SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

STXXL: standard template library for XXL data sets

R. Dementiev[1,*], L. Kettner[23] and P. Sanders[1]

[1] *Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany*
[2] *Max Planck Institut für Informatik, Saarbrücken, Germany*
[3] *Now at mental images GmbH, Berlin, Germany*

SUMMARY

We present the software library STXXL **that is an implementation of the C++ standard template library STL for processing huge data sets that can fit only on hard disks. It supports parallel disks, overlapping between disk I/O and computation and it is the first I/O-efficient algorithm library that supports the pipelining technique that can save more than half of the I/Os.** STXXL **has been applied both in academic and industrial environments for a range of problems including text processing, graph algorithms, computational geometry, gaussian elimination, visualization, and analysis of microscopic images, differential cryptographic analysis, etc. The performance of** STXXL **and its applications is evaluated on synthetic and real-world inputs. We present the design of the library, how its performance features are supported, and demonstrate how the library integrates with STL.**

KEY WORDS: very large data sets; software library; C++ standard template library; algorithm engineering

1. INTRODUCTION

Massive data sets arise naturally in many domains. Spatial data bases of geographic information systems like GoogleEarth and NASA's World Wind store terabytes of geographically-referenced information that includes the whole Earth. In computer graphics one has to visualize highly complex scenes using only a conventional workstation with limited memory [1]. Billing systems of telecommunication companies evaluate terabytes of phone call log files [2]. One is interested in analyzing huge network instances like a web graph [3] or a phone call graph. Search engines like Google and Yahoo provide fast text search in their data bases indexing billions of web pages. A precise simulation of the Earth's climate needs to manipulate with petabytes of data [4]. These examples are only a sample of numerous applications which have to process vast amounts of data.

*Correspondence to: Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, Email: dementiev@ira.uka.de
Contract/grant sponsor: Partially supported by DFG; contract/grant number: SA 933/1-2

Received ???
Revised ???

The *internal memories*[†] of computers can keep only a small fraction of these large data sets. During the processing the applications need to access the *external memory*[‡] (e. g. hard disks) very frequently. One such access can be about 10^6 times slower than a main memory access. Therefore, the disk accesses (I/Os) become the main bottleneck.

The data is stored on the magnetic surface of a hard disk that rotates 4 200–15 000 times per minute. In order to read or write a designated track of data, the disk controller moves the read/write arm to the position of this track (seek latency). If only a part of the track is needed, there is an additional rotational delay. The total time for such a disk access is an average of 3–10 ms for modern disks. The latency depends on the size and rotational speed of the disk and can hardly be reduced because of the *mechanical* nature of hard disk technology. After placing the read/write arm, the data is streamed at a high speed which is limited only by the surface data density and the bandwidth of the I/O interface. This speed is called sustained throughput and achieves up to 80 MByte/s nowadays. In order to amortize the high seek latency, one reads or writes the data in blocks. The block size is balanced when the seek latency is a fraction of the sustained transfer time for the block. Good results show blocks containing a full track. For older low density disks of the early 90's the track capacities were about 16-64 KB. Nowadays, disk tracks have a capacity of several megabytes.

Operating systems implement the virtual memory mechanism that extends the working space for applications, mapping an external memory file (page/swap file) to virtual addresses. This idea supports the Random Access Machine model [5] in which a program has an infinitely large main memory. With virtual memory the application does not know where its data is located: in the main memory or in the swap file. This abstraction does not have large running time penalties for simple sequential access patterns: The operating system is even able to predict them and to load the data in ahead. For more complicated patterns these remedies are not useful and even counterproductive: the swap file is accessed very frequently; the executable code can be swapped out in favor of unnecessary data; the swap file is highly fragmented and thus many random I/O operations are needed even for scanning.

1.1. I/O-efficient algorithms and models

The operating system cannot adapt to complicated access patterns of applications dealing with massive data sets. Therefore, there is a need of explicit handling of external memory accesses. The applications and their underlying algorithms and data structures should care about the pattern and the number of external memory accesses (I/Os) which they cause.

Several simple models have been introduced for designing I/O-efficient algorithms and data structures (also called *external memory* algorithms and data structures). The most popular and realistic model is the Parallel Disk Model (PDM) of Vitter and Shriver [6]. In this model, I/Os are handled explicitly by the application. An I/O operation transfers a block of B consecutive elements[§] from/to

[†]The *internal memory*, or primary memory/storage, is a computer memory that is accessible to the CPU without the use of the computer's input/output (I/O) channels.

[‡]The *external memory*, or secondary memory/storage, is a computer memory that is not directly accessible to the CPU, requiring the use of the computer's input/output channels. In computer architecture and storage research the term of *"external storage"* is used more frequently. However, in the field of theoretical algorithm research the term of *"external memory"* is more established.

[§]In this paper B denotes the number of bytes in a block.

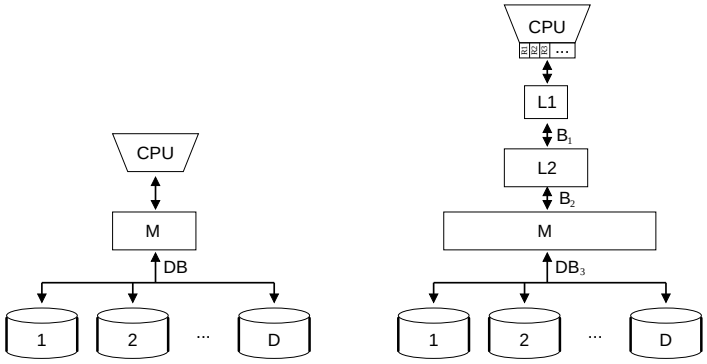

Figure 1. Schemes of parallel disk model (left) and memory hierarchy (right).

a disk to amortize the latency. The application tries to transfer D blocks between the main memory of size M bytes and D independent disks in one I/O step to improve bandwidth, see Figure 1. The input size is N bytes which is (much) larger than M. The main complexity metrics of an I/O-efficient algorithm in PDM are the number of I/O steps (main metric) and the number of operations executed by the CPU. If not I/O but a slow internal CPU processing is the limiting factor of the performance of an application, we call such behavior *CPU-bound*.

The PDM has become the standard theoretical model for designing and analyzing I/O-efficient algorithms. For this model the following matching upper and lower bounds for I/O complexity are known. Scanning a sequence of N items takes $\mathrm{scan}(N) = \Theta(N/(DB))$ I/Os. Sorting a sequence of N items takes $\mathrm{sort}(N) = \Theta(\frac{N}{DB}\log_{M/B}(N/M))$ I/Os. Online search among N items takes $\mathrm{search}(N) = \Theta(\log_{DB}(N))$ I/Os.

1.2. Memory hierarchies

The PDM measures the transfers between the main memory and the hard disks, however, in modern architectures, the CPU does not access the main memory directly. There are a few levels of faster memory caches in-between (Figure 1): CPU registers, level one (L2), level two (L2) and even level three (L3) caches. The main memory is cheaper and slower than the caches. Cheap dynamic random access memory, used in the majority of computer systems, has an access latency up to 60 ns whereas L1 has a latency of less than a ns. However, for a streamed access a high bandwidth of several GByte/s can be achieved. The discrepancy between the speed of CPUs and the latency of the lower hierarchy levels grows very quickly: the speed of processors is improved by about 55 % yearly, the hard disk access latency only by 9 % [7]. Therefore, the algorithms which are aware of the memory hierarchy will continue to benefit in the future and the development of such algorithms is an important trend in computer science.

The PDM model only describes a single level in the hierarchy. An algorithm tuned to make a minimum number of I/Os between two particular levels could be I/O-inefficient on other levels. The

cache-oblivious model in [8] avoids this problem by not providing the knowledge of the block size B and main memory size M to the algorithm. The benefit of such an algorithm is that it is I/O-efficient on all levels of the memory hierarchy across many systems without fine tuning for any particular real machine parameters. Many basic algorithms and data structures have been designed for this model [8, 9, 10, 11]. A drawback of cache-oblivious algorithms playing a role in practice is that they are only *asymptotically* I/O-optimal. The constants hidden in the O-notation of their I/O-complexity are significantly larger than the constants of the corresponding I/O-efficient PDM algorithms (on a particular memory hierarchy level). For instance, a tuned cache-oblivious funnel sort implementation [12] is 2.6–4.0 times slower than our I/O-efficient sorter from STXXL (Section 6) for out-of-memory inputs [13]. A similar funnel sort implementation [14] is up to two times slower than the I/O-efficient sorter from the TPIE library (Section 1.7) for large inputs. The reason for this is that these I/O-efficient sorters are highly optimized to minimize the number of transfers between the main memory and the hard disks where the imbalance in the access latency is the largest. Cache-oblivious implementations tend to lose on the inputs, exceeding the main memory size, because they do (a constant factor) more I/Os at the last level of memory hierarchy. In this paper we concentrate on extremely large *out-of-memory* inputs, therefore we will design and implement algorithms and data structures efficient in the *PDM*.

1.3. Algorithm engineering for large data sets

Theoretically, I/O-efficient algorithms and data structures have been developed for many problem domains: graph algorithms, string processing, computational geometry, etc. (see the surveys [15, 16]). Some of them have been implemented: sorting, matrix multiplication [17], search trees [18, 19, 20, 21], priority queues [22], text processing [23]. However only few of the existing I/O-efficient algorithms have been studied experimentally. As new algorithmic results rely on previous ones, researchers, which would like to engineer practical implementations of their ideas and show the feasibility of *external memory* computation for the solved problem, need to invest much time in the careful design of unimplemented underlying external algorithms and data structures. Additionally, since I/O-efficient algorithms deal with hard disks, a good knowledge of low-level operating system issues is required when implementing details of I/O accesses and file system management. This delays the transfer of theoretical results into practical applications, which will have a tangible impact for industry. Therefore one of the primary goals of algorithm engineering for large data sets is to create software frameworks and libraries which handle both the low-level I/O details efficiently and in an abstract way, and provide well-engineered and robust implementations of basic external memory algorithms and data structures.

1.4. C++ standard template library

The Standard Template Library (STL) [24] is a C++ library which is included in every C++ compiler distribution. It provides basic data structures (called containers) and algorithms. STL containers are generic and can store any built-in or user data type that supports some elementary operations (e.g. copying and assignment). STL algorithms are not bound to a particular container: an algorithm can be applied to any container that supports the operations required for this algorithm (e.g. random access to its elements). This flexibility significantly reduces the complexity of the library.

STL is based on the C++ template mechanism. The flexibility is supported using compile-time polymorphism rather than the object oriented run-time polymorphism. The run-time polymorphism is implemented in languages like C++ with the help of virtual functions that usually cannot be inlined by C++ compilers. This results in a high per-element penalty of calling a virtual function. In contrast, modern C++ compilers minimize the abstraction penalty of STL inlining many functions.

STL containers include: `std::vector` (an unbounded array), `std::priority_queue`, `std::list`, `std::stack`, `std::deque`, `std::set`, `std::multiset` (allows duplicate elements), `std::map` (allows mapping from one data item (a key) to another (a value)), `std::multimap` (allows duplicate keys), etc. Containers based on hashing (`hash_set`, `hash_multiset`, `hash_map` and `hash_multimap`) are not yet standardized and distributed as an STL extension.

Iterators are an important part of the STL library. An iterator is a kind of handle used to access items stored in data structures. Iterators offer the following operations: read/write the value pointed by the iterator, move to the next/previous element in the container, move by some number of elements forward/backward (random access).

STL provides a large number of algorithms that perform scanning, searching and sorting. The implementations accept iterators that posses a certain set of operations described above. Thus, the STL algorithms will work on any container with iterators following the requirements. To achieve flexibility, STL algorithms are parameterized with objects, overloading the function operator (`operator()`). Such objects are called *functors*. A functor can, for instance, define the sorting order for the STL sorting algorithm or keep the state information in functions passed to other functions. Since the type of the functor is a template parameter of an STL algorithm, the function operator does not need to be virtual and can easily be inlined by the compiler, thus avoiding the function call costs.

The STL library is well accepted and its generic approach and principles are followed in other famous C++ libraries like Boost [25] and CGAL [26].

1.5. The goals of STXXL

Several external memory software library projects (LEDA-SM [27] and TPIE [28]) were started to reduce the gap between theory and practice in external memory computing. They offer frameworks which aim to speed up the process of implementing I/O-efficient algorithms, abstracting away the details of how I/O is performed. See Section 1.7 for an overview of these libraries.

The motivation for starting another project, namely STXXL, was that we wanted an easier to use and higher performance library. Here are a number of key new or improved features of STXXL:

- Transparent support of parallel disks. The library provides implementations of basic *parallel* disk algorithms. STXXL is the only external memory algorithm library supporting parallel disks. Such a feature was announced for TPIE in 1996 [29, 28].
- The library is able to handle problems of a *very large* size (up to dozens of terabytes).
- Improved utilization of computer resources. STXXL explicit supports *overlapping* between I/O and computation. STXXL implementations of external memory algorithms and data structures benefit from the overlapping of I/O and computation.
- STXXL achieves small constant factors in I/O volume. In particular, *"pipelining"* can save more than *half* the number of I/Os performed by many algorithms.

- Short *development times* due to well known STL-compatible interfaces for external memory algorithms and data structures. STL algorithms can be directly applied to STXXL containers (code reuse); moreover, the I/O complexity of the algorithms remains optimal in most cases.

1.6. Software facts

STXXL is distributed under the Boost Software License[¶] which is an open source license allowing free commercial use. The source code, installation instructions and documentations are available under http://stxxl.sourceforge.net/. The release branch of the STXXL project not including applications has about 25,000 physical source lines of code[‖].

1.7. Related work

TPIE [29, 30] was the first large software project implementing I/O-efficient algorithms and data structures. The library provides implementations of I/O efficient sorting, merging, matrix operations, many (geometric) search data structures (B^+-tree, persistent B^+-tree, R-tree, K-D-B-tree, KD-tree, Bkd-tree) and the logarithmic method. The work on the TPIE project is in progress. For our experiments we have used a TPIE version from September 2005.

The LEDA-SM [27] external memory library was designed as an extension to the LEDA library [31] for handling large data sets. The library offers implementations of I/O-efficient sorting, external memory stack, queue, radix heap, array heap, buffer tree, array, B^+-tree, string, suffix array, matrices, static graph, and some simple graph algorithms. However, the data structures and algorithms cannot handle more than 2^{31} elements. The development of LEDA-SM has been stopped. LEDA releases later than version 4.2.1 are not supported by the last LEDA-SM version 1.3. To experiment with LEDA-SM we have used the `g++` version 2.95, since it it is the most recent compiler supported by LEDA-SM 1.3.

LEDA-SM and TPIE currently offer only single disk external memory algorithms and data structures. They are not designed to explicitly support overlapping between I/O and computation. The overlapping largely relies on the operating system that caches and prefetches data according to a general purpose policy, which often cannot be as efficient as the explicit approach (see Section 6.3). Furthermore, in most of the operating systems, the overlapping based on the system cache requires additional copies of the data, which leads to computational and internal memory overhead.

A list of algorithms and data structures available in TPIE, LEDA-SM and STXXL is shown in Table I.

Database engines use I/O-efficient search trees and sorting to execute SQL queries, evaluating huge sets of table records. The idea of pipelined execution of the algorithms which process large data sets not fitting into the main memory is well known in relational database management systems [32]. The pipelined execution strategy executes a database query with a minimum number of external memory accesses, to save memory space to store intermediate results, and to obtain the first result as soon as possible.

[¶] http://www.boost.org/more/license_info.html
[‖] According to the SLOCCount tool http://www.dwheeler.com/sloccount/.

Table I. Algorithms and data structures of the external memory libraries.

Function	TPIE	LEDA-SM	STXXL
sorting	√	√	√
stack	√	√	√
queue	√	√	√
deque	—	—	√
array/vector	—	√	√
matrix operations	√	√	—
suffix array	—	√	√(extension)
search trees	B^+-tree, K-D-B-tree, Bkd-tree persist. B^+-tree, R-tree, KD-tree	B^+-tree buffer tree	B^+-tree
priority queue	—	array and radix heap	sequence heap
pipelined algorithms	—	—	√

The design framework FG [33] is a programming environment for parallel programs running on clusters. In this framework, parallel programs are split into series of asynchronous stages which are executed in a pipelined fashion with the help of multithreading. The pipelined execution allows to mitigate disk latency of external data accesses and communication network latency of remote data accesses. I/O and communication can be automatically overlapped with computation stages by the scheduler of the FG environment.

Berkeley DB [34] is perhaps the best open source external memory B^+-tree implementation. It has a dual license and is not free for industry. Berkeley DB has a very large user base, among those are `amazon.com`, Google and Motorola. Many free open source programs use Berkeley DB as their data storage backend (e.g. MySQL data base system).

