LEDA-SM
Extending LEDA to Secondary Memory

Andreas Crauser and Kurt Mehlhorn*

Max-Planck-Institut für Informatik
Saarbrücken, Germany.
{crauser,mehlhorn}@mpi-sb.mpg.de

April 8, 1999

Abstract

During the last years, many software libraries for *in-core* computation have been developed. Most internal memory algorithms perform very badly when used in an *external memory* setting. We introduce LEDA-SM that extends the LEDA-library [21] towards secondary memory computation. LEDA-SM uses I/O-efficient algorithms and data structures that do not suffer from the so called *I/O bottleneck*. LEDA is used for in-core computation. We explain the design of LEDA-SM and report on performance results.

1 Introduction

Current computers have large main memories, but some applications need to manipulate data sets that are too large to fit into main memory. Very large data sets arise, for example, in geographic information systems [3], text indexing [5], WWW-search, and scientific computing. In these applications, *secondary memory* (mostly disks) [23] provides the required workspace. It has two features that distinguish it from main memory:

- Access to secondary memory is slow. An access to a data item in external memory takes much longer than an access to the same item in main memory; the relative speed of a fast internal cache and a slow external memory is close to one million.
- Secondary memory rewards locality of reference. Main memory and secondary memory exchange data in *blocks*. The transfer of a block between main memory and secondary memory is called an *I/O-operation* (short I/O).

*research partially supported by the EU ESPRIT LTR Project Nr. 20244 (ALCOM-IT)

Standard data structures and algorithms are not designed for locality of reference and hence frequently suffer an intolerable slowdown when used in conjunction with external memory. They simply use the virtual memory (provided by the operating system) and address this memory as if they would operate in internal memory, thus performing huge amounts of I/Os. In recent years the algorithms community has addressed this issue and has developed I/O-efficient algorithms for many data structure, graph-theoretic, and geometric problems [27, 6, 10, 18, 17]. Implementations and experimental work are lacking behind.

External memory algorithms move data in the memory hierarchy and process data in main memory. A platform for external memory computation therefore has to address two issues: movement of data and co-operation with internal memory algorithms.

We propose *LEDA-SM* (LEDA secondary memory) as a platform for external memory computation. It extends LEDA [21, 22] to secondary memory computation and is therefore directly connected to an efficient internal-memory library of data structures and algorithms. LEDA-SM is portable, easy to use and efficient. It consists of:

- a kernel that gives an abstract view of secondary memory and provides a convenient infrastructure for implementing external memory algorithms and data structures. We view secondary memory as a collection of disks and each disk as a collection of blocks. There are currently four implementations of the kernel, namely by standard I/O (`stdio`), system call I/O (`syscall`), memory mapped I/O (`mmapio`) and memory disks (`memory`). All four implementations are portable across Unix-based (also Linux) platforms, and `stdio` and `syscall` are also for Microsoft operating systems.

- a collection of external memory data structures. An external memory data structure offers an interface that is akin to the interface of the corresponding internal memory data structures (of LEDA), uses only a small amount of internal memory, and offers efficient access to secondary memory. For example, an *external stack* offers the stack operations *push* and *pop*, requires only slightly more than two blocks of internal memory, and needs only $O(1/B)$ I/O-operations per push and pop, where B is the number of data items that fit into a block.

- algorithms operating on these data structures. This includes basic algorithms like sorting as well as matrix multiplication, text indexing and simple graph algorithms.

- a precise and readable specification for all data structures and algorithms. The specifications are short and abstract so as to hide all details of the implementation.

The external memory data structures and algorithms of LEDA-SM (items (2) and (3)) are based on the kernel; however, their use requires little knowledge of the kernel. LEDA-SM[1] supports fast prototyping of secondary memory algorithms and therefore can be used to experimentally analyze new data structures and algorithms in a secondary memory setting.

The database community has a long tradition of dealing with external memory. Efficient index structures, e.g, B-trees [7] and extendible hashing [15], have been designed and highly optimized implementations are available. "General purpose" external memory computation has never been a major concern for the database community.

