Stxxl : Standard Template Library for XXL Data Sets

Roman Dementiev*   Lutz Kettner†   Peter Sanders*

July 14, 2005

Abstract

We present a software library Stxxl, that enables practice-oriented experimentation with huge data sets. Stxxl is an implementation of the C++ standard template library STL for external memory computations. It supports parallel disks, overlapping between I/O and computation and is the first external memory algorithm library that supports the *pipelining* technique that can save more than *half* of the I/Os. Stxxl has already been used for the following applications: implementations of external memory algorithms for computing minimum spanning trees, connected components, breadth-first search decompositions, constructing suffix arrays, and computing social network analysis metrics for huge graphs.

1 Introduction

Massive data sets arise naturally in many domains: geographic information systems, computer graphics, database systems, telecommunication billing systems [20], network analysis [23], and scientific computing [28]. Applications working in those domains have to process terabytes of data. However, the internal memories of computers can keep only a small fraction of these huge data sets. During the processing the applications need to access the external storage (e.g. hard disks). One such access can be about 10^6 times slower than a main memory access. For any such access to the hard disk, accesses to the next elements in the external memory are much cheaper. In order to amortize the high cost of a random access one can read or write contiguous chunks of size B. The I/O becomes the main bottleneck for the applications dealing with the large data sets, therefore one tries to minimize the number of I/O operations performed. In order to increase I/O bandwidth, applications use multiple disks, in parallel. In each I/O step the algorithms try to transfer D blocks between the main memory and disks (one block from each disk). This model has been formalized by Vitter and Shriver as Parallel Disk Model (PDM) [38] and is the standard theoretical model for designing and analyzing I/O-efficient algorithms.

Theoretically, I/O-efficient algorithms and data structures have been developed for many problem domains: graph algorithms, string processing, computational geometry, etc. (for a survey see [27]). Some of them have been implemented: sorting, matrix multiplication [36], search trees [8, 30, 4, 1], priority queues [7], text processing [10]. However there is an ever increasing gap between theoretical achievements of external memory algorithms and their practical usage. Several external memory software library projects (LEDA-SM [11] and TPIE [22]) have been started

---

*Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, `[dementiev,sanders]@ira.uka.de`
†Max Planck Institut für Informatik, Saarbrücken, Germany, `kettner@mpi-sb.mpg.de`

to reduce this gap. 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.

TPIE and LEDA-SM projects are excellent proofs of concept of external memory paradigm, but have some drawbacks which impede their practical usage. This led us to start the development of a performance–oriented library of external memory algorithms and data structures STXXL, which tries to avoid those obstacles. The following are some key 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 [35, 22].

- The library is able to handle problems of *very large* size (up to dozens of terabytes).

- Improved utilization of computer resources. STXXL supports explicitly *overlapping* between I/O and computation. STXXL implementations of external memory algorithms and data structures benefit from overlapping of I/O and computation.

- Small constant factors in I/O volume. A unique library feature *"pipelining"* can save more than *half* the number of I/Os performed by many algorithms.

- Shorter *development times* due to well known STL-compatible interfaces for external memory algorithms and data structures. STL – Standard Template Library [34] is the library of algorithms and data structures which is a part of the C++ standard. STL algorithms can be directly applied to STXXL containers (code reuse); moreover the I/O complexity of the algorithms remains optimal in most of the cases.

STXXL library is open source and available under the Boost Software License 1.0 (`http://www.boost.org/LICENSE_1_0.txt`). The latest version of the library, a user tutorial and a programmer documentation can be downloaded at `http://stxxl.sourceforge.net`. Currently the size of the library is about 15 000 lines of code.

The remaining part of this paper is organized as follows. Section 2 discusses the design of STXXL. We explain how the generic interfaces of the library support high performance external memory computations using parallel disks, overlapping between I/O and computation, and pipelining. The section provides several examples. In Section 3 we implement a short benchmark that computes a maximal independent set of a graph and use it to study the performance of STXXL. Section 4 gives a short overview of the projects using STXXL. We make some concluding remarks and point out the directions of future work in Section 5.