There are several libraries for advanced models of computation which follow the interface of STL. The Standard Template Adaptive Parallel Library (STAPL) is a parallel library designed as a superset of the STL. It is sequentially consistent for functions with the same name and performs on uni- or multi-processor systems that utilize shared or distributed memory [35].

MCSTL (Multi-Core Standard Template Library) project [36] was started in 2006 at the University of Karlsruhe. The library is an implementation of the STL which uses multiple processors and multiple cores of a processor with shared memory. It already has implementations of parallel sorting, merging, random permutation, searching, scanning. We are currently working on using MCSTL to parallelize the internal work of STXXL algorithms and data structures.

There is a number of libraries which provide *persistent* containers [37, 38, 39, 40, 41, 42]. Persistent STL-compatible containers are implemented in [43, 44]. These containers can keep (some of) the elements in external memory transparently to the user. In contrast to STXXL, these libraries do not guarantee the *I/O-efficiency* of the containers, e. g. the PSTL [44] library implements search trees as I/O-inefficient red-black trees.

1.8. Overview

The paper is structured as follows. Section 2 overviews the design of STXXL. We explain the design decisions we have made to achieve high performance in Sections 3–5 in detail. Section 6 engineers an efficient parallel disk sorting, which is the most important component of an external memory library. The concept of algorithm pipelining is described in Section 7. The design of our implementation of pipelining is presented in Section 8. In Section 9 we implement a short benchmark that computes a maximal independent set of a graph and uses it to study the performance of STXXL. Section 10 gives a short overview of the projects using STXXL. We make some concluding remarks and point out the directions of future work in Section 11.

The shortened preliminary material of this paper has been published in conference proceedings as [45] and [46].

2. THE DESIGN OF THE STXXL LIBRARY

STXXL is a layered library consisting of three layers (see Figure 2). The lowest layer, the Asynchronous I/O primitives layer (AIO layer), abstracts away the details of how asynchronous I/O is performed on a particular operating system. Other existing external memory algorithm libraries only rely on synchronous I/O APIs [27] or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API [28]. These libraries also rely on uncontrolled operating system I/O caching and buffering in order to overlap I/O and computation in some way. However, this approach has significant performance penalties for accesses without locality. Unfortunately, the asynchronous I/O APIs are very different for different operating systems (e.g. POSIX AIO and Win32 Overlapped I/O). Therefore, we have introduced the AIO layer to make porting STXXL easy. Porting the whole library to a different platform requires only reimplementing the thin AIO layer using native file access methods and/or native multithreading mechanisms.

STXXL already has several implementations of the AIO layer which use different file access methods under POSIX/UNIX and Windows systems (see Table II). Porting STXXL to Windows took only a few days. The main effort was to write the AIO layer using native Windows calls. Rewriting the thread-related code was easy using the Boost thread library [25] whose interfaces are similar to POSIX threads. There were little header file and compiler-specific incompatibilities; those were solved by conditional compilation using the C++ preprocessor. The POSIX version of STXXL had run immediately on the all listed operating systems after changing some Linux-specific header file includes to more common POSIX headers.

The Block Management layer (BM layer) provides a programming interface emulating the *parallel* disk model. The BM layer provides an abstraction for a fundamental concept in the external memory algorithm design — a block of elements. The block manager implements block allocation/deallocation, allowing several block-to-disk assignment strategies: striping, randomized striping, randomized cycling, etc. The block management layer provides an implementation of parallel disk buffered writing [47], optimal prefetching [47], and block caching. The implementations are fully asynchronous and designed to explicitly support overlapping between I/O and computation.

The top of STXXL consists of two modules. The STL-user layer provides external memory sorting, external memory stack, external memory priority queue, etc. which have (almost) the same interfaces

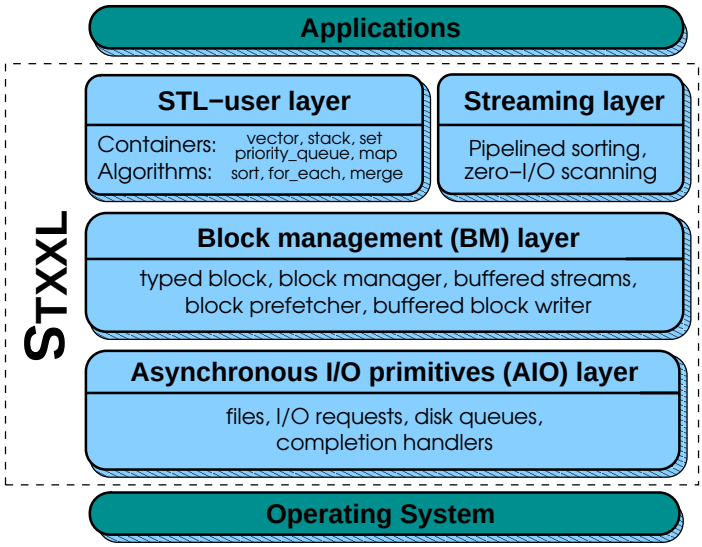

Figure 2. Structure of STXXL

Table II. Supported operating systems.

STXXL version	POSIX	MS Visual C++
OS	Linux, Cygwin, SunOS, Solaris, FreeBSD, NetBSD, MacOS	Windows 2000, Windows XP
Compiler	g++ 3.4+	MS VC++ 7.1+
Dependencies	Posix Threads	Boost library

(including syntax and semantics) as their STL counterparts. The Streaming layer provides efficient support for *pipelining* external memory algorithms. Many external memory algorithms, implemented using this layer, can save a factor of 2–3 in I/Os. For example, the algorithms for external memory suffix array construction implemented with this module [48] require only $1/3$ of the number of I/Os which must be performed by implementations that use conventional data structures and algorithms (either from the STXXL STL-user layer, LEDA-SM, or TPIE). The win is due to an efficient interface that couples the input and the output of the algorithm–components (scans, sorts, etc.). The output from an algorithm is directly fed into another algorithm as input, without needing to store it on the disk in-between. This generic pipelining interface is the first of this kind for external memory algorithms.

3. AIO LAYER

The purpose of the AIO layer is to provide a unified approach to asynchronous I/O. The layer hides details of native asynchronous I/O interfaces of an operating system. Studying the patterns of I/O accesses of external memory algorithms and data structures, we have identified the following functionality that should be provided by the AIO layer:

- To issue read and write requests without having to wait for them to be completed.
- To wait for the completion of a subset of issued I/O requests.
- To wait for the completion of at least one request from a subset of issued I/O requests.
- To poll the completion status of any I/O request.
- To assign a callback function to an I/O request which is called upon I/O completion (asynchronous notification of completion status), with the ability to co-relate callback events with the issued I/O requests.

The AIO layer exposes two user objects: `file` and `request_ptr`. Together with the I/O waiting functions `wait_all`, `wait_any`, and `poll_any` they provide the functionality mentioned above. Using a `file` object, the user can submit asynchronous read and asynchronous write requests (methods `file::aread` and `file::awrite`). These methods return a `request_ptr` object which is used to track the status of the issued request. The AIO layer functions `wait_all`, `wait_any`, and `poll_any` facilitate tracking a set of `request_ptr`s. The last parameter of the methods `file::aread` and `file::awrite` is a reference to a callback function object (callback functor). The functor's `operator()(request_ptr )` method is called when the I/O request is completed.

As a part of the AIO layer, the STXXL library provides various I/O performance counters (`stats` class). The class counts the number and the duration of the performed I/O operations as well as the transferred volume. Read and write operations are counted separately. STXXL also measures the time spent by the processing thread(s) waiting for the completions of I/Os (I/O wait time). This metric helps to evaluate the degree and the impact of overlapping between I/O and computation in an application.

Listing 1 shows a simple example of how to use AIO objects to perform asynchronous I/O. All STXXL library objects are defined in the namespace `stxxl`. For convenience, in Line 1 we bring all names from the STXXL namespace to the local scope. In Line 9 a file object `myfile` is constructed. `syscall_file` is an implementation of the STXXL `file` interface which uses UNIX/POSIX `read` and `write` system calls to perform I/O. The file named `"storage"` in the current directory is opened in read-only mode. In Line 10 an asynchronous read of the 1 MB region of the file starting at position 0 is issued. The data will be read into the array `mybuffer`. When the read operation is completed, `my_handler::operator()` will be called with a pointer to the completed request. The execution stops at Line 12 waiting for the completion of the issued read operation. Note that the work done in the function `do_something1()` is overlapped with reading. When the I/O is finished, one can process the read buffer (Line 13) and free it (Line 14).

3.1. AIO layer implementations

There are several implementation strategies for the STXXL AIO layer. Some asynchronous I/O related APIs (and underlying libraries implementing them) already exist. The most well known framework is POSIX AIO, which has an implementation on almost every UNIX/POSIX system. Its disadvantage

Listing 1. Example of how to program with the AIO layer.

```
1  using namespace stxxl;
2  struct my_handler { // I/O completion handler
3      void operator () (request_ptr ptr) {
4          std::cout << "Request '"<< *ptr <<"' completed."
5              <<std::endl;
6      }
7  };
8  char * mybuffer = new char[1024*1024]; // allocate 1MB buffer
9  syscall_file myfile("./storage", file::RDONLY );
10 request_ptr myreq = myfile.aread(mybuffer, 0, 1024*1024,my_handler());
11 do_something1(); //do_something1() is overlapped with reading
12 myreq->wait();    //wait for read completion
13 do_something2(mybuffer);// process the read buffer
14 delete [] mybuffer; // free the buffer
```

is that it has only limited support for I/O completion event mechanism**. The Linux AIO kernel side implementation †† of POSIX AIO does not have this deficit, but is not portable since it works under Linux only.

The STXXL AIO layer follows a different approach. It does not rely on any asynchronous I/O API. Instead we use synchronous I/O calls running asynchronously in separate threads. For each file there is one read and one write request queue and one thread. The main thread posts requests (invoking `file::aread` and `file::awrite` methods) to the file queues. The thread associated with the file executes the requests in FIFO order. This approach is very flexible and it does not suffer from limitations of native asynchronous APIs.

Our POSIX implementation of the AIO layer is based on POSIX threads and supports several Unix file access methods: the `syscall` method uses `read` and `write` system calls, the `mmap` method uses memory mapping (`mmap` and `munmap` calls), the `sim_disk` method simulates I/O timings of a hard disk provided a big internal memory. To avoid superfluous copying of data between the user and kernel buffer memory, the `syscall` method has the option to use unbuffered file system access. These file access methods can also be used for raw disk I/O, bypassing the file system. In this case, instead of files, raw *device handles* are open. The `read/write` calls using direct unbuffered access (`O_DIRECT` option) have shown the best performance under Linux. The direct unbuffered access bypasses the operating system buffering and caching mechanisms performing I/Os on user memory without copying the data to the system memory regions using direct memory access (DMA). The disadvantage of the `mmap` call is that programs using this method have less control over I/O: In most operating systems 4 KBytes data pages of a `mmap`ed file region are brought to the main memory "lazily", only when they are accessed for the first time. This means if one `mmap`s a 100 KBytes block and touches only the first and the last element of the block then *two* I/Os are issued by the operating system. This will slow down

**The Linux `glibc` implementation of POSIX AIO also has a performance drawback. It launches one user thread for each I/O operation. STXXL starts one thread for each disk during the library initialization, avoiding the thread start-up overhead for each I/O.

††http://freshmeat.net/projects/linux-aio/

many I/O-efficient algorithms, since for modern disks the seek time is much longer than the reading of 100 KBytes of contiguous data.

The POSIX implementation does not need to be ported to other UNIX compatible systems, since POSIX threads is the standard threading API on all POSIX-compatible operating systems.

AIO file and request implementation classes are derived from the generic `file` and `request` interface classes with C++ pure virtual functions. These functions are specialized for each access method in implementation classes to define the read, write, wait for I/O completion and other operations. The desired access method implementation for a file is chosen dynamically at running time. One can add the support of an additional access method (e.g. for a DAFS distributed filesystem) just providing classes implementing the `file` and `request` interfaces. We have decided to use the virtual function mechanism in the AIO layer because this mechanism is very flexible and *will not sacrifice* the performance of the library, since the virtual functions of the AIO layer need to be called only once per *large* chunk of data (i.e. B bytes). The inefficiencies of C++ virtual functions are explained in Section 1.4. Similar to STL, the higher layers of STXXL do not rely on run-time polymorphism with virtual functions to avoid the high per–element penalties.

4. BM LAYER

The BM layer includes a toolbox for allocating, deallocating, buffered writing, prefetching, and caching of blocks. The external memory manager (object `block_manager`) is responsible for allocating and deallocating external memory space on disks. The manager supports four parallel disk allocation strategies: simple striping, fully randomized, simple randomized [49], and randomized cycling [50].

The BM layer also delivers a set of helper classes that efficiently implement frequently used sequential patterns of interaction with parallel disks. The optimal parallel disk queued writing [47] is implemented in the `buffered_writer` class. The class operates on blocks. The `buf_ostream` class is built on top of `buffered_writer` and has a high-level interface, similar to the interface of STL output iterators. Analogously, the classes `block_prefetcher` and `buf_istream` contain an implementation of an optimal parallel disk *prefetching* algorithm [47]. The helper objects of the BM layer support overlapping between I/O and computation, which means that they are able to perform I/O in the background, while the user thread is doing useful computations.

The BM layer views external memory as a set of large AIO files — one for each hard disk. We will refer to these files as *disks*. The other approach would be to map a related subset of blocks (e. g. those belonging to the same data structure) to a separate file. This approach has some performance problems. One of them is that since those (numerous) files are created dynamically, during the run of the program, the file system allocates the disk space on demand, that might in turn introduce severe disk space fragmentation. Therefore we have chosen the "one-large-file-per-disk" approach as our major scheme. However, the design of our library does not forbid data structures to store their content in separate user data files (e.g., as an option, `stxxl::vector` can be mapped to a user file, see Section 5).

The external memory manager (object `block_manager`) is responsible for allocating and deallocating external memory space on the disks. When allocating a bunch of blocks, a programmer can specify how the blocks will be assigned to disks, passing an allocation strategy function object. When an application requests several blocks from a disk, the manager tries to allocate the blocks contiguously. This reduces the *bulk* access time. On allocation requests, the `block_manager` returns `BID` objects

– Block IDentifiers. An object of the type `BID` describes the physical location of an allocated block, including the disk and offset of a region of storage on disk.

The full signature of the STXXL "block of elements" class is `typed_block<RawSize,T,NRef,InfoType>`. The template parameter `RawSize` defines the total size of the block in bytes. Since block size is not a single global constant in STXXL, a programmer can simultaneously operate with several block types having different blocks sizes. Such flexibility is often required for good performance. For example, B^+-tree leaves might have a size different from the size of the internal nodes. We have made the block size a template parameter and not a member variable for the sake of efficiency. The values of the template parameters are known to the compiler, therefore for the power of two values (a very common choice) it can replace many arithmetic operations, like divisions and multiplications, by more efficient *binary shifts*. A critical requirement for many external memory data structures is that a block must be able to store links to other blocks. An STXXL block can store NRef objects of type `BID`. Additionally, one can equip a block with a field of the type `InfoType`, that can hold some per-block information.

In Listing 2, we give an example of how to program block I/O using objects of the BM layer. In Line 2 we define the type of block: Its size is one megabyte and the type of elements is `double`. The pointer to the only instance of the singleton object `block_manager` is obtained in Line 5. Line 6 asks the block manager to allocate 32 blocks in external memory. The `new_blocks` call writes the allocated BIDs to the output iterator, given by the last parameter. The `std::back_inserter` iterator adapter will insert the output BIDs at the end of the array `bids`. The manager assigns blocks to disks in a round-robin fashion as the `striping()` strategy suggests. Line 7 allocates 32 internal memory blocks. The internal memory allocator `new_alloc<block_type>` of STXXL allocates blocks on a virtual memory page boundary, which is a requirement for unbuffered file access. Along lines 8–10 the elements of blocks are filled with some values. Then, the blocks are submitted for writing (lines 11–12). The request objects are stored in an `std::vector` for the further status tracking. As in the AIO example, I/O is overlapped with computations in the function `do_something()`. After the completion of all write requests (Line 14) we perform some useful processing with the written data (function `do_something1()`). Finally we free the external memory space occupied by the 32 blocks (Line 16).

5. STL-USER LAYER

When we started to develop the library we decided to equip our implementations of external memory data structures and algorithms with well known generic interfaces of the Standard Template Library, which is a part of the C++ standard. This choice would shorten the application development times, since the time to learn new interfaces is saved. Porting an internal memory code that relies on STL would also be easy, since interfaces of STL-user layer data structures (containers in the STL terminology) and algorithms have the same syntax and semantics.

We go over the containers currently available in STXXL.

