

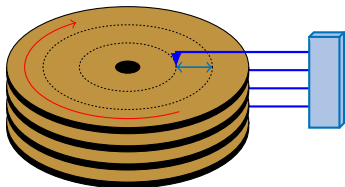
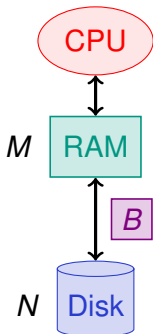
# STXXL 1.4.0 and Beyond: External Memory Algorithms

Timo Bingmann | June 22nd, 2014 @ 3rd LSDMA Topical Meeting

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS



# External Memory (EM) Model

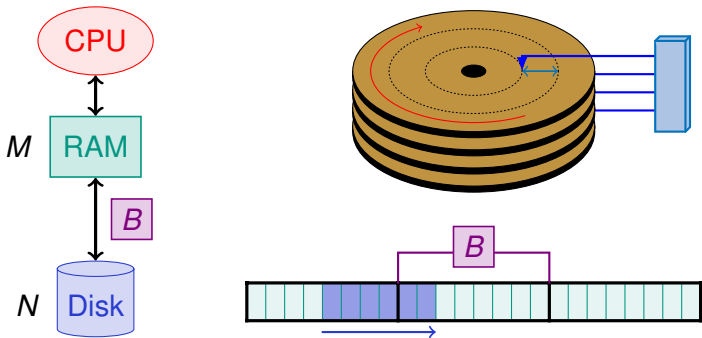


$$M = 1 \text{ GiB} = 2^{30}$$

$$B = 1 \text{ MiB} = 2^{20}$$

$$N = 1 \text{ TiB} = 2^{40}$$

# External Memory (EM) Model

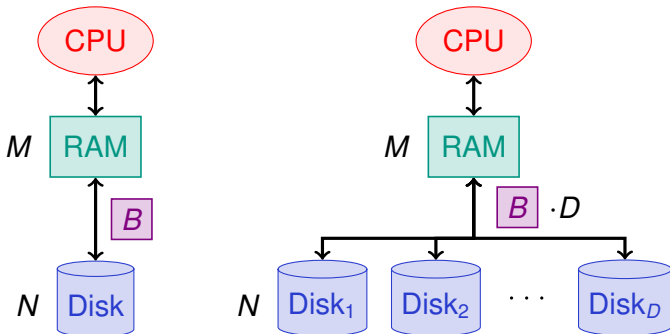


Scanning:  $\frac{N}{B}$  I/Os  $M = 1 \text{ GiB} = 2^{30}$

Random Access:  $N$  I/Os  $B = 1 \text{ MiB} = 2^{20}$

Sorting:  $O\left(\frac{N}{B} \lceil \log_{\frac{M}{B}} \frac{N}{M} \rceil\right)$  I/Os  $N = 1 \text{ TiB} = 2^{40}$

# External Memory (EM) Model



Scanning:  $\frac{N}{DB}$  I/Os

Random Access:  $N$  I/Os

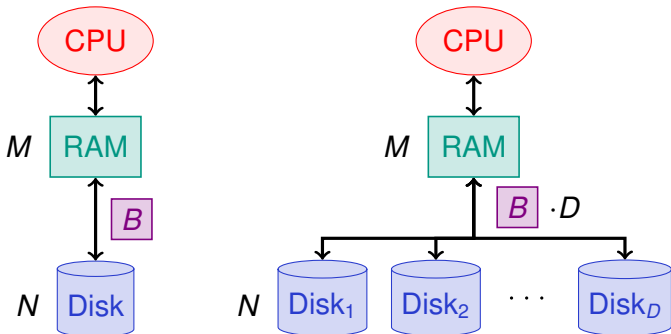
Sorting:  $\mathcal{O}\left(\frac{N}{DB} \lceil \log_{\frac{M}{B}} \frac{N}{M} \rceil\right)$  I/Os