The shortened version of this paper is also published as [14].

Related Work

TPIE [35, 5] was the first large software project implementing I/O-efficient algorithms and data structures. The library provides implementation 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.

LEDA-SM [11] external memory library was designed as an extension to the LEDA library [25] 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 can not handle more than 2^{31} bytes. The development of LEDA-SM has been stopped.

LEDA-SM and TPIE libraries 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 relies largely on the operating system that caches and prefetches data according to a general purpose policy, which can not be as efficient as the explicit approach. Furthermore, overlapping based on system cache on most of the operating systems requires additional copies of the data, which leads to CPU and internal memory overhead.

The idea of pipelined execution of the algorithms which process large data sets not fitting into main memory is well known in relational database management systems [33]. The pipelined execution strategy allows to execute a database query with minimum number of external memory accesses, to save memory space to store intermediate results, and to obtain the first result as soon as possible.

The design framework FG [13] 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 the 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 could be automatically overlapped with computation stages by the scheduler of FG environment.

2 STXXL Design

STXXL is a layered library; it consists of three layers (see Figure 1). 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 rely only on synchronous I/O APIs [11] or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API [22]. 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, asynchronous I/O APIs are very different on 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 (for example Windows) requires only reimplementing the AIO layer using native file access methods and/or native multithreading mechanisms. STXXL has already several implementations of the layer which use different file access methods under POSIX/UNIX systems.

The Block management layer (BM layer) provides a programming interface simulating the *parallel* disk model. The BM layer provides abstraction for a fundamental concept in the external memory algorithm design – 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 implementation of parallel disk buffered writing [21], optimal prefetching [21], 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 (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 factor of 2–3 in I/Os. For example, the algorithms for external memory suffix array construction implemented with this module [15] require only 1/3 of I/Os which must be performed by implementations that use conventional data structures and algorithms (either from STXXL STL-user layer, or LEDA-SM, or TPIE). The win is due to an efficient interface, that couples the input and the output of the algorithm–components
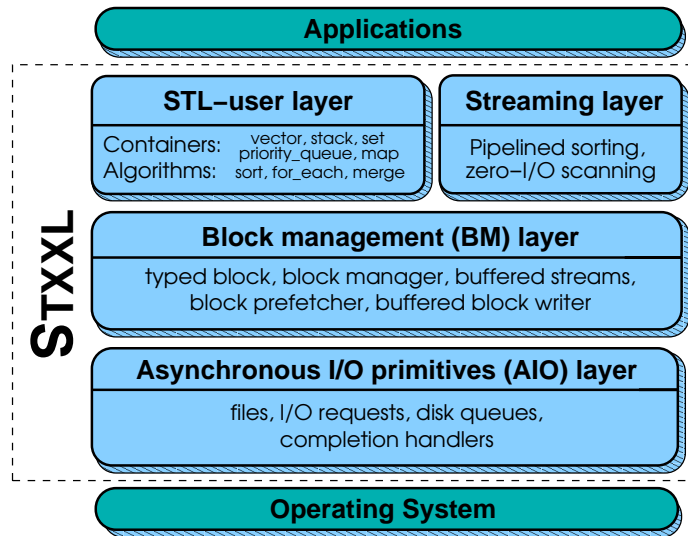
Figure 1: Structure of STXXL

(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.

2.1 Asynchronous I/O Primitives (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 complete,

- 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 to an I/O request a callback function, which is called upon completion (asynchronous notification of completion status), with 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 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 an 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 STXXL library provides various I/O performance counters (`stats` class). The class counts the number and the duration of performed I/O operations as well as the

transferred volume. The counting of read and write operations is done separately. Stxxl also measures the time spent by the processing thread(s) waiting for the completions of I/Os. 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 how to use AIO objects to program 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 the Line 8 a file object `myfile` is constructed. `syscall_file` is an implementation of 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 9 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 completes, `my_handler::operator()` will be called with a pointer to the completed request. The execution stops at Line 11 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 12) and free it (Line 13).