In the algorithms community implementation work is still in its infancy. There are implementations of particular data structures [9, 19, 4] and there is TPIE [26], the transparent I/O-environment. The former work aimed at investigating the relative merits of different data structures, but not at external memory computation at-large. TPIE provides some *external programming paradigms* like scanning, sorting and merging. It does not offer external memory data structures and it has no support for internal memory computation. Both features were missed by users of TPIE [19], it is planned to add both features to TPIE (L. Arge, personal communication, February 99).

This paper is structures as follows. In Section 2 we explain the software architecture of LEDA-SM and discuss the main layers of LEDA-SM (kernel, data structures and algorithms). We describe the kernel and give an overview of the currently available data structures and algorithms. In Section 3 we give two examples to show (i) how the kernel is used for implementing an external data structure and (ii) the ease of use of LEDA-SM and its with LEDA. Some experimental results are given Section 4. We close with a discussion of future developments.

2 LEDA-SM

LEDA-SM is a C++ class library that uses LEDA for internal memory computations. LEDA-SM is designed in a modular way and consists of 4 layers:

Layer	Major Classes
algorithms	sorting, graph algorithms, ...
data structures	*ext_stack*, *ext_array*, ...
abstract kernel	*block*<*E*>, *B_ID*, *U_ID*
concrete kernel	*ext_memory_manager*, *ext_disk*, *ext_free_list*, *name_server*

We use application layer for the upper two layers and kernel layer for the lower two layers.

The *kernel layer* of LEDA-SM manages secondary memory and movement of data between secondary memory and main memory. It is divided into the *abstract* and the *concrete* kernel. The abstract kernel consists of 3 C++ classes that give an abstract view of secondary memory. Two classes model disk block locations and users of disk blocks while the third class is a container class that is able to transfer elements of type E to and from secondary memory in a simple way. This class provides a typed view of data stored in secondary memory, data on disk is always untyped (type *void∗*).

The concrete kernel is responsible for performing I/Os and managing disk space in secondary memory and consists of 4 C++ classes. LEDA-SM provides several realizations of the concrete kernel; the user can choose one of them at run-time. The concrete kernel defines functions for performing I/Os and managing disk blocks, e.g. read/write a block, read/write k consecutive blocks, allocate/free a disk block etc. These functions are used by the abstract kernel or directly by data structures and algorithms.

The *application layer* consists of data structures and algorithms. LEDA is used to implement the in-core part of the applications and the kernel of LEDA-SM is used to perform the I/Os. The physical I/O calls are completely hidden in the concrete kernel, the applications use the container class of the abstract kernel to transport data to and from

[1]LEDA-SM can be downloaded from `www.mpi-sb.mpg.de/~crauser/leda-sm.html`

secondary memory. This makes the data structures simple, easy to use and still efficient.

We now explain in more detail the kernel layer. Examples for the application layer are given in Section 2.2.

2.1 The Kernel

The abstract kernel models secondary memory as files of the underlying file system. Secondary memory consists of NUM_OF_DISKS logical disks (files of the file system), $max\_blocks[d]$ blocks can reside on the d-th disk, $0 \leq d < NUM\_OF\_BLOCKS$. 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 $0 \leq d < NUM\_OF\_DISKS$ 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. 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 an *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 a block is created. The number of variables of type E is denoted by *blk_sz* and is equal to $(BLK\_SZ * sizeof(GenPtr) - num\_of\_bids * sizeof(B\_ID))/sizeof(E)$ where *GenPtr* is the generic pointer type of the machine (usually $void*$) and *BLK_SZ* is the physical disk block size in bytes. Both arrays are indexed starting at zero.

Every block has an associated user identifier and an associated block 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 container class *block*<*E*> is directly connected to functions of the concrete kernel (by use of the class *ext_memory_manager*), i.e. function *write* of class *block*<*E*> uses the write-function of the concrete kernel to initiate the physical I/Os.

The concrete kernel is responsible for performing I/Os and managing disk space in secondary memory and consists of classes *ext_memory_manager*, *ext_disk*, *ext_free_list*, *name_server*. The class *ext_disk* is responsible for file I/O and models an external disk device. There are several choices for file I/O: system call I/O, standard file I/O, memory mapped I/O and memory I/O (this is an in-memory disk simulation). Each of these methods has different advantages and disadvantages. Besides the file I/O method, there also exists the possibility to switch to the raw disk drive[2] (underlying the file system). By this, we simply drop the overhead introduced by the file system layer, on the other hand we also loose the caching effects that the file system performs. This feature is not portable across all platforms and is therefore not directly implemented in the library[3]. The class *ext_freelist* is responsible for managing allocated and free disk blocks in external memory. Disk blocks can be free or in-use by a dedicated user, the system must manage this. This is done by a specific data structure which is currently implemented in four different ways.