Listing 2. Example of how to program using the BM layer.

```
1  using namespace stxxl;
2  typedef typed_block<1024*1024,double> block_type;
3  std::vector<block_type::bid_type> bids;//empty array of BIDs
4  std::vector<request_ptr> requests;
5  block_manager * bm = block_manager::get_instance ();
6  bm->new_blocks<block_type>(32,striping(), std::back_inserter(bids));
7  std::vector<block_type,new_alloc<block_type> > blocks(32);
8  for (int ii = 0; ii < 32; ii++)
9      for (int jj=0; jj < block_type::size ; jj++)
10         blocks[ii][jj] = some_value(ii,jj);
11 for (int i = 0; i < 32; i++)
12     requests.push_back( blocks[i].write(bids[i]) );
13 do_something(); // do_something() is overlapped with writing
14 wait_all(requests.begin(), requests.end()); //wait until all I/Os finish
15 do_something1(bids.begin(),bids.end());
16 bm->delete_blocks(bids.begin(), bids.end()); // deallocate external memory
```

5.1. Vector

STL vector is an array, supporting random access to elements, constant time insertion and removal of elements at the end. The size of a vector may vary dynamically. The implementation of `stxxl::vector` is similar to the LEDA-SM array [27]. The content of a vector is striped block-wise over the disks, using an assignment strategy given as a template parameter. Some of the blocks are cached in a vector cache of fixed size (also a parameter). The replacement of cache blocks is controlled by a specified page-replacement strategy. STXXL has implementations of LRU and random replacement strategies. The user can provide his/her own strategy as well. The STXXL `vector` has STL-compatible Random Access Iterators. One random access costs $\mathcal{O}(1)$ I/Os in the worst case. Sequential scanning of the vector costs $\mathcal{O}(1/DB)$ amortized I/Os per vector element.

5.2. Stack

A stack is a LIFO data structure that provides insertion, removal, and inspection of the element at the top of the stack. Due to the restricted set of operations a stack can be implemented I/O-efficiently and applied in many external memory algorithms (e.g. [51, 52]). Four implementations of a stack are available in STXXL, which are optimized for different access patterns (long or short random insert/remove sequences) and manage their memory space differently (own or shared block pools). Some of the implementations (e.g. `stxxl::grow_shrink_stack2`) are optimized to prefetch data ahead and to queue writing, efficiently overlapping I/O and computation. The amortized I/O complexity for `push` and `pop` stack operations is $\mathcal{O}(1/DB)$.

We compare the performance of STXXL stack with performance of LEDA-SM and TPIE stacks in a simple test: we insert records to the stacks and afterwards delete them all. We try 4- and 32-byte records to evaluate different CPU processing overheads. For the experiments we used a 3 GHz Pentium 4 processor, 1 GB of main memory and a SATA disk dedicated to the experiments. The measured maximum I/O bandwidth of the hard disk was 72 MB/s for writing and 65 MByte/s for reading. Before

the insertion, we allocated a 768 MByte array and filled it with zeros to prevent this memory to be used for file system buffering, which would distort the measurements. This also simulates the memory consumption of other algorithms and data structures used in a real application. The rest of the memory was used for buffer blocks for stacks in the libraries and for operating system buffers in the case of LEDA-SM and TPIE. STXXL has used its own buffer and prefetching mechanism. The block size was set to two MBytes. Preliminary experiments have shown that larger blocks did not help any of the libraries. STXXL and TPIE implementations have been compiled with `g++ 3.3`. The compiler optimization level was set to `-O3` for both codes.

Tables III and IV show the bandwidth achieved in the experiments, which was computed as $n \cdot$ `sizeof(T)`$/t$, where n is the number of elements to insert/delete, `T` is the data type and t is the time to insert/delete all n elements. The experiments were conducted on input volumes of 1-8 GBytes. Since we have only registered insignificant variations, the average bandwidths are presented. We considered the following stack implementations:

- STXXL `grow_shrink_stack2` (GS2) using block pools for asynchronous prefetching and buffering. The name of this stack stems from the access pattern it supports with the best efficiency: An empty stack is first filled with elements up to the maximum number of elements and then all elements are removed one after another. In the first step the implementation is operating in the "grow" mode, using a pool of blocks for buffering the incoming elements. In the second step it is operating in the "shrink" mode, reading ahead some number of blocks (user-defined value) using a prefetch block pool. This type of access (or similar) is frequently used in applications, see e.g. [52]. `grow_shrink_stack2` and `grow_shrink_stack1` implementations differ in the usage mode of the block pools: the former uses pools, shared between (possibly) several STXXL data structures, the latter has an exclusive access to its own pools.
- STXXL `normal_stack` is a classical implementation with synchronous reading and writing,
- LEDA-SM stack implementation,
- TPIE stack implementation using the `mmap` function for disk access,
- TPIE stack implementation using standard Unix calls for disk access.

TPIE stack operations on 4-byte elements are CPU-bound as seen in Table III. STXXL stacks have the best bandwidths and achieve 57 MB/s even for this small record size. GS2 stacks perform better than STXXL normal stacks because of the better overlapping of I/O and computation. The LEDA-SM stack performs significantly better than the TPIE stack as it is probably less CPU-bound, but still its performance is worse than the performance of the STXXL stacks, since it does not use direct I/O and relies on system buffering which incurs superfluous data block copying.

Experiments with a larger record size (32 bytes) decrease the per-record CPU overhead. This helps TPIE to improve the bandwidth, which again indicates that the TPIE stack is highly CPU-bound. Note that for our system the `mmap` access method was not the best for TPIE implementations, but the `ufs` access method that is similar to the STXXL `syscall`. The TPIE stack achieves the performance of normal STXXL for inserting. STXXL writes records at 67 MB/s which is 93 % of maximum disk bandwidth.

Table IV shows the results of the above described tests for GS2 stacks on the SATAOpteron system (Table V) with one GByte of main memory. The measured maximum raw single-disk bandwidths for

Table III. Bandwidth for stack operations with a single disk.

	STXXL GS2	STXXL normal	LEDA-SM	TPIE `mmap`	TPIE `ufs`
insert 4-byte el.	57	50	39	3.5	7.3
delete 4-byte el.	51	50	34	3.5	6.9
insert 32-byte el.	67	60	60	24	37
delete 32-byte el.	52	51	40	27	29

Table IV. Bandwidth for stack operations with multiple disks.

	$D = 1$	$D = 2$	$D = 4$
insert 4-byte elements	64	130	260
delete 4-byte elements	46	91	168

these disks are 79 and 59 MByte/s for writing and reading respectively. We see that almost perfect speedups could be obtained. Only for the deletions running on four disks the speedup was about 3.7.

The stack benchmarks underline the advantages of the STXXL library:

- low CPU overhead (use low-level optimizations like copying operands in CPU registers, unroll loops, etc.),
- use of direct I/O to avoid unneeded data copying (i.e. use `syscall_file` with `O_DIRECT` flags),
- use of own prefetching/buffering mechanisms for overlapping I/O and computation (`buf_ostream` and `buf_istream` classes from the BM layer),
- support of parallel disks.

The source code of all stack tests described above is distributed with the STXXL library.

5.3. Queue and deque

The design STXXL FIFO `queue` of is similar to `stxxl::grow_shrink_stack2`. The implementation holds the head and the tail blocks in the main memory. Prefetch and write block pools are used to overlap I/O and computation during `queue` operations.

The STXXL implementation of external memory `deque` is an adaptor of an (external memory) vector. This implementation wraps the elements around the end of the vector *circularly*. It provides the pop/push operations from/to both ends of the `deque` in $\mathcal{O}(1/DB)$ amortized I/Os if parameterized with a properly configured `stxxl::vector`.

5.4. Priority queue

External memory priority queues are the central data structures for many I/O efficient graph algorithms [53, 51, 15]. The main technique in these algorithms is time-forward processing [51, 54], easily realizable by an I/O efficient priority queue. This approach evaluates a DAG with labeled nodes. The

Table V. Specifications of our experimental computers.

Code name	Xeon	SCSIOpteron	SATAOpteron
Processor	2×Xeon 2.0GHz	4×Opteron 1.8GHz	Dual-Core Opteron 2.0GHz
Main memory	1 GByte	8 GBytes	4 GBytes
Exp. disks	8	10	4
Disk interface	PATA	SCSI	SATA
Disk manufacturer	IBM	Seagate	Seagate
Disk RPM	7200	15000	7200
Single disk capacity	80 GBytes	70 GBytes	250 GBytes
Measured max. bandwidth of a disk	48 MByte/s	75 MByte/s	79 MByte/s
Total max. bandwidth achieved	375 MByte/s	640 MByte/s	214 MB/s
Approx. price (year)	3000 EURO (2002)	15000 EURO (2005)	3500 EURO (2006)

output of the processing is another node labelling: The output label of node v is computed from its input label and the messages received from the incoming edges. After computing the output label, node v sends messages along its outgoing edges. I/O efficient priority queues also find application in large-scale discrete event simulation and online sorting. The STXXL implementation of priority queues is based on [55]. Every operation of this priority queue, called sequence heap, takes $\mathcal{O}\left(\frac{1}{B}\log_{M/B}(I/B)\right)$ amortized I/Os, where I is the total number of insertions into the priority queue. This queue needs less than a third of I/Os used by other similar cache (I/O) efficient priority queues (e.g. [22, 56]).

A sequence heap, shown in Figure 3, maintains R *merge groups* $G_1,\ldots, G_R$ where G_i holds up to k sorted sequences of size up to mk^{i-1}, $m << M$. When group G_i overflows, all its sequences are merged, and the resulting sequence is put into group G_{i+1}. Each group is equipped with a *group buffer* of size m to allow batched deletion from the sequences. The smallest elements of these buffers are deleted in small batches and stored in the *deletion buffer*. The elements are first inserted into the *insertion priority queue*. On deletion, one checks the minimum elements stored in the insertion priority queue and the deletion buffer.

The difference between our implementation and [55] is that a number of larger merge groups are explicitly kept in external memory. The sorted sequences in those groups only hold their *first* blocks in the main memory. The implementation supports parallel disks and overlaps I/O and computation. As in [55], the internal merging is based on loser trees [57]. However, our implementation does not use *sentinel* elements.

In the following we compare the performance of the STXXL priority queue with the general-purpose array heap implementation of LEDA-SM [22]. TPIE does not provide an I/O-efficient priority queue in the distributed library version. We run the implementation on synthetic inputs following [22]. The

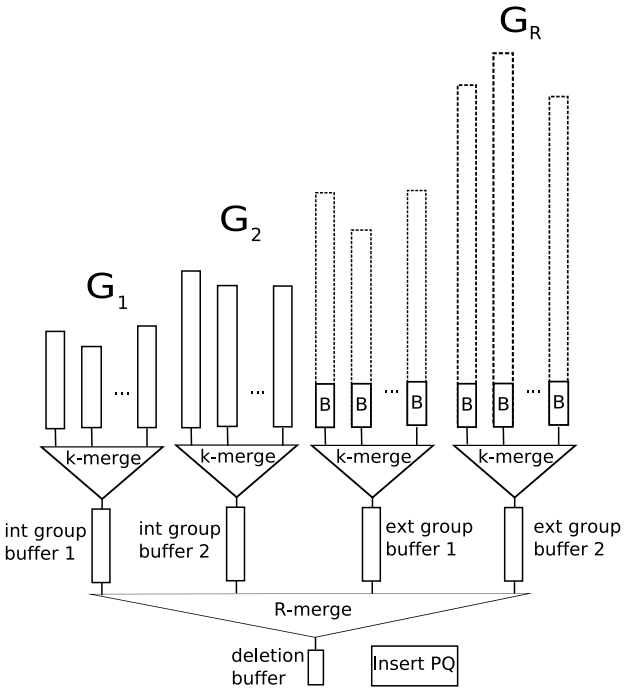

Figure 3. The structure of STXXL priority queue.

comparison of the data structures in a real graph algorithm is presented in Section 9, where we implement an I/O-efficient maximal independent set algorithm.

The first test performs n insertions, followed by n delete-min operations. The elements inserted are pairs of 4-byte integers where the key component is drawn randomly from the range $[0, 2^{31})$. Figure 4 shows the running time of this experiment running on a system with a 3.00 GHz Pentium 4 processor, 1 GB of main memory and a SATA disk dedicated to the experiments. The priority queues were given 512 MB of main memory at their disposal, the rest was used by the Linux operating system for buffering in the case of LEDA-SM and for prefetch and write pools in the case of STXXL. The STXXL implementation was compiled with `g++ 3.3`. The compiler optimization level was set to `-O3`. We have tuned the external memory block size (256 KB) for LEDA-SM to obtain the best running times for the array heap. However, the differences in running times with different block sizes were negligible, which is a symptom of its CPU-boundedness. The STXXL priority queue used 2 MByte blocks. Note the drop of the LEDA-SM delete curve after $n = 2^{23}$ is not an artifact of the measurements; it has been also reported in the original study [22]. However, we do not devote much attention to the results with $n \leq 2^{24}$ since those inputs fit in the internal memory. LEDA-SM containers cannot hold more than $2^{31} - 1$ items, therefore we have stopped at input size $n = 2000 \cdot 2^{20}$, which corresponds to about 16 GBytes. This input is 32 times larger than the main memory size and it is reasonable to be handled

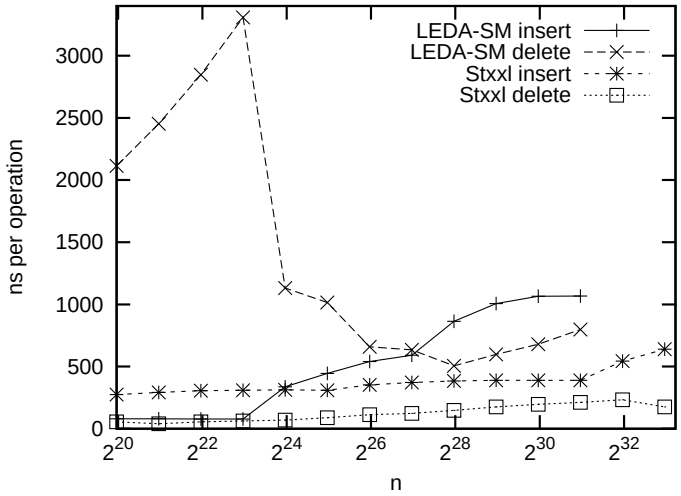

Figure 4. The running time of the insert-all-delete-all test for priority queues.

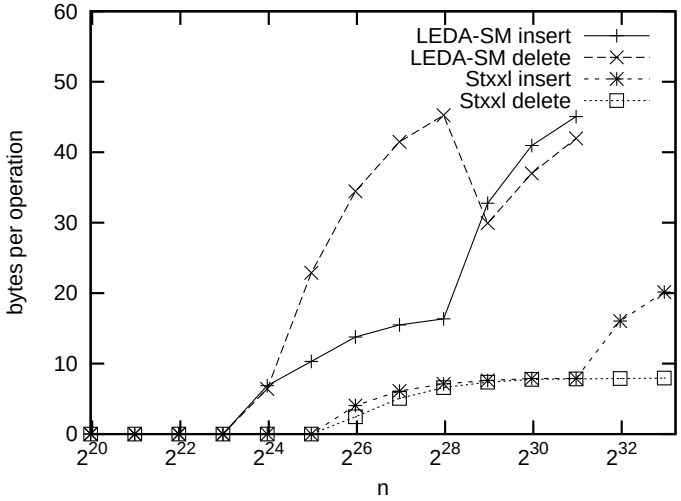

Figure 5. The I/O volume of the insert-all-delete-all test for priority queues.

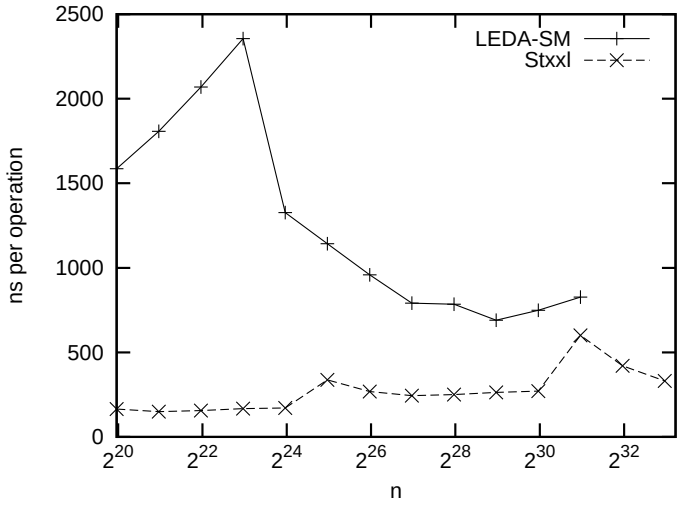

Figure 6. The running times of intermixed insertions with deletions (priority queues).

in external memory. The STXXL priority queue is up to 2.7 times faster for insertions and 3.7 times faster for deletions. This can be explained by more expensive CPU work taking place in the LEDA-SM implementation, its larger I/O volume and also better *explicit* overlapping of I/O and computation of STXXL. Note that LEDA-SM relies on (general purpose) operating system caching and buffering. LEDA-SM uses plain binary heaps for merging internal memory data; the STXXL priority queue reduces the internal memory processing overhead applying techniques from [55] (loop-unrolling, 2/4-way register mergers, loser trees with less comparisons). The insertion and deletion phases for the STXXL priority queue need almost the same number of I/Os for $n \leq 2^{31}$. The insertion phase has also the advantage that writing is faster than reading, nonetheless it is almost two times slower than the deletion phase. This is explained by the higher CPU work needed for merging and filling the buffers during insertions. The insertion phase is highly CPU-bound which is confirmed by the I/O-wait time counter, whose value was close to zero. According to the I/O-wait time counter, the deletion phase is less CPU-bound. For $n \geq 2^{32}$ the insertion phase needs to merge and insert *external* sequences, which implies more I/O operations and results in the observed running time escalation. This is confirmed by Figure 5, which also shows that the I/O volume of STXXL priority queue is 2–5.5 times smaller than the I/O volume of the LEDA-SM array heap. This difference was predicted in the original paper [55].