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."<<std::endl;
5       }
6   };
7   char * mybuffer = new char[1024*1024]; // allocate 1 MB buffer
8   syscall_file myfile("./storage", file::RDONLY );
9   request_ptr myreq = myfile.aread(mybuffer, 0, 1024*1024,my_handler ());
10  do_something1 ();          // do_something1() is overlapped with reading
11  myreq->wait ();            // wait for read completion
12  do_something2 (mybuffer);// process the read buffer
13  delete [] mybuffer; // free the buffer
```

AIO Layer Implementations. There are several implementation strategies for 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 implementation on almost every UNIX/POSIX system. Its disadvantage is that it has only limited support for I/O completion event mechanism [1]. The Linux AIO kernel side implementation [2] of POSIX AIO does not have this deficit, but is not portable since it works only under Linux.

Our AIO layer implementation 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. The main thread posts requests (invoking `file::aread` and `file::awrite` methods) in 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 API.

Our 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 memory disk. To avoid superfluous copying of data between user and

---

[1] The Linux `glibc` implementation of POSIX AIO also has a performance drawback. It launches one user thread for each I/O operation.

[2] http://freshmeat.net/projects/linux-aio/

kernel buffer memory, the `syscall` method has an option to use unbuffered file system access. Alternatively, the file access methods can be used for raw disk I/O bypassing the file system.

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

2.2 Block Management (BM) Layer

As already mentioned above, the BM layer provides an implementation of the central concept in I/O-efficient algorithms and data structures: block of elements (`typed_block` object). Besides, it 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 [6], and randomized cycling [37].

The BM layer also delivers a set of helper classes that efficiently implement frequently used sequential patterns of interaction with (parallel disk) external memory. The optimal parallel disk queued writing [21] is implemented in the `buffered_writer` class. The class operates on blocks. `buf_ostream` class is build on top of `buffered_writer` and has a high level interface, similar to the interface of STL output iterators. Analogously, classes `block_prefetcher` and `buf_istream` contain an implementation of an optimal parallel disk *prefetching* algorithm [21]. 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, when user thread is doing useful computations.

The BM layer views external memory as a set of large AIO files — one for each 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 as follows: 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 uncontrolled 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 for 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 2.3).

The external memory manager (object `block_manager`) is responsible for allocating and deallocating external memory space on the disks. The `block_manager` reads information about available disks from the STXXL configuration file. This file contains the location of each disk file, the sizes of the disks, and file access method for each disk. When allocating a bunch of blocks, a programmer can specify how the blocks will be assigned to disks, passing an allocation strategy function object. The `block_manager` implements the first fit allocation heuristic. 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 type BID describes the physical location of an allocated block, including the disk and offset of a region of storage on disk. One can load or store the data that resides at the given by BID location using asynchronous `read` and `write` methods of a `typed_block` object.

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 fixed global constant in STXXL, a programmer can simultaneously operate with several block types having different blocks sizes. 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 type InfoType, that can hold some per-block information. Block elements of type T can be easily accessed by the array `operator []` and via random access iterators.

In Listing 2 we give an example 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). As in the AIO example, I/O is overlapped with computations in 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).

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 ext. memory
```

2.3   STL-user Layer

When we started to develop the library we decided to equip our implementations of external memory data structure and algorithms with well known generic interfaces of data structures and algorithms from the Standard Template Library, which is a part of C++ standard. This choice would shorten the application development times, since the time to learn new interfaces is saved. Porting 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. Another advantage is that compatible interfaces will allow to reuse the I/O-efficient code, existing in STL (e.g. scanning and selection algorithms).

Containers

We go over the containers currently available in Stxxl.