We can now summarize the software layout of LEDA-SM. The concrete kernel consists of class *ext_memory_manager*, class *name_server* and of the two kernel implementation classes for disk access and for disk block management. The abstract kernel consists of classes *block*, *B_ID* and *U_ID*. All data structures and algorithms of LEDA-SM are implemented using only the kernel classes and data structures or algorithms available in LEDA. In the next subsection we will give an overview of the currently available data structures and algorithms.

[2] This is the very reason why we model disk positions by an own class.

[3] In Solaris systems, this is just a change of a few code lines, but it also requires super-user rights.

2.2 Data Structures and Algorithms

We survey the data structures and algorithms currently available in LEDA-SM. Theoretical I/O-bounds are given in the classical external memory model of [27], where M is the size of the main memory, B is the size of a disk block, and N is the input size measured in disk blocks.

Stacks and Queues: External stacks and queues are simply the secondary memory counterpart to the corresponding internal memory data structures. Operations *push*, *pop* and *append* are implemented in optimal $O(1/B)$ amortized I/Os.

Priority Queues: Secondary memory priority queues can be used for large-scale event simulation, in graph algorithms or for online-sorting. Three different implementations are available. Buffer trees [2] achieve optimal $O(1/B \log_{M/B}(N/B))$ amortized I/Os for operations *insert* and *del_min*. Radix heaps are an extension of [1] towards secondary memory. This integer-based priority queue achieves $O(1/B)$ I/Os for *insert* and $O(1/B \log_{M/B}(C))$ I/Os for *del_min* where C is the maximum allowed difference between the last deleted minimum key and the actual keys in the queue. Array heaps [8, 13, 24] achieve $O(1/B \log_{M/B}(N/B))$ amortized I/Os for *insert* and $O(1/B)$ amortized I/Os for *del_min*.

Arrays: Arrays are a widely used data structure in internal memory. The main drawback of internal-memory arrays is the fact that when used in secondary memory, it is not possible to control the paging. Our external array data structure uses an internal-memory cache of fixed size that is multi-segmented. This allows us to efficiently support multiple data streams that operate on the array. Several page-replacement strategies are supported like LRU, random, fixed, etc. The user can also implement his/her own page-replacement strategy.

Sorting: Sorting is implemented by multiway-mergesort. Internal sorting during the run-creation phase is realized by LEDA-quicksort which is a fast and generic code-inlined template sorting routine. Sorting N items takes optimal $O(N/B \log_{M/B} (N/B))$ I/Os.

B-Trees: B-Trees [7] are the classical secondary memory online search trees. We use a B^+-implementation and support operations *insert*, *delete*, *delete_min* and *search* in optimal $O(\log_B(N))$ I/Os.

Suffix arrays and strings: Suffix arrays [20] are a full-text indexing data structure for large text strings. We provide several different construction algorithms [12] for suffix arrays as well as exact searching, 1- and 2-mismatch searching, and 1- and 2-edit searching.

Matrix operations: We provide matrices with entry type `double`. The operations $+$, $-$, and $*$ for dense matrices are realized with optimal I/O-bounds [25].

Graphs and graph algorithms: We provide a data type external graph and simple graph algorithms like depth-first search, topological sorting and Dijkstra's shortest path computation. External graphs are static, i.e. graph updates are expensive.

2.3 Further Features

Secondary-memory data structures and algorithms also use internal memory. LEDA-SM allows the user to control the amount of memory that each data structure and/or algorithm uses. The amount of memory is either specified at the time of construction of the data structure or it is an additional parameter of a function call. If we look at our stack example of Section 3 we see that the constructor of data type *ext_stack* has a parameter a, the number of blocks of size *blk_size* that are held in internal memory. We therefore immediately know that the internal memory space occupancy is $a \cdot \mathit{blz\_size} + O(1)$ bytes.