$M = 1 \text{ GiB} = 2^{30}$

$B = 1 \text{ MiB} = 2^{20}$

$N = 1 \text{ TiB} = 2^{40}$

# External Memory (EM) Model



Scanning:  $\mathcal{O}\left(\frac{N}{DB}\right)$  I/Os

Random Access:  $\mathcal{O}(N)$  I/Os

Sorting:  $\mathcal{O}\left(\frac{N}{DB} \lceil \log_{\frac{M}{B}} \frac{N}{M} \rceil\right)$  I/Os

$M = 1 \text{ GiB} = 2^{30}$

$B = 1 \text{ MiB} = 2^{20}$

$N = 1 \text{ TiB} = 2^{40}$

## Basic Properties:

- C++ [template library](#) of efficient external memory algorithms
- Licensed under the [Boost Software License](#), Version 1.0
- Supports [Linux](#), [Windows](#), and [Mac OS X](#).
- Primary authors: Roman Dementiev, Johannes Singler, Andreas Beckmann. Initiator: Peter Sanders.

## Main Features:

- **STL-compatible** containers and algorithms.
- Efficient, highly optimized **sorting** implementation.
- Transparent **parallel disk** support.
- Pipelined Sorting, efficient priority queue, matrix operations.
- **Partially parallelized** shared memory algorithms.

## Containers:

- `std::vector<T>`  
Operations: `push_back()`, `operator[]`, `begin()`, ...
- `std::stack<T>`, `std::queue<T>`, `std::deque<T>`  
Operations: `push()`, `top()`, `pop()`, ...
- `std::priority_queue<T, C, Cmp>`  
Operations: `push()`, `top()`, `pop()`, ...
- `std::map<K, V>`, `std::set<K>`  
Operations: `insert()`, `find()`, ...

## Algorithms:

- `std::sort(begin, end, Cmp)`
- `std::for_each(begin, end, Functor)`



## Containers:

- `stxxl::vector<T>`  
Operations: `push_back()`, `operator[]`, `begin()`, ...
- `stxxl::stack<T>`, `stxxl::queue<T>`, `stxxl::deque<T>`  
Operations: `push()`, `top()`, `pop()`, ...
- `stxxl::priority_queue<T, C, Cmp>`  
Operations: `push()`, `top()`, `pop()`, ...
- `stxxl::map<K, V>`, `stxxl::set<K>`  
Operations: `insert()`, `find()`, ...

## Algorithms:

- `stxxl::sort(begin, end, Cmp, memory)`
- `stxxl::for_each(begin, end, Functor, buffers)`

# stxxl::vector Architecture

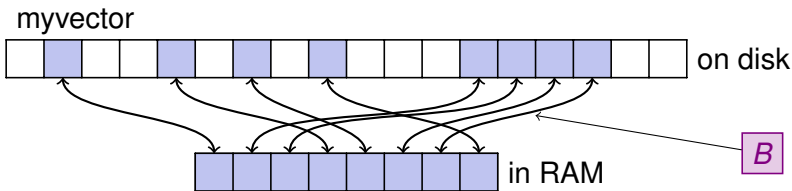
```
struct MyData { int a, b; };  
std::vector<MyData> myvector;  
for (int i = 0; i < N; i++) {  
    myvector[i].a = 42 + i;  
}
```

myvector

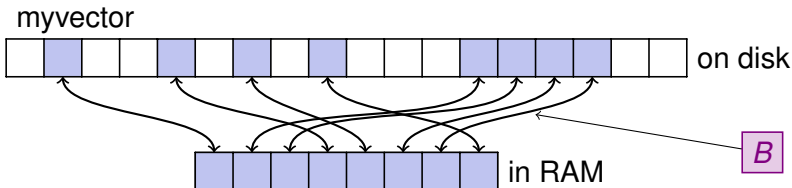


# stxxl::vector Architecture

```
struct MyData { int a, b; };  
stxxl::VECTOR_GENERATOR<MyData>::result myvector;  
for (int i = 0; i < N; i++) {  
    myvector[i].a = 42 + i;  
}
```