The most universal Stxxl container is `stxxl::vector`. Vector is an array whose size can vary dynamically. The implementation of `stxxl::vector` is similar to LEDA-SM array [11]. 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. 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. In the paper we use the classical notation from [38], where M is the size of main memory, B is the block size in bytes, D is the number of disks, and N is the input size measured in bytes.

The I/O-efficient `stxxl::stack` is perhaps the simplest external memory data structure. Four different implementations of stack are available in STXXL. Some of the implementations 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 $\mathcal{O}(1/DB)$.

External memory priority queues are the central data structures for many I/O-efficient graph algorithms [39, 9, 27]. The main technique in these algorithms is time-forward processing [9, 3], easily realizable by an I/O efficient priority queue. I/O efficient priority queues also find their application in large-scale discrete event simulation and online sorting. The STXXL implementation of `priority_queue` is based on [31]. This queue needs less than a third of I/Os used by other similar cache (I/O) efficient priority queues (e.g. [7, 18]). The implementation supports parallel disks and overlaps I/O and computation.

The current version of STXXL also has an implementation of external memory `map` (based on B^+-tree) and external memory FIFO `queue`.

As in other external memory algorithm libraries [11, 22] STXXL has the restriction that the data types stored in the containers can not 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, 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 external memory. When dereferencing an external memory iterator, the pointed object is loaded from external memory by the library on demand (if the object is not already in the cache of the data structure).

STXXL containers differ in treating allocation and distinguishing between uninitialized and initialized memory from the STL containers. 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 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 can not cope with custom dynamic memory management anyway, the common use of custom constructors/destructors. However, we plan to implement special versions of STXXL containers which will support not only PODs and handle construction/destruction appropriately.

Algorithms

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

Scanning algorithms. These are the algorithms that 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 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 4). 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

Listing 3: 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)
13              Edge.dst = random() - 1 ;  //no self-loops
14          return Edge;
15      }
16  };
17  struct edge_cmp {  // edge comparison functor
18      edge min_value() const {
19          return edge(std::numeric_limits<int>::min(),0);  };
20      edge max_value() const {
21          return edge(std::numeric_limits<int>::max(),0);  };
22      bool operator () (const edge & a, const edge & b) const {
23          return a.src < b.src || (a.src == b.src && a.dst < b.dst);
24      }
25  };
```

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 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.

Random access algorithms. These algorithms require RandomAccess iterators, hence may perform many random I/Os [3]. For such algorithms, STXXL provides specialized I/O-efficient implementations that work with STL-user layer external memory containers. 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 highly efficient parallel disk implementations. The algorithm they implement guarantees close to optimal I/O volume and almost perfect overlapping between I/O and computation [16]. The performance of the sorters scales well. With eight disks which have peak bandwidth of 380 MB/s it sorts 128 byte elements with 32 bit keys achieving I/O bandwidth of 315 MB/s.

Listing 4 shows how to program using the STL-user layer and how STXXL containers can be used together with both STXXL algorithms and STL algorithms. Definitions of classes edge, random_edge and edge_cmp are in Listing 3. The purpose of our example is to generate a huge random directed graph in sorted edge array representation. 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 for it 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

---

[3]The `std::nth_element` algorithm is an exception. It needs a linear number of I/Os on average.

Listing 4: 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(),
4       512*1024*1024);
5  stxxl::vector<edge>::iterator NewEnd =
6              std::unique(ExtEdgeVec.begin(),ExtEdgeVec.end());
7  ExtEdgeVec.resize(NewEnd − ExtEdgeVec.begin());
```

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 6 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 (Line 7), we chop the vector at the NewEnd boundary. Now we count the number of I/Os performed by this example: 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 $7N/DB$.

2.4 Streaming Layer

The streaming layer provides a framework for *pipelined* processing of large sequences. Many external memory algorithms implemented with the STXXL streaming layer save a factor at least two in I/Os. The pipelined processing technique is well known in the database world [33]. To the best of our knowledge we are the first who apply this method systematically in the domain of external memory algorithms. We introduce it in the context of an external memory software library.