The second feature of LEDA-SM is accounting of I/Os. The kernel supports counting read- and write operations. Some of the reads may be logical and can be served by the buffer of the operating system. It is also possible to count consecutive I/Os (also introduced by [16, 12] as bulk I/Os). This allows the user to experimentally classify the I/O-structure of algorithms and in this way to compare algorithms with the same asymptotic I/O-complexity (see also [12]).

3 Examples

We discuss the implementation of secondary memory stacks and secondary memory graph search. The first example shows how the kernel of LEDA-SM is used to implement data structures and algorithms. The second example shows that secondary memory algorithms can be coded in a natural way in LEDA-SM and LEDA. It also shows the interplay between LEDA and LEDA-SM.

3.1 External Memory Stacks

We discuss the implementation of external memory stacks. It is simple, conceptually and programming-wise. The point of this section is to show how easy it is to translate an algorithmic idea into a program.

A external memory stack S for elements of type E (*ext_stack*<E>) is realized by an array (a LEDA data structure) A of $2a$ blocks of type *block*<E>

and a linear list of disk blocks. Each block in A can hold *blk_sz* elements, i.e, A can hold up to $2a \cdot blk\_sz$ elements. We may view A as a one-dimensional array of elements of type E. The slots 0 to *top* of this array are used to store elements of S with *top* designating the top of the stack. The older elements of S, i.e., the ones that do not fit into A, reside on disk. We use *bid* to store the block identifier of the elements moved to disk most recently. Each disk block has one link; it is used to point to the block below. The number of elements stored on disk is always a multiple of $a \cdot blk\_sz$.

⟨*ext_stack*⟩≡

```
template <class E>
class ext_stack
{
  array< block<E> > A;
  int top_cnt, a_sz, s_sz, blk_sz;
  B_ID bid;

public:

ext_stack(int a = 1);
void push(E x);
E pop();
E top();
int size() { return s_sz; };
void clear();
~ext_stack();
};
```

We next discuss the implementation of the operations *push* and *pop*. We denote by $a\_sz = 2a$ the size of array A. A push operation $S.push(x)$ writes x into the location $top + 1$ of A except if A is full. If A is full ($top\_cnt == a\_sz * blk\_sz - 1$), the first half of A is moved to disk, the second half of A is moved to the first half, and x is written to the first slot in the second half.

⟨*ext_stack*⟩+≡

```
template<class E>
void ext_stack<E>::push(E x)
{
  int i;
  if (top_cnt ==  a_sz*blk_sz - 1)
  {
    A[0](0) = bid;
    bid = ext_mem_mgr.new_blocks(myid,a_sz/2);
    block<E>::write_array(A,bid,a_sz/2);
    for(i=0;i<a_sz/2;i++)
      A[i] = A[i+a_sz/2];
    top_cnt = (a_sz/2)*blk_sz-1;
  }
  top_cnt++;
  A[top_cnt/blk_sz][top_cnt%blk_sz] = x;
  s_sz++;
}
```

The interesting case of *push* is the one where we have to write the first half of A to disk. In this step we have to do the following: we must reserve $a = a\_sz/2$ disk blocks on disk and we must add the first a blocks of array A to the linked list of blocks on disk. The blocks are linked by using the array (of length one) of type *B_ID* of class *block* (see Section 2.1) and the block least recently written is identified by block identifier *bid*. The commands $A[0](0) = bid$ creates a backwards linked list of disk blocks which we use during pop-operations later. We then use the kernel to allocate a consecutive free disk blocks by the command *ext_mem_mgr.new_blocks*. The return value is the first allocated block identifier. The first half of array A is written to disk by calling *write_array* of class *block* which tells the kernel to initiate the necessary physical I/Os. In the next step we copy the last a blocks of A to the first a blocks and reset *top_cnt*. Now the normal push can continue by copying element x to its correct location inside A.

A pop operation $S.pop(\,)$ is also trivial to implement. We read out the element in slot *top* except if A is empty. If A is empty and there are elements residing on disk we move $a \cdot blk\_sz$ elements from disk into the left half of A.

