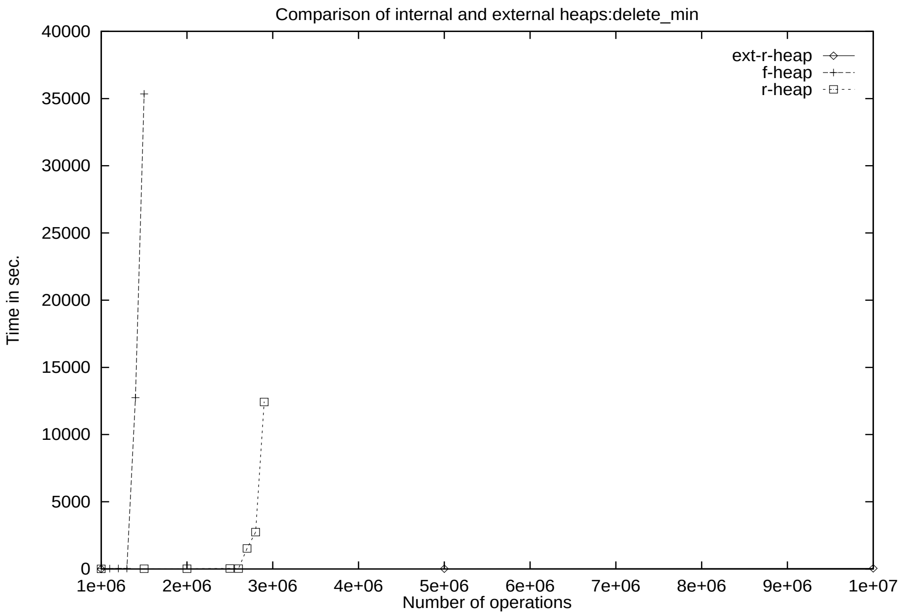

Figure 4.8: *Internal heaps versa external R–heaps:del_min*

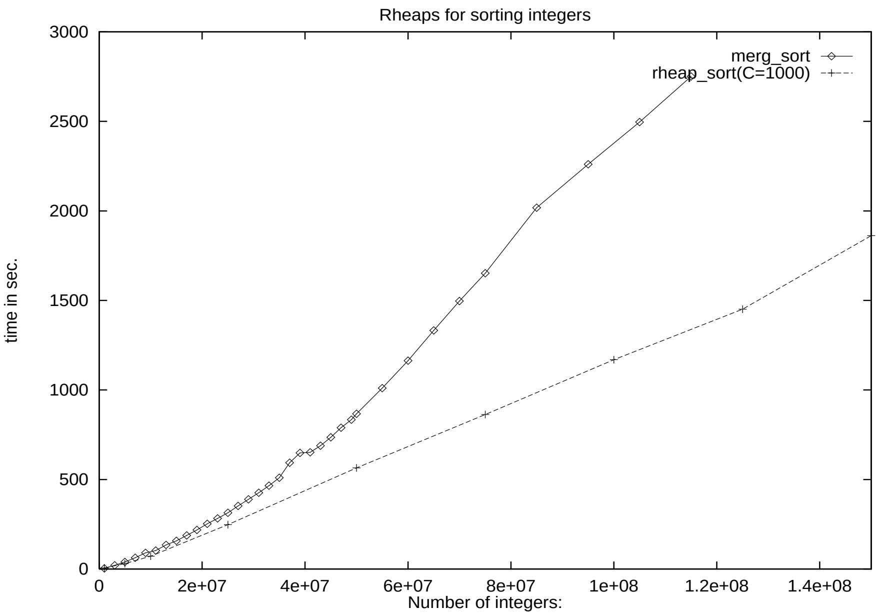

Figure 4.9: *Radix heaps for sorting integers*

4.2 Buffer Trees as Priority Queues

Buffer trees [Arg96a] are lazy search trees, based on internal (a,b)–trees. Using a fanout of M/B per node together with buffering of data, it is possible to achieve an amortized I/O–bound of $\mathcal{O}(1/B \log_{M/B} n/B)$ per operation. In the absence of online-changes this is optimal [Arg96a]. As buffer trees are general search trees they can be used for a variety of problems like sorting, priority queues, etc. For example if they are used in a tree-sort algorithm, this immediately leads to an I/O–optimal sorting algorithm. Unfortunately, the data structure is quite complicated. We implemented the buffer tree using Arge's general buffer emptying strategy [Arg96a]. During the development phase it turned out that several details that Arge proposed do not behave well in a practical setting (see also [HMSV97]):

- Arge's strategy for emptying buffers guarantees that only the leaf buffers can contain more than M elements at a time (see [Arg96a]). However it turned out during practical tests that this leads to a slowdown of the buffer emptying process. Therefore as in [HMSV97] we also propose not to restrict the internal buffer size. Thus we allow our implementation to store more than M elements in an internal buffer.
- It is critical for the performance to choose good values for the (a, b)–tree rebalancing and for the size of the internal buffers. We performed some tests where we changed b and the internal buffer size. We observed the following:
  1. If we increase the internal buffer size as much as possible, insert is getting faster. If we additionally decrease b this speed up improves.
  2. If one wants to speed up the time used for a delete_min operation, the original setting of the parameters a, b and the buffer size for the buffer tree should be used.