Usually the interface of an external memory algorithm assumes that it reads the input from external memory container(s) and writes output in 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 external memory. Logically, the input of an external memory algorithm does not have to reside in external memory, it could be rather 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 $G = (V = F \cup S \cup R, E)$. The *file nodes* F represent physical data sources and data sinks, which are stored on disks (e.g. in the external memory containers of STL-user layer). A file node outputs 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 a pipelined execution of the computations in a data flow graph G is possible in an I/O-efficient way?" is analyzed in [15].

In STXXL, all data flow node implementations have an STXXL stream interface, which is similar

to STL Input iterators[4]. 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 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 STXXL stream concept in the style of STL documentation.

Notation

X, X1, ..., Xn	A type that is a model of 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

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 dereference and increment operator of the STXXL stream interface. The input nodes procede 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.

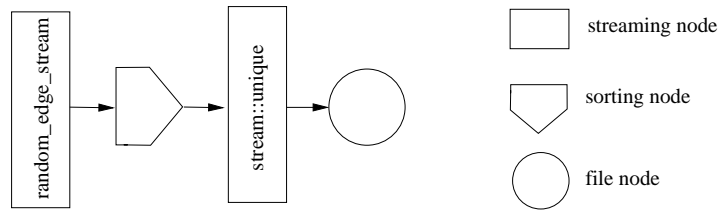[4]Do not confuse with the stream interface of the C++ iostream library.

Figure 2: Data flow graph for the example in Listing 5.

This style of pipelined execution scheduling is I/O-efficient, it allows to keep the intermediate results in-memory without needing to store them in external memory.

Streaming layer of 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: 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 STXXL stream interface. As the STL-user layer sorter, the pipelined sorter is an implementation of parallel disk merge sort [16] that overlaps I/O and computation. The implementation of `stream::sort` relies on two classes that encapsulate the two phases of the algorithm: sorted run formation (class `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 efficient implementation of loops and recursions: the input for the next iteration or recursion can be the sorted runs stored on disks [26, 15]. The templated class `runs_creator` has several specializations which have input interfaces different from 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 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.

As mentioned above, 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. The other outputs must then be passed to file nodes (e.g. via calling the method `push_back` of `stxxl::vector`) or sorting nodes (they have a `push_back` interface too).

Now we "pipeline" the random graph generation example shown in the previous chapter. The data flow graph of the algorithm is presented in Figure 2 in the appendix. Listing 5 shows the pipelined code of the algorithm, the definitions of `edge`, `random_edge`, and `edge_cmp` are in Listing 3. Since the sorter of the streaming layer accepts an STXXL stream input, we do not need to

Listing 5: Generating a random graph using the Streaming layer.

```
1  using namespace stxxl;
2  class random_edge_stream {
3      int64 counter;
4      edge current;
5      random_edge_stream();
6  public:
7      typedef edge value_type;
8      random_edge_stream(int64 elements):
9          counter(elements),current(random_edge()()){  }
10     const edge & operator * () const { return current; }
11     const edge * operator ->() const { return &current; }
12     random_edge_stream & operator ++ () {
13         --counter;
14         current = random_edge()();
15         return *this;
16     }
17     bool empty() const { return counter==0; }
18 };
19 random_edge_stream RandomStream(10000000000ULL);
20 typedef stream::sort<random_edge_stream,edge_cmp> sorted_stream;
21 sorted_stream SortedStream(RandomStream,edge_cmp(), 512*1024*1024);
22 typedef stream::unique<sorted_stream> unique_stream_type;
23 unique_stream_type UniqueStream(SortedStream);
24 stxxl::vector<edge> ExtEdgeVec(10000000000ULL);
25 stxxl::vector<edge>::iterator NewEnd =
26     stream::materialize(UniqueStream,ExtEdgeVec.begin());
27 ExtEdgeVec.resize(NewEnd - ExtEdgeVec.begin());
```

output the random edges. Rather, we generate them on the fly. The random_edge_stream object (model of 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 bounded 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 of the UniqueStream into the external memory vector. As in the Listing 4 (Line 27), we cut the vector at the NewEnd boundary. 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. Totally the program incurs only $3N/DB$ I/Os, compared to $7N/DB$ for the nonpipelined code in Section 2.3.