Figure 6 presents the running times of another synthetic test: we insert n random elements into the priority queues and then measure the running time of n operations: We insert a random pair with probability $\frac{1}{3}$ and delete the minimum with probability $\frac{2}{3}$. The behavior of the LEDA-SM curve is similar to the deletions. The STXXL curve has two steps: The first step occurs when the internal sequences have to be merged and put into the external memory for the first time ($n = 2^{25}$), the next step happens at $n = 2^{31}$ when sequences from the first external merger have to be merged and put into

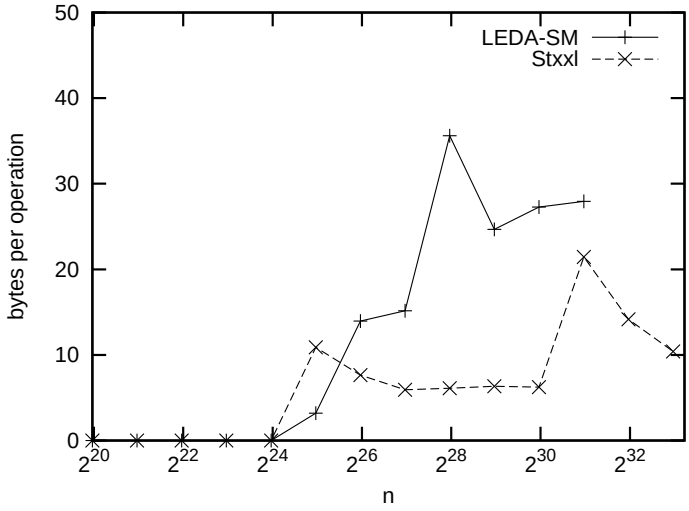

Figure 7. The I/O volume of intermixed insertions with deletions (priority queues).

a sequence of the second external merger for the first time (see Figure 7). These steps are hardly seen in Figure 4 (insertions) because of the different access pattern and amortization (n versus $\frac{4n}{3}$ insertions).

The operations on the STXXL priority queue did not scale using more hard disks. For four disks we have observed speedups at most 1.4. We expect a better speedup for element types that incur larger I/O volume and less CPU work, e.g. when the satellite data field is large and the comparison function is simple.

5.5. Map

The `map` is an STL interface for search trees with unique keys. Our implementation of `map` is a variant of a B$^+$-tree data structure [58] supporting the operations `insert`, `erase`, `find`, `lower_bound` and `upper_bound` in optimal $\mathcal{O}\big(\log_B(n)\big)$ I/Os. Operations of `map` use *iterators* to refer to the elements stored in the container, e.g. `find` and `insert` return an iterator pointing to the data. Iterators are used for range queries: an iterator pointing to the smallest element in the range is returned by `lower_bound`, the element which is next to the maximum element in the range is returned by `upper_bound`. Scanning through the elements of the query can be done by using `operator++` or `operator--` of the obtained iterators in $\mathcal{O}(R/B)$ I/Os, where R is the number of elements in the result. Our current implementation does not exploit disk parallelism. The flexibility of the iterator-based access has some complications for an external memory implementation: iterators must return correct data after reorganizations in the data structure even when the pointed data is moved to a different external memory block.

The way iterators are used for accessing a `map` is similar to the use of database *cursors* [59]. STXXL is the first C++ template library that provides an *I/O-efficient* search tree implementation with iterator-based access.

Our design of B$^+$-tree permits different implementations for leaves and internal nodes to be used. For example, leaves could be represented internally as sparse arrays [10]. Currently, only the classic sorted array implementation is available. For the sake of flexibility and tuning, leaves and nodes can have different external memory block sizes. Each leaf has links to the predecessor and successor leaves to speed up scanning. Our B$^+$-tree is able to prefetch the neighbor leaves when scanning, obtaining a higher bandwidth by overlapping I/O and computation. The root node is always kept in the main memory and implemented using the `std::map`. To save I/Os, the most frequently used nodes and leaves are cached in corresponding node and leaf *caches* that are implemented in a single template class. An iterator keeps the BID of the external block where the pointed data element is contained, the offset of the data element in the block and a pointer to the B$^+$-tree. In case of reorganizations of the data in external memory blocks (rebalancing, splitting, fusing blocks), all iterators pointing to the moved data have to be updated. For this purpose, the addresses of all instantiated iterators are kept in the *iterator map* object. The iterator map facilitates fast accounting of iterators, mapping BID and block offsets of iterators to its main memory addresses using an *internal* memory search tree. Therefore, the number of "alive" B$^+$-tree iterators must be kept reasonably small. The parent pointers in leaves and nodes can be useful for finger search[‡‡] and insertions using a finger, however, that would require to store the whole B-tree path in the iterator data structure. This might make the iterator accounting very slow, therefore we do not maintain the parent links. The implementation can save I/Os when `const_iterator`s are used: no flushing of supposedly changed data is needed (e. g. when scanning or doing other read-only operations). Our implementation of B$^+$-tree supports bulk bottom-up construction from the presorted data given by an iterator range in $\mathcal{O}(n/B)$ I/Os.

We have compared the performance of our B$^+$-tree implementation with the performance of the Berkeley DB B$^+$-tree (BDB) implementation version 4.4 [34], which is a commercial product. This implementation is known to be one of the fastest implementations available. For the experiments we have configured Berkeley DB with disabled transactions and concurrency support. We also measured the performance of the TPIE B$^+$-tree implementation, but not the LEDA-SM implementation because it does not support predecessor/successor and range queries. For the experiments we used the SATAOpteron machine (Table V) using one GByte of main memory. Each implementation used a separate hard disk for storing its external data. STXXL map and TPIE have used a cache of 750 MBytes and BDB's cache was slightly less because it has used more memory than the given cache size [*]. The block size was set to 32 KBytes for all implementations.

The B$^+$-trees indexed records with eight character random keys (letters 'a'-'z') and 32 bytes data field. First, B$^+$-trees have been constructed from a sorted set of n records. To make the comparison fair we configured BDB to store records with *unique* keys, since the `map` interface does keep multiple records with equal keys. TPIE B$^+$-tree supports the bulk construction from a pre-sorted TPIE stream

[‡‡] The program can help the search tree to find an element by giving some "position close by" which has been determined by an earlier search.

[*] The BDB process has been killed by an out-of-memory exception when run with 750 MByte cache. Therefore we had to reduce the cache size.

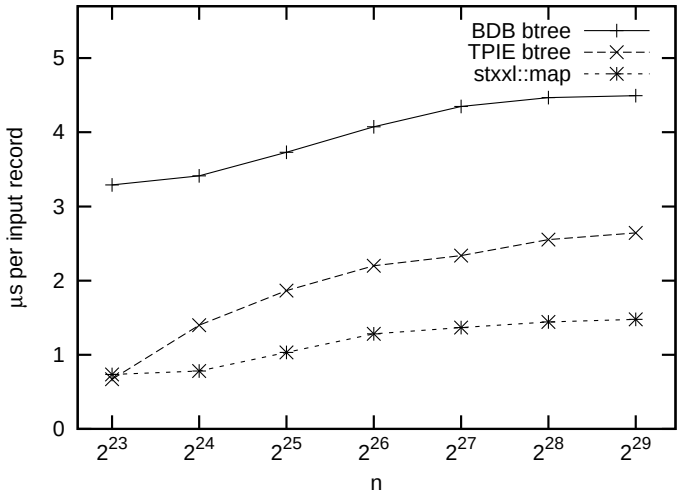

Figure 8. The running times of B^+-tree construction.

of data which must reside on a hard disk (`AMI_STREAM`). According to the Berkeley DB support team, BDB lacks the bulk load capability, therefore as suggested we inserted the records one by one in ascending order for the best performance. This insertion pattern leads to nearly 100 % cache hits and produces a BDB B^+-tree with a fill factor of about 100 % in the leaves. The `stxxl::map` and the TPIE bulk construction were configured to achieve the 100 % fill factor, too. Figure 8 shows the construction time without the pre-sorting. In this test, STXXL map is about three times faster than BDB. The obvious reason for this is that BDB has to do many searches in the leaves and nodes to find an appropriate place to insert, and thus is highly CPU-bound. STXXL map bulk construction performs only a small constant number of operations per input element. The TPIE bulk construction was up to 70 % slower than the construction of the `stxxl::map`, because, in fact, it repeatedly inserts all input elements into the leaves doing a *binary search* of the place to insert them into the last leaf over and over from the scratch. This inefficiency makes the construction more CPU-bound.

After the construction of the base element set index we generated 100,000 random records and inserted them. The running times of this experiment are shown in Figure 9. For large inputs one has to load up to two leaves and flush up to three leaves per insertion since the leaves are full after the construction and they need to be split. Note that the links between the leaves must be updated, too. This can explain the long 25 ms of the STXXL and TPIE B^+-trees since a random access to this hard disk takes up to 8 ms on average according to its specifications paper. BDB is about 30% faster for large inputs; this advantage could to some extent be due to the adaptive compression of sequences of keys with the same prefix, an optimization exploited in the recent versions of BDB. Other reasons could be the highly tuned node/leaf splitting strategy of BDB [34] or better exploitation of seek time optimizations for write accesses.

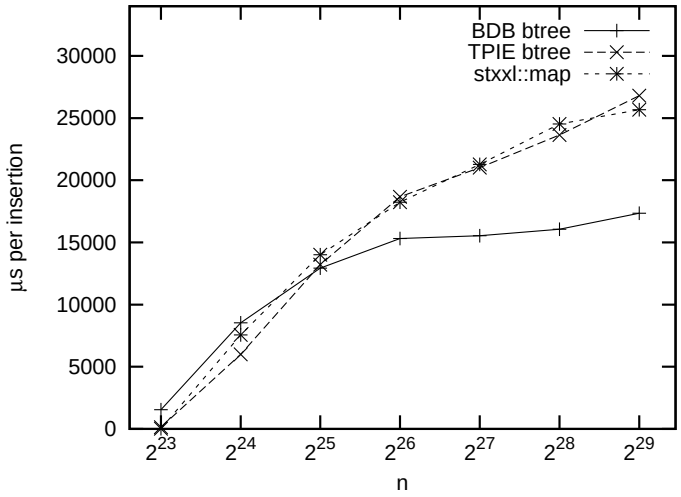

Figure 9. The running times of B^+-tree insertions.

The next test (Figure 10) is run after the insertions and performs 100,000 random *locate* queries of the smallest record that is not smaller than the given key. For large inputs, in almost every locate query, a random leaf has to be loaded. This is the reason of the observed latency of 10–13 ms. BDB is faster again, but the advantage is smaller here (below 20 %). The STXXL B^+-tree is slightly faster than the TPIE B^+-tree.

Figure 11 (left) shows the running time of random range queries with scanning. We sample the possible ranges uniformly at random and scan the obtained ranges until about n records are scanned after all queries. This way only 3–5 queries suffice. In order to explain the trends in the running times we have drawn Figure 11 (right) that shows the leaf space overuse factor computed as $\frac{\text{number of leaves}\cdot B}{(n+a)\cdot\text{size of record}}$ before this test, where $B = 32$ KBytes is the block size and $a = 100,000$ is the number of the additional records inserted. This metric shows how inefficiently the leaves are filled with data. After the bulk construction the overuse factor is very close to 1 for the STXXL and BDB implementations. We could not generate such statistic for the TPIE B^+-tree since it does not offer the required counters. Adding the 100,000 random elements to small inputs worsens the overuse factor: For `stxxl::map`, it almost reaches 2, and for BDB it is larger than 2.5. This difference plays a big role in the ratio of the records that can be kept in the memory for $n = 2^{23}$. STXXL can keep almost the whole data set, and thus the scanning time is very small. For larger inputs $n \geq 2^{24}$ only a small fraction of the input can reside in the main memory for the implementations. This increases the scanning time significantly, because many leaves have to be loaded and flushed from/to the hard disk. However, the overuse factor of STXXL is still considerably smaller for $n = 2^{24}$. This contributes to a speedup about 2 over the BDB. In the middle region, all data structures are equally fast and the overuse factor is about the same. For the largest inputs, the STXXL map has again a small advantage of about 20 % in scanning speed due

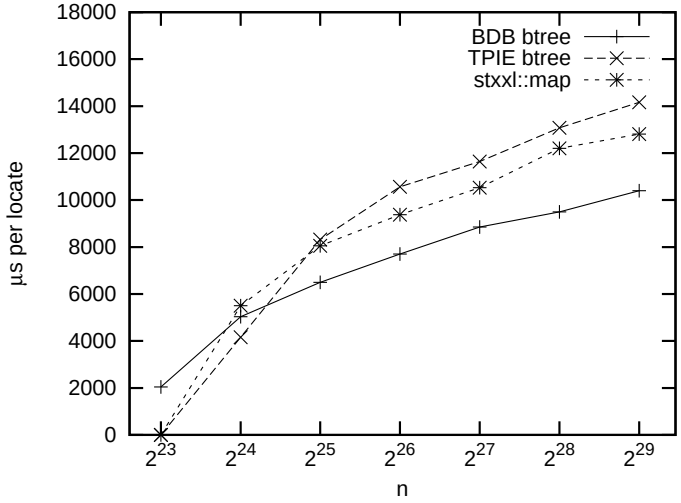

Figure 10. The running times of B^+-tree locates.

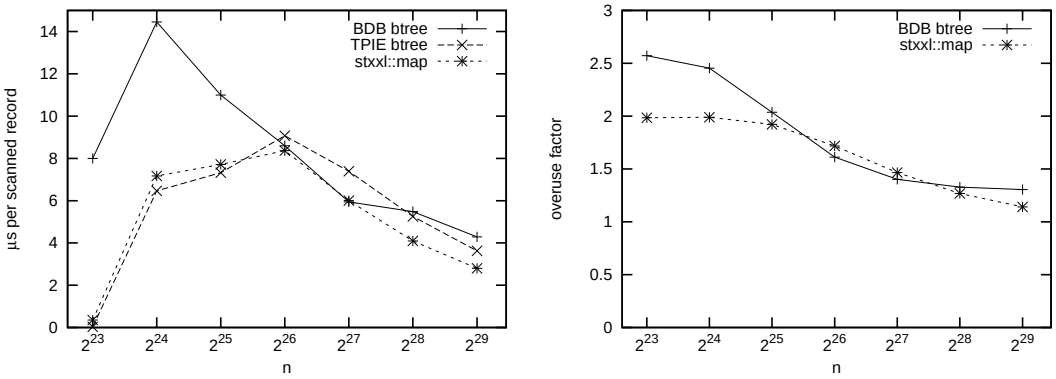

Figure 11. The running times of B^+-tree range queries with scanning (left). The space overuse factor in leaves of B^+-tree (right).

to the better leaf fill factor and due to the better overlapping of I/O and computation. The latter might also be the reason of the small advantage of the STXXL B^+-tree over the TPIE B^+-tree for $n \geq 2^{26}$.

After the tests described above we delete the 100,000 records inserted after the bulk construction from the B^+-trees. Here, we need to load and store one leaf for almost every delete operation. Fusing and rebalancing of leaves and nodes should not occur frequently. Again, the BDB is faster by about

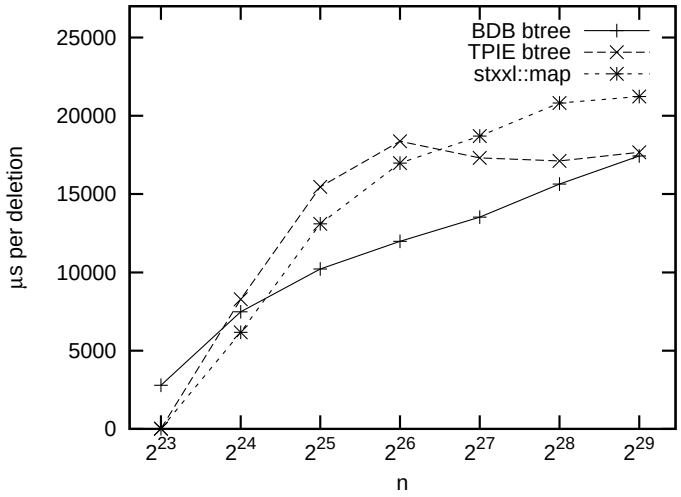

Figure 12. The running times of B^+-tree deletions.