# stxxl::vector Interface



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

```
vector_type vec;
```

```
vec.push_back(data);   vec[42] = data;
```

```
vec.resize(1024);     vec.size();
```

```
for (vector_type::iterator it = vec.begin();
     it != vec.end(); ++it)
{ *it = 42; }
```

# stxxl::stack **and** stxxl::queue

```
stxxl::stack<MyData> mystack;
```

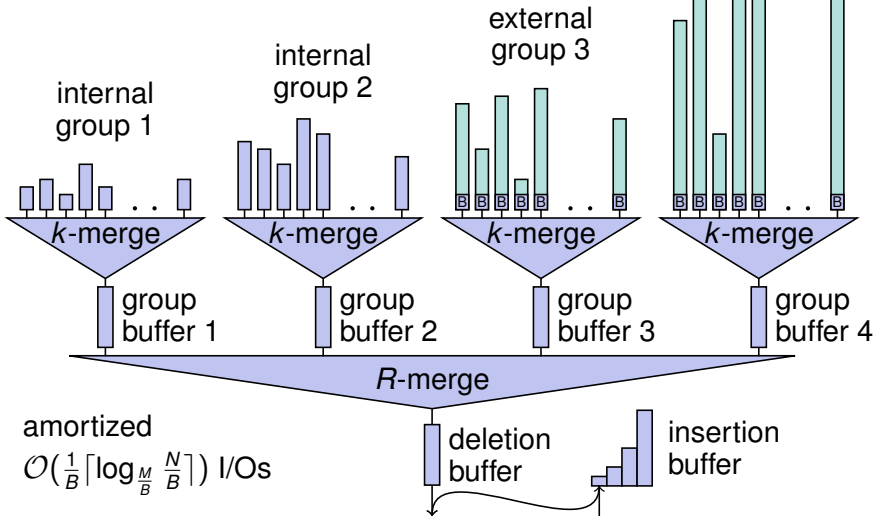


```
stxxl::queue<MyData> myqueue;
```

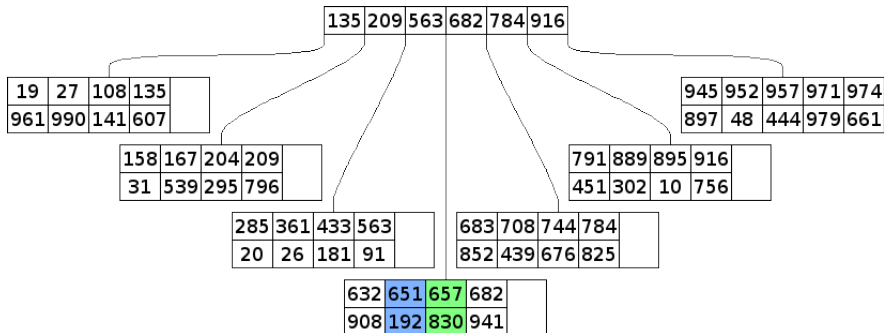


# stxxl::priority\_queue

external  
group 4



# stxxl::map – a B-Tree



# Sorting in STXXL

```
struct MyData { int a, b; }; MyData data = { 42, 6*9 };  
stxxl::sorter<MyData, ComparisonFunctor> sorter;
```

```
// push all data into sorter  
for (int i = 0; i < N; ++i)  
    sorter.push(data);
```

```
// switch to reading state  
sorter.sort();
```

```
// get back, in sorted order  
while ( !sorter.empty() )  
    data = *sorter, ++sorter;
```



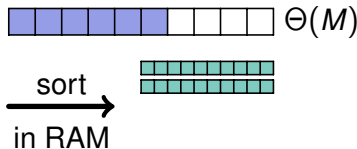
# Sorting in STXXL

```
struct MyData { int a, b; }; MyData data = { 42, 6*9 };
stxxl::sorter<MyData, ComparisonFunctor> sorter;
```

```
// push all data into sorter
for (int i = 0; i < N; ++i)
    sorter.push(data);
```

```
// switch to reading state
sorter.sort();
```

```
// get back, in sorted order
while ( !sorter.empty() )
    data = *sorter, ++sorter;
```