3   Performance

We demonstrate some performance characteristics of STXXL using the external memory maximal independent set (MIS) algorithm from [39] as an example. This algorithm is based on the time-forward processing technique. As the input for the MIS algorithm, we use the random graph computed by the examples in the previous Sections (Listings 4 and 5). Our benchmark also

Listing 6: Computing a Maximal Independent Set using STXXL.

```cpp
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, node_greater,
7              PQ_MEM, INPUT_SIZE/1024>::result pq_type;
8  pq_type depend(PQ_PPOOL_MEM,PQ_WPOOL_MEM); // keeps "not in MIS" events
9  stxxl::vector<node_type> MIS; // output
10 for(;!edges.empty();++edges) {
11     while(!depend.empty() && edges->src > depend.top())
12         depend.pop(); // delete old events
13     if(depend.empty() || edges->src != depend.top() ) {
14         if(MIS.empty() || MIS.back() != edges->src )
15             MIS.push_back(edges->src);
16         depend.push(edges->dst);
17     }
18 }
```

includes the running time of the input generation.

Now we describe the MIS algorithm implementation in Listing 6, which is only nine lines long not including 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 neighbours is already in the MIS. The neighbour nodes of the MIS nodes are stored as events in a priority queue. In Lines 6-7, the template metaprogram [12] PRIORITY_QUEUE_GENERATOR computes the type of priority queue that will store events. 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, 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 type node_type and minimum value that a node object can have, such that the top() method will return the smallest contained element. The last template parameter tells that the priority queue can not contain more than INPUT_SIZE elements (in 1024 units). Line 8 creates the priority queue depend having prefetch buffer pool of size PQ_PPOOL_MEM bytes and 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 neighbour of a MIS node (the comparison with the current event depend.top(), Line 13), then it is included in MIS (if it was not there before, Line 15). All neighbour nodes edges->dst of a node in MIS edges->src are inserted in the event priority queue depend (Line 16). Lines 11-12 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 TPIE and LEDA-SM libraries. The MIS algorithm was implemented in LEDA-SM using its array heap data structure as a priority queue. The 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 report only the running time of the graph generation. The source code of all our implementations is available under http://i10www.ira.uka.de/dementiev/stxxl/paper/index.shtml.

To make the benchmark closer to real applications, we have added two 32-bit integer fields in the edge data structure, which can store some additional information associated with the edge.

The implementations of 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 6), such that it also has a 4-byte dummy info field.

The experiments were run on a 2-processor workstation, having 2 GHz Xeon processors (only one processor was used) and 1 GB of main memory (swapping was switched off). The OS was Debian with Linux kernel 2.4.20. The computer had four 80 GB IDE (IBM/Hitachi 120 GXP series) hard disks formatted with the XFS file system and dedicated solely for the experiments. We used LEDA-SM version 1.3 with LEDA version 4.2.1[5] and TPIE of January 21, 2005. For compilation of STXXL and TPIE sources, the `g++` compiler version 3.3 was used. LEDA-SM and LEDA were compiled with `g++` compiler version 2.95, because they could not be compiled by later `g++` versions. The compiler optimization level was set to `-O3`. For sorting we used library sorters that use C++ comparison operators to compare elements. 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 [22]. The input size (the length of the random edge sequence, see Listing 4) for all tests was 2000 MB[6]. The benchmark programs were limited to use only 512 MB of main memory. The remaining 512 MB are given to operating system kernel, daemons, shared libraries and file system buffer cache, from which TPIE and LEDA-SM might benefit. The STXXL implementations do not use the file system cache.