18 % for large inputs (Figure 12). For $n \geq 2^{26}$ the delete time of the TPIE B^+-tree goes *down* and approaches the time of the BDB B^+-tree.

The source code of all B^+-tree tests described above is distributed with the STXXL library.

Discussion: The tests have shown that the STXXL map is somewhat slower than the BDB in insert, locate and delete operations. These operations have been highly tuned in commercial BDB implementations over the last 15 years of development [34]. However, an advantage of STXXL is that it can do fast bulk construction, which is not available in BDB. The speed of scanning of the records in a range is competitive with the BDB. STXXL is also arguably simpler to use and offers stronger typing. For example, compare Listings 3 and 4. The complete code of the STXXL map test is about two times shorter than the equivalent BDB test code.

5.6. General issues concerning STXXL containers

Similar to other external memory algorithm libraries [27, 28], STXXL has the restriction that the data types stored in the containers cannot have C/C++ pointers or references to other elements of external memory containers. The reason is that these pointers and references get invalidated when the blocks containing the elements they point/refer to are written to disk. To get around this problem, the links can be kept in the form of external memory iterators (e.g. `stxxl::vector::iterator`). The iterators remain valid while storing to and loading from the external memory. When dereferencing an external memory iterator, the referenced object is loaded from external memory by the library on demand (if the object is not in the cache of the data structure already).

Listing 3. Locates with the Berkeley DB.

```
1  struct my_key { char keybuf[KEY_SIZE]; };
2  struct my_data { char databuf[DATA_SIZE]; };
3
4  Dbc *cursorp; // data base cursor
5  // db is the BDB B-tree object
6  db.cursor(NULL, &cursorp, 0); // initialize cursor
7
8  for (int64 i = 0; i < n_locates; ++i)
9  {
10     rand_key(key_storage); // generate random key
11     // initialize BDB key object for storing the result key
12     Dbt keyx(key_storage.keybuf,KEY_SIZE);
13     // initialize BDB key object for storing the result data
14     Dbt datax(data_storage.databuf,DATA_SIZE);
15     cursorp->get(&keyx, &datax,DB_SET_RANGE); // perform locate
16 }
```

Listing 4. Locates with the STXXL map.

```
1  struct my_key { char keybuf[KEY_SIZE]; };
2  struct my_data { char databuf[DATA_SIZE]; };
3
4  std::pair<my_key,my_data> element; // key-data pair
5
6  for (i = 0; i < n_locates; ++i)
7  {
8      rand_key(i,element.first); // generate random key
9
10     // perform locate, CMap is a constant reference to a map object
11     map_type::const_iterator result = CMap.lower_bound(element.first);
12 }
```

STXXL containers differ from STL containers in treating allocation and distinguishing between uninitialized and initialized memory. STXXL containers assume that the data types they store are plain old data types (POD) [†]. The constructors and destructors of the contained data types are not called when a container changes its size. The support of constructors and destructors would imply a significant I/O cost penalty, e.g. on the deallocation of a non-empty container, one has to load all contained objects and call their destructors. This restriction sounds more severe than it is, since external memory data structures cannot cope with custom dynamic memory management anyway, which is the common use of custom constructors/destructors.

[†]Plain old data structures (PODs) are data structures represented only as passive collections of field values, without using aggregation of variable-size objects (as references or pointers), polymorphism, virtual calls or other object-oriented features. Most naturally they are represented as `C/C++ structs`.

Listing 5. Definitions of classes.

```
1  struct edge { // edge class
2      int src,dst; // nodes
3      edge() {}
4      edge(int src_, int dst_): src(src_), dst(dst_) {}
5      bool operator == (const edge & b) const {
6          return src == b.src && dst == b.dst;
7      }
8  };
9  struct random_edge { // random edge generator functor
10     edge operator () () const {
11         edge Edge(random()-1,random()-1);
12         while(Edge.dst == Edge.src) Edge.dst = random() - 1 ; //no self-loops
13         return Edge;
14     }
15 };
16 struct edge_cmp { // edge comparison functor
17     edge min_value() const { return edge(std::numeric_limits<int>::min(),0); };
18     edge max_value() const { return edge(std::numeric_limits<int>::max(),0); };
19     bool operator () (const edge & a, const edge & b) const {
20         return a.src < b.src || (a.src == b.src && a.dst < b.dst);
21     }
22 };
```

5.7. Algorithms

The algorithms of the STL can be divided into two groups by their memory access pattern: *scanning* algorithms and *random access* algorithms.

5.7.1. Scanning algorithms

Scanning algorithms work with Input, Output, Forward, and Bidirectional iterators only. Since random access operations are not allowed with these kinds of iterators, the algorithms inherently exhibit a strong spatial locality of reference. STXXL containers and their iterators are STL-compatible, therefore one can directly apply STL scanning algorithms to them, and they will run I/O-efficiently (see the use of `std::generate` and `std::unique` algorithms in the Listing 6). Scanning algorithms are the majority of the STL algorithms (62 out of 71). STXXL also offers specialized implementations of some scanning algorithms (`stxxl::for_each`, `stxxl::generate`, etc.), which perform better in terms of constant factors in the I/O volume and internal CPU work. These implementations benefit from accessing lower level interfaces of the BM layer instead of using iterator interfaces, resulting in a smaller CPU overhead. Being aware of the sequential access pattern of the applied algorithm, the STXXL implementations can do prefetching and use queued writing, thereby leading to the overlapping of I/O with computation.

Listing 6. Generating a random graph using the STL-user layer.

```
1 stxxl::vector<edge> ExtEdgeVec(10000000000ULL);
2 std::generate(ExtEdgeVec.begin(),ExtEdgeVec.end(), random_edge());
3 stxxl::sort(ExtEdgeVec.begin(),ExtEdgeVec.end(),edge_cmp(),512*1024*1024);
4 stxxl::vector<edge>::iterator NewEnd =
5             std::unique(ExtEdgeVec.begin(),ExtEdgeVec.end());
6 ExtEdgeVec.resize(NewEnd - ExtEdgeVec.begin());
```

5.7.2. *Random access algorithms*

Random access algorithms require random access iterators, hence may perform (many) random I/Os. Some of these algorithms are I/O efficient anyway, e.g., `std::nth_element` needs only $\mathcal{O}(\text{scan}(n))$ I/Os on average. Some algorithms, like binary search and binary heap functions are not really needed anymore because more efficient external data structures like B-trees and priority queues can usually replace them. What remains are sorting and random shuffling which we both implement. Currently, the library provides two implementations of sorting: an `std::sort`-like sorting routine – `stxxl::sort`, and a sorter that exploits integer keys – `stxxl::ksort`. Both sorters are implementations of parallel disk algorithms described in Section 6.

Listing 6 shows how to program using the STL-user layer and how STXXL containers can be used together with both STXXL algorithms and STL algorithms. The definitions of the classes `edge`, `random_edge` and `edge_cmp` are in Listing 5. The purpose of our example is to generate a huge random directed graph in a sorted edge array representation, i. e. the edges in the edge array must be sorted lexicographically. A straightforward procedure to do this is to: 1) generate a sequence of random edges, 2) sort the sequence, 3) remove duplicate edges from it. If we ignore definitions of helper classes the STL/STXXL code of the algorithm implementation is only five lines long: Line 1 creates an STXXL external memory vector with 10 billion edges. Line 2 fills the vector with random edges (`generate` from the STL is used). In the next line the STXXL external memory sorter sorts randomly generated edges using 512 megabytes of internal memory. The lexicographical order is defined by functor `my_cmp`, `stxxl::sort` also requires the comparison functor to provide upper and lower bounds for the elements being sorted. Line 5 deletes duplicate edges in the external memory vector with the help of the STL `unique` algorithm. The `NewEnd` vector iterator points to the right boundary of the range without duplicates. Finally (in Line 6), we chop the vector at the `NewEnd` boundary. Now we count the number of I/Os performed by this example as we will need it later: external vector construction takes no I/Os; filling with random values requires a scan — N/DB I/Os; sorting will take $4N/DB$ I/Os; duplicate removal needs no more than $2N/DB$ I/Os; chopping a vector is I/O-free. The total number of I/Os is $7N/DB$.

6. PARALLEL DISK SORTING

Sorting is the first component we have designed for STXXL, because it is *the* fundamental tool for I/O-efficient processing of large data sets. Therefore, an efficient implementation of sorting largely defines the performance of an external memory software library as a whole. To achieve the best performance

our implementation [46] uses parallel disks, has an optimal I/O volume $\mathcal{O}(\frac{N}{DB}\log_{M/B}\frac{N}{B})$, and guarantees almost perfect overlap between I/O and computation.

No previous implementation has all these properties, which are needed for a good practical sorting. LEDA-SM [27] and TPIE [30] concentrate on single disk implementations. For the overlapping of I/O and computation they rely on prefetching and caching provided by the operating system, which is suboptimal since it does not use prefetching information.

Barve and Vitter implemented a parallel disk algorithm [49] that can be viewed as the immediate ancestor of our algorithm. Innovations with respect to our sorting are: a different allocation strategy that enables better theoretical I/O bounds [47, 60]; a prefetching algorithm that optimizes the number of I/O steps and never evicts data previously fetched; overlapping of I/O and computation; a completely asynchronous implementation that reacts flexibly to fluctuations in disk speeds; and an implementation that sorts many GBytes and does not have to limit internal memory size artificially to obtain a nontrivial number of runs. Additionally, our implementation is not a prototype, it has a generic interface and is a part of the software library STXXL.

Algorithms in [61, 62, 63] have the theoretical advantage of being deterministic. However, they need three passes over data even for relatively small inputs.

Prefetch buffers for disk load balancing and overlapping of I/O and computation have been intensively studied for external memory merge sort [64, 65, 66, 47, 60, 67]. But we have not seen results that guarantee overlapping of I/O and computation during the parallel disk merging of arbitrary runs.

There are many good practical implementations of sorting (e.g. [68, 69, 70, 71]) that address parallel disks, overlapping of I/O and computation, and have a low internal overhead. However, we are not aware of fast implementations that give good performance guarantees for all inputs. Most practical implementations use a form of striping that requires $\Omega(\frac{N}{DB}\log_{\Theta(\frac{M}{DB})}\frac{N}{B})$ I/Os rather than the optimal $\Theta(\frac{N}{DB}\log_{\Theta(M/B)}\frac{N}{B})$. This difference is usually considered insignificant for practical purposes. However, already on our Xeon experimental system (Table V) we have to go somewhat below the block sizes that give the best performance in Figure 18 if the input size is 128 GBytes. Another reduction of the block size by a factor of eight (we have eight disks) could increase the run time significantly. We are also not aware of high performance implementations that guarantee overlap of I/O and computation during merging for inputs such as the one described in Section 6.1.3.

On the other hand, many of the practical merits of our implementation are at least comparable with the best current implementations: We are close to the peak performance of our Xeon system.

The Sort Benchmark competition [72] is held yearly and includes several categories; some of them define restrictions on the cost of the hardware used for sorting. In the "Terabyte category", the goal is to sort quickly a terabyte of data. As this benchmark type is not limited by the hardware costs, distributed memory sorting algorithms win running on expensive clusters with SAN disks. Distributed memory sorters also lead in the "Minute category" which asks to sort as much data as possible in a minute. In the "Penny category" the cost of the hardware is spread over three years. Then, it is measured how much data can be sorted in an interval of time that costs one US-cent. Competition participants are responsible for the choice of their hardware. Each category has two subcategories: Daytona (for general-purpose sorting) and Indy (sort 100-byte records with 10-byte random keys).

The most interesting category for us is the "Penny category" because it addresses the cost-efficiency aspect of sorting, and it turns out that the cheapest way to sort is to use an I/O-efficient sorting. All

winners in this category since the announcement in 1998 were external memory sorters. We overview the past "Penny sort" winners [72]. The NTSort (won Indy in 1998) is a command line sorting utility of Windows NT implementing multi-way merge sort. The implementation adopted for the competition relied on direct unbuffered I/O, but used no overlapping between I/O and computation. PostmanSort (Daytona winner in 1998 and 2005) is a commercial sorting utility; it is a variant of the bucket sort. The recent version utilizes asynchronous I/O of Windows to overlap I/O and computation. HMSort (winner in 1999 and 2000) also exploits overlapping of I/O and computation, however, other details about the algorithm are not published. DMSort (Indy 2002) is based on the most-significant byte radix sort. The implementation works with two disks: The input data elements are read from the first disk and distributed to the bins on the second disk, explicitly overlapping I/O and computation. The second phase reads the bins from the second disk, sorts them, and writes the sorted result to the first disk in an overlapped fashion. Only few algorithmic details of THSort (Daytona 2004), Sheenksort (Indy 2004, 2005) and Byte-Split-Index Sort (Daytona 2006) are published. The authors of the THSort, SheenkSort and Byte-Split-Index have used systems with four disks. The GpuTeraSort (Indy 2006) [73] uses a graphic processing unit (GPU) for internal sorting, mapping a bitonic sorting network to GPU rasterization operations and using the GPUs programmable hardware and high bandwidth memory interface. The implementation accesses the files directly and explictly overlaps I/O and computation. To achieve a higher I/O-bandwidth, a RAID-0 has been used.

We have participated in the "Penny sort" competition in 2003 with an earlier variant of the implementation presented below. We took the second place, sorting 25 GBytes in our time budget, which is a two-fold improvement over the previous year. The "Penny sort" external sorters mentioned above are very impressive pieces of engineering. However, these algorithms and implementations do not give *worst case guarantees* for all inputs, including the overlapping of I/O and computation and the optimal use of parallel disks.

6.1. Multi-way merge sort with overlapped I/Os

Perhaps the most widely used external memory sorting algorithm is k-way merge sort: During *run formation*, chunks of $\Theta(M)$ elements are read, sorted internally, and written back to the disk as sorted *runs*. The runs are then merged into larger runs until only a single run is left. $k = \mathcal{O}(M/B)$ runs can be sorted in a single pass by keeping up to B of the smallest elements of each run in internal memory. Using randomization, prediction of the order in which blocks are accessed, a prefetch buffer of $\mathcal{O}(D)$ blocks, and an optimal prefetching strategy, it is possible to implement k-way merging using D disks in a load balanced way [47]. However, the rate at which new blocks are requested is more difficult to predict so that this algorithm does not guarantee overlapping of I/O and computation. In this section, we derive a parallel disk algorithm that compensates these fluctuations in the block request rate by a FIFO buffer of $k + \Theta(D)$ blocks.

6.1.1. Run formation

There are many ways to overlap I/O and run formation. We start with a very simple method that treats internal sorting as a black box and therefore can use the fastest available internal sorters. Two threads cooperate to build k runs of size $M/2$:

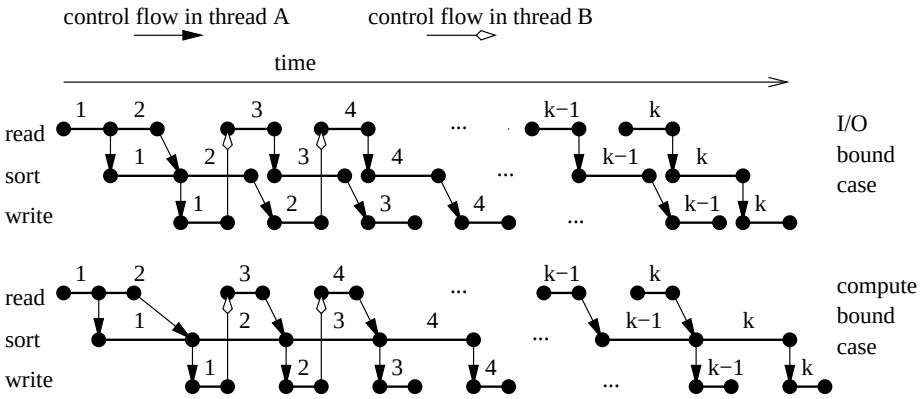

Figure 13. Overlapping I/O and computation during run formation.

```
post a read request for runs 1 and 2
thread A:                  | thread B:
for r:=1 to k do           | for r:=1 to k-2 do
  wait until               |   wait until
    run r is read          |     run r is written
  sort run r               |   post a read for run r+2
  post a write for run r   |
```

Figure 13 illustrates how I/O and computation is overlapped by this algorithm. Formalizing this figure, we can prove that using this approach an input of size N can be transformed into sorted runs of size $M/2 - \mathcal{O}(DB)$ in time $\max(2T_{\text{sort}}(M/2)N/M, \frac{2LN}{DB}) + \mathcal{O}\left(\frac{LM}{DB}\right)$, where $T_{\text{sort}}(x)$ denotes the time for sorting x elements internally and where L is the time needed for a parallel I/O step.

