STXXL Tutorial

for STXXL 1.1

Roman Dementiev

under development

Contents

Chapter 1

Introduction

There exist many application that have to process data sets which can not fit into the main memory of a computer, but external memory (e.g. hard disks). The examples are Geographic Information Systems, Internet and telecommunication billing systems, Information Retrieval systems manipulating terabytes of data.

The most of engineering efforts have been spent on designing algorithms which work on data that *completely* resides in the main memory. The algorithms assume that the execution time of any memory access is a *small* constant (1–20 ns). But it is no more true when an application needs to access external memory (EM). Because of the mechanical nature of the position seeking routine, a random hard disk access takes about 3–20 ms. This is about **1 000 000** longer than a main memory access. Since the I/Os are apparently the major bottleneck of applications that handle large data sets, they minimize the number of performed I/Os. A new measure of program performance is becoming sound – the I/O complexity.

Vitter and Shriver [8] came up with a model for designing I/O efficient algorithms. In order to amortize the high cost of a random disk access[1], external data loaded in contiguous chunks of size B. To increase bandwidth external memory algorithms use multiple parallel disks. The algorithms try in each I/O step transfer D blocks between the main memory and disks (one block per each disk).

I/O efficient algorithms have been developed for many problem domains, including fundamental ones like sorting [], graph algorithms [], string processing [], computational geometry [].

However there is the ever increasing gap between theoretical nouveau of external memory algorithms and their use in practice. Several EM software library projects (LEDA-SM [2] and TPIE [1]) attempted to reduce this gap. They offer frameworks which aim to speed up the process of implementing I/O efficient algorithms giving a high level abstraction away the details of how I/O is performed. Implementations of many EM algorithms and data structures are offered as well.

Those projects are excellent proofs of EM paradigm, but have some drawbacks which *impede* their practical use.

Therefore we started to develop STXXL library, which tries to avoid those obstacles. The objectives of STXXL project (distinguishing it from other libraries):

---

[1] Modern disks after locating the position of the data on the surface can deliver the contiguous data blocks at speed 50-60 MiB/s. For example with the seek time 10 ms, 1 MiB can be read or written in $10 + 1000 \times 1/50 = 30$ ms, 1 byte – in 10.02 ms.

- Make the library able to handle problems of *real world size* (up to dozens of terabytes).
- Offer *transparent* support of parallel disks. This feature although announced has not been implemented in any library.
- Implement *parallel* disk algorithms. LEDA-SM and TPIE libraries offer only implementations of single disk EM algorithms.
- Use computer resources more efficiently. STXXL allows transparent *overlapping* of I/O and computation in many algorithms and data structures.
- Care about constant factors in I/O volume. A unique library feature *"pipelining"* can *half* the number of I/Os performed by an algorithm.
- Care about the *internal work*, improve the in-memory algorithms. Having many disks can hide the latency and increase the I/O bandwidth, s.t. internal work becomes a bottleneck.
- Care about operating system overheads. Use *unbuffered disk access* to avoid superfluous copying of data.
- Shorten *development times* providing well known interface for EM algorithms and data structures. We provide STL-compatible[2] interfaces for our implementations.

[2]STL – Standard Template Library [7] is freely available library of algorithms and data structures delivered with almost any C++ compiler.

Chapter 2

Prerequisites

The intended audience of this tutorial are developers or researchers who develop applications or implement algorithms processing large data sets which do not fit into the main memory of a computer. They must have basic knowledge in the theory of external memory computing and have working knowledge of C++ and an experience with programming using STL. Familiarity with key concepts of generic programming and C++ template mechanism is assumed.

Chapter 3

Installation

See the STXXL home page `stxxl.sourceforge.net` for the installation instruction for your compiler and operating system.

Chapter 4

A Starting Example

Let us start with a toy but pretty relevant problem: the phone call billing problem. You are given a sequence of event records. Each record has a time stamp (time when the event had happened), type of event ('call begin' or 'call end'), the callers number, and the destination number. The event sequence is time-ordered. Your task is to generate a bill for each subscriber that includes cost of all her calls. The solution is uncomplicated: sort the records by the callers number. Since the sort brings all records of a subscriber together, we *scan* the sorted result computing and summing up the costs of all calls of a particular subscriber. The phone companies record up to 300 million transactions per day. AT&T billing system Gecko [4] has to process databases with about 60 billion records, occupying 2.6 terabytes. Certainly this volume can not be sorted in the main memory of a single computer[1]. Therefore we need to sort those huge data sets out-of-memory. Now we show how STXXL can be useful here, since it can handle large volumes I/O efficiently.

4.1 STL Code

If you are familiar with STL your the `main` function of bill generation program will probably look like this:

```
int main(int argc, char * argv[])
{
  if(argc < 4) // check if all parameters are given
  {            // in the command line
          print_usage(argv[0]);
          return 0;
  }
  // open file with the event log
  std::fstream in(argv[1],std::ios::in);
  // create a vector of log entries to read in
  std::vector<LogEntry> v;
  // read the input file and push the records
  // into the vector
  std::copy(std::istream_iterator<LogEntry>(in),
            std::istream_iterator<LogEntry>(),
```

[1]Except may be in the main memory of an expensive *super*computer.

```
             std::back_inserter(v));
  // sort records by callers number
  std::sort(v.begin(),v.end(),SortByCaller());
  // open bill file for output
  std::fstream out(argv[3],std::ios::out);
  // scan the vector and output bills
  std::for_each(v.begin(),v.end(),ProduceBill(out));
  return 0;
}
```

To complete the code we need to define the log entry data type `LogEntry`, input operator >> for `LogEntry`, comparison functor `SortByCaller`, unary functor `ProduceBills` used for computing bills, and the `print_usage` function.

```
#include <algorithm> // for STL std::sort
#include <vector>    // for STL std::vector
#include <fstream>   // for std::fstream
#include <limits>
#include <ctime>     // for time_t type
#define CT_PER_MIN 2 // subscribers pay 2 cent per minute

struct LogEntry // the event log data structure
{
  long long int from; // callers number (64 bit integer)
  long long int to;   // destination number (64 bit int)
  time_t timestamp;   // time of event
  int event;          // event type 1 - call started
                      //            2 - call ended
};

// input operator used for reading from the file
std::istream & operator >> (std::istream & i,
                            LogEntry & entry)
{
  i >> entry.from;
  i >> entry.to;
  i >> entry.timestamp;
  i >> entry.event;
  return i;
}







struct SortByCaller // comparison function
{
  bool operator() (      const LogEntry & a,
                         const LogEntry & b) const
  {
        return  a.from < b.from ||
        (a.from == b.from && a.timestamp < b.timestamp) ||
        (a.from == b.from && a.timestamp == b.timestamp &&
```

```
            a.event < b.event);
  }
  static LogEntry min_value()
  {
    LogEntry dummy;
    dummy.from = (std::numeric_limits<long long int>::min)();
    return dummy;
  }
  static LogEntry max_value()
  {
    LogEntry dummy;
    dummy.from = (std::numeric_limits<long long int>::max)();
    return dummy;
  }

}

// unary function used for producing the bills
struct ProduceBill
{
        std::ostream & out; // stream for outputting
                            // the bills
        unsigned sum;       // current subscribers debit
        LogEntry last;      // the last record

        ProduceBill(std::ostream & o_):out(o_),sum(0)
        {
                last.from = -1;
        }

        void operator () (const LogEntry & e)
        {
                if(last.from == e.from)
                {
                    // either the last event was 'call started'
                    // and current event is 'call ended' or the
                    // last event was 'call ended' and current
                    // event is 'call started'
                    assert( (last.event == 1 && e.event == 2) ||
                            (last.event == 2 && e.event == 1));

                    if(e.event == 2) // call ended
                            sum += CT_PER_MIN*
                            (e.timestamp - last.timestamp)/60;
                }
                else if(last.from != -1)
                {
                   // must be 'call ended'
                   assert(last.event == 2);
                   // must be 'call started'
                   assert(e.event == 1);

                   // output the total sum
                   out << last.from <<"; "<< (sum/100)<<" EUR "
```

```
                              << (sum%100)<< "␣ct"<< std::endl;

                    sum = 0; // reset the sum
                 }

                 last = e;
         }
};


void print_usage(const char * program)
{
          std::cout << "Usage:␣"<<program<<
                "␣logfile␣main␣billfile" << std::endl;
          std::cout <<"␣logfile␣␣-␣file␣name␣of␣the␣input"
                << std::endl;
          std::cout <<"␣main␣␣␣␣␣-␣memory␣to␣use␣(in␣MiB)"
                << std::endl;
          std::cout <<"␣billfile␣-␣file␣name␣of␣the␣output"
                << std::endl;
}
```