In a normal priority queue setting, we perform mixed insert and delete_min operations. So, to speed up one single operation while slowing down another is not the choice. We found that increasing the internal buffer size can speed up insert without slowing down delete_min too much. Unfortunately the setting of the parameters a, B and of the buffer size is application-dependent and can't be chosen in a general way.

4.2.1 Performance of Buffer Tree Heaps

We tested the performance of buffer trees with a setting where $M = 4$ Mbytes and $B = 8$ kbytes. The priorities were integers as well as the information. We first performed insertions and then delete_min operations (see

Figure 4.12). The transfer rate is really poor. For insert we achieve about 100 kbytes per second. For delete_min we achieve approximately 187 kbytes per second. This is not astonishing as a lot of CPU-operations are done when emptying a buffer. We note that due to the design of buffer trees we need 12 bytes per element since every elements also stores a time stamp. The complexity of the structure causes this slowdown. We note that our running time is conform with the results obtained by [HMSV97]. Therefore we conclude that the data structure itself is slow and not our implementation. However, the advantage of buffer trees in a priority queue setting is that the priority data type is not restricted to integers as it is the case for the radix heaps.

4.2.2 Sorting with Buffer Trees

We also tested buffer trees as a sorting procedure. We directly used the buffer tree heap by performing first all insertions and then all deletions. Even there, the buffer tree is not a good choice. Buffer tree sort is outperformed by every sorting algorithm we used (see Figure 4.13). Even the quicksort with virtual memory is faster than buffer–tree sort. It turns out that the buffer tree sort is a factor 14.5 slower than multiway mergesort [2].

4.2.3 Buffer Tree Heaps versus Radix Heaps

We compared the buffer heaps to our radix heap implementation (see Figure 4.14). To be fair we increased C to the maximal possible value. The insert of radix heaps is much faster than that of buffer heaps. It is the best that can be achieved due to its theoretical I/O bound. Even the delete_min of radix heaps is faster. This is amazing because the logarithmic terms differ. Radix heaps have a $\log_{M/B} C$ term and buffer trees $\log_{M/B} n/B$. Although $C > n/B$, radix heaps are faster. This implies that the constants for radix heaps are better. However this also implies that there is a input range where the delete_min of buffer heaps will be faster than that of radix heaps.

[2] If we take into account that the total data size for the Buffer tree that must be transported during the sorting is 1.625 larger than that of mergesort, we see that there is still a slowdown factor of 9.

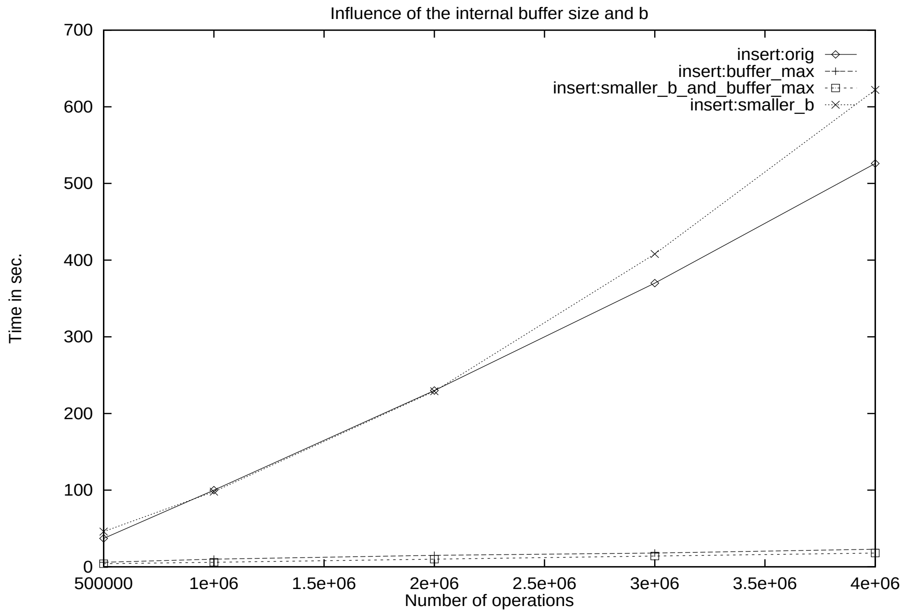

Figure 4.10: *The influence of 'b' and the internal buffer-size on the performance of insert for buffer trees*