In [46] one can find an algorithm which generates longer runs of average length $2M$ and overlaps I/O and computation.

6.1.2. *Multi-way merging*

We want to merge k sorted sequences comprising N' elements stored in N'/B blocks. In practical situations, where a single merging phase suffices, we will have $N' = N$. In each iteration, the merging thread chooses the smallest remaining element from the k sequences and hands it over to the I/O thread. Prediction of read operations is based on the observation that the merging thread does not need to access a block until its smallest element becomes the smallest unread element. We therefore record the *smallest* keys of each block during run formation. By merging the resulting k sequences of smallest elements, we can produce a sequence σ of block identifiers that indicates the exact order in which blocks are logically read by the merging thread. The overhead for producing and storing the prediction data structure is negligible because its size is a factor at least B smaller than the input.

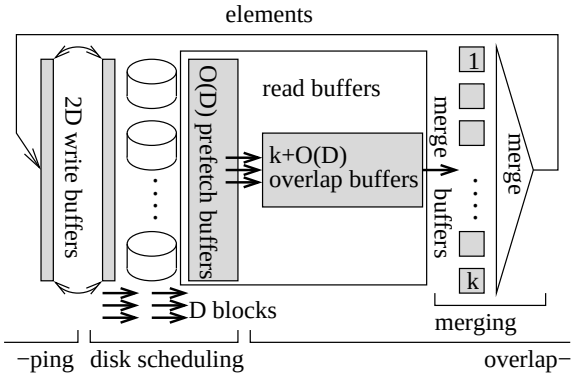

Figure 14. Data flow through the different kinds of buffers for overlapped parallel disk multi-way merging. Data is moved in units of blocks except between the merger and the write buffer.

The prediction sequence σ is used as follows. The merging thread maintains the invariant that it always buffers the k first blocks in σ that contain unselected elements, i.e., initially, the first k blocks from σ are read into these *merge buffers*. When the last element of a merge buffer block is selected, the now empty buffer frame is returned to the I/O thread and the next block in σ is read.

The keys of the smallest elements in each buffer block are kept in a tournament tree data structure [57] so that the currently smallest element can be selected in time $\mathcal{O}(\log k)$. Hence, the total internal work for merging is $\mathcal{O}\left(N' \log k\right)$.

We have now defined multi-way merging from the point of view of the sorting algorithm. Our approach to merging slightly deviates from previous approaches that keep track of the run numbers of the merge blocks and pre-assign each merge block to the corresponding input sequence. In these approaches also the *last* key in the *previous* block decides about the position of a block in σ. The correctness of our approach is shown in [46]. With respect to performance, both approaches should be similar. Our approach is somewhat simpler, however — note that the merging thread does not need to know anything about the k input runs and how they are allocated. Its only input is the prediction sequence σ. In a sense, we are merging individual blocks and the order in σ makes sure that the overall effect is that the input runs are merged. A conceptual advantage is that data *within* a block decides about when a block is read.

6.1.3. Overlapping I/O and merging

Although we can predict the order in which blocks are read, we cannot easily predict how much internal work is done between two reads. For example, consider k identical runs storing the sequence $\boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots$. After initializing the merge buffers, the merging thread will consume $k(B-1)$ values '1' before it posts another read. Then it will post one read after selecting each of the next k values (2). Then there will be a pause of another $k(B-1)$ steps and another k reads are following

each other quickly, etc. We explain how to overlap I/O and computation despite this irregularity using the I/O model of Aggarwal and Vitter [74] that allows access to D *arbitrary* blocks from D disks within one parallel I/O step. To model overlapping of I/O and computation, we assume that an I/O step takes time L and can be done in parallel with internal computations. We maintain an *overlap buffer* that stores up to $k+3D$ blocks in a FIFO manner (see Figure 14). Whenever the overlap buffer is non-empty, a read can be served from it without blocking. Writing is implemented using a *write buffer* FIFO with $2DB$ elements capacity. An *I/O thread* inputs or outputs D blocks in time L using the following strategy: Whenever no I/O is active and at least DB elements are present in the write buffer, an output step is started. When no I/O is active, less than D output blocks are available, and at least D overlap buffers are unused, then the next D blocks from σ are fetched into the overlap buffer. This strategy guarantees that merging k sorted sequences with a total of N' elements can be implemented to run in time $\max\left(\frac{2LN'}{DB}, \ell N'\right) + \mathcal{O}\left(L\left\lceil\frac{k}{D}\right\rceil\right)$ where ℓ is the time needed by the merging thread to produce one element of output and L is the time needed to input or output D arbitrary blocks [46].

6.1.4. Disk scheduling

The I/Os for the run formation and for the output of merging are perfectly balanced over all disks if all sequences are *striped* over the disks, i.e., sequences are stored in blocks of B elements each and the blocks numbered $i,\ldots,i+D-1$ in a sequence are stored on different disks for all i. In particular, the original input and the final output of sorting can use any kind of striping.

The merging algorithm presented above is optimal for the unrealistic model of Aggarwal and Vitter [74] which allows to access any D blocks in an I/O step. This facilitates good performance for fetching very irregularly placed input blocks. However, this model can be simulated using D independent disks using *randomized striping allocation* [50] and a prefetch buffer of size $m=\Theta(D)$ blocks: In almost every input step, $(1-\mathcal{O}(D/m))D$ blocks from prefetch sequence σ can be fetched [46].

Figure 14 illustrates the data flow between the components of our parallel disk multi-way merging.

6.2. Implementation details

Run Formation. We build runs of a size close to $M/2$ but there are some differences to the simple algorithm from Section 6.1.1. Overlapping of I/O and computation is achieved using the call-back mechanism supported by the I/O layer. Thus, the sorter remains portable over different operating systems with different interfaces to threading.

We have two implementations with respect to the internal work: `stxxl::sort` is a comparison based sorting using `std::sort` from STL to sort the runs internally; `stxxl::ksort` exploits integer keys and has smaller internal memory bandwidth requirements for large elements with small key fields. After reading elements, we extract pairs (key, pointerToElement), sort these pairs, and only then move elements in sorted order to write buffers from where they are output. For reading and writing we have used unbuffered direct file I/O.

Furthermore, we exploit random keys. We use two passes of MSD (most significant digit) radix sort of the key-pointer pairs. The first pass uses the m most significant bits where m is a tuning parameter depending on the size of the processor caches and of the TLB (translation look-aside buffer). This pass consists of a counting phase that determines bucket sizes and a distribution phase that moves pairs. The

counting phase is fused into a single loop with pair extraction. The second pass of radix sort uses a number of bits that brings us closest to an expected bucket size of two. This two-pass algorithm is much more cache efficient than a one-pass radix sort.[‡] The remaining buckets are sorted using a comparison based algorithm: Optimal straight line code for $n \leq 4$, insertion sort for $n \in \{5..16\}$, and quicksort for $n > 16$.

Multi-way Merging. We have adapted the tuned multi-way merger from [55], i.e. a tournament tree stores pointers to the current elements of each merge buffer.

Overlapping I/O and Computation. We integrate the prefetch buffer and the overlap buffer to a *read buffer*. We distribute the buffer space between the two purposes of minimizing disk idle time and overlapping I/O and computation indirectly by computing an optimal prefetch sequence for a smaller buffer space.

Asynchronous I/O. I/O is performed without any synchronization between the disks. The prefetcher computes a sequence σ' of blocks indicating the order in which blocks should be fetched. As soon as a buffer block becomes available for prefetching, it is used to generate an asynchronous read request for the next block in σ'. The AIO layer queues this request at the disk storing the block to be fetched. The thread for this disk serves the queued request in FIFO manner. All I/O is implemented without superfluous copying. Blocks, fetched using unbuffered direct I/O, travel to the prefetch/overlap buffer and from there to a merge buffer simply by passing pointers to blocks. Similarly, when an element is merged, it is directly moved from the merge buffer to the write buffer and a block of the write buffer is passed to the output queue of a disk simply by passing a block pointer to the AIO layer that then uses unbuffered direct I/O to output the data.

6.3. Experiments

Hardware. For the experiments we have used the system with two 2GHz Xeon processors, one GByte of RAM and eight IDE disks, described in Table V. The maximum parallel disk bandwidth from the outermost (fastest) zones was about 375 MB/s.

Software. The system ran the Debian Linux distribution with kernel 2.4.20 and the `ext2` file system. All programs were compiled with `g++` version 3.2 and the optimization level `-O3`.

If not otherwise mentioned, we use random 32 bit integer keys to keep internal work limited. Runs of size 256 MByte[§] are built using key sorting with an initial iteration of 10 bit MSD radix sort. We choose block sizes in such a way that a single merging phase using 512 MBytes for all buffers suffices. Input sizes are powers of two between 2 GByte and 128 GByte with a default of 16 GByte[¶]. When not otherwise stated, we use eight disks, 2 MByte blocks, and the input is stored on the fastest zones.

To compare our code with previous implementations, we have to run them on the same machine because technological development in recent years has been very fast. Unfortunately, the

[‡]On the Xeon system we get a factor of 3.8 speedup over the one pass radix sort and a factor of 1.6 over STL's sort which in turn is faster than a hand tuned quicksort (for sorting 2^{21} pairs storing a random four byte key and a pointer).

[§]This leaves space for two runs build in an overlapped way, buffers, operating system, code, and, for large inputs, the fact that the implementation of the `ext2` file system needed 1 byte of internal memory for each KBytes of disk space accessed via `O_DIRECT`.

[¶]We have a few measurements with 256 GBytes but the problem with `ext2` mentioned above starts to distort the results for this input size.

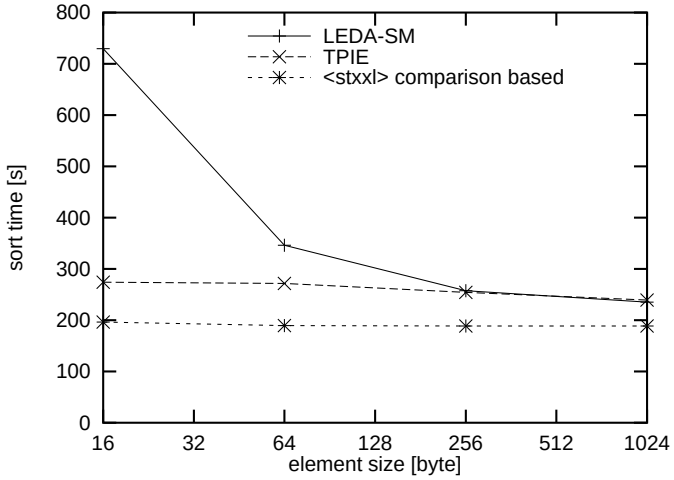

Figure 15. Comparison of the single disk performance of STXXL, LEDA-SM, and TPIE.

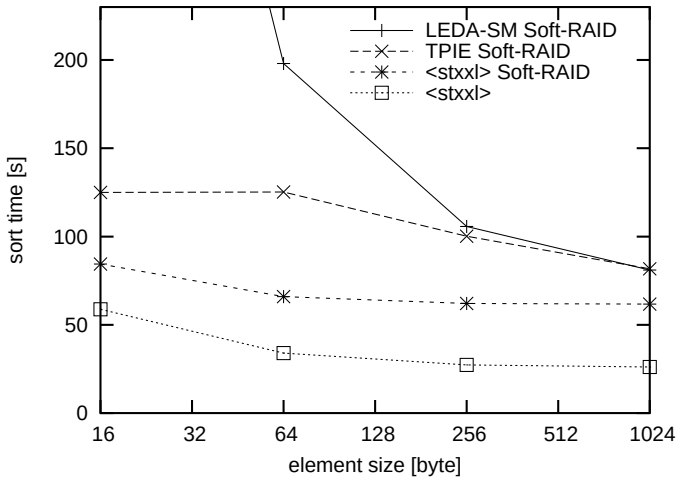

Figure 16. Comparison of of STXXL, LEDA-SM, and TPIE for eight disks.

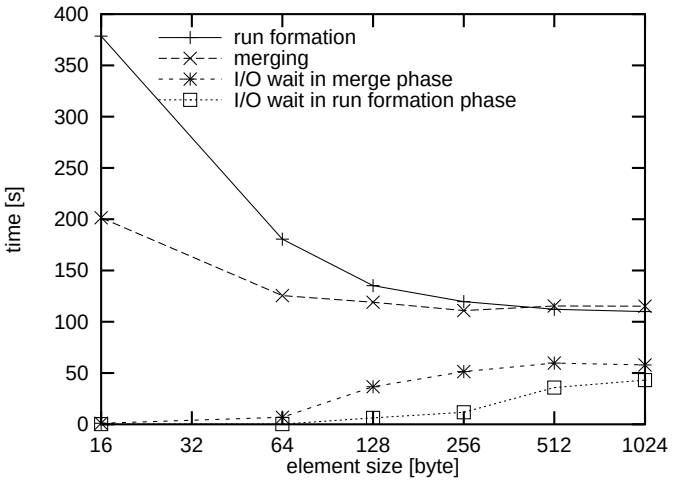

Figure 17. Dependence execution time and I/O wait time on the element size.

implementations we could obtain, LEDA-SM and TPIE, are limited to inputs of size 2 GByte which, for our machine, is a rather small input. Figure 15 compares the single disk performance of the three libraries using the best block size for each library. The flat curves for TPIE and STXXL indicate that both codes are I/O bound even for small element sizes. This is even true for the fully comparison based version of STXXL. Still, the STXXL sorter is significantly faster than the TPIE sorter. This could be due to better overlapping of I/O and computation or due to the higher bandwidth of the file system calls we use. STXXL sustains an I/O bandwidth of 45.4 MByte/s, which is 95 % of the 48 MByte/s peak bandwidth of the disk at their outermost zone. LEDA-SM is compute-bound for small keys and has the same performance as TPIE for large keys.

To get some kind of comparison for parallel disks, we ran the other codes using Linux Software-RAID 0.9 and 8 × 128KBytes stripes (larger stripes did not improve performance). Here, the STXXL sorter is between two and three times faster than the TPIE sorter and sustains an I/O bandwidth of 315 MByte/s for large elements. Much of this advantage is lost when STXXL also runs on the Software-RAID. Although we view it as likely that the Software-RAID driver can be improved, this performance difference might also be an indication that treating disks as independent devices is better than striping as predicted by theory.

Figure 17 shows the dependence of the performance on the element size in more detail. For element sizes ≥ 64, the merging phase starts to wait for I/Os and hence is I/O-bound. The run formation phase only becomes I/O-bound for element sizes above 128. This indicates areas for further optimization. For small elements, it should be better to replace key sorting by sorters that always (or more often) move the entire elements. For example, we have observed that the very simple loop that moves elements to the write buffer when the key-pointer pairs are already sorted can take up to 45 % of the CPU time

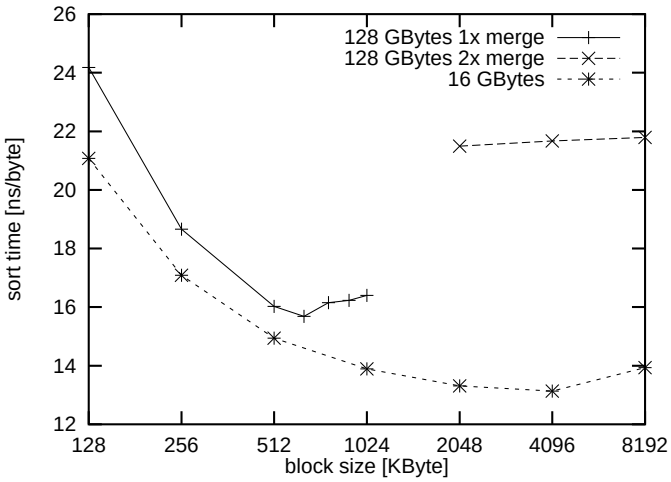

Figure 18. Dependence of sorting time on the block size.

of the run formation. For small keys it also looks promising to use parallelism. Already our cheap machine supports four parallel threads.

Figure 18 shows the dependence of the execution time on the block size. We see that block sizes of several MBytes are needed for good performance. The main reason is the well known observation that blocks should consist of several disk tracks to amortize seeks and rotational delays over a large consecutive data transfer. This value is much larger than the block sizes used in older studies because the data density on hard disks has dramatically increased in the last years. This effect is further amplified in comparison to the SCSI disks used in most other studies because modern IDE disks have even higher data densities but larger rotational delays and less opportunities for seek time optimization.

Nevertheless, the largest possible block size is not optimal because it leaves too little room for read and write buffers. Hence, in most measurements we use the heuristics to choose half the largest possible block size that is a power of two.

For very large inputs, Figure 18 shows that we already have to go below the "really good" block sizes because of the lack of buffer space. Still, it is not a good idea to switch to two merge passes because the overall time increases even if we are able to stick to large block sizes with more passes. The large optimal block sizes are an indicator that "asymptotically efficient" can also translate into "practically relevant" because simpler suboptimal parallel disk algorithms often use logical blocks striped over the disks. On our system, this leads to a further reduction of the possible block size by a factor of about eight.