measure the running time for in-core and out-of-core case, point the I/O inefficiency of the code

4.2 Going Large – Use STXXL

In order to make the program I/O efficient we will replace the STL internal memory data structures and algorithms by their STXXL counterparts. The changes are underlined.

```
#include <stxxl.h>
// the rest of the code remains the same
int main(int argc, char * argv[])
{
  if(argc < 4) // check if all parameters are given
  {            // in the command line
          print_usage(argv[0]);
          return 0;
  }
  // open file with the event log
  std::fstream in(argv[1],std::ios::in);
  // create a vector of log entries to read in
  stxxl::vector<LogEntry> v;
  // read the input file and push the records
  // into the vector
  std::copy(std::istream_iterator<LogEntry>(in),
            std::istream_iterator<LogEntry>(),
            std::back_inserter(v));
  // bound the main memory consumption by M
  // during sorting
  const unsigned M = atol(argv[2])*1024*1024;
  // sort records by callers number
```

```
  stxxl::sort(v.begin(),v.end(),SortByCaller(),M);
  // open bill file for output
  std::fstream out(argv[3],std::ios::out);
  // scan the vector and output bills
  // the last parameter tells how many buffers
  // to use for overlapping I/O and computation
  stxxl::for_each(v.begin(),v.end(),ProduceBill(out),2);
  return 0;
}
```

As you note the changes are minimal. Only the namespaces and some memory specific parameters had to be changed.

To compile the STXXL billing program you may use the following `Makefile`:

```
all: phonebills
# path to stxxl.mk file
# from your stxxl installation
include ~/stxxl/stxxl.mk

phonebills: phonebills.cpp
        $(STXXL_CXX) -c phonebills.cpp $(STXXL_CPPFLAGS)
        $(STXXL_CXX) phonebills.o -o phonebills.bin $(STXXL_LDLIBS)
clean:
        rm -f phonebills.bin phonebills.o
```

Do not forget to configure you external memory space in file `.stxxl`. You can copy the `config_example` (Windows: `config_example_win`) from the STXXL installation directory, and adapt it to your configuration.

Chapter 5

Design of STXXL

STXXL is a layered library. There are three layers (see Fig. 5.1). The lowest layer, *Asynchronous I/O primitives layer* hides the details of how I/Os are done. In particular, the layer provides abstraction for *asynchronous* read and write operations on a *file*. The completion status of I/O operations is is facilitated by *I/O request* objects returned by read and write file operations. The layer has several implementations of file access for Linux. The fastest one is based on `read` and `write` system calls which operate directly on user space memory pages[1]. To support asynchrony the current Linux implementation of the layer uses standard `pthread` library. Porting STXXL library to a different platform (for example Windows) involves only reimplementing the Asynchronous I/O primitives layer using native file access methods and/or native multithreading mechanisms[2].

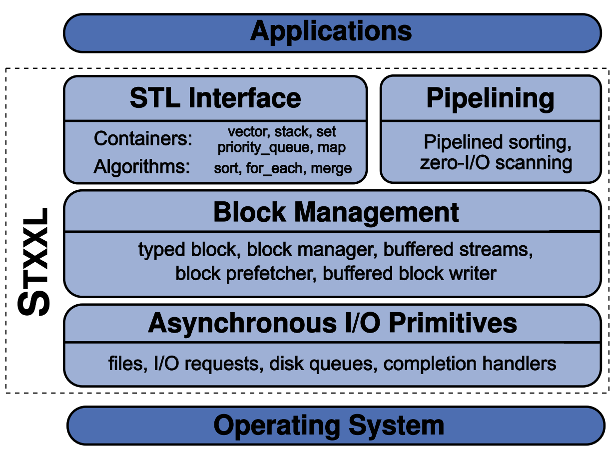

Figure 5.1: The STXXL library structure

The middle layer, *Block management layer* provides a programming interface simulating the *parallel* disk model. The layer provides abstraction for a fundamental concept in the external memory algorithm design – block of elements. Block manager implements block allocation/deallocation allowing several block-to-disk assignment

[1]`O_DIRECT` option when opening a file.
[2]Porting STXXL to Windows platform is not finished yet.

strategies: striping, randomized striping, randomized cycling, etc. The block management layer provides implementation of *parallel* disk buffered writing and optimal prefetching [5], and block caching. The implementations are fully asynchronous and designed to explicitly support overlapping of I/O and computation.

The top of STXXL consists of two modules (see Fig. 5.1). STL-user layer implements the functionality and interfaces of the STL library. The 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 external memory algorithms with mostly *sequential* I/O pattern, i.e. scan, sort, merge, etc. A user algorithm, implemented using this module can save many I/Os[3]. The win is due to an efficient interface, that couples the input and the output of the algorithms-components (scans, sorts, etc.). The output from an algorithm is directly fed into another algorithm as the input, without the need to store it on the disk.

[3]The doubling algorithm for external memory suffix array construction implemented with this module requires only $1/3$ of I/Os which must be performed by an implementation that uses conventional data structures and algorithms (from STXXL STL-user layer, or LEDA-SM, or TPIE).

Chapter 6

STL-User Layer

STXXL library was designed to ease the access to external memory algorithms and data structures for a programmer. We decided to equip our implementations of *out-of-memory* data structure and algorithms with well known generic interfaces of *internal memory* data structures and algorithms from the Standard Template Library. Currently we have implementation of the following data structures (in STL terminology *containers*): `vector`, `stack`, `priority_queue`. We have implemented a *parallel* disk sorter which have syntax of STL `sort` [3]. Our `ksort` is a specialized implementation of `sort` which efficiently sorts elements with integer keys[1]. STXXL currently provides several implementations of scanning algorithms (`generate`, `for_each`, `find`) optimized for external memory. However, it is possible (with some constant factor degradation in the performance) to apply internal memory scanning algorithms from STL to STXXL containers, since STXXL containers have iterator based interface.

STXXL has a restriction that the data types stored in the containers can not have pointers or references to other elements of external memory containers. The reason is that those pointers/references get invalidated when the blocks containing the elements they point/refer to are written on the disks.

6.1 Vector

External memory vector (array) `stxxl::vector` is a data structure that supports random access to elements. The semantics of the basic methods of `stxxl::vector` is kept to compatible with STL `std::vector`. Table 6.1 shows the internal work and the I/O worst case complexity of the `stxxl::vector`.