⟨*ext_stack*⟩+≡

```
template<class E>
E ext_stack<E>::pop()
{
  if (top_cnt == -1 && s_sz > 0)
  {
    B_ID oldbid = bid;
    block<E>::read_array(A,oldbid,a_sz/2);
    bid = A[0](0);
    top_cnt = (a_sz/2)*blk_sz - 1;
    ext_mem_mgr.free_blocks(oldbid,myid,a_sz/2);
  }
  s_sz--;
  top_cnt--;
  return
    A[(top_cnt+1)/blk_sz][(top_cnt+1)%blk_sz];
}
```

If array A is empty ($top\_cnt = -1$) we load a blocks from disk into the first a array positions of A by calling *read_array*. These disk blocks are identified by *bid*. We then restore the invariant that block identifier *bid* stores the block identifier of the blocks least recently written to disk. As the disk blocks are linked backwards, we can retrieve this block identifier from the first entry of the array of block identifiers of the first loaded disk block ($A[0](0)$). Array A now stores $a \cdot blk\_sz$ data items of type E. Variable *top_cnt* is reset to this value. The just loaded disk blocks are now stored internally, therefore there is no reason to keep them again on disk. These disk blocks are freed by calling the kernel routine *ext_mem_mgr.free_blocks*. Return-value of operation *pop* is the top-most element of A.

Operations *push* and *pop* move a blocks at a time. As the read and write requests for the a blocks always target consecutive disk locations, we can choose a in such a way that it maximizes disk-to-host throughput rate. After the movement A is half-full and hence there are no I/Os for the next $a \cdot blk\_sz$ stack operations. Thus the amortized number of I/Os per operations is $1/blk\_sz$ which is optimal. Stacks with fewer than $2a \cdot blk\_sz$ elements reside in-core.

3.2 Graph Search

We give a simple example that shows how to use both, LEDA data structures and LEDA-SM data structures and how they interact. Our example computes the number of nodes of a graph G reachable from a source node v by using graph search. We assume that a bit vector for all nodes of graph G can be stored in internal memory.

⟨*graph search*⟩≡
```
template<class T>
long graph_search( T& G, ext_node<T> v,
                    int_set Visited )
{
  ext_stack< ext_node<T> > S;
  ext_edge<T> e;
  long i = 1;

  Visited.insert(v);
  S.push(v);

  while( S.size())
  {
    v = S.pop();

    forall_out_edges(e,v,G)
    if( !Visited.member( G.target(e)) )
    {
      Visited.insert( G.target(e) );
      S.push( G.target(e) );
      i++;
    }
  }
  return i;
}


ext_graph<int,char,int,char> G;
int_set visited(1,G.number_of_nodes());
ext_node<ext_graph<int,char,int,char> > v;
int reachable = graph_search( G, v, visited );
```

The LEDA-SM graph data type *ext_graph* is parameterized so that algorithm-dependent information can be associated with nodes and edges. For efficiency, it is important that node and edge labels are stored directly in the nodes and edges and are not accessed through pointers or indices (as this would imply an I/O-operation for each label). Different algorithms need different labels and hence it is crucial that the space allocated for labels can be reused. We have chosen the following design: each node (and similarly edge) stores an information of an arbitrary type X and an array of characters; X and the size of the array are fixed when the graph is defined. We use the array of characters to store arbitrary information of fixed length (through appropriate casting) and use the field of type X for "particularly important" information; we could also do without it. In our example, each node and edge has an *int* and a single *char* associated with it. Edges and nodes are parameterized with the graph data type to which they belong.

The example shows the interaction between LEDA and LEDA-SM. Data structures from both libraries are used. The bit vector *Visited* is implemented by LEDA data type *int_set*. Reached nodes are directly inserted into the *int_set*. The necessary type conversion from type *ext_node* to type *int* is performed by LEDA-SM, i.e. type *ext_node* can be converted to type *int*, because the nodes of an external graph are consecutively numbered. We have implemented *graph_search* in a recursion-free way by using a stack (type *ext_stack*). We have chosen an external stack with minimum internal space requirement.

The example shows that LEDA-SM and LEDA support a natural programming style and allow the user to freely combine the required data structures and algorithms. The I/O operations of LEDA-SM are completely hidden in the data and not visible in the algorithms layer.