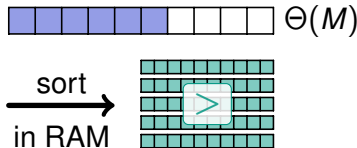
# Sorting in STXXL

```
struct MyData { int a, b; }; MyData data = { 42, 6*9 };  
stxxl::sorter<MyData, ComparisonFunctor> sorter;
```

```
// push all data into sorter  
for (int i = 0; i < N; ++i)  
    sorter.push(data);
```

```
// switch to reading state  
sorter.sort();
```

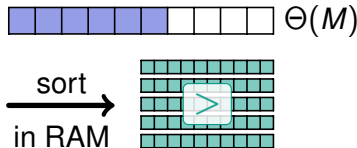
```
// get back, in sorted order  
while ( !sorter.empty() )  
    data = *sorter, ++sorter;
```



# Sorting in STXXL

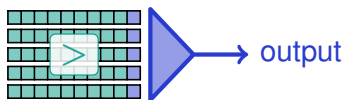
```
struct MyData { int a, b; }; MyData data = { 42, 6*9 };  
stxxl::sorter<MyData, ComparisonFunctor> sorter;
```

```
// push all data into sorter  
for (int i = 0; i < N; ++i)  
    sorter.push(data);
```



```
// switch to reading state  
sorter.sort();
```

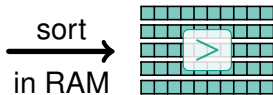
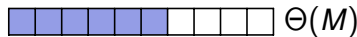
```
// get back, in sorted order  
while ( !sorter.empty() )  
    data = *sorter, ++sorter;
```



# Sorting in STXXL

```
struct MyData { int a, b; }; MyData data = { 42, 6*9 };  
stxxl::sorter<MyData, ComparisonFunctor> sorter;
```

```
// push all data into sorter  
for (int i = 0; i < N; ++i)  
    sorter.push(data);
```

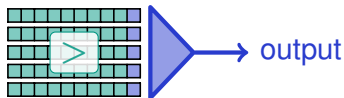


```
// switch to reading state
```

```
sorter.sort();
```

$\mathcal{O}\left(\frac{N}{B} \lceil \log_{\frac{M}{B}} \frac{N}{M} \rceil\right)$  I/Os, usually  $2\frac{N}{B}$  I/Os

```
// get back, in sorted order  
while ( !sorter.empty() )  
    data = *sorter, ++sorter;
```



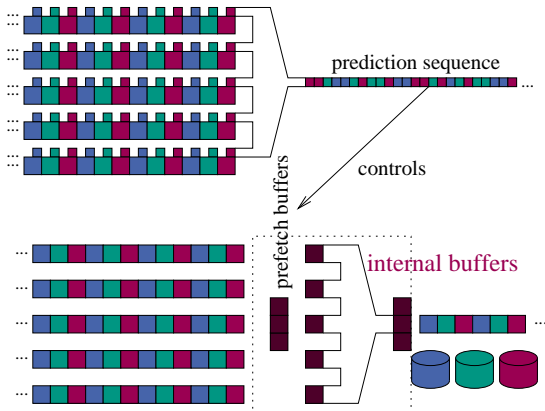
## Prediction

[Folklore, Knuth]

Smallest Element  
of each block  
triggers fetch.