[1] `ksort` is not STL compatible, it extends the syntax of STL.

Table 6.1: Running times of the basic operations of `stxxl::vector`

	int. work	I/O (worst case)
random access	$O(1)$	$O(1)$
insertion at the end	$O(1)$	$O(1)$
removal at the end	$O(1)$	$O(1)$

6.1.1 The Architecture of `stxxl::vector`

The `stxxl::vector` is organized as a collection of blocks residing on the external storage media (parallel disks). Access to the external blocks is organized through the fully associative *cache* which consist of some fixed amount of in-memory pages[2]. The schema of `stxxl::vector` is depicted in the Fig. 6.1. When accessing an element the implementation of `stxxl::vector` access methods (`[·]` operator, `push_back`, etc.) first checks whether the page to which the requested element belongs is in the vector's cache. If it is the case the reference to the element in the cache is returned. Otherwise the page is brought into the cache[3]. If there was no free space in the cache, then some page is to be written out. Vector maintains a *pager* object, that tells which page to kick out. STXXL provides LRU and random paging strategies. The most efficient and default one is LRU. For each page vector maintains the *dirty* flag, which is set when *non-constant* reference to one of the page's elements was returned. The dirty flag is cleared each time when the page is read into the cache. The purpose of the flag is to track whether any element of the page is modified and therefore the page needs to be written to the disk(s) when it has to be evicted from the cache.

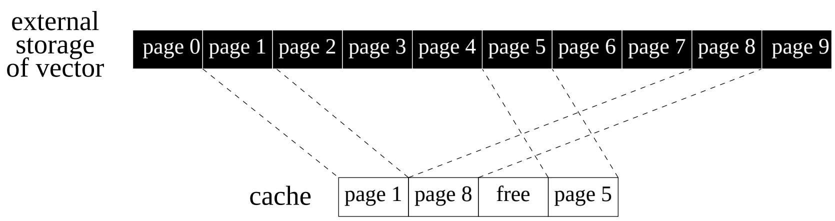

Figure 6.1: The schema of `stxxl::vector` that consists of ten external memory pages and has a cache with the capacity of four pages. The first cache page is mapped to external page 1, the second page is mapped to external page 8, and the fourth cache page is mapped to page 5. The third page is not assigned to any external memory page.

In the worst case scenario when vector elements are read/written in the random order each access takes $2 \times$ *blocks_per_page* I/Os. The factor *two* shows up here because one has to write the replaced from cache page and read the required one). However the scanning of the array costs about n/B I/Os using constant vector iterators or const reference to the vector[4] (read-only access). Using non-const vector access methods leads to $2 \times n/B$ I/Os because every page becomes dirty when returning a non const reference. If one needs only to sequentially write elements to the vector in n/B I/Os the currently fastest method is `stxxl::generate` (see section 6.8.1). Sequential writing to an untouched before vector[5] or alone adding elements at the end of the vector[6] leads also to n/B I/Os.

Example of use

```
stxxl::vector<int> V;
```

[2]The page is a collection of consecutive blocks. The number of blocks in the page is constant.

[3]If the page of the element has not been touched so far, this step is skipped. To keep an eye on such situations there is a special flag for each page.

[4]n is the number of elements to read or write.

[5]For example writing in the vector that has been created using `vector(size_type n)` constructor.

[6]Using `void push_back(const T&)` method.

```
V.push_back(3);
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);
```

6.1.2 stxxl::VECTOR_GENERATOR

Besides the type of the elements `stxxl::vector` has many other template parameters (block size, number of blocks per page, pager class, etc.). To make the configuration of the vector type easier STXXL provides special type generator template meta programs for its containers.

The program for `stxxl::vector` is called `stxxl::VECTOR_GENERATOR`.

Example of use

```
typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;
vector_type V;
V.push_back(3);
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);
```

Table 6.2: Template parameters of `stxxl::VECTOR_GENERATOR` from left to right.

parameter	description	default value	recommended value
Tp_	element type		
PgSz_	number of blocks in a page	4	$\geq D$
Pages_	number of pages in the cache	8	≥ 2
BlkSize_	block size B in bytes	$2 \times 1024 \times 1024$	larger is better
AllocStr_	parallel disk assignment strategy (Table 6.3)	RC	RC
Pager_	paging strategy (Table 6.4)	lru	lru

Table 6.3: Supported parallel disk assignment strategies.

strategy	identifier
striping	striping
simple randomized	SR
fully randomized	FR
randomized cycling	RC

Notes:

- All blocks of a page are read and written from/to disks together. Therefore to increase the I/O bandwidth, it is recommended to set the PgSz_ parameter to multiple of D.

Since there are defaults for the last five of the parameters, it is not necessary to specify them all. **Examples:**

Table 6.4: Supported paging strategies.

strategy	identifier
random	random
least recently used	lru

- `VECTOR_GENERATOR<double>::result` – external vector of **double**'s with four blocks per page, the cache with eight pages, 2 MiB blocks, Random Allocation and lru cache replacement strategy
- `VECTOR_GENERATOR<double,8>::result` – external vector of **double**'s , with **eight** blocks per page, the cache with eight pages, 2 MiB blocks, Random Allocation and lru cache replacement strategy
- `VECTOR_GENERATOR<double,8,2,524288,SR>::result` – external vector of **double**'s, with **eight** blocks per page, the cache with **two** pages, **512 KiB** blocks, **Simple Randomized** allocation and lru cache replacement strategy

6.1.3 Internal Memory Consumption of `stxxl::vector`

The cache of `stxxl::vector` largely dominates in its internal memory consumption. Other members consume very small fraction of `stxxl::vector`s memory even when the vector size is large. Therefore, the internal memory consumption of `stxxl::vector` can be estimated as $BlkSize\_ \times Pages\_ \times PgSz\_$ bytes.

6.1.4 Members of `stxxl::vector`

See Tables 6.5 and 6.6.

Notes:

a) In opposite to STL, `stxxl::vector`'s iterators do not get invalidated when the vector is resized or reallocated.

b) Dereferencing a non-const iterator makes the page of the element to which the iterator points to *dirty*. This causes the page to be written back to the disks(s) when the page is to be kicked off from the cache (additional write I/Os). If you do not want this behavior, use const iterators instead. Example:

```
vector_type V;

// ... fill the vector here

vector_type::iterator iter = V.begin();

// ... advance the iterator
a = *iter; // causes write I/Os,
           // although *iter is not changed
vector_type::const_iterator citer = V.begin();
// ... advance the iterator
a = *citer; // read-only access, causes no write I/Os
```

Table 6.5: Members of `stxxl::vector`. Part 1.

member	description
`value_type`	The type of object, Tp_, stored in the vector.
`pointer`	Pointer to Tp_.
`reference`	Reference to Tp_.
`const_reference`	Const reference to Tp_.
`size_type`	An unsigned 64-bit[7] integral type.
`iterator`	Iterator used to iterate through a vector. See notes a,b.
`const_iterator`	Const iterator used to iterate through a vector. See notes a,b.
`block_type`	type of the block used in disk-memory transfers
`iterator begin()`	Returns an iterator pointing to the beginning of the vector. See notes a,b.
`iterator end()`	Returns an iterator pointing to the end of the vector. See notes a,b.
`const_iterator begin() const`	Returns a const_iterator pointing to the beginning of the vector. See notes a,b.
`const_iterator end() const`	Returns a const_iterator pointing to the end of the vector. See notes a,b.
`size_type size() const`	Returns the size of the vector.
`size_type capacity() const`	Number of elements for which *external* memory has been allocated. `capacity()` is always greater than or equal to `size()`.
`bool empty() const`	true if the vector's size is 0.
`reference operator[](size_type n)`	Returns (the reference to) the n'th element. See note c.
`const_reference operator[](size_type n) const`	Returns (the const reference to) the n'th element. See note c.

```
*citer = b; // does not compile, citer is const
```

c) Non const `[·]` operator makes the page of the element *dirty*. This causes the page to be written back to the disks(s) when the page is to be kicked off from the cache (additional write I/Os). If you do not want this behavior, use const `[·]` operator. For that you need to access the vector via a const reference to it. Example:

```
vector_type V;

// ... fill the vector here
```