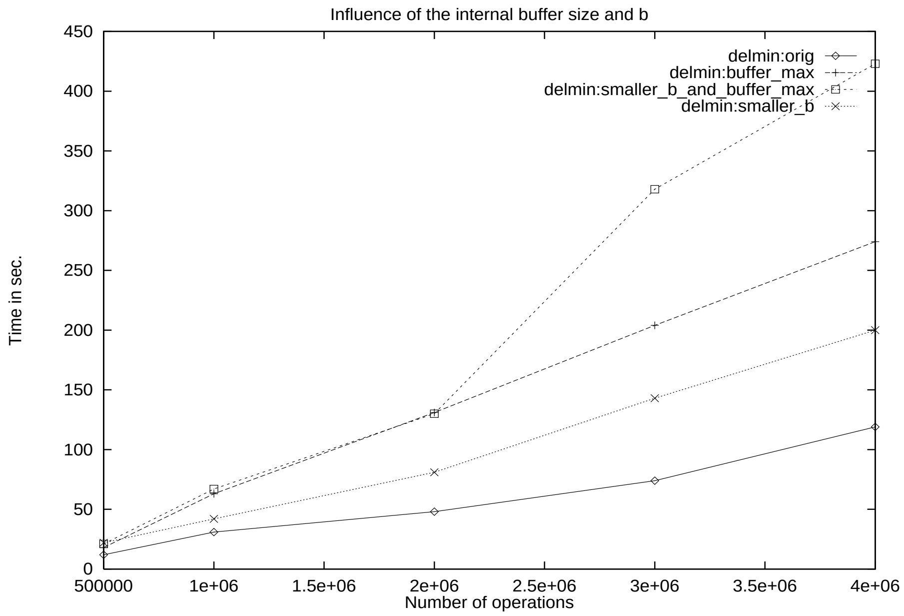

Figure 4.11: *The influence of 'b' and the internal buffer-size on the performance of delete_min for buffer trees*