Prefetch buffers

allow parallel access  
of next blocks



# Tournament Trees for Multiway Merging

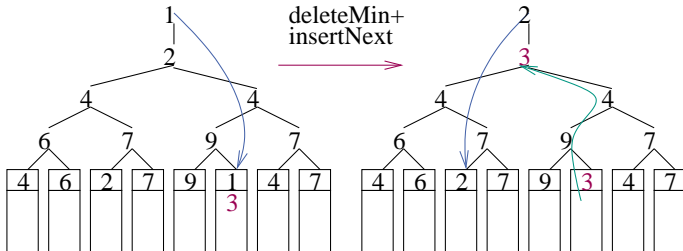
Assume  $k = 2^K$  runs

$K$  level complete binary tree

Leaves: smallest current element of each run

Internal nodes: loser of a competition for being smallest

Above root: global winner



# Sorting in STXXL – Rotational Disks

```
$ stxxl_tool benchmark_sort 256gib -M 16gib
```

```
Parameter size set to 274877906944.
```

```
Option -M, --ram set to 17179869184.
```

```
[STXXL-MSG] STXXL v1.4.99 (prerelease/Release) (git refs/heads/master)
```

```
[STXXL-MSG] Disk '/dev/sde1' is allocated, space: 2861587 MiB, I/O implementa
```

```
[STXXL-MSG] Disk '/dev/sdf1' is allocated, space: 2861587 MiB, I/O implementa
```

```
[STXXL-MSG] Disk '/dev/sdg1' is allocated, space: 2861587 MiB, I/O implementa
```

```
[STXXL-MSG] Disk '/dev/sdh1' is allocated, space: 2765655 MiB, I/O implementa
```

```
[STXXL-MSG] In total 4 disks are allocated, space: 11350417 MiB
```

```
#! running sorting test with pair of uint64 = 16 bytes.
```

```
# materialize random_stream into vector of size 17179869184
```

```
finished in 483.831 seconds @ 541.809 MiB/s
```

```
# stxxl::sort vector of size 17179869184
```

```
finished in 1831.44 seconds @ 143.135 MiB/s
```

```
# stxxl::ksort vector of size 17179869184
```

```
finished in 2269.05 seconds @ 115.53 MiB/s
```

```
# stxxl::stream::sort of size 17179869184
```

```
finished in 996.842 seconds @ 262.974 MiB/s
```

# Sorting in STXXL – SSD Disks

```
$ stxxl_tool benchmark_sort 256gib -M 16gib
```

```
Parameter size set to 274877906944.
```

```
Option -M, --ram set to 17179869184.
```

```
[STXXL-MSG] STXXL v1.4.99 (prerelease/Release) (git refs/heads/master)
```

```
[STXXL-MSG] Disk '/dev/sda1' is allocated, space: 953868 MiB, I/O implementat
```

```
[STXXL-MSG] Disk '/dev/sdb1' is allocated, space: 953868 MiB, I/O implementat
```

```
[STXXL-MSG] Disk '/dev/sdc1' is allocated, space: 953868 MiB, I/O implementat
```

```
[STXXL-MSG] Disk '/dev/sdd1' is allocated, space: 953868 MiB, I/O implementat
```

```
[STXXL-MSG] In total 4 disks are allocated, space: 3815474 MiB
```

```
#! running sorting test with pair of uint64 = 16 bytes.
```

```
# materialize random_stream into vector of size 17179869184
```

```
finished in 171.482 seconds @ 1528.7 MiB/s
```

```
# stxxl::sort vector of size 17179869184
```

```
finished in 870.682 seconds @ 301.079 MiB/s
```

```
# stxxl::ksort vector of size 17179869184
```

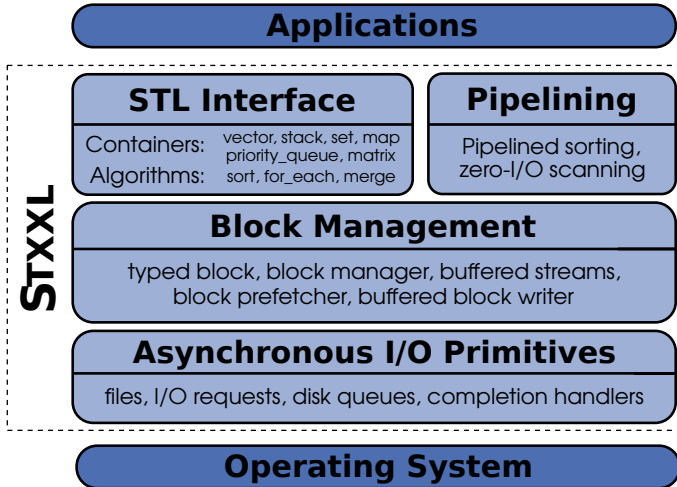
```
finished in 1038.68 seconds @ 252.383 MiB/s
```

```
# stxxl::stream::sort of size 17179869184
```

```
finished in 908.607 seconds @ 288.512 MiB/s
```