Table 6.6: Members of `stxxl::vector`. Part 2.

member	description
`vector()`	Creates an empty vector.
`vector(size_type n)`	Creates a vector with n elements.
`vector(const vector&)`	Not yet implemented
`~vector()`	The destructor.
`void reserve(size_type n)`	If n is less than or equal to `capacity()`, this call has no effect. Otherwise, it is a request for allocation of additional *external* memory. If the request is successful, then `capacity()` is greater than or equal to n; otherwise, `capacity()` is unchanged. In either case, `size()` is unchanged.
`reference front()`	Returns (the reference to) the first element. See note c.
`const_reference front() const`	Returns (the const reference to) the first element. See note c.
`reference back()`	Returns (the reference to) the last element. See note c.
`const_reference back() const`	Returns (the const reference to) the last element. See note c.
`void push_back(const T&)`	Inserts a new element at the end.
`void pop_back()`	Removes the last element.
`void clear()`	Erases all of the elements and deallocates all external memory that vector occupied.
`void flush()`	Flushes the cache pages to the external memory.
`vector (file * from)`	Create the vector from the file. The construction causes no I/O.

```
a = V[index]; // causes write I/Os,
              // although V[index] is not changed

const vector_type & CV = V; // const reference to V
a = CV[index]; // read-only access, can cause no write I/Os
CV[index] = b; // does not compile, CV is const
```

This issue also concerns `front()` and `back()` methods.

6.2 Stacks

Stacks provide only restricted subset of sequence operations: insertion, removal, and inspection of the element at the top of the stack. Stacks are a ”last in first out” (LIFO) data structures: the element at the top of a stack is the one that was most recently added. Stacks does not allow iteration through its elements.

The *I/O efficient* stack is perhaps the simplest external memory data structure. The basic variant of EM stack keeps the top k elements in the main memory buffer, where $k \leq 2B$. If the buffers get empty on a removal call, one block is brought from the disk to the buffers. Therefore at least B removals are required to make one I/O reading a block. Insertions cause no I/Os until the internal buffers get full. In this case to make space the first B elements are written to the disk. Thus a block write happens only after at least B insertions. If we choose the unit of disk transfer to be a multiple of DB (we denote it as a *page*), set the stack buffer size to $2D$ pages, and evenly assign the blocks of a page to disks we obtain the running times shown in Table 6.7.

Table 6.7: Amortized running times of the basic operations of `stxxl::stack`

	int. work	I/O (amortized)
insertion at the end	$O(1)$	$O(1/DB)$
removal at the end	$O(1)$	$O(1/DB)$

STXXL has several implementations of the external memory stack. Each implementation is specialized for a certain access pattern:

- The **Normal** stack (`stxxl::normal_stack`) is a general purpose implementation which is the best if the access pattern to the stack is an irregular mix of push'es and pop's, i.e. the stack grows and shrinks without a certain rule.
- The **Grow-Shrink** stack is a stack that is optimized for an access pattern where the insertions are (almost) not intermixed with the removals, and/or vice versa, the removals are (almost) not intermixed with the insertions. In other words the stack first grows to its maximal size, then it shrinks, then it might again grow, then shrink, and so forth, i.e. the pattern is $(push^{i_j} pop^{r_j})^k$, where $k \in N$, $1 \leq j \leq k$, and i_j, r_j are *large*.
- The **Grow-Shrink2** stack is a "grow-shrink" stack that allows the use of common prefetch and write buffer pools. The pools are shared between several "grow-shrink" stacks.
- The **Migrating** stack is a stack that migrates from internal memory to external when its size exceeds a certain threshold.

6.2.1 `stxxl::normal_stack`

The `stxxl::normal_stack`is a general purpose implementation of the external memory stack. The stack has two pages, the size of the page in blocks is a configuration constant and can be given as a template parameter. The implementation of the methods follows the description given in Section 6.2.

Internal Memory Consumption of `stxxl::normal_stack`

The cache of `stxxl::normal_stack` largely dominates in its internal memory consumption. Other members consume very small fraction of `stxxl::normal_stacks` memory even when the stack size is large. Therefore, the internal memory consumption of `stxxl::normal_stack` can be estimated as $2 \times BlkSize\_ \times PgSz\_$ bytes, where $BlkSize\_$ is the block size and $PgSz\_$ is the page size in blocks (see Section 6.2.5).

Members of `stxxl::normal_stack`

See Table 6.8.

Table 6.8: Members of `stxxl::normal_stack`.

member	description
`value_type`	The type of object, Tp_, stored in the vector.
`size_type`	An unsigned 64-bit[8] integral type.
`block_type`	type of the block used in disk-memory transfers
`bool empty() const`	Returns true if the stack contains no elements, and false otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the stack.
`value_type& top()`	Returns a mutable reference to the element at the top of the stack. Precondition: `empty()` is false.
`const value_type& top() const`	Returns a const reference to the element at the top of the stack. Precondition: `empty()` is false.
`void push(const value_type& x)`	Inserts x at the top of the stack. Postconditions: `size()` will be incremented by 1, and `top()` will be equal to x.
`void pop()`	Removes the element at the top of the stack. Precondition: `empty()` is false. Postcondition: size() will be decremented by 1.
`normal_stack()`	he default constructor. Creates an empty stack.
`template <class stack_type> normal_stack(const stack_type & stack_)`	The copy constructor. Accepts any *stack concept* data type.
`~normal_stack()`	The destructor.

The running times of the push/pop stack operations are given in Table 6.7. Other operations except copy construction perform constant internal work and no I/Os.

6.2.2 stxxl::grow_shrink_stack

The stxxl::grow_shrink_stack stack specialization is optimized for an access pattern where the insertions are (almost) not intermixed with the removals, and/or vice versa, the removals are (almost) not intermixed with the insertions. In other words the stack first grows to its maximal size, then it shrinks, then it might again grow, then shrink, and so forth, i.e. the pattern is $(push^{i_j} pop^{r_j})^k$, where $k \in N$, $1 \leq j \leq k$, and i_j, r_j are *large*. The implementation efficiently exploits the knowledge of the access pattern that allows *prefetching* the blocks beforehand while the stack shrinks and *buffered writing* while the stack grows. Therefore the *overlapping* of I/O and computation is possible.

Internal Memory Consumption of stxxl::grow_shrink_stack

The cache of stxxl::grow_shrink_stack largely dominates in its internal memory consumption. Other members consume very small fraction of stxxl::grow_shrink_stack's memory even when the stack size is large. Therefore, the internal memory consumption of stxxl::grow_shrink_stack can be estimated as $2 \times BlkSize\_ \times PgSz\_$ bytes, where $BlkSize\_$ is the block size and $PgSz\_$ is the page size in blocks (see Section 6.2.5).

