Stxxl : Standard Template Library for XXL Data Sets

Roman Dementiev[1], Lutz Kettner[2], and Peter Sanders[1,⋆]

[1] Fakultät für Informatik, Universität Karlsruhe,
Karlsruhe, Germany
{dementiev, sanders}@ira.uka.de
[2] Max Planck Institut für Informatik,
Saarbrücken, Germany
kettner@mpi-sb.mpg.de

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 *pipelining* technique that can save more than *half* of the I/Os. Stxxl has already been used for computing minimum spanning trees, connected components, breadth-first search decompositions, constructing suffix arrays, and computing social network analysis metrics.

1 Introduction

Massive data sets arise naturally in many domains: geographic information systems, computer graphics, database systems, telecommunication billing systems, network analysis, and scientific computing. 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. One minimizes the number of I/Os performed, and 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 of size M and D disks (one block from each disk). This model has been formalized by Vitter and Shriver as Parallel Disk Model (PDM) [1] and is the standard theoretical model for designing and analyzing I/O-efficient algorithms. In this model, N is the input size and B is the block size measured in bytes.

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 [2]). Some of them have been implemented:

⋆ Partially supported by DFG grant SA 933/1-2.

sorting, matrix multiplication [3], (geometric) search trees [3], priority queues [4], suffix array construction [4]. However there is an increasing gap between theoretical achievements of external memory (EM) algorithms and their practical usage. Several EM software library projects (LEDA-SM [4] and TPIE [5]) have been started 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.

We have started to develop an external memory library Stxxl making more emphasis on performance, trying to avoid the drawbacks of the previous libraries impeding their practical usage. The following are some key features of Stxxl:

– Transparent support of parallel disks.
– The library is able to handle problems of size up to dozens of terabytes.
– Explicit *overlapping* between I/O and computation.
– A library feature *"pipelining"* can save more than *half* the number of I/Os performed by many algorithms, directly feeding the output from an EM algorithm into another EM algorithm, without needing to store it on the disk in between.
– The library avoids superfluous copying of data blocks, e.g. in I/O subsystem.
– Short *development times* due to well known STL-compatible interfaces for EM algorithms and data structures. STL – Standard Template Library is the library of algorithms and data structures that is a part of the C++ standard. STL algorithms can be directly applied to Stxxl containers; 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. In Section 3 we implement a short benchmark 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.

Related Work. TPIE [3] 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 [4] EM library was designed as an extension to the LEDA library for handling large data sets. The library offers implementations of I/O-efficient sorting, EM 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 EM 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 that process large data sets not fitting into main memory is very well known in relational database management systems. The pipelined execution strategy allows to execute a database query with minimum number of EM accesses, to save memory space to store intermediate results, and to obtain the first result as soon as possible.

FG [6] is a design framework for parallel programs running on clusters, where parallel programs are split into series of asynchronous stages, which are executed in the pipelined fashion with the help of multithreading. This allows to mitigate disk access latency, communication network latency, and overlap I/O and communication.

2 Stxxl Design

Stxxl 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 EM algorithm libraries rely only on synchronous I/O APIs [4] or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API [5]. 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 synchronous file access methods under POSIX/UNIX systems. The `read/write` calls using direct access (`O_DIRECT` option) have shown the best performance under Linux. To provide asynchrony we use POSIX threads or Boost threads.

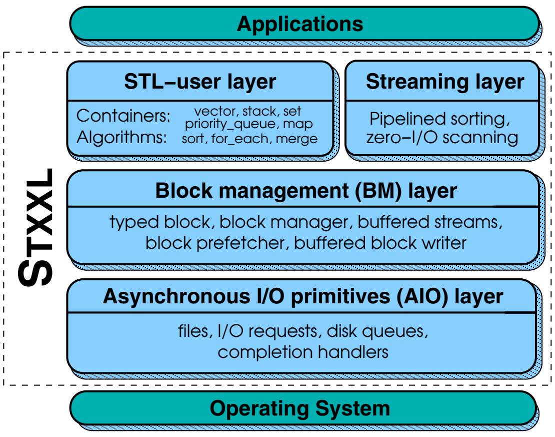
Applications
STL–user layer
Containers: vector, stack, set priority_queue, map
Algorithms: sort, for_each, merge
Streaming layer
Pipelined sorting, zero–I/O scanning
STXXL
Block management (BM) layer
typed block, block manager, buffered streams, block prefetcher, buffered block writer
Asynchronous I/O primitives (AIO) layer
files, I/O requests, disk queues, completion handlers
Operating System

Fig. 1. Structure of Stxxl

The Block Management layer (BM layer) provides a programming interface simulating the *parallel* disk model. The block manager implements block allocation/deallocation allowing several block-to-disk assignment strategies: striping, randomized striping, randomized cycling, etc. The BM layer provides implementation of parallel disk buffered writing [7], optimal prefetching [7], 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 EM data structures which have (almost) the same interfaces (including syntax and semantics) as their STL counterparts. The Streaming layer provides efficient support for *pipelining* EM algorithms. The algorithms for external memory suffix array construction implemented with this module [8] 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 rest of this section discusses the STL-user and Streaming layers in more detail. The detailed description of the BM and AIO layers can be found in the extended version of the paper [9].