# STXXL Design – Layers



# Pipelining – Materialize Sorted Stream

```
typedef std::pair<uint64_t,uint64_t> MyData
stxxl::vector<MyData> myvector(N);
for (int i = 0; i < N; ++i)
    myvector[i] = std::make_pair(rand(),rand());
stxxl::sort(myvector.begin(), myvector.end());
```

# Pipelining – Materialize Sorted Stream

```
typedef std::pair<uint64_t,uint64_t> MyData
stxxl::vector<MyData> myvector(N);
for (int i = 0; i < N; ++i)
    myvector[i] = std::make_pair(rand(),rand());
stxxl::sort(myvector.begin(), myvector.end());
```

or we can ...

```
stxxl::sorter<MyData,MyCompare> mysorter(N);
for (int i = 0; i < N; ++i)
    mysorter.push( std::make_pair(rand(),rand()) );
mysorter.sort();
stxxl::vector<MyData> myvector(N);
stxxl::stream::materialize(mysorter,
    myvector.begin(), myvector.end());
```

# Pipelining – Materialize Sorted Stream

```
typedef std::pair<uint64_t,uint64_t> MyData  
stxxl::vector<MyData> myvector(N);  
for (int i = 0; i < N; ++i)  
    myvector[i] = std::make_pair(rand(),rand());  
stxxl::sort(myvector.begin(), myvector.end());
```

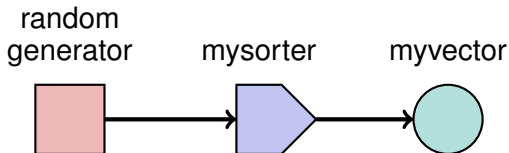
Write +  
2x Read  
2x Write

or we can ...

```
stxxl::sorter<MyData,MyCompare> mysorter(N);  
for (int i = 0; i < N; ++i)  
    mysorter.push( std::make_pair(rand(),rand()));  
mysorter.sort();  
stxxl::vector<MyData> myvector(N);  
stxxl::stream::materialize(mysorter,  
    myvector.begin(), myvector.end());
```

1x Write  
1x Read  
+ Write

# Pipelining – Materialize Sorted Stream



```
stxxl::sorter<MyData,MyCompare> mysorter(N);  
for (int i = 0; i < N; ++i)  
    mysorter.push( std::make_pair(rand(), rand()));  
mysorter.sort();  
stxxl::vector<MyData> myvector(N);  
stxxl::stream::materialize(mysorter,  
    myvector.begin(), myvector.end());
```

1x Write  
1x Read  
+ Write

# Pipelining – Stream Concept

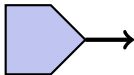
```
concept Stream {  
    bool empty();  
    const value_type & operator* () const;  
    Stream & operator++ ();  
};
```

# Pipelining – Stream Concept

```
concept Stream {  
    bool empty();  
    const value_type & operator* () const;  
    Stream & operator++ ();  
};
```

```
stxxl::sorter<MyData, CmpType> sorter;  
// ... fill sorter
```

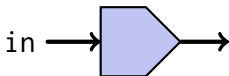
```
// get back, in sorted order  
while ( !sorter.empty() )  
    data = *sorter, ++sorter;
```



# Pipelining – Stream Concept

```
concept Stream {  
    bool empty();  
    const value_type & operator* () const;  
    Stream & operator++ ();  
};
```

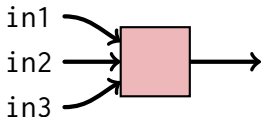
```
class stxxl::stream::sort : Stream {  
    typedef Input::value_type value_type;  
    sort(Input in, CmpType cmp);  
    // implements Stream methods  
};
```