Members of stxxl::grow_shrink_stack

The stxxl::grow_shrink_stackhas the same set of members as the stxxl::normal_stack (see Table 6.8). The running times of stxxl::grow_shrink_stack are the same as stxxl::normal_stack except that when the stack switches from growing to shrinking (or from shrinking to growing) $PgSz\_$ I/Os can be spent additionally in the worst case.[9]

6.2.3 stxxl::grow_shrink_stack2

The stxxl::grow_shrink_stack2 is optimized for the same kind of access pattern as stxxl::grow_shrink_stack. The difference is that each instance of stxxl::grow_shrink_stack uses an own internal buffer to overlap I/Os and computation, but stxxl::grow_shrink_stack2 is able to share the buffers from the pool used by several stacks.

Internal Memory Consumption of stxxl::grow_shrink_stack2

Not counting the memory consumption of the shared blocks from the pools, the stack alone consumes about $BlkSize\_$ bytes.[10]

Members of stxxl::grow_shrink_stack2

The stxxl::grow_shrink_stack2 has almost the same set of members as the stxxl::normal_stack (Table 6.8), except that it does not have the default constructor. The stxxl::grow_shrink_stack2 requires prefetch and write pool

[9] This is for the single disk setting, if the page is perfectly striped over parallel disk the number of I/Os is $PgSz\_/D$.

[10] It has the cache that consists of only a single block.

objects (see Sections 8.1.1 and 8.1.2 for the documentation for the pool classes) to be specified in the creation time. The new members are listed in Table 6.9.

Table 6.9: New members of `stxxl::grow_shrink_stack2`.

member	description
`grow_shrink_stack2 (prefetch_pool< block_type > & p_pool_, write_pool< block_type > &w_pool_, unsigned prefetch_aggressiveness=0)`	Constructs stack, that will use `p_pool_` for prefetching and `w_pool_` for buffered writing. `prefetch_aggressiveness` parameter tells how many blocks from the prefetch pool the stack is allowed to use.
`void set_prefetch_aggr (unsigned new_p)`	Sets level of prefetch aggressiveness (number of blocks from the prefetch pool used for prefetching).
`unsigned get_prefetch_aggr () const`	Returns the number of blocks used for prefetching.

6.2.4 `stxxl::migrating_stack`

The `stxxl::migrating_stack` is a stack that migrates from internal memory to external when its size exceeds a certain threshold (template parameter). The implementation of internal and external memory stacks can be arbitrary and given as a template parameters.

Internal Memory Consumption of `stxxl::migrating_stack`

The `stxxl::migrating_stack` memory consumption depends on the memory consumption of the stack implementations given as template parameters. The the current state is internal (external), the `stxxl::migrating_stack` consumes almost exactly the same space as internal (external) memory stack implementation.[11]

Members of `stxxl::migrating_stack`

The `stxxl::migrating_stack` extends the member set of `stxxl::normal_stack` (Table 6.8). The new members are listed in Table 6.10.

6.2.5 `stxxl::STACK_GENERATOR`

To provide an easy way to choose and configure the `stxxl::stack` implementations STXXL offers a template meta program called `stxxl::STACK_GENERATOR`. See Table 6.11.

Example:

[11]The `stxxl::migrating_stack` needs only few pointers to maintain the switching from internal to external memory implementations.

Table 6.10: New members of `stxxl::migrating_stack`.

member	description
`bool internal () const`	Returns true if the current implementation is internal, otherwise false.
`bool external () const`	Returns true if the current implementation is external, otherwise false.

```
typedef stxxl::STACK_GENERATOR<int>::result stack_type;

int main()
{
  stack_type S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

Example for `stxxl::grow_shrink_stack2` :

```
typedef STACK_GENERATOR<int,external,grow_shrink2>::result stack_type;
typedef stack_type::block_type block_type;

stxxl::prefetch_pool p_pool(10); // 10 read buffers
stxxl::write_pool w_pool(6);     // 6 write buffers
stack_type S(p_pool,w_pool,0);   // no read buffers used

for(long long i=0;i < max_value;++i)
     S.push(i);

S.set_prefetch_aggressiveness(5);
/* give a hint that we are going to
   shrink the stack from now on,
   always prefetch 5 buffers
   beforehand */
```

Table 6.11: Template parameters of `stxxl::STACK_GENERATOR` from left to right.

parameter	description	default value	recommended value
ValTp	element type		
Externality	tells whether the vector is internal, external, or migrating (Table 6.12)	external	
Behavior	chooses *external* implementation (Table 6.13)	normal	
BlocksPerPage	defines how many blocks has one page of internal cache of an *external* implementation	4	$\geq D$
BlkSz	external block size in bytes	$2 \times 1024 \times 1024$	larger is better
IntStackTp	type of internal stack (used for the migrating stack)	`std::stack<ValTp>`	
MigrCritSize	threshold value for number of elements when `migrating_stack` migrates to the external memory	$2 \times BlocksPerPage \times BlkSz$	
AllocStr	parallel disk assignment strategy (Table 6.3)	RC	RC
SzTp	size type	`off_t`	`off_t`

Table 6.12: The Externality parameter.

identifier	comment
internal	chooses IntStackTp implementation
external	external container, implementation is chosen according to the Behavior parameter
migrating	migrates from internal implementation given by IntStackTp parameter to external implementation given by Behavior parameter when size exceeds MigrCritSize

```
for(long long i=0; i< max_value;++i)
        S.pop();

S.set_prefetch_aggressiveness(0);
// stop prefetching
```

6.3 Priority Queue

A priority queue is a data structure that provides a restricted subset of container functionality: it provides insertion of elements, and inspection and removal of the top element. It is guaranteed that the top element is the largest element in the priority

Table 6.13: The Behavior parameter.

identifier	comment
normal	conservative version, implemented in `stxxl::normal_stack`
grow_shrink	chooses `stxxl::grow_shrink_stack`
grow_shrink2	chooses `stxxl::grow_shrink_stack2`

queue, where the function object `Cmp_` is used for comparisons. Priority queue does not allow iteration through its elements.

STXXL priority queue is an external memory implementation of [6]. The difference to the original design is that the last merge groups keep their sorted sequences in the external memory. The running times of `stxxl::priority_queue` data structure is given in Table 6.14. The theoretic guarantees on I/O performance are given only for a single disk setting, however the queue also performs well in practice for multi-disk configuration.

Table 6.14: Amortized running times of the basic operations of `stxxl::priority_queue` in terms of $I =$ the number of performed operations.

	int. work	I/O (amortized)
insertion	$O(\log I)$	$O(1/B)$
deletion	$O(\log I)$	$O(1/B)$

6.3.1 Members of `stxxl::priority_queue`

See Table 6.15.

6.3.2 `stxxl::PRIORITY_QUEUE_GENERATOR`

Since the `stxxl::priority_queue` has many setup parameters (internal memory buffer sizes, arity of mergers, number of internal and external memory merger groups, etc.) which are difficult to guess, STXXL provides a helper meta template program that searches for the optimum settings for user demands. The program is called `stxxl::PRIORITY_QUEUE_GENERATOR`. The parameter of the program are given in Table 6.16.

Notes:

a) If `Cmp_(x,y)` is true, then x is smaller than y. The element returned by `Q.top()` is the largest element in the priority queue. That is, it has the property that, for every other element x in the priority queue, `Cmp_(Q.top(), x)` is false. `Cmp_` must also provide `min_value` method, that returns value of type `Tp_` that is smaller than any element of the queue x , i.e. `Cmp_(Cmp_.min_value(),x))` is always true.

   Example, a comparison object for priority queue where `top()` returns the *smallest* contained integer:

```
struct CmpIntGreater
{
  bool operator () (const int & a, const int & b)
  { return a<b; }
  int min_value() const
  { return (std::numeric_limits<int>::max)(); }
};
```

Example, a comparison object for priority queue where `top()` returns the *largest* contained integer:

```
struct CmpIntLess: public std::less<int>
{
   int min_value() const
   { return (std::numeric_limits<int>::min)(); }
};
```

Note that `Cmp_` must define the Strict Weak Ordering.

b) Example: if you are sure that priority queue contains no more than one million elements any time, then the right parameter for you is $(1000000/1024) = 976$.

c) Try to play with the Tune_ parameter if the your code does not compile (larger than default value 6 might help). The reason that the code does not compile is that no suitable internal parameters were found for given IntM_ and MaxS_. It might also happen that given IntM_ is too small for given MaxS_, try larger values.

   `PRIORITY_QUEUE_GENERATOR` searches for 7 configuration parameters of `stxxl::priority_queue` that both minimize internal memory consumption of the priority queue to match IntM_ and maximize the performance of priority queue operations. Actual memory consumption might be slightly larger (use `stxxl::priority_queue::mem_cons()` method to track it), since the search assumes rather optimistic schedule of push'es and pop'es for the estimation of the maximum memory consumption. To keep actual memory requirements low, increase the value of MaxS_ parameter.

d) For the functioning, a priority queue object requires two pools of blocks (See the constructor of `priority_queue`). To construct STXXL block pools you need the block type that is used by priority queue. Block's size and hence it's type is generated by the `PRIORITY_QUEUE_GENERATOR` in compile type from IntM_, MaxS_ and `sizeof(Tp_)` and it can not be given directly by the user as a template parameter. The block type can be accessed as `PRIORITY_QUEUE_GENERATOR<parameters>::result::block_type`.

Example:

```
struct Cmp
{
  bool operator () (const int & a,
                    const int & b) const
  { return a>b; }
  int min_value() const
```

```
  { return (std::numeric_limits<int>::max)(); }
};

typedef stxxl::PRIORITY_QUEUE_GENERATOR<int,
                                        Cmp,
/* use 64 MiB on main memory */         64*1024*1024,
/* 1 billion items at most  */          1024*1024
                                        >::result pq_type;
typedef pq_type::block_type block_type;


int main() {
  // use 10 block read and write pools
  // for enable overlapping of I/O and
  // computation
  stxxl::prefetch_pool<block_type> p_pool(10);
  stxxl::write_pool<block_type>    w_pool(10);

  pq_type Q(p_pool,w_pool);
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);

  assert(Q.top() == 8);
  Q.pop();

  assert(Q.top() == 7);
  Q.pop();

  assert(Q.top() == 5);
  Q.pop();

  assert(Q.top() == 4);
  Q.pop();

  assert(Q.top() == 2);
  Q.pop();

  assert(Q.top() == 1);
  Q.pop();

  assert(Q.empty());
}
```

6.3.3 Internal Memory Consumption of `stxxl::priority_queue`

Internal memory consumption of `stxxl::priority_queue` is bounded by the IntM_ parameter in most situations.

6.4 STXXL Algorithms

Iterators of `stxxl::vector` are STL compatible. `stxxl::vector::iterator` is a model of Random Access Iterator concept from STL. Therefore it is possible to use the `stxxl::vector` iterator ranges with STL algorithms. However such use is not I/O efficient if an algorithm accesses the sequence in a random order. For such kind of algorithms STXXL provides I/O efficient implementations described in this chapter (Sections 6.5–6.7). If an algorithm does only a scan (or a constant number of scans) of a sequence (or sequences) the implementation that calls STL algorithm is nevertheless I/O efficient. However one can save constant factors in I/O volume and internal work if the the access pattern is known (read-only or write-only scan for example). This knowledge is used in STXXL specialized implementations of STL algorithms (Section 6.8).

Example: STL Algorithms Running on STXXL containers

```
typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;

// Replace every number in an array with its negative.
const int N = 1000000000;
vector_type A(N);
std::iota(A.begin(), A.end(), 1);
std::transform(A, A+N, A, negate<double>());

// Calculate the sum of two vectors,
// storing the result in a third vector.

const int N = 1000000000;
vector_type V1(N);
vector_type V2(N);
vector_type V3(N);

std::iota(V1.begin(), V1.end(), 1);
std::fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() &&
       V3.size() >= V1.size());
std::transform(V1.begin(),
               V1.end(),
               V2.begin(),
               V3.begin(),
               plus<int>());
```

6.5 Sorting

`stxxl::sort` is an external memory equivalent to STL `std::sort`. The design and implementation of the algorithm is described in detail in [3].

Prototype

```
template < typename ExtIterator_,
           typename StrictWeakOrdering_
         >
void sort ( ExtIterator_        first,
            ExtIterator_        last,
            StrictWeakOrdering_ cmp,
            unsigned            M
          )
```

Description

`stxxl::sort` sorts the elements in [first, last) into ascending order, meaning that if `i` and `j` are any two valid iterators in [first, last) such that `i` precedes `j`, then `*j` is not less than `*i`. Note: as `std::sort`, `stxxl::sort` is not guaranteed to be stable. That is, suppose that `*i` and `*j` are equivalent: neither one is less than the other. It is not guaranteed that the relative order of these two elements will be preserved by `stxxl::sort`.

The order is defined by the `cmp` parameter. The sorter's internal memory consumption is bounded by `M` bytes.

Requirements on Types

- `ExtIterator_` is a model of External Random Access Iterator[13].
- `ExtIterator_` is mutable.
- `StrictWeakOrdering_` is a model of Strict Weak Ordering and must provide min and max values for the elements in the input:
  - `max_value` method that returns an object that is *strictly greater* than all other objects of user type according to the given ordering.
  - `min_value` method that returns an object that is *strictly less* than all other objects of user type according to the given ordering.

  Example: a comparison object for ordering integer elements in the ascending order

```
struct CmpIntLess: public std::less<int>
{
   static int min_value() const
   { return (std::numeric_limits<int>::min)(); }
   static int max_value() const
   { return (std::numeric_limits<int>::max)(); }
};
```

  Example: a comparison object for ordering integer elements in the descending order

[13] In STXXL currently only `stxxl::vector` provides iterators that are models of External Random Access Iterator.