2.1 STL-User Layer

Containers
Vector is an array whose size can vary dynamically. Similar to LEDA-SM arrays [4], the user has the choice over the block striping strategy of `vector`, the size of the vector cache, the cache replacement strategy (LRU, random, user-defined). 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.

EM priority queues are used for time-forward processing technique in external graph algorithms [10,2] and online sorting. The STXXL implementation of `priority_queue` is based on [11]. This queue needs less than a third of I/Os used by other similar cache (I/O) efficient priority queues. The implementation supports parallel disks and overlaps I/O and computation.

The current version of STXXL also has an implementation of EM `map` (based on B^+-tree), FIFO `queue`, and several efficient implementations of `stack`.

STXXL allows to store the references to objects located in EM using EM iterators (e.g. `stxxl::vector::iterator`). The iterators remain valid while storing to and loading from EM. When dereferencing an EM iterator, the pointed object is loaded from EM by the library on demand.

STXXL containers differ from the STL containers in their treatment of memory and distinction of 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 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 EM 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 1.1). 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. Being aware of the sequential access pattern of the applied algorithm, the Stxxl implementations can do prefetching and use queued writing, thereby enabling overlapping of I/O with computation.

Random access algorithms. These algorithms require RandomAccess iterators, hence may perform many random I/Os [1]. 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 [7]. The performance of the sorter 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 1.1 shows how to program using the STL-user layer and how Stxxl containers can be used together with both Stxxl algorithms and STL algorithms. This example generates a huge random directed graph in sorted edge array representation. The edges 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. The STL/Stxxl code for it is only five lines long: Line 1 creates an Stxxl EM vector with 10 billion edges. Line 2 fills the vector with random edges (`generate` from STL is used, `random_edge` functor returns random edge objects). In the next line the Stxxl sorter sorts randomly generated edges using 512 megabytes of internal memory. The lexicographical order is defined by functor `my_cmp`. Line 6 deletes duplicate

[1] The `std::nth_element` algorithm is an exception. It needs $\mathcal{O}(N/B)$ I/Os on average.

edges in the EM 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.

Listing 1.1. Generating a random graph using the STL-user layer

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

2.2 Streaming Layer

The streaming layer provides a framework for *pipelined* processing of large sequences. The pipelined processing technique is well known in the database world. To the best of our knowledge we are the first to apply this method systematically in the domain of EM algorithms. We introduce it in the context of an EM software library.

Usually the interface of an EM algorithm assumes that it reads the input from EM container(s) and writes output to EM container(s). The idea of pipelining is to equip the EM 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 EM. Logically, the input of an EM algorithm does not have to reside in EM, it could be rather a data stream produced by another EM algorithm.

Many EM 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 EM containers of STL-user layer). A file node outputs or/and reads one stream of elements. Streaming nodes S are equivalent to scan operations in non-pipelined EM algorithms, but do not perform any I/O, unless a node needs to access EM data structures. 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. A pipelined execution of the computations in a data flow is possible in an I/O-efficient way [8].

In Stxxl, all data flow node implementations have an Stxxl stream interface which is similar to STL Input iterators[2]. 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. Stxxl stream has a boolean member function `empty()` which returns `true` iff the end of the stream is reached. The binding of a Stxxl stream object to its input streams (incoming edges in a data flow graph G) happens at compile time using templates, such that we benefit from function inlining in

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

C++. After constructing 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. 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 EM.

In the extended version of the paper [9] we show how to "pipeline" the random graph generation example from the previous chapter, such that the number of I/Os is more than halved.

3 Performance

We demonstrate some performance characteristics of STXXL using the EM maximal independent set (MIS) algorithm from [10] 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 Section (Listings 1.1 and its pipelined version [9]). Our benchmark includes the running time of the input generation.

The MIS algorithm given in Listing 1.2 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, EM 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.