We conclude this section by explaining how one can eliminate the use of an internal bit-field with size equal to the number of nodes of G. If the main memory cannot store a bit-field, we use external depth-first search that proceeds in rounds (see [10]). In each round, we store visited nodes in an internal dictionary (LEDA data type *dictionary*) of a fixed size. If the dictionary is full (no space left in main memory), we compact the edges of the external graph by deleting (marking) those edges that point to nodes that were already visited. We then delete the dictionary and start the next round.

4 Performance Results

We report on the performance of LEDA-SM. In our tests, we compare secondary memory data structures and algorithms of LEDA-SM to their internal memory counterparts (of LEDA) and we compare TPIE to LEDA-SM. For more detailed results on

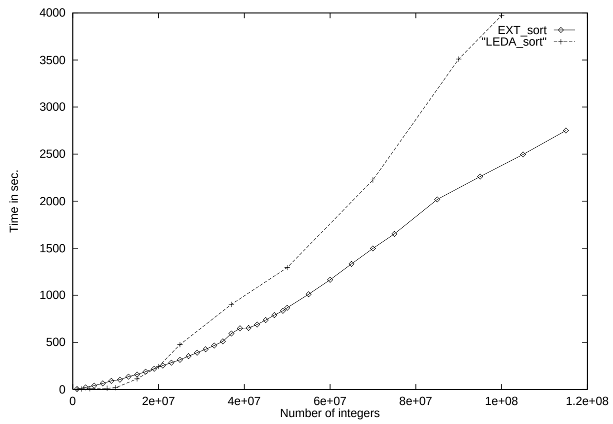

Figure 1: *Comparison of Multiway-Mergesort of LEDA-SM and quicksort of LEDA.*

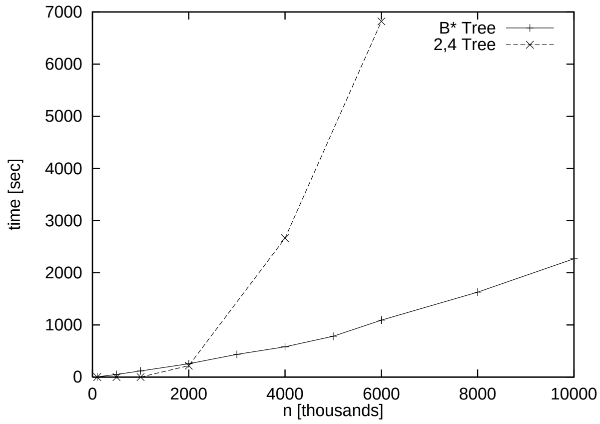

Figure 3: *Performance of insert-operation for LEDA-SM B-Trees and 2-4-trees of LEDA.*

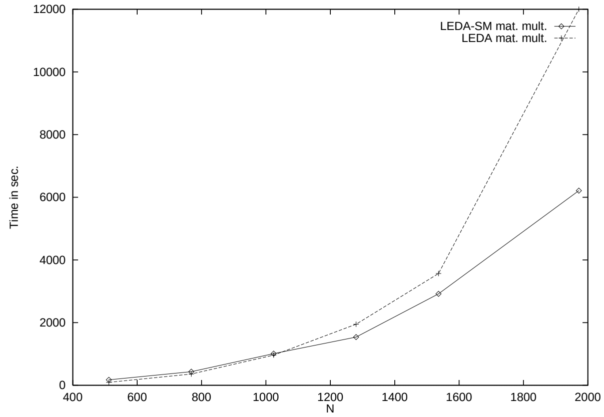

Figure 2: *Comparison of LEDA-SM matrix multiplication against LEDA matrix multiplication.*