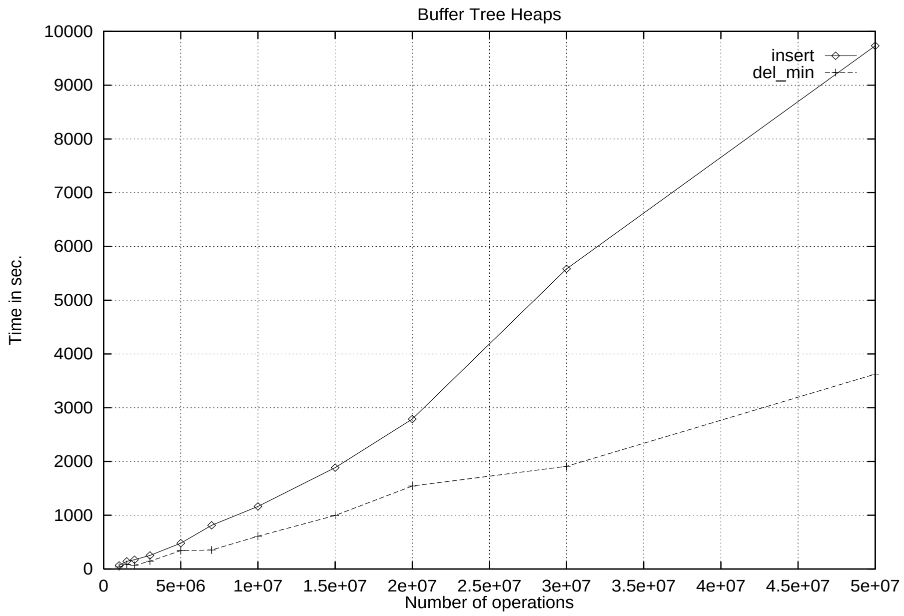

Figure 4.12: *Buffer Trees as priority queues*

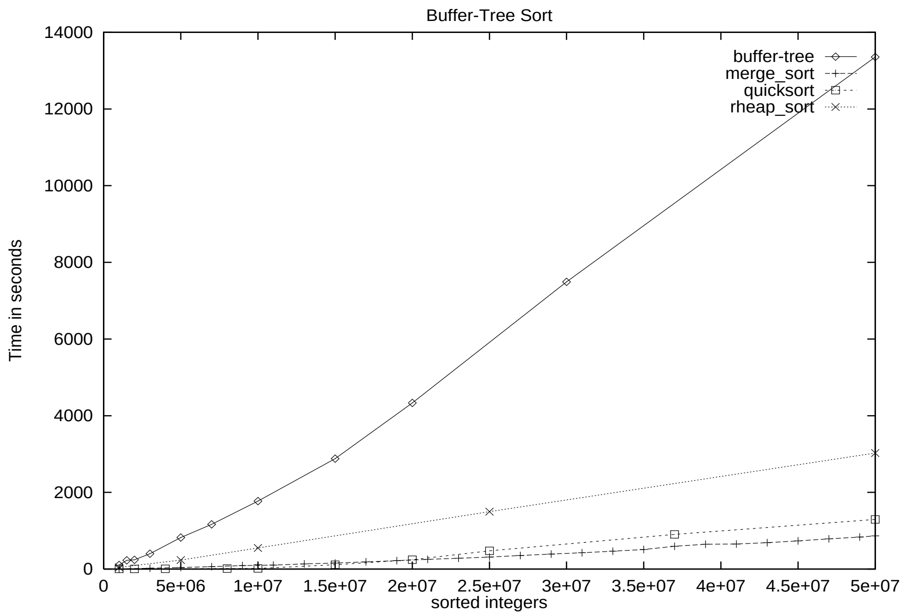

Figure 4.13: *Buffer Tree sort in comparison to other methods*

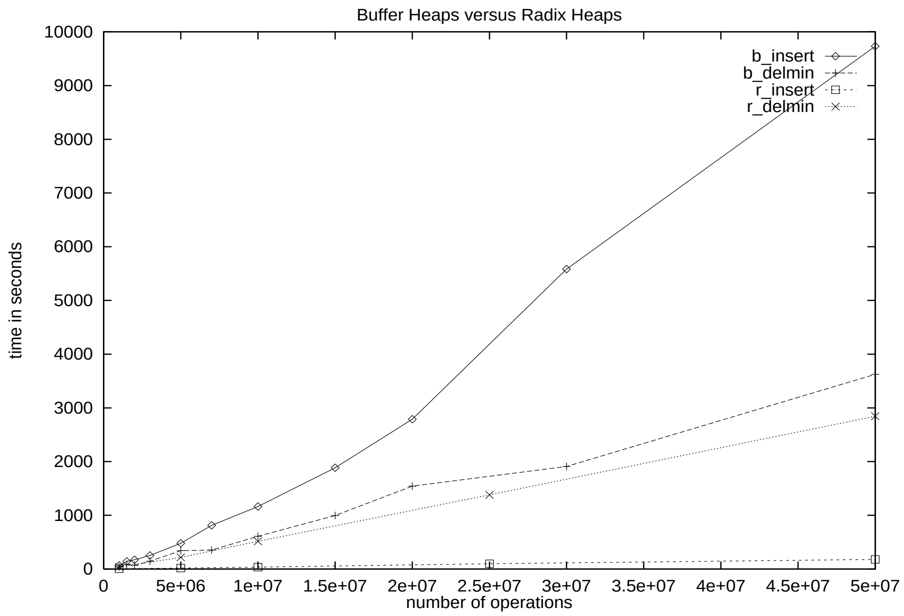

Figure 4.14: *Buffer Heaps versus Radix Heaps*

5 Matrix Operations

Matrix operations are a classical external memory application. We introduce a simple external memory algorithm for matrix multiplication. It is based on [UY91]. So far, we do not provide special algorithms for sparse matrix multiplication. Our algorithm is quite simple. We divide each matrix in $\sqrt{M} \times \sqrt{M}$ sub-matrices. After this preprocessing step we simple multiply sub-matrices with the standard matrix multiplication algorithm. This leads to a $\mathcal{O}(n^3/\sqrt{M}B)$ I/Os algorithm [UY91]. There are several implementation problems. Dividing into sub-matrices needs a rearrangement of the input. This is done by a permutation which requires sorting time and I/Os. We want to choose quadratic sub-matrices since this prevents us from handling a lot special cases. Since the number of rows/columns is normally not a multiple of the number of the sub-matrices subsize, we have to pad the input matrices to a multiple of the subsize. This leads to two preprocessing and two post-processing steps. They have overall I/O–complexity of 6 scans and 3 sorting steps. An additional 4 scans and 2 sorting steps are necessary to bring back the input matrices to their original size and ordering. To show the simplicity of code we give the multiplication example:

ext_matrix ext_matrix::operator(ext_matrix& D)*

```
int i,j,k;
int subsize;
subsize = choose_subsize(mem_size,row,col);  // choose subsize
cout << "Subsize is " << subsize << endl;
matrix op1(subsize,subsize);                 //allocate submatrices
matrix op2(subsize,subsize);
matrix sub_res(subsize,subsize);


int old_row = row;

ext_matrix::pad(subsize);                   //Padding of matrices
D.pad(subsize);

ext_matrix res(row,row);
```

```
ext_matrix::to_blocks(subsize);            //Dividing into sub-matrices
D.to_blocks(subsize);

for(i= 0;i< row/subsize;i++)              //start multiplication
  {
    for(j=0;j< D.col/subsize;j++)
      {
        for(int ii=0;ii<sub_res.dim1();ii++)
          for(int jj=0;jj<sub_res.dim2();jj++) sub_res(ii,jj) = 0;


        for(k=0;k<D.row/subsize;k++)
          {
            load(i,k,subsize,subsize,op1);

            D.load(k,j,subsize,subsize,op2);

            sub_res += op1*op2;
          }

        res.write(sub_res,i,j,subsize,subsize);
    }
  }

res.out_blocks(subsize);              // permute to row major order

res.unpad(old_row,old_row);           // shrink to old size

return res;
```