Listing 1.2. Computing a Maximal Independent Set using Stxxl

```
struct node_greater : public std::greater<node_type> {
    node_type min_value() const {
        return std::numeric_limits<node_type>::max();
    }
};
typedef stxxl::PRIORITY_QUEUE_GENERATOR<node_type,
 node_greater, PQ_MEM, INPUT_SIZE/1024>::result pq_type;
pq_type depend(PQ_PPOOL_MEM,PQ_WPOOL_MEM);
stxxl::vector<node_type> MIS; // output
for(;!edges.empty();++edges) {
    while(!depend.empty() && edges->src > depend.top())
        depend.pop(); // delete old events
    if(depend.empty() || edges->src != depend.top() ) {
        if(MIS.empty() || MIS.back() != edges->src )
            MIS.push_back(edges->src);
        depend.push(edges->dst);
    }
}
```

To make a comparison with other EM libraries, we have implemented the graph generation algorithm using TPIE and LEDA-SM. 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, the `edge` data structure has two 32-bit integer fields, which can store some additional information associated with the edge. The priority queues 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, such that it also has a 4-byte dummy info field.

The experiments were run on a 2-processor Xeon (2 GHz) workstation (only one processor was used) and 1 GB of main memory (swapping was switched off). The OS was Debian Linux with 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[3] and TPIE of January 21, 2005. For compilation of Stxxl and TPIE sources, the `g++` version 3.3 was used. LEDA-SM and LEDA were compiled with `g++` version 2.95, because they could not be compiled by later `g++` versions. The optimization level was set to `-O3`. 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

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

file access methods and disk block sizes. In order to tune the TPIE benchmark implementation, we followed the performance tuning Section of [5]. The input size (the length of the random edge sequence, see Listing 1.1) for all tests was 2000 MB[4]. 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, the Stxxl implementation based on the STL-user level, a pipelined Stxxl implementation, and a TPIE implementation (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) 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 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

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

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

errors, such as I/O errors[5]. The running time of the filling stage of Stxxl-STL implementation is much higher than of TPIE and LEDA-SM because they 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 to 33 MB/s by replacing the call `std::generate` by the native `stxxl::generate` call that efficiently overlaps I/O and computation. 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 [11].

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, run formation in the sorting, 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.

[5] 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.

Close to zero I/O wait time indicates that the Stxxl-Pipelined implementation is fully compute bound when running with two or four disks. The longest MIS computation, requiring the entire space of four disks (360 GBytes), for the graph with $4.3 \cdot 10^9$ nodes and $13.4 \cdot 10^9$ edges took 2h 44min on an Opteron system.

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 EM *suffix array construction* algorithms was investigated in [8]. The experimentation with pipelined Stxxl implementations of the algorithms has shown that computing suffix arrays in EM 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 [13] has compared experimentally two EM *breadth-first search* (BFS) algorithms. The pipelining technique of Stxxl has helped to save a factor of 2–3 in I/O volume. Using Stxxl, it became possible to compute BFS decomposition of large grid graphs with 128 million edges in less than a day, and for random sparse graphs within an hour.

Simple algorithms for computing *minimum spanning trees* (MST), *connected components*, and *spanning forests* were developed in [14]. 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 8h 40min.

The number of triangles in a graph is a very important metric in social network analysis. 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 [6]. 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 EM algorithms. Several projects using

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

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 EM algorithms. We plan to submit STXXL to the collection of the Boost C++ libraries (`www.boost.org`) which includes a Windows port.

References

1. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory, I/II. Algorithmica **12** (1994) 110–169
2. Meyer, U., Sanders, P., Sibeyn, J., eds.: Algorithms for Memory Hierarchies. Volume 2625 of LNCS Tutorial. Springer (2003)
3. Arge, L., Procopiuc, O., Vitter, J.S.: Implementing I/O-efficient Data Structures Using TPIE. In: 10th European Symposium on Algorithms (ESA). Volume 2461 of LNCS., Springer (2002) 88–100
4. Crauser, A.: LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice. PhD thesis, Universität des Saarlandes, Saarbrücken (2001) `http://www.mpi-sb.mpg.de/~crauser/diss.pdf`.
5. L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, R. Wickeremesinghe: TPIE: User manual and reference. (2003)
6. Davidson, E.R., Cormen, T.H.: 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.
7. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: 15th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego (2003) 138–148
8. Dementiev, R., Mehnert, J., Kärkkäinen, J., Sanders, P.: 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`.
9. Dementiev, R., Kettner, L., Sanders, P.: Stxxl: Standard Template Library for XXL Data Sets. Technical Report 18, Fakultät für Informatik, University of Karlsruhe (2005)
10. Zeh, N.R.: I/O Efficient Algorithms for Shortest Path Related Problems. PhD thesis, Carleton University, Ottawa (2002)
11. Sanders, P.: Fast priority queues for cached memory. ACM Journal of Experimental Algorithmics **5** (2000)
12. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley Professional (2000)
13. Ajwani, D.: Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (2005)
14. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an External Memory Minimum Spanning Tree Algorithm. In: IFIP TCS, Toulouse (2004) 195–208