```
struct CmpIntGreater: public std::greater<int>
{
   int min_value() const
   { return (std::numeric_limits<int>::max)(); }
   int max_value() const
   { return (std::numeric_limits<int>::min)(); }
};
```

Note, that according to the `stxxl::sort` requirements `min_value` and `max_value` **can not** be present in the input sequence.

- `ExtIterator_`'s value type is convertible to `StrictWeakOrdering_`'s argument type.

Preconditions

[first, last) is a valid range.

Complexity

- Internal work: $O(N \log N)$, where
  $N = (last - first) \cdot$ `sizeof(ExtIterator_::value_type)`.

- I/O complexity: $(2N/DB)(1 + \lceil \log_{M/B}(2N/M) \rceil)$ I/Os

`stxxl::sort` chooses the block size (parameter B) equal to the block size of the container, the last and first iterators pointing to (e.g. `stxxl::vector`'s block size).

The second term in the I/O complexity accounts for the merge phases of the external memory sorting algorithm [3]. Avoiding multiple merge phases speeds up the sorting. In practice one should choose the block size B of the container to be sorted such that there is only one merge phase needed: $\lceil \log_{M/B}(2N/M) \rceil) = 1$. This is possible for $M > DB$ and $N < M^2/2DB$. But still this restriction gives a freedom to choose a variety of blocks sizes. The study [3] has shown that optimal B for sorting lies in the range $[M^2/(4N), 3M^2/(8N)]$. With such choice of the parameters the `stxxl::sort` always performs $4N/DB$ I/Os.

Internal Memory Consumption

The `stxxl::sort` consumes slightly more than M bytes of internal memory.

External Memory Consumption

The `stxxl::sort` is not in-place. It requires about N bytes of external memory to store the sorted runs during the sorting process [3]. After the sorting this memory is freed.

Example

```
struct MyCmp: public std::less<int> // ascending
{                                    // order
   static int min_value() const
   { return (std::numeric_limits<int>::min)(); }
   static int max_value() const
   { return (std::numeric_limits<int>::max)(); }
};
typedef stxxl::VECTOR_GENERATOR<int>::result vec_type;

vec_type V;
// ... fill here the vector with some values

/*
   Sort in ascending order
   use 512 MiB of main memory
*/
stxxl::sort(V.begin(),V.end(),MyCmp(),512*1024*1024);
// sorted
```

6.6 Sorted Order Checking

STXXL gives an ability to automatically check the order in the output of STXXL [14] sorters and intermediate results of sorting (the order and a meta information in the sorted runs). The check is switched on if the source codes and the library are compiled with the option `-DSTXXL_CHECK_ORDER_IN_SORTS` and the option `-DNDEBUG` is not used. For details see the `compiler.make` file in the STXXL tar ball. Note, that the checking routines require more internal work as well as additional N/DB I/Os to read the sorted runs. Therefore for the final non-debug version of a user application on should switch this option off.

6.7 Sorting Using Integer Keys

`stxxl::ksort` is a specialization of external memory sorting optimized for records having integer keys.

Prototype

```
template < typename ExtIterator_>
void ksort ( ExtIterator_ first,
             ExtIterator_ last,
             unsigned     M
           )

template < typename ExtIterator_, typename KeyExtractor_>
void ksort ( ExtIterator_  first,
```

[14] This checker checks the `stxxl::sort`, `stxxl::ksort` (Section 6.7), and the pipelined sorter from Section 7.6.

```
            ExtIterator_  last,
            KeyExtractor_ keyobj,
            unsigned      M
          )
```

Description

`stxxl::ksort` sorts the elements in [first, last) into ascending order, meaning that if `i` and `j` are any two valid iterators in [first, last) such that `i` precedes `j`, then `*j` is not less than `*i`. Note: as `std::sort`and `stxxl::sort`, `stxxl::ksort` is not guaranteed to be stable. That is, suppose that `*i` and `*j` are equivalent: neither one is less than the other. It is not guaranteed that the relative order of these two elements will be preserved by `stxxl::ksort`.

The two versions of `stxxl::ksort` differ in how they define whether one element is less than another. The first version assumes that the elements have `key()` member function that returns an integral key (32 or 64 bit), as well as the minimum and the maximum element values. The second version compares objects extracting the keys using `keyobj` object, that is in turn provides min and max element values.

The sorter's internal memory consumption is bounded by `M` bytes.

Requirements on Types

- `ExtIterator_` is a model of External Random Access Iterator[15].
- `ExtIterator_` is mutable.
- `KeyExtractor_` must implement `operator ()` that extracts the key of an element and provide min and max values for the elements in the input:
  - `key_type` typedef for the type of the keys.
  - `max_value` method that returns an object that is *strictly greater* than all other keys of the elements in the input.
  - `min_value` method that returns an object that is *strictly less* than all other keys of the elements in the input.

  Example: a key extractor object for ordering elements having 64 bit integer keys:

```
struct MyType
{
   typedef unsigned long long key_type;
   key_type _key;
   char _data[32];
   MyType() {}
   MyType(key_type __key):_key(__key) {}
};
struct GetKey
{
   typedef MyType::key_type key_type;
```

[15] In STXXL currently only `stxxl::vector` provides iterators that are models of External Random Access Iterator.

```
    key_type operator() (const MyType & obj)
    { return obj._key; }
    MyType min_value() const
    { return MyType(
        (std::numeric_limits<key_type>::min)()); }
    MyType max_value() const
    { return MyType(
        (std::numeric_limits<key_type>::max)()); }
};
```

Note, that according to the `stxxl::sort` requirements `min_value` and `max_value` **can not** be present in the input sequence.

- `ExtIterator_`'s value type is convertible to `KeyExtractor_`'s argument type.
- `ExtIterator_`'s value type has a typedef `key_type`.
- For the first version of `stxxl::ksort` `ExtIterator_`'s value type must have the `key()` function that returns the key value of the element, and the `min_value()` and `max_value()` member functions that return minimum and maximum element values respectively. Example:

```
struct MyType
{
    typedef unsigned long long key_type;
    key_type _key;
    char _data[32];
    MyType() {}
    MyType(key_type __key):_key(__key) {}
    key_type key() { return _key; }
    MyType min_value() const
    { return MyType(
        (std::numeric_limits<key_type>::min)()); }
    MyType max_value() const
    { return MyType(
        (std::numeric_limits<key_type>::max)()); }
};
```

Preconditions

The same as for `stxxl::sort` (section 6.5).

Complexity

The same as for `stxxl::sort` (Section 6.5).

Internal Memory Consumption

The same as for `stxxl::sort` (Section 6.5)

External Memory Consumption

The same as for `stxxl::sort` (Section 6.5).

Example

```
struct MyType
{
   typedef unsigned long long key_type;
   key_type _key;
   char _data[32];
   MyType() {}
   MyType(key_type __key):_key(__key) {}
   key_type key() { return obj._key; }
   static MyType min_value() const
   { return MyType(
        (std::numeric_limits<key_type>::min)()); }
   static MyType max_value() const
   { return MyType(
        (std::numeric_limits<key_type>::max)()); }
};

typedef stxxl::VECTOR_GENERATOR<MyType>::result vec_type;

vec_type V;
// ... fill here the vector with some values

/*
   Sort in ascending order
   use 512 MiB of main memory
*/
stxxl::ksort(V.begin(),V.end(),512*1024*1024);
// sorted
```

6.8 Other STXXL Algorithms

STXXL offers several specializations of STL algorithms for `stxxl::vector` iterators. The algorithms while accessing the elements bypass the vector's cache and access the vector's blocks directly. Another improvement is that algorithms from this chapter are able to overlap I/O and computation. With standard STL algorithms the overlapping is not possible. This measures save constant factors both in I/O volume and internal work.

6.8.1 `stxxl::generate`

The semantics of the algorithm is equivalent to the STL `std::generate`.

Prototype

```
template<typename ExtIterator, typename Generator>
void generate ( ExtIterator first,
                ExtIterator last,
                Generator gen,
                int nbuffers
              )
```

Description

Generate assigns the result of invoking `gen`, a function object that takes no arguments, to each element in the range [first, last). To overlap I/O and computation `nbuffers` are used (a value at least D is recommended). The size of the buffers is derived from the container that is pointed by the iterators.

Requirements on types

- `ExtIterator` is a model of External Random Access Iterator.
- `ExtIterator` is mutable.
- `Generator` is a model of STL Generator.
- `Generator`'s result type is convertible to `ExtIterator`'s value type.

Preconditions

[first, last) is a valid range.

Complexity

- Internal work is linear.
- External work: close to N/DB I/Os (write-only).

Example