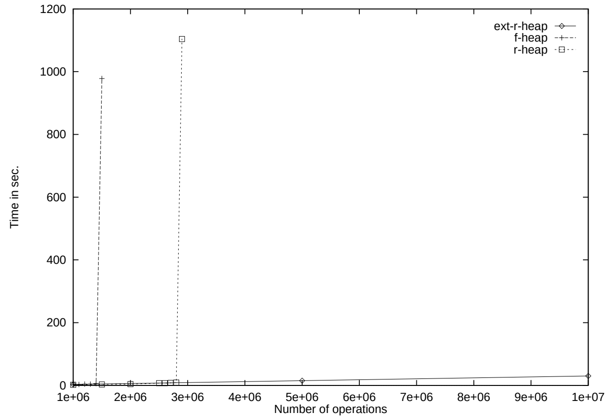

Figure 4: *Comparison of LEDA-SM radix heaps against LEDA's radix-heaps and Fibonacci-heaps.*

the performance of LEDA-SM see [11]. All tests were performed on a SUN UltraSparc1/143 using 64 Mbytes main memory and a single SCSI-disk connected to a fast-wide SCSI controller. All tests used a disk block size of 8 kbytes. We note that this is not the optimal disk block size for the disk according to data throughput versus service time. However, this disk block size allows us to compare the secondary memory algorithms in a fair way to internal memory algorithms in swap space because the page size of the machine is also 8 kbytes.

Figure 1 compares external multiway mergesort and LEDA quicksort. External multiway mergesort uses LEDA quicksort to partition the data into sorted runs before merging proceeds. The sorted runs are merged using a priority queue. The external sorting routine uses approx. 16 Mbytes of main memory. As soon as the input to be sorted reaches the size of the main memory, the external sorting routine is faster than the internal sorting routine. The sharp bend in the curve of external multiway mergesort occurs because we change the priority queue implementation inside the merging routine.

Figure 2 compares internal and external matrix multiplication of dense matrices of type *double*. The external matrix multiplication code uses tiling and matrix reordering (see also [26]). External matrix multiplication is faster than internal matrix multiplication even if the total input size is smaller than the main memory. This effect is due to the better cache behavior of the external matrix code (although it uses the same internal matrix multiplication code as LEDA).

Figure 3 compares the performance of operation *insert* of B-Trees and 2-4 trees. Both data structures are classical online search trees. The internal data structure is slowed down dramatically if it is running in swap space. The 2-4 tree does not ex-

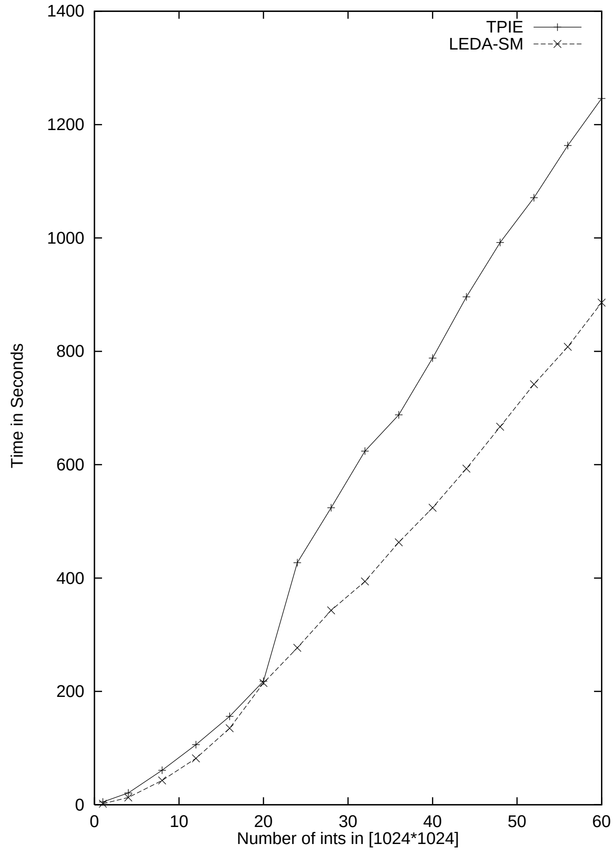

Figure 5: *Comparison of sorting efficiency for TPIE and LEDA-SM.*

hibit locality of reference and its use of pointers leads to many page faults.

Figure 4 compares the performance of operation *insert* of external radix heaps [13] to Fibonacci heaps and radix heaps [1] of LEDA. We see that the internal priority queues fail if the main memory size is exceeded.