Finally, Figure 19 shows the overall performance for different input sizes. Although we can stick to two passes, the execution time per element goes up because we need to employ increasingly slow zones, because the block sizes go down, and because the seek times go during merging up.

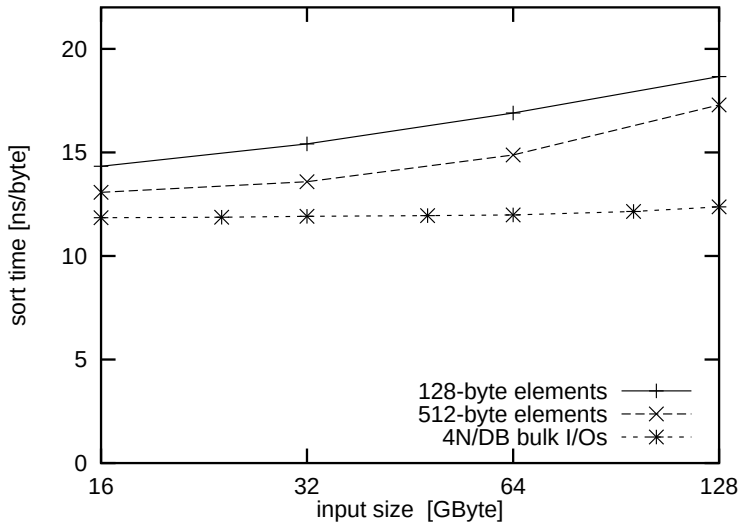

Figure 19. Dependence of sorting time on the input size.

6.4. Discussion

We have engineered a sorting algorithm that combines a very high performance on state of the art hardware with theoretical performance guarantees. This algorithm is compute-bound although we use small random keys and a tuned linear time algorithm for the run formation. Similar observations apply to other external memory algorithms that exhibit a good spatial locality, i.e. those dominated by scanning, sorting, and similar operations. This indicates that bandwidth is no longer a limiting factor for most external memory algorithms if parallel disks are used.

7. ALGORITHM PIPELINING

The pipelined processing technique is very well known in the database world [32].

Usually, the interface of an external memory algorithm assumes that it reads the input from (an) external memory container(s) and writes output into (an) external memory container(s). The idea of pipelining is to equip the external memory algorithms with a new interface that allows them to feed the output as a data stream directly to the algorithm that consumes the output, rather than writing it to the external memory first. Logically, the input of an external memory algorithm does not have to reside in the external memory, rather, it could be a data stream produced by another external memory algorithm.

Many external memory algorithms can be viewed as a data flow through a directed acyclic graph G with node set $V = F \cup S \cup R$ and edge set E. The *file nodes* F represent physical data sources and data sinks, which are stored on disks (e.g. in the external memory containers of the STL-user layer). A file node writes or/and reads one stream of elements. The *streaming nodes* S read zero, one or several streams and output zero, one or several new streams. Streaming nodes are equivalent to scan operations in non-pipelined external memory algorithms. The difference is that non-pipelined conventional scanning needs a linear number of I/Os, whereas streaming nodes usually do not perform any I/O, unless a node needs to access external memory data structures (stacks, priority queues, etc.). The sorting nodes R read a stream and output it in a sorted order. Edges E in the graph G denote the directions of data flow between nodes. The question "When is a pipelined execution of the computations in a data flow graph G possible in an I/O-efficient way?" is analyzed in [48].

8. STREAMING LAYER

The streaming layer provides a framework for the *pipelined* processing of large sequences. To the best of our knowledge we are the first who apply the pipelining method systematically in the domain of external memory algorithms. We introduce it in the context of an external memory software library.

In STXXL, all data flow node implementations have an STXXL stream interface which is similar to the STL Input iterators[‖]. As an input iterator, an STXXL stream object may be dereferenced to refer to some object and may be incremented to proceed to the next object in the stream. The reference obtained by dereferencing is read-only and must be convertible to the `value_type` of the STXXL stream. The concept of the STXXL stream also defines a boolean member function `empty()` which returns `true` iff the end of the stream is reached.

Now we tabulate the valid expressions and the expression semantics of the STXXL stream concept in the style of the STL documentation.

Notation

`X`, `X1`, ..., `Xn`	A type that is a model of the STXXL stream
`T`	The value type of `X`
`s`, `s1`, ..., `sn`	Object of type `X`, `X1`, ..., `Xn`
`t`	Object of type `T`

Valid expressions

Name	Expression	Type requirements	Return type
Constructor	`X s(s1,...,sn)`	`s1`, ..., `sn` are convertible to `X1&`, ..., `Xn&`	
Dereference	`*s`		Convertible to `T`
Member access	`s->m`	`T` is a type for which `t.m` is defined	
Preincrement	`++s`		`X&`
End of stream check	`s.empty()`		`bool`

[‖] Not be confused with the stream interface of the C++ `iostream` library.

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Constructor	`X s(s1,...,sn)`	`s1, ..., sn` are the n input streams of `s`		
Dereference	`*s`	`s` is incrementable		
Member access	`s->m`	`s` is incrementable	Equivalent to `(*s).m`	
Preincrement	`++s`	`s` is incrementable		`s` is incrementable or past-the-end

The binding of a STXXL stream object to its input streams (incoming edges in a data flow graph G) happens at compile time, i.e. statically. The other approach would be to allow binding at running time using the C++ virtual function mechanism. However this would result in a severe performance penalty because most C++ compilers are not able to inline virtual functions. To avoid this disadvantage, we follow the static binding approach using C++ templates. For example, assuming that streams `s1`, ..., `sn` are already constructed, construction of stream `s` with constructor `X::X(X1& s1,..., Xn& sn)` will bind `s` to its inputs `s1`, ..., `sn`.

After creating all node objects, the computation starts in a "lazy" fashion, first trying to evaluate the result of the topologically latest node. The node reads its intermediate input nodes, element by element, using the dereference and increment operator of the STXXL stream interface. The input nodes proceed in the same way, invoking the inputs needed to produce an output element. This process terminates when the result of the topologically latest node is computed. This style of pipelined execution scheduling is I/O-efficient and allows to keep the intermediate results in-memory without needing to store them in external memory.

The Streaming layer of the STXXL library offers generic classes which implement the functionality of sorting, file, and streaming nodes:

- File nodes: Function `streamify` serves as an adaptor that converts a range of ForwardIterators into a compatible STXXL stream. Since iterators of `stxxl::vector` are RandomAccessIterators, `streamify` can be used to read external memory. The set of (overloaded) `materialize` functions implement data sink nodes, they flush the content of a STXXL stream object to an output iterator. The library also offers specializations of `streamify` and `materialize` for `stxxl::vector`, which are more efficient than the generic implementations due to the support of overlapping between I/O and computation.
- Sort nodes: The Stream layer `stream::sort` class is a generic pipelined sorter which has the interface of an STXXL stream. The input of the sorter may be an object complying to the STXXL stream interface. The pipelined sorter implements the parallel disk merge sort [46] described in a previous section. The `stream::sort` class relies on two classes that encapsulate the two phases of the algorithm: sorted run formation (`runs_creator`) and run merging (`runs_merger`). The separate use of these classes breaks the pipelined data flow: the `runs_creator` must read the entire input to compute the sorted runs. This facilitates an

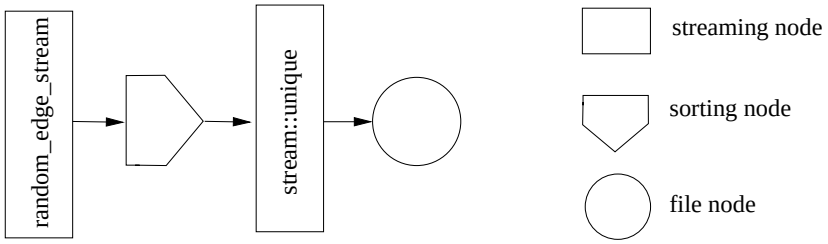

Figure 20. Data flow graph for the example in Listing 7.

efficient implementation of loops and recursions: the input for the next iteration or recursion can be the sorted runs stored on disks [75, 48]. The templated class `runs_creator` has several specializations which have input interfaces different from the STXXL stream interface: a specialization where elements to be sorted are `push_back`'ed into the `runs_creator` object and a specialization that accepts a set of presorted sequences. All specializations are compatible with the `runs_merger`.

- Streaming nodes: In general, most of the implementation effort for algorithms with the streaming layer goes to the streaming nodes. The STXXL library exposes generic classes that help to accelerate coding the streaming node classes. For example `stream::transform` is similar to the `std::transform` algorithm: it reads n input streams `s1`, ..., `sn` and returns the result of a user-given n-ary function object `functor(*s1,...,*sn)` as the next element of the output stream until one of the input streams gets empty.

STXXL allows streaming nodes to have more than one output. In this case, only one output of a streaming node can have the STXXL stream interface (it is an iterator). The other outputs can be passed to other nodes using a "push-item" interface. Such an interface is offered by file nodes (e.g. the method `push_back` of `stxxl::vector`) and sorting nodes (`push_back`-specializations). Streaming nodes do not have such methods by definition, however, it is always possible to reimplement all streaming nodes between sorting and/or file nodes as a single streaming node that will `push_back` the output elements to the corresponding sorting/file nodes.

Now we "pipeline" the random graph generation example shown in a previous section. The data flow graph of the algorithm is presented in Figure 20. Listing 7 shows the pipelined code of the algorithm, the definitions of `edge`, `random_edge`, and `edge_cmp` are in Listing 5. Since the sorter of the streaming layer accepts an STXXL stream input, we do not need to write the random edges on disks. Rather, we generate them on the fly. The `random_edge_stream` object (model of the STXXL stream) constructed in Line 19 supplies the sorter with a stream of random edges. In Line 20, we define the type of the sorter node; it is parameterized by the type of the input stream and the type of the comparison function object. Line 21 creates a `SortedStream` object attaching its input to the `RandomStream`. The internal memory consumption of the sorter stream object is limited to 512 MB. The `UniqueStream` object filters the duplicates in its input edge stream (Line 23). The generic `stream::unique` stream class stems from the STXXL library. Line 26 records the content

Listing 7. Generating a random graph using the Streaming layer.

```
using namespace stxxl;
class random_edge_stream {
    int64 counter;
    edge current;
    random_edge_stream();
public:
    typedef edge value_type;
    random_edge_stream(int64 elements):
        counter(elements),current(random_edge()()){ }
    const edge & operator * () const { return current; }
    const edge * operator ->() const { return &current; }
    random_edge_stream & operator ++ () {
        --counter;
        current = random_edge()();
        return *this;
    }
    bool empty() const { return counter==0; }
};
random_edge_stream RandomStream(10000000000ULL);
typedef stream::sort<random_edge_stream,edge_cmp> sorted_stream;
sorted_stream SortedStream(RandomStream,edge_cmp(), 512*1024*1024);
typedef stream::unique<sorted_stream> unique_stream_type;
unique_stream_type UniqueStream(SortedStream);
stxxl::vector<edge> ExtEdgeVec(10000000000ULL);
stxxl::vector<edge>::iterator NewEnd =
    stream::materialize(UniqueStream,ExtEdgeVec.begin());
ExtEdgeVec.resize(NewEnd - ExtEdgeVec.begin());
```

of the `UniqueStream` into the external memory vector. Let us count the number of I/Os the program performs: random edge generation by `RandomStream` costs no I/O; sorting in `SortedStream` needs to store the sorted runs and read them again to merge — $2N/DB$ I/Os; `UniqueStream` deletes duplicates on the fly, it does not need any I/O; and materializing the final output can cost up to N/DB I/Os. All in all, the program only incurs $3N/DB$ I/Os, compared to $7N/DB$ for the nonpipelined code in Section 5.

9. EXAMPLE: COMPUTING A MAXIMAL INDEPENDENT SET

We demonstrate some performance characteristics of STXXL using the external memory maximal independent set (MIS)* algorithm from [53] as an example. A MIS is computed, for example, for scheduling dependent parallel jobs. As input for the MIS algorithm we use the random graph computed

* An independent set I is a set of nodes on a (multi)graph G such that no edge in G joins two nodes in I, i.e. the nodes in I are not neighbors. A *maximal* independent set is an independent set such that adding any other node would cause the set not to be independent anymore.

Listing 8. Computing a Maximal Independent Set using STXXL.

```
1  struct node_greater: public std::greater<node_type> {
2      node_type min_value() const  {
3          return std::numeric_limits<node_type>::max();
4      }
5  };
6  typedef stxxl::PRIORITY_QUEUE_GENERATOR<node_type,
7      node_greater,PQ_MEM, INPUT_SIZE/1024>::result pq_type;
8
9  // keeps "not in MIS" events
10 pq_type depend(PQ_PPOOL_MEM,PQ_WPOOL_MEM);
11 stxxl::vector<node_type> MIS; // output
12 for(;!edges.empty();++edges) {
13     while(!depend.empty() && edges->src > depend.top())
14         depend.pop(); // delete old events
15     if(depend.empty() || edges->src != depend.top() ) {
16         if(MIS.empty() || MIS.back() != edges->src )
17             MIS.push_back(edges->src);
18         depend.push(edges->dst);
19     }
20 }
```

by the examples in the previous Sections (Listings 6 and 7). Our benchmark also includes the running time of the input generation.

Now we describe the MIS algorithm implementation in Listing 8 which is only nine lines long not including `typedef` declarations. The algorithm visits the graph nodes scanning lexicographically sorted input edges. When a node is visited, we add it to the maximal independent set if none of its visited neighbors is already in the MIS. The neighbor nodes of the MIS nodes are stored as events in a priority queue. In Lines 6-8, the template metaprogram [76] `PRIORITY_QUEUE_GENERATOR` computes the type of STXXL priority queue. The metaprogram finds the optimal values for numerous tuning parameters (the number and the maximum arity of external/internal mergers, the size of merge buffers, the external memory block size, etc.) under the constraint that the total size of the priority queue internal buffers must be limited by `PQ_MEM` bytes. The `node_greater` comparison functor defines the order of nodes of the type `node_type` and the minimum value that a node object can have, such that the `top()` method will return the smallest contained element. The last template parameter assures that the priority queue cannot contain more than the `INPUT_SIZE` elements (in 1024 units). Line 10 creates the priority queue `depend` having a prefetch buffer pool of size `PQ_PPOOL_MEM` bytes and a buffered write memory pool of size `PQ_WPOOL_MEM` bytes. The external vector `MIS` stores the nodes belonging to the maximal independent set. Ordered input edges come in the form of an STXXL stream called `edges`. If the current node `edges->src` is not a neighbor of a MIS node (the comparison with the current event `depend.top()`, Line 15), then it is included in `MIS` (if it was not there before, Line 17). All neighbor nodes `edges->dst` of a node in MIS `edges->src` are inserted in the event priority queue `depend` (Line 18). Lines 13-14 remove the events already passed through from the priority queue.

To make a comparison with other external memory libraries, we have implemented the graph generation algorithm using the TPIE and LEDA-SM libraries. The MIS algorithm was implemented

Table VI. Running time (in seconds)/I/O bandwidth (in MB/s) of the MIS benchmark running on single disk. For TPIE, only the graph generation is shown (marked with *). The running time of the input graph generation is split into the three phases: filling, sorting and duplicate removal.

	LEDA-SM	STXXL-STL	STXXL-Pipel.	TPIE
Filling	51/41	89/24	100/20	40/52
Sorting	371/23	188/45		307/28
Dup. removal	160/26	104/40	128/26	109/39
MIS computation	513/6	153/21		–N/A–
Total	1095/16	534/33	228/24	456*/32*

in LEDA-SM using its array heap data structure as a priority queue. An I/O-efficient implementation of the MIS algorithm was not possible in TPIE, since it does not have an I/O-efficient priority queue implementation. For TPIE, we only report the running time of the graph generation[†].

To make the benchmark closer to real applications, we have added two 32-bit integer fields to the `edge` data structure, which can store some additional information associated with the edge. The implementations of a priority queue of LEDA-SM always store a pair <key,info>. The info field takes at least four bytes. Therefore, to make a fair comparison with STXXL, we have changed the event data type stored in the priority queue (Listing 8) such that it also has a 4-byte dummy info field.

The experiments were run on the Xeon system described in Table V. For the compilation of the STXXL and TPIE sources the `g++` compiler version 3.3 was used. The compiler optimization level was set to `-O3`. For sorting we used library sorters that use C++ comparison operators to compare elements, which has the best performance. All programs have been tuned to achieve their maximum performance. We have tried all available file access methods and disk block sizes. In order to tune the TPIE benchmark implementation, we followed the performance tuning Section of [28]. The input size (the length of the random edge sequence, see Listing 6) for all tests was 2000 MB[‡]. The benchmark programs were limited to use only 512 MB of main memory. The remaining 512 MB were given to the operating system kernel, daemons, shared libraries and the file system buffer cache, from which TPIE and LEDA-SM might benefit. The STXXL implementations do not use the file system cache.