```
// Fill a vector with random numbers, using the
// standard C library function rand.
typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;
vector_type V(some_size);
// use 20 buffer blocks
stxxl::generate(V.begin(), V.end(), rand, 20);
```

6.8.2 `stxxl::for_each`

The semantics of the algorithm is equivalent to the STL `std::for_each`.

Prototype

```
template<typename ExtIterator, typename UnaryFunction>
UnaryFunction for_each ( ExtIterator   first,
                         ExtIterator   last,
                         UnaryFunction f,
                         int nbuffers
                       )
```

Description

`stxxl::for_each` applies the function object `f` to each element in the range [first, last); `f`'s return value, if any, is ignored. Applications are performed in forward order, i.e. from first to last. `stxxl::for_each` returns the function object after it has been applied to each element. To overlap I/O and computation `nbuffers` are used (a value at least D is recommended). The size of the buffers is derived from the container that is pointed by the iterators.

Requirements on types

- `ExtIterator` is a model of External Random Access Iterator.
- `UnaryFunction` is a model of STL Unary Function.
- `UnaryFunction` does not apply any non-constant operations through its argument.
- `ExtIterator`'s value type is convertible to `UnaryFunction`'s argument type.

Preconditions

[first, last) is a valid range.

Complexity

- Internal work is linear.
- External work: close to N/DB I/Os (read-only).

Example

```
template<class T> struct print :
     public unary_function<T, void>
{
  print(ostream& out) : os(out), count(0) {}
  void operator() (T x) { os << x << '␣'; ++count; }
  ostream& os;
  int count;
};
typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;
```

```
int main()
{
  vector_type A(N);
  // fill A with some values
  // ...

  print<int> P = stxxl::for_each(A.begin(), A.end(),
                                 print<int>(cout));
  cout << endl << P.count << "␣objects␣printed." << endl;
}
```

6.8.3 stxxl::for_each_m

stxxl::for_each_m is a *mutating* version of stxxl::for_each, i.e. the restriction that Unary Function f can not apply any non-constant operations through its argument does not exist.

Prototype

```
template<typename ExtIterator, typename UnaryFunction>
UnaryFunction for_each ( ExtIterator   first,
                         ExtIterator   last,
                         UnaryFunction f,
                         int nbuffers
                       )
```

Description

stxxl::for_each applies the function object f to each element in the range [first, last); f's return value, if any, is ignored. Applications are performed in forward order, i.e. from first to last. stxxl::for_each returns the function object after it has been applied to each element. To overlap I/O and computation nbuffers are used (a value at least $2D$ is recommended). The size of the buffers is derived from the container that is pointed by the iterators.

Requirements on types

- ExtIterator is a model of External Random Access Iterator.
- UnaryFunction is a model of STL Unary Function.
- ExtIterator's value type is convertible to UnaryFunction's argument type.

Preconditions

[first, last) is a valid range.

Complexity

- Internal work is linear.
- External work: close to $2N/DB$ I/Os (read and write).

Example

```
struct AddX
{
  int x;
  AddX(int x_): x(x_) {}
  void operator() (int & val)
  { val += x; }
};

typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;
int main()
{
  vector_type A(N);
  // fill A with some values
  // ...

  // Add 5 to each value in the vector
  stxxl::for_each(A.begin(), A.end(), AddX(5));
}
```

6.8.4 `stxxl::find`

The semantics of the algorithm is equivalent to the STL `std::find`.

Prototype

```
template< typename ExtIterator,
          typename EqualityComparable>
ExtIterator find ( ExtIterator                  first,
                   ExtIterator                  last,
                   const EqualityComparable & value,
                   int                          nbuffers
                 )
```

Description

Returns the first iterator `i` in the range [first, last) such that `*i == value`. Returns last if no such iterator exists. To overlap I/O and computation `nbuffers` are used (a value at least D is recommended). The size of the buffers is derived from the container that is pointed by the iterators.

Requirements on types

a) `EqualityComparable` is a model of STL EqualityComparable concept.

b) `ExtIterator` is a model of External Random Access Iterator.

c) Equality is defined between objects of type `EqualityComparable` and objects of `ExtIterator`'s value type.

Preconditions

[first, last) is a valid range.

Complexity

- Internal work is linear.
- External work: close to N/DB I/Os (read-only).

Example

```
typedef stxxl::VECTOR_GENERATOR<int>::result vector_type;

vector_type V;
// fill the vector


// find 7 in V
vector_type::iterator result = find(V.begin(), V.end(), 7);
if(result != V.end())
 std::cout << ``Found at position ''<<
      (result - V.begin()) << std::endl;
else
 std::cout << ``Not found'' << std::endl;
```

Table 6.15: Members of `stxxl::priority_queue`.

member	description
`value_type`	The type of object, Tp_, stored in the vector.
`size_type`	An unsigned 64-bit[12] integral type.
`block_type`	type of the block used in disk-memory transfers
`priority_queue( prefetch_pool<block_type>& p_pool_, write_pool<block_type>& w_pool_)`	Creates an empty priority queue. Prefetch pool `p_pool_` and write pools `w_pool_` will be used for overlapping of I/O and computation during external memory merging (see Sections 8.1.1 and 8.1.2 for the documentation for the pool classes).
`bool empty() const`	Returns true if the `priority_queue` contains no elements, and false otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `priority_queue`.
`const value_type& top() const`	Returns a const reference to the element at the top of the `priority_queue`. The element at the top is guaranteed to be the largest element in the priority queue, as determined by the comparison function `Cmp_`. That is, for every other element x in the `priority_queue`, `Cmp_(Q.top(), x)` is false. Precondition: `empty()` is false.
`void push(const value_type& x)`	Inserts x into the `priority_queue`. Postcondition: `size()` will be incremented by 1.
`void pop()`	Removes the element at the top of the `priority_queue`, that is, the largest element in the `priority_queue`. Precondition: `empty()` is false. Postcondition: `size()` will be decremented by 1.
`unsigned mem_cons () const`	Returns number of bytes consumed by the `priority_queue` in the internal memory not including the pools.
`~priority_queue()`	The destructor. Deallocates all occupied internal and external memory.

Table 6.16: Template parameters of `stxxl::PRIORITY_QUEUE_GENERATOR` from left to right.

parameter	description	default value	recommended value
Tp_	element type		
Cmp_	the comparison type used to determine whether one element is smaller than another element. See note a.		
IntM_	upper limit for internal memory consumption in bytes		larges is better
MaxS_	upper limit for number of elements contained in the priority queue (in units of 1024 items). See note b.		
Tune_	a tuning parameter. See note c.	6	

Chapter 7

Pipelined/Stream Interfaces

7.1 Preliminaries

7.2 Node Interface

7.3 Scheduling

7.4 File Nodes – `streamify` and `materialize`

7.5 Streaming Nodes

7.6 Sorting Nodes

7.6.1 Runs Creator – `stxxl::stream::runs_creator`

7.6.2 Specializations of `stxxl::stream::runs_creator`

7.6.3 Runs Merger – `stxxl::stream::runs_merger`

7.6.4 A Combination: `stxxl::stream::sort`

7.7 A Pipelined Version of the Billing Application

Chapter 8

Internals

8.1 Block Management Layer

8.1.1 `stxxl::prefetch_pool`

8.1.2 `stxxl::write_pool`

8.2 I/O Primitives Layer

8.3 Utilities

Chapter 9

Miscellaneous

9.1 STXXL Compile Flags

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

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

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