We tested matrix multiplication performance by comparing it to LEDA matrix multiplication code for data type `double`. The external matrix algorithm was restricted to use at most 4 Mbytes internal memory and a block size of 8 kbytes; while the LEDA matrix multiplication algorithm was allowed to use all the available internal memory (64 Mbytes). We did not count the time for post-processing the input matrices. We note that an external matrix needs 12 bytes per entry, the LEDA matrix needs 8 bytes per entry. Internal matrix multiplication in the external algorithm was done by LEDA matrix multiplication. The external algorithm was a magnitude faster than the internal one although we only used 4 Mbytes of internal memory per matrix and the input data had to be read from disk before multiplication (see Figure 5.1). At an input size of 0.5 times the size of the main memory, the external algorithm was faster than the internal one. As both algorithms have a CPU-time of $\mathcal{O}(N^3)$, we know that the matrix multiplication is I/O–bound and therefore the external algorithm outperforms the internal one. As the external algorithm operates on blocks of data, the cache behavior is much better than for the standard internal matrix multiplication algorithm. This is the reason why the external algorithm is already faster for inputs that should fit into main memory.

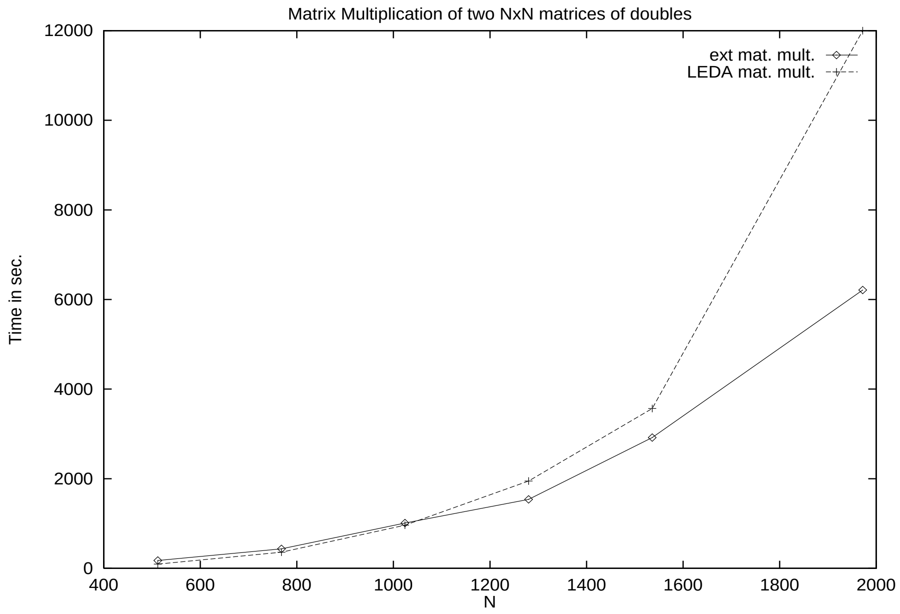

Figure 5.1: *Comparison of internal and external matrix multiplication*

6 Summary

We introduced a new library prototype for external memory data structures and algorithms. Based on the operating system's file system we showed that I/O can be implemented efficiently if we rely on the performance of the operating system. Several abstract data types as well as basic operations were tested. We summarize the main results:

- Sorting
  We implemented sorting by multiway–mergesort, buffer tree sort, radix heap sort and compared to standard virtual memory quicksort. Multiway–mergesort was the fastest sorting algorithm. The quicksort itself is a good choice for sorting if one allows it to use enough swap space. Buffer tree sort is the slowest sorting routine and should not be used for sorting in external memory. Radix heap sort outperforms every sorting algorithm if we know that the input are positive integers in a small interval $\{0, \ldots, C\}$. We note that if we compare quicksort with mergesort in such a way that both use the same amount of internal memory, then mergesort will be much faster than quicksort. But even with restricted main memory, mergesort could beat one of the fastest internal sorting algorithms.

- Priority Queues
  We tested buffer heaps and radix heaps. Radix heaps are always faster than buffer heaps for the input interval that we could test. The main advantages of radix heaps are small constants. However they are restricted to positive integer priorities (or at least to positive floating point numbers). Internal priority queues in virtual memory completely fail. For buffer trees in general, we note that tuning the data structure to achieve better performance is not easy. It seems that although this data structure leads to a lot of optimal algorithms in theory, in practice this data structure is not fast enough to achieve good performance.