Table VI compares the MIS benchmark performance of the LEDA-SM implementation with the array heap priority queue, the STXXL implementation based on the STL-user layer, a pipelined STXXL implementation, and a TPIE implementation with only input graph generation. The running times, averaged over three runs, and average I/O bandwidths are given for each stage of the benchmark. The running time of the different stages of the pipelined implementation cannot be measured separately. However, we show the values of time and I/O counters from the beginning of the execution till the time when the sorted runs are written to the disk(s) in the run formation phase of sorting, and from this point to the end of the MIS computation. The total running time numbers show that the pipelined STXXL

[†]The source code of all our implementations is available under `http://algo2.ira.uka.de/dementiev/stxxl/paper/index.shtml`.
[‡]Algorithms and data structures of LEDA-SM are limited to inputs of size 2 GB.

Table VII. Running time (in seconds)/I/O bandwidth (in MB/s) of the MIS benchmark running on multiple disk.

		STXXL-STL		STXXL-Pipelined	
Disks		2	4	2	4
Input graph generation	Filling	72/28	64/31	98/20	98/20
	Sorting	104/77	80/100		
	Dup. removal	58/69	34/118	112/30	110/31
MIS computation		127/25	114/28		
Total		360/50	291/61	210/26	208/27

implementation is significantly faster than the other implementations. It is 2.4 times faster than the second leading implementation (STXXL-STL). The win is due to the reduced I/O volume: the STXXL-STL implementation transfers 17 GB, the pipelined implementation only needs 5.2 GB. However, the 3.25 fold I/O volume reduction does not imply equal reduction of the running time because the run formation fused with the filling/generating phase becomes compute-bound. This is indicated by the almost zero value of the STXXL I/O wait counter, which measures the time the processing thread waited for the completion of an I/O. The second reason is that the fusion of merging, duplicate removal and CPU intensive priority queue operations in the MIS computation is almost compute-bound. Comparing the running times of the total input graph generation, we conclude that the STXXL-STL implementation is about 20 % faster than TPIE and 53 % faster than LEDA-SM. This could be due to better (explicit) overlapping between I/O and computation. The running time of the the filling stage of the STXXL-STL implementation is much higher than that of TPIE and LEDA-SM. This is because those libraries rely on the operating system cache. The filled blocks do not go to the disk(s) immediately but remain in the main memory until other data needs to be cached by the system. An indication for this is the very high bandwidth of 52 MB/s for the TPIE implementation, which is even higher than the maximum physical disk bandwidth (48 MB/s) at its outermost zone. However, the cached blocks need to be flushed in the sorting stage and then the TPIE implementation pays the remaining due. The unsatisfactory bandwidth of 24 MB/s of the STXXL-STL filling phase could be improved by replacing the call `std::generate` by the native `stxxl::generate` call that efficiently overlaps I/O and computation. With a single disk it fills the vector in 60 seconds with a bandwidth of 33 MB/s. The STXXL STL-user sorter sustains an I/O bandwidth of about 45 MB/s which is 95 % of the disk's peak bandwidth. The high CPU load in the priority queue and the less then perfect overlapping between I/O and computation explain the low bandwidth of the MIS computation stage in all three implementations. We also run the graph generation test on 16 GByte inputs. All implementations scale with the input size almost linearly: the TPIE implementation finishes in 1h 3min, the STXXL-STL in 49min, and the STXXL-Pipelined in 28min.

The MIS computation of STXXL, which is dominated by PQ operations, is 3.35 times faster than LEDA-SM. The main reason for this big speedup is likely to be the more efficient priority queue.

Table VII shows the parallel disk performance of the STXXL implementations. The STXXL-STL implementation achieves a speedup of about 1.5 using two disks and 1.8 using four disks. The reason

Table VIII. Running time of the STXXL-pipelined implementation on very large random graphs (*SCSIOpteron* system).

Input volume	N/M	n	m	m/n	D	Running time
100 GB	200	$2.1 \cdot 10^9$	$13.4 \cdot 10^9$	6.25	4	2h 34min
100 GB	200	$4.3 \cdot 10^9$	$13.4 \cdot 10^9$	3.13	4	2h 44min

for this low speedup is that many parts of the code become compute-bound: priority queue operations in the MIS computation stage, the run formation in the sorting stage, and the generating random edges in the filling stage. The STXXL-pipelined implementation was almost compute-bound in the single disk case, and, with two disks the first phase shows no speedup as expected. However the second phase has a small improvement in speed due to faster I/O. A close to zero I/O wait time indicates that the STXXL-pipelined implementation is fully compute-bound when running with two or four disks. We had run the STXXL-pipelined implementation on very large graphs that require the entire space of four hard disks (360 GBytes). The results of this experiment, using the faster *SCSIOpteron* system (Table V), are shown in Table VIII.

10. STXXL APPLICATIONS

The STXXL library already has users both in academia and industry. We know of at least 17 institutions which apply the library for a wide range of problems including text processing [77, 48], graph algorithms [13, 78], gaussian elimination [79], visualization and analysis of 3D and 4D microscopic images, differential cryptographic analysis, computational geometry [80], topology analysis of large networks, statistics and time series analysis, and analysis of seismic files.

STXXL has been successfully applied in implementation projects that studied various I/O-efficient algorithms from the practical point of view. The fast algorithmic components of STXXL library gave the implementations an opportunity to solve problems of very large size on a low-cost hardware in a record time. For the tests many real-world and synthetic inputs have been used. It has been shown that external memory computation for these problems is *practically feasible* now. We overview some computational results of these projects.

The performance of external memory *suffix array construction* algorithms was investigated in [48]. The experimentation with pipelined STXXL implementations of the algorithms has shown that computing suffix arrays in external memory is feasible even on a low-cost machine. Suffix arrays for long strings up to 4 billion characters could be computed in hours.

The project [81] has compared experimentally two external memory *breadth-first search* (BFS) algorithms [82, 83]. The pipelining technique of STXXL has helped to save a factor of 2–3 in I/O volume of the BFS implementations. Using STXXL, it became possible to compute BFS decomposition of node-set of large grid graphs with 128 million edges in less than a day, and for random sparse graph class within an hour. Recently, the results have been significantly improved [13].

Simple algorithms for computing *minimum spanning trees* (MST), *connected components*, and *spanning forests* were developed in [52, 84]. Their implementations were built using STL-user-level algorithms and data structures of STXXL. The largest solved MST problem had 2^{32} nodes, the input graph edges occupied 96 GBytes. The computation on a PC took only 8h 40min.

The number of triangles in a graph is a very important metric in social network analysis [85]. An external memory algorithm that counts and lists all triangles in a graph is designed and implemented in [86, Section 4.6]. Using this implementation the author [86] has counted the number of triangles of a large web crawl graph with 135 million nodes and 1.2 billion edges in 4h 46min detecting 10.6 billion triangles. The experiment has been conducted on the SCSIOpteron machine described in Table V.

In [86, Section 4.7] simple and fast I/O-efficient heuristics for huge general graphs are designed. One of these heuristics turns out to be a practical 7-coloring algorithm for planar graphs. The implementations make a heavy use of STXXL pipelining and could color synthetic and real graph instances with 1.5 billion edges in less than an hour on cheap machines.

11. CONCLUSION

We have described STXXL: a library for external memory computation that aims for high performance and ease-of-use. The library supports parallel disks and explicitly overlaps I/O and computation. The library is easy to use for people who know the C++ Standard Template Library. The library implementations outperform or can at least compete with the best available practical implementations on real and random inputs. STXXL supports algorithm pipelining, which saves many I/Os for many external memory algorithms.

The experiments conducted within the STXXL projects indicate that a tuned I/O-efficient code can become CPU-bound and that disk bandwidth is no longer a limiting factor if many disks are used. Such observations apply to external memory algorithms that exhibit a good spatial locality, i.e. those dominated by scanning, sorting, and similar operations.

On the other hand, the fact that it is challenging to sustain a peak bandwidth for eight disks during sorting on a dual processor system implies that using even more disks requires a more aggressive use of parallel processing. Therefore the library has room for a speedup by using parallel processing. We have already achieved good results with the MCSTL library [36] parallelizing the CPU work in external memory sorters. For example on the *SCSIOpteron* system (Table V) with eight fast SCSI disks we have sorted *small* 8-byte records with a speedup close to *two* using four processors. We are also working on parallezing external priority queues and on using task based parallelism to implement pipelining.

REFERENCES

1. Ricardo Farias and Claudio T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42 – 50, July 2001.
2. Andrew Hume. *Handbook of massive data sets*, chapter “Billing in the large”, pages 895 – 909. Kluwer Academic Publishers, 2002.
3. D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J. F. Sibeyn. Algorithms and experiments for the webgraph. *Journal of Graph Algorithms and Applications*, 10(2), 2006. to appear.
4. R. W. Moore. Enabling petabyte computing. http://www.nap.edu/html/whitepapers/ch-48.html, 2000.

5. J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945. http://www.histech.rwth-aachen.de/www/quellen/vnedvac.pdf.
6. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I/II. *Algorithmica*, 12(2/3):110–169, 1994.
7. David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.
8. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.
9. Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 268–276. ACM Press, 2002.
10. M. Bander, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *13th Annual ACM-SIAM Symposium On Descrete Algorithms (SODA-02)*, 2002.
11. Gerth S. Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *SWAT 2004 : 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *LNCS*, pages 480–492, Humlebaek, Denmark, 2004. Springer.
12. Frederik Juul Christiani. Cache-oblivious graph algorithms. Master's thesis, Department of Mathematics and Computer Science (IMADA) University of Southern Denmark, Odense, 2005.
13. Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementations. In *9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. ACM-SIAM, 2007. to appear.
14. G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.
15. U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.
16. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.
17. D. E. Vengroff and J. S. Vitter. I/O-Efficient Scientific Computation using TPIE. In *Goddard Conference on Mass Storage Systems and Technologies*, volume 2, pages 553–570, 1996. published in NASA Conference Publication 3340.
18. Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Algorithms*. PhD thesis, Brown University, 1995.
19. O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A Dynamic Scalable KD-Tree. In *8th International Symposium on Spatial and Temporal Databases (SSTD '03)*, pages 46–65.
20. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient Bulk Operations on Dynamic R-trees. In *1st Workshop on Algorithm Engineering and Experimentation (ALENEX '99)*, Lecture Notes in Computer Science, pages 328–348. Springer-Verlag, 1999.
21. P. K. Agarwal, L. Arge, and S. Govindarajan. CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries. In *9th International Conference on Database Theory (ICDT '03)*, pages 143–157, 2003.
22. Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithms*, 5(17), 2000.
23. A. Crauser and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.
24. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Silicon Graphics Inc., Hewlett Packard Laboratories, 1994.
25. Bjrn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley, 2005.
26. Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the Design of CGAL, a Computational Geometry Algorithms Library. *Software - Practice and Experience*, 30(11):1167–1202, September 2000.
27. A. Crauser and K. Mehlhorn. LEDA-SM, extending LEDA to secondary memory. In *3rd International Workshop on Algorithmic Engineering (WAE)*, volume 1668 of *LNCS*, pages 228–242, 1999.
28. Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopiuc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. *TPIE: User manual and reference*, November 2003.
29. D. E. Vengroff. A Transparent Parallel I/O Environment. In *Third DAGS Symposium on Parallel Computation*, pages 117–134, Hanover, NH, July 1994.
30. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient Data Structures Using TPIE. In *10th European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 88–100. Springer, 2002.
31. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
32. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2001.
33. Elena Riccio Davidson and Thomas H. Cormen. Building on a Framework: Using FG for More Flexibility and Improved Performance in Parallel Programs. In *IPDPS*, 2005.

34. Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB White Paper. http://dev.sleepycat.com/resources/whitepapers.html, 2000.
35. Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, August 2001.
36. Felix Putze, Peter Sanders, and Johannes Singler. The Multi-Core Standard Template Library. In *Euro-Par 2007 Parallel Processing*. to appear. http://algo2.iti.uni-karlsruhe.de/singler/mcstl/.
37. University of California, Berkeley. *dbm(3) Unix Programmer's Manual*.
38. University of California, Berkeley. *ndbm(3) 4.3BSD Unix Programmer's Manual*.
39. P. Gaumond, P. A. Nelson, and J. Downs. *GNU dbm: A Database Manager*, 1999.
40. T. Nelson. Disk-based container objects. *C/C++ Users Journal*, pages 45–53, April 1998.
41. K. Knizhnik. Persistent Object Storage for C++. http://www.garret.ru/~knizhnik/post/readme.htm.
42. V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Fifth International Workshop on Persistent Object Systems*, September 1992.
43. A. Stevens. The persistent template library. *Dr. Dobb's*, pages 117–120, March 1998.
44. T. Gschwind. PSTL: A C++ Persistent Standard Template Library. In *Sixth USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, Texas, USA, January-February 2001.
45. R. Dementiev, L. Kettner, and P. Sanders. Stxxl: Standard Template Library for XXL Data Sets. In *13th Annual European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 640–651. Springer, 2005.
46. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.
47. D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. In *9th European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 62–73. Springer, 2001.
48. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 2006. to appear.
49. R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.
50. J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *12th ACM-SIAM Symposium on Discrete Algorithms*, pages 77–86, 2001.
51. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.
52. R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an External Memory Minimum Spanning Tree Algorithm. In *IFIP TCS*, pages 195–208, Toulouse, 2004.
53. N. Zeh. *I/O Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, Carleton University, Ottawa, April 2002.
54. L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In *4th Workshop on Algorithms and Data Structures*, number 955 in LNCS, pages 334–345. Springer, 1995.
55. Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
56. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. External heaps combined with effective buffering. In *4th Australasian Theory Symposium*, volume 19-2 of *Australian Computer Science Communications*, pages 72–78. Springer, 1997.
57. D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
58. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, page 173189, 1972.
59. M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference*, page 183192, June 1999.
60. M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O systems. In *13th Symposium on Parallel Algorithms and Architectures*, pages 219–228, 2001.
61. S. Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 88–98, 1998.
62. G. Chaudhry and T. H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, number 2409 in LNCS, pages 143–154, 2002.
63. G. Chaudhry, T. H. Cormen, and L. F. Wisniewski. Columnsort lives! an efficient out-of-core sorting program. In *13th ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 2001.
64. V. S. Pai and P. J. Varman. Prefetching with multiple disks for external mergesort: Simulation and analysis. In *ICDE*, pages 273–282, 1992.
65. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.
66. S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 454–462, New York, May 23–26 1998. ACM Press.

67. Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29(4):1051–1082, 2000.
68. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD*, pages 233–242, 1994.
69. R. Agarwal. A super scalar sort algorithm for RISC processors. In *ACM SIGMOD International Conference on Management of Data*, pages 240–246, 1996.
70. C. Nyberg, C. Koester, and J. Gray. Nsort: A parallel sorting program for NUMA and SMP machines, 2000. http://www.ordinal.com/lit.html.
71. J. Wyllie. SPsort: How to sort a terabyte quickly . http://research.microsoft.com/barc/SortBenchmark/SPsort.pdf, 1999.
72. Jim Gray. Sort Benchmark Home Page. http://research.microsoft.com/barc/sortbenchmark/.
73. Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. Technical Report MSR-TR-2005-183, Microsoft, December 2005. revised March 2006.
74. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
75. Jens Mehnert. External Memory Suffix Array Construction. Master's thesis, University of Saarland, Germany, November 2004. http://algo2.iti.uka.de/dementiev/esuffix/docu/data/diplom.pdf.
76. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley Professional, 2000. http://www.generative-programming.org/.
77. Haixia Tang. A suffix array based n-gram extraction algorithm. Master's thesis, Faculty of Computer Science, Dalhousie University, 2005.
78. Andrea Ribichini. Tradeoff algorithms in streaming models. Progress report, University of Rome "La Sapienza", 2006. http://www.dis.uniroma1.it/~dottorato/db/relazioni/relaz_ribichini_2.pdf.
79. Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework and experimental evaluation. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 236–236, New York, NY, USA, 2006. ACM Press.
80. Amit Mhatre and Piyush Kumar. Projective clustering and its application to surface reconstruction: extended abstract. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 477–478, New York, NY, USA, 2006. ACM Press.
81. Deepak Ajwani. Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, January 2005.
82. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *10th Symposium on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.
83. K. Mehlhorn and U. Meyer. External-Memory Breadth-First Search with Sublinear I/O. In *10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735, 2002.
84. Dominik Schultes. External Memory Spanning Forests and Connected Components. http://i10www.ira.uka.de/dementiev/files/cc.pdf, September 2003.
85. F. Harary and H. J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 6:199–210, 1979.
86. R. Dementiev. *Algorithm Engineering for Large Data Sets*. Dissertation, University of Saarland, 2006.