Figure 5 compares the sorting routines of TPIE and LEDA-SM. Both sorting routines use external-multiway mergesort. An internal memory of 2 Mbytes is used and data type *int* is sorted. Both libraries use memory mapped I/O to transfer data. The block size is 8 kbytes. The test was performed on a SUN Ultra-10 with 128 Mbytes of main memory, a 333 MHz Ultra-SparcIIi processor and a 8.5 Gbytes local IDE hard disk. TPIE was compiled with GNU g++-2.7.2.3 and LEDA-SM was compiled with GNU eg++-1.1.2. We used both compilers with optimizer option `-O`. Unfortunately, it was not possible to use the same compilers; the TPIE implementation does not comply with the new C++ standard and hence cannot be compiled with any newer GNU compiler. The sorting code of LEDA-SM is always faster than that of TPIE. There is a jump in the TPIE-curve at an input size of roughly 22 million where the sorting code of TPIE is getting slower. However, even for smaller inputs, LEDA-SM provides the faster sorting routine.

5 Conclusions

We proposed LEDA-SM, an extension of LEDA towards secondary memory. The library follows LEDA's main features: portability, ease-of-use and efficiency. The performance results of LEDA-SM are promising. Although we use a high-level implementation for the library, we are orders of magnitudes faster than the corresponding internal data structures. The speedup increases for larger disk block sizes. The performance of many secondary memory data structures is determined by the speed of their internally used data structures; recall that the goal of algorithm design for external memory is to overcome the I/0-bottleneck. If succesful, external memory algorithms are compute-bound. LEDA-SM profits from the efficient in-core data structures and algorithms of LEDA[4]. LEDA-SM is still growing. Future directions of research cover geometric computation [14] as well as graph applications. Important practical experiments should be done for parallel disks (RAID arrays) as well as low-level disk device access.

Acknowledgements. The first author wants to thank Stefan Schirra for his help on C++ language problems and the Rechnerbetriebsgruppe of the Max-Planck-Institute for providing the necessary test equipment.

References

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

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

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

[4] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In C. C. McGeoch and M. T. Goodrich, editors, *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALENEX '99)*,

[4] LEDA's quicksort routine was recenctly improved (as a consequence of our tests of LEDA-SM-mergesort). For user-defined data types this led to a speed-up of almost five. LEDA-SM's multiway-mergesort profited immediately; its running time was reduced by a factor of two.

volume (to appear) of *Lecture Notes in Computer Science*. Springer, 1999.

[5] R. Baeza-Yates, E. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[6] R. Barve, E. Grove, and J. Vitter. Simple randomized merge sort on parallel disks. In *SPAA*, 1996.

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

[8] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In S. Arnborg and L. Ivansson, editors, *Proceedings 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118, Stockholm, Sweden, July 1998. Springer.

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

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

[11] Crauser, Mehlhorn, Althaus, Brengel, Buchheit, Keller, Krone, Lambert, Schulte, Thiel, Westphal, and Wirth. On the performance of LEDA-SM. Technical Report MPI-I-98-1-028, Max-Planck Institut für Informatik, 1998.

[12] A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. *European Symposium on Algorithms (ESA'99)*, 1999.

[13] A. Crauser, P. Ferragina, and U. Meyer. Efficient priority queues for external memory. Working Paper, Oktober 1997.

[14] A. Crauser, K. Mehlhorn, and U. Meyer. Kürzeste-Wege-Berechnung bei sehr großen Datenmengen. In O. Spaniol, editor, *Promotion tut not : Innovationsmotor "Graduiertenkolleg" (Aachener Beiträge zur Informatik; Band 21)*, pages 113–132S., Aachen, 1997. Verlag der Augustinus Buchhandlung.

[15] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, Sept. 1979. Also published in/as: IBM, Research Report RJ2305, Jul. 1978.

[16] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. *IEEE Symposium on Foundations of Computer Science*, 1998.

[17] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 693–702. ACM, May 1995.

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

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

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

[21] K. Mehlhorn and S. Näher. The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press, 1999. Draft versions of some chapters are available at `http://www.mpi-sb.mpg.de/~mehlhorn`.

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

[23] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, pages 17–28, 1994.

[24] P. Sanders. Fast priority queues for cached memory. In *Workshop in Algorithmic Engineering and Experimentation (ALENEX)*, Lecture Notes in Computer Science. Springer, 1999.

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

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

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