- Matrix operations
  We implemented the classical matrix multiplication algorithm of [UY91] and compared it to the standard matrix multiplication algorithm in

virtual memory. The algorithm of [UY91] is one order of magnitude faster in the theoretical I/O–bound and even leads to a faster multiplication if the input is smaller than the main memory. This implies that the cache–miss ratio of the external algorithm is much better.

It turned out that many internal data structures fail in virtual memory even if the input size, counted in the number of elements, was "small". Therefore external data structures are useful to solve this dilemma even if we do not consider inputs in the range of Giga- and Terabytes.

6.1 Future Work

The work in this project is still ongoing. At the moment we are implementing suffix arrays and external graphs. We also developed a new priority queue which is only based on merging. This new data structure has been analyzed theoretically but it has not been implemented up to now. In the last months, we focussed our work on constructing large suffix arrays [MM93]. We have developed three new construction algorithms for suffix arrays and the results that we obtained are promising.

6.2 Acknowledgments

I want to thank my students Sven Thiel, Klaus Brengel, Henning Krone, Oliver Lambert, Ernst Althaus, Joerg Keller, Robert Wirth, Ralph Schulte, Thomas Buchheit and Mark Westphal for implementing most of the stuff. Furthermore thanks to Paolo Ferragina for many discussions on the string based stuff. I also want to thank Michael Seel from the LEDA research team who answered all my questions concerning the LEDA library. My special thank is to the RBG of the Max-Planck institute, especially to Jörg Herrmann and Thomas Hirtz who gave me all their support for maintaining our test system and to Wolfram Wagner who explained me the usage of the virtual memory monitoring tools. Last but not least my very special thank is to Kurt Mehlhorn for all his advice.

Appendix A The Elite-9 Fast SCSI-2 Wide Hard Disk

ST-410800W Elite 9	
Unformatted capacity	10800
Formatted capacity(512 byte blk)	9090
Average sectors per track	133
Actuator type	rotary voice coil
Tracks	132975
Cylinders	4925
Heads	27
Disks(5.25 in)	14
Media type	thin film
Head type	thin film
Recording method	ZBR RLL (1,7)
Internal transfer rate(mbits/sec)	44-65
Internal transfer rate avg(mbyte/sec)	7.2
External transfer rate(mbyte/sec)	20 (burst)
Spindle speed	5400
Average latency(msec)	5.55
Command overhead(msec)	<0.5
Buffer	1024 Kbyte
Bytes per track	63000 - 91000
Sectors per drive	17845731
Bytes per cylinder	1058400 to 1587600
TPI(tracks per inch)	3921
Average access(msec) read/write	11/12
Single rack seek(msec) read/write	0.9/1.7
Max full seek(msec) read/write	23/24

Table A.1: Technical Data of the Seagate Elite-9 Fast SCSI-2 Wide Disk

A.1 Low-Level transfer rate

We analyze the transfer rate of LEDA-SM by performing consecutive read and write operations to the underlying disk using different block sizes. We give below some logging files that are created by *iostat*. In detail, we perform the following test: We write 100000 consecutive blocks to disk and then read them again consecutively. The block size is 8 kbytes for the first test and 32 kbytes for the second. The fields in the tables have the following meaning:

disk	name of the disk
r/s	reads per second
w/s	writes per second
Kr/s	kilobytes read per second
Kw/s	kilobytes written per second
wait	average number of transactions waiting for service (queue length)
actv	average number of transactions actively being serviced (removed from the queue but not yet completed)
svc_t	average service time, in milliseconds
%w	percent of time there are transactions waiting for service (queue non-empty)
%b	percent of time the disk is busy (transactions in progress)