Table 1: Running time (in seconds)/I/O bandwidth (in MB/s) of the MIS benchmark running on single disk. For TPIE only graph generation is shown (marked with *).

		LEDA-SM	STXXL-STL	STXXL-Pipel.	TPIE
Input graph generation	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*

Table 1 compares the MIS benchmark performance of the LEDA-SM implementation with array heap priority queue, the STXXL implementation based on the STL-user level, 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 time numbers show that the pipelined STXXL 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 reduced I/O volume: the STXXL-STL implementation transfers 17 GB, the pipelined implementation needs only 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 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

---

[5]Later versions of the LEDA are not supported by the last LEDA-SM version 1.3.

[6]Algorithms and data structures of LEDA-SM are limited to inputs of size 2 GB.

queue operations in the MIS computation is almost compute bound. Comparing the running times of the total input graph generation we conclude that 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. Another possible reason could be that TPIE uses a more expensive way of reporting run-time errors, such as I/O errors[7]. The running time of the filling stage of Stxxl-STL implementation is much higher than of TPIE and LEDA-SM. This is due to the fact that those libraries rely on operating system cache. The filled blocks do not go immediately to the disk(s) but remain in the main memory until other data needs to be cached by the system. The indication of this is the very high bandwidth of 52 MB/s for 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. 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 not very 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 almost linearly with the input size: the TPIE implementation finishes in 1h 3min, Stxxl-STL in 49min, and 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 algorithm from [31].

Table 2: 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

Table 2 shows the parallel disk performance of the Stxxl implementations. The Stxxl-STL implementation achieves speedup of about 1.5 using two disks and 1.8 using four disks. The reason for this low speedup is that many parts of the code become compute bound: priority queue operations in the MIS computation stage, run formation in the sorting stage, and generating random edges in the filling stage. The Stxxl-Pipelined implementation was almost compute bound in the single disk case, and as expected, with two disks the first phase shows no speedup. However the second phase has a small improvement in speed due to faster I/O. 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

---

[7]TPIE uses function return types for error codes and diagnostics, which can become quite expensive at the level of the single-item interfaces (e.g. `read_item` and `write_item`) that is predominantly used in TPIEs algorithms. Instead, Stxxl checks (I/O) errors on the per-block basis. We will use C++ exceptions to propagate errors to the user layer without any disadvantage for the library users. First experiments indicate that this will have negligible impact on runtime.

require the entire space of four hard disks (360 GBytes). The results of this experiment, using a faster Opteron system, are shown in Table 3.

Table 3: Running time of the STXXL-Pipelined implementation running on very large random graphs (Opteron system).

Input volume	N/M	#nodes	#edges	#edges/#nodes	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

4 Applications

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.

The performance of external memory *suffix array construction* algorithms was investigated in [15]. 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 [2] has compared experimentally two external memory *breadth-first search* (BFS) algorithms [29, 24]. 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.

Simple algorithms for computing *minimum spanning trees* (MST), *connected components*, and *spanning forests* were developed in [17, 32]. 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 [19]. We have designed and implemented an external memory algorithm that counts and lists all triangles in a graph. Using our implementation we have counted the number of triangles of a web crawl graph from the WebBase project [8]. In this graph the nodes are web pages and edges are hyperlinks between them. For the computation we ignored the direction of the links. Our crawl graph had 135 million nodes and 1.2 billion edges. During computation on an Opteron SMP which took only 4h 46min we have detected 10.6 billion triangles. Total volume of 851 GB was transferred between 1GB of main memory and seven hard disks. The details about the algorithm and the source code are available under `http://i10www.ira.uka.de/dementiev/tria/algorithm.shtml`.