Each entry in the listing accounts for a time interval of 15 seconds.

```
Writing blocks
                                         extended disk statistics
disk      r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b
sd7       0.1 88.4    0.4 4815.0  0.0  5.8   66.0   0 100
sd7       0.1 88.1    0.4 4852.4  0.0  5.6   63.6   0 100
sd7       0.0 87.1    0.0 4738.5  0.0  5.9   67.8   0 100
sd7       0.1 86.0    0.8 4722.1  0.0  6.3   72.6   0 100
sd7       0.1 85.6    0.4 4650.0  0.0  6.0   70.1   0 100
sd7       0.1 83.7    0.8 4635.1  0.0  5.8   69.2   0 100
sd7       0.1 88.4    0.8 4882.9  0.0  5.6   63.8   0 100
sd7       0.1 82.3    0.4 4477.3  0.0  5.8   69.9   0 100
sd7       0.1 82.2    0.4 4524.2  0.0  5.6   68.6   0 100
sd7       0.1 80.4    0.8 4369.1  0.0  6.0   74.1   0 100
sd7       0.1 78.7    0.4 4355.8  0.0  5.7   72.9   0 100
sd7      56.2  8.1  449.6  397.2  0.0  1.4   21.1   0  89

Reading blocks

sd7      63.3  0.0  506.7    0.0  0.0  0.9   13.9   0  88
sd7      63.3  0.0  506.1    0.0  0.0  0.9   13.9   0  88
sd7      62.5  0.8  500.3   34.1  0.0  0.9   14.6   0  88
```

```
sd7      62.4  0.1  499.2    0.5  0.0  0.9   14.1   0  88
sd7      61.0  1.3  488.0   51.2  0.0  1.0   16.7   0  89
sd7      61.3  1.0  490.7   42.1  0.0  0.9   15.2   0  88
sd7      61.3  0.0  490.7    0.0  0.0  0.9   14.3   0  88
.
.
.
sd7      62.5  0.1  499.7    3.7  0.0  0.9   14.3   0  88
sd7      61.6  0.5  492.8   20.3  0.0  0.9   14.7   0  88
sd7      62.9  0.1  502.9    2.1  0.0  0.9   13.9   0  87
sd7      62.8  0.0  502.4    0.0  0.0  0.9   14.0   0  88
sd7      62.1  0.0  496.5    0.0  0.0  0.9   14.1   0  88
sd7      63.4  0.0  507.2    0.0  0.0  0.9   13.9   0  87
sd7      62.8  0.0  502.4    0.0  0.0  0.9   14.0   0  88
sd7      63.7  0.0  509.3    0.0  0.0  0.9   13.8   0  88
sd7      61.8  0.0  494.4    0.0  0.0  0.9   14.2   0  88
sd7      63.4  0.0  507.2    0.0  0.0  0.9   13.8   0  88
sd7      62.4  0.0  499.2    0.0  0.0  0.9   14.1   0  88
```

We see that for read, the disk is not busy for 100% of the time. The average service time is short (14 milliseconds). As the disk is not busy all the time, we hope to achieve higher transfer rates for read by using larger blocks. The test for $B = 32$ kbytes follows .

```
Writing blocks
                                   extended disk statistics
disk       r/s  w/s   Kr/s   Kw/s wait  actv  svc_t  %w  %b
sd7        0.0 85.9    0.0 4744.1  0.0  5.9   68.2   0 100
sd7        0.1 84.8    0.4 4593.4  0.0  6.0   70.1   0 100
sd7        0.1 83.9    0.4 4650.7  0.0  5.6   66.9   0 100
sd7        0.1 86.7    0.4 4713.7  0.0  6.0   69.2   0 100
sd7        0.1 82.3    0.4 4553.2  0.0  5.6   68.4   0 100
sd7        0.0 81.4    0.0 4408.8  0.0  5.8   70.9   0 100
sd7        0.1 80.2    0.4 4422.5  0.0  5.9   73.6   0 100
sd7        0.1 78.1    0.8 4310.4  0.0  5.7   72.4   0 100
sd7        0.1 91.3    0.8 4978.1  0.0  5.9   64.7   0 100
sd7        0.2 109.9   1.2 6090.6  0.0  5.5   50.3   0 100
sd7        0.3 108.6   1.6 5907.5  0.0  5.7   52.8   0 100
sd7        0.5 108.2   3.2 6002.7  0.0  5.7   52.1   0 100
sd7        0.4 106.7   2.4 5761.0  0.0  6.0   56.2   0 100
sd7       43.1 40.7 1657.0 2250.4  0.0  2.9   34.9   0  99
sd7       65.2  3.2 2523.3  132.6  0.0  1.6   22.7   0  98

Reading blocks

sd7       67.2  0.7 2618.9    4.2  0.0  1.4   21.2   0  98
sd7       69.5  0.0 2735.0    0.0  0.0  1.3   19.2   0  98
sd7       70.9  0.0 2804.5    0.0  0.0  1.4   19.2   0  99
sd7       68.9  0.0 2706.6    0.0  0.0  1.3   19.6   0  98
```

```
sd7       61.9  0.1 2255.7    4.7  0.0  1.7   26.7   0  99
sd7       68.2  0.3 2669.5   13.5  0.0  1.4   20.2   0  98
sd7       67.2  0.5 2630.7   25.5  0.0  1.4   20.1   0  98
sd7       69.5  0.0 2726.8    0.0  0.0  1.3   19.2   0  98
sd7       69.1  0.0 2697.2    0.0  0.0  1.4   19.6   0  98
sd7       68.6  0.1 2687.0    0.5  0.0  1.3   19.3   0  98
.
.
.
sd7       69.7  0.0 2719.0    0.0  0.0  1.4   19.5   0  98
sd7       68.5  0.1 2678.2    2.7  0.0  1.4   19.8   0  98
sd7       66.0  0.6 2547.5   23.3  0.0  1.5   21.9   0  98
sd7       69.2  0.1 2724.1    4.2  0.0  1.3   19.4   0  98
sd7       68.3  0.0 2674.9    0.0  0.0  1.4   19.9   0  98
sd7       70.2  0.0 2754.3    0.0  0.0  1.3   19.0   0  98
sd7       64.8  0.1 2520.2    1.6  0.0  1.3   20.3   0  98
sd7       69.4  0.0 2712.9    0.0  0.0  1.3   19.4   0  98
sd7       69.1  0.1 2686.7    4.7  0.0  1.4   20.1   0  98
sd7       65.2  0.0 2463.3    0.0  0.0  1.5   23.8   0  99
sd7       68.8  0.4 2694.7   12.4  0.0  1.4   19.8   0  98
sd7       68.9  0.5 2709.0   15.9  0.0  1.4   19.8   0  98
sd7       69.8  0.2 2684.8    5.6  0.0  1.4   19.9   0  98
sd7       69.0  0.0 2719.8    0.0  0.0  1.3   19.2   0  98
sd7       67.6  0.1 2643.3    0.9  0.0  1.3   19.4   0  98
sd7       70.0  0.1 2738.8    1.1  0.0  1.3   18.9   0  98
sd7       68.9  0.0 2706.3    0.0  0.0  1.4   20.0   0  98
sd7       64.4  0.2 2419.9    5.0  0.0  1.5   24.0   0  98
sd7       69.6  0.0 2733.6    0.0  0.0  1.4   19.5   0  98
```

For blocks of size 32 kbytes, the disk is nearly busy 100% of the time during read. The average service time is about 20 milliseconds for a read which is higher than before. This leads to the conclusion that it is further possible to speed up the I/O performance of the library LEDA-SM by carefully choosing the best block size. For our example this is for $B = 32$ kbytes.

Bibliography

[AEH84] T.O. Alanko, H.H.A. Erkiö, and I.J. Haikala. Virtual memory behavior of some sorting algorithms. *IEEE Tansactions on Software Engineering*, SE-10:422–431, 1984.

[AMOT90] R. Ahuja, K. Mehlhorn, J.B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *JACM*, 3(2):213–223, 1990.

[Arg96a] L. Arge. *Efficient external-memory data structures and applications.* PhD thesis, University of Aarhus, 1996.

[Arg96b] L. Arge. External-memory algorithms with applications in geographic information systems. CS Department, Duke Univeristy, technical report, 1996.

[AV88] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, 1988.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indizes. *Acta Informatica*, 1:173–189, 1972.

[CGG+95] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.

[CGS97] B.V. Cherkassky, A.V. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *SODA*, pages 83–92, 1997.

[Chi95] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Algorithms.* PhD thesis, Brown University, 1995.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill Book Company, 1990.

[Gro98] R. Grossi. Influence of block size on i/o performance. Personal Communication at BRICS, Aarhus, DK, 1998.

[HMSV97] D. Hutchinson, A. Makeshwari, J-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. *Workshop on Algorithmic Engineering*, 1997.

[HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for online string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[MN95] K. Mehlhorn and S. Näher. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[UY91] Ullman and Yannakakis. The Input/Output Complexity of Transitive Closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360, 1991.

[VS94] J.S. Vitter and E.A.M. Shriver. Optimal algorithms for parallel memory i:two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[VV95] D.E. Vengroff and J.S. Vitter. I/O-efficient scientific computation using tpie. In *IEEE Symposium on Parallel and Distributed Computing*, 1995.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid E-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems

MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values
MPI-I-97-2-009	A. Bockmayr, F. Eisenbrand	On the Chvátal Rank of Polytopes in the 0/1 Cube
MPI-I-97-2-008	A. Bockmayr, T. Kasper	A Unifying Framework for Integer and Finite Domain Constraint Programming
MPI-I-97-2-007	P. Blackburn, M. Tzakova	Two Hybrid Logics
MPI-I-97-2-006	S. Vorobyov	Third-order matching in $\lambda \rightarrow$-Curry is undecidable
MPI-I-97-2-005	L. Bachmair, H. Ganzinger	A Theory of Resolution
MPI-I-97-2-004	W. Charatonik, A. Podelski	Solving set constraints for greatest models
MPI-I-97-2-003	U. Hustadt, R.A. Schmidt	On evaluating decision procedures for modal logic
MPI-I-97-2-002	R.A. Schmidt	Resolution is a decision procedure for many propositional modal logics
MPI-I-97-2-001	D.A. Basin, S. Matthews, L. Viganò	Labelled modal logics: quantifiers
MPI-I-97-1-028	M. Lermen, K. Reinert	The Practical Use of the $\mathcal{A}^*$ Algorithm for Exact Multiple Sequence Alignment
MPI-I-97-1-027	N. Garg, G. Konjevod, R. Ravi	A polylogarithmic approximation algorithm for group Steiner tree problem
MPI-I-97-1-026	A. Fiat, S. Leonardi	On-line Network Routing - A Survey