5 Conclusions

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. STXXL supports algorithm pipelining, which saves many I/Os for many external memory

---

[8]`http://www-diglib.stanford.edu/~testbed/doc2/WebBase/`

algorithms. Several projects using STXXL have been finished already. With help of STXXL, they have solved very large problem instances externally using a low cost hardware in a record time. The work on the project is in progress. Future directions of STXXL development cover the implementation of the remaining STL containers, improving the pipelined sorter with respect to better overlapping of I/O and computation, implementations of graph and text processing external memory algorithms. We plan to submit STXXL to the collection of the Boost C++ libraries (`www.boost.org`) which includes a Windows port.

References

[1] P. K. Agarwal, L. Arge, and S. Govindarajan. CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries. In *Proc. 9th Int'l Conference on Database Theory (ICDT '03)*, pages 143–157, 2003.

[2] D. Ajwani. Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, January 2005.

[3] 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.

[4] 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.

[5] 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.

[6] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[7] 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.

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

[9] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren E. Vengroff, and Jeffey S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[10] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[11] 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.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley Professional, 2000.

[13] E. R. Davidson and T. H. Cormen. Building on a Framework: Using FG for More Flexibility and Improved Performance in Parallel Programs. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*. to appear.

[14] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: Standard Template Library for XXL Data Sets. In *13th Ann. European Symposium on Algorithms (ESA)*, 2005. to appear.

[15] R. Dementiev, J. Mehnert, J. Kärkkäinen, and P. Sanders. Better External Memory Suffix Array Construction. In *Workshop on Algorithm Engineering & Experiments*, Vancouver, 2005. `http://i10www.ira.uka.de/dementiev/files/DKMS05.pdf` see also `http://i10www.ira.uka.de/dementiev/esuffix/docu/data/diplom.pdf`.

[16] 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.

[17] 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.

[18] 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.

[19] Frank Harary and Helene J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 6:199–210, 1979.

[20] Andrew Hume. *Handbook of massive data sets*, chapter Billing in the large, pages 895 – 909. Kluwer Academic Publishers, 2002.

[21] 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.

[22] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, R. Wickeremesinghe. *TPIE: User manual and reference*, November 2003.

[23] L. Laura, S. Leonardi, S. Millozzi, U. Meyer, , and J.F. Sibeyn. Algorithms and Experiments for the Webgraph. In *11th Ann. European Symposium on Algorithms (ESA)*, pages 703–714, 2003.

[24] K. Mehlhorn and U. Meyer. External-Memory Breadth-First Search with Sublinear I/O. In *10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735, 2002.

[25] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[26] Jens Mehnert. External Memory Suffix Array Construction. Master's thesis, University of Saarland, Germany, November 2004. `http://i10www.ira.uka.de/dementiev/esuffix/docu/data/diplom.pdf`.

[27] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

[28] R. W. Moore. Enabling petabyte computing. `http://www.nap.edu/html/whitepapers/ch-48.html`, 2000.

[29] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th Symp. on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.

[30] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A Dynamic Scalable KD-Tree. In *Proc. 8th Int'l Symposium on Spatial and Temporal Databases (SSTD '03)*, pages 46–65.

[31] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[32] Dominik Schultes. External Memory Spanning Forests and Connected Components. `http://i10www.ira.uka.de/dementiev/files/cc.pdf`, September 2003.

[33] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2001.

[34] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Silicon Graphics Inc., Hewlett Packard Laboratories, 1994.

[35] D. E. Vengroff. A Transparent Parallel I/O Environment. In *Third DAGS Symposium on Parallel Computation*, pages 117–134, Hanover, NH, July 1994.

[36] 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.

[37] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *12th ACM-SIAM Symposium on Discrete Algorithms*, pages 77–86, 2001.

[38] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I/II. *Algorithmica*, 12(2/3):110–169, 1994.

[39] Norbert Ralf Zeh. *I/O Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, Carleton University, Ottawa, April 2002.