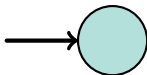
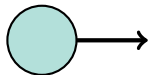
# Pipelining – Stream Concept

```
concept Stream {  
    bool empty();  
    const value_type & operator* () const;  
    Stream & operator++ ();  
};  
  
class stxxl::stream::transform : Stream {  
    typedef Operation::value_type value_type;  
    transform(Operation op,  
              Input1 in1, Input2 in2, Input3 in3, ...);  
    // implements Stream methods  
};
```

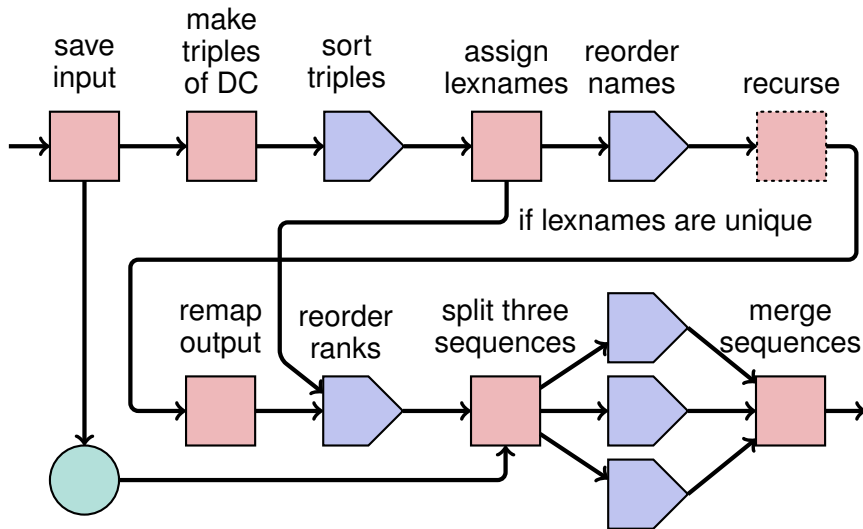


# Pipelining – Stream Concept

```
concept Stream {  
    bool empty();  
    const value_type & operator* () const;  
    Stream & operator++ ();  
};  
  
class stxxl::stream::vector_iterator2stream : Stream {  
    typedef Iterator::value_type value_type;  
    vector_iterator2stream(Iterator begin,  
                           Iterator end);  
    // implements Stream methods  
};  
  
stxxl::stream::streamify(begin, end);  
stxxl::stream::materialize(stream, begin, end);
```



# Pipelining – DC3 Suffix Sorting

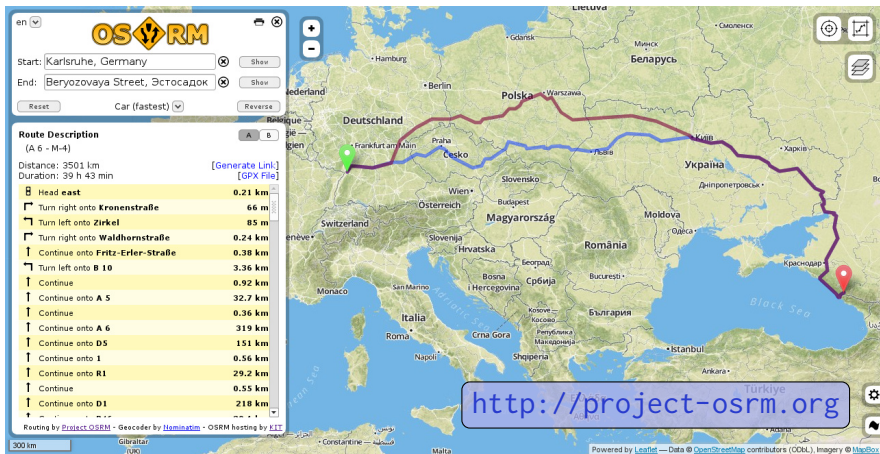


## Past:

- Minimum **spanning tree** and **breadth-first search** in EM.
- Large **matrix operations**: multiplication and inversion.
- EM suffix array construction with DC3.

# Present and Future of STXXL

- OSRM – Open Source Routing Machine:  
make **shortest path preprocessing** scale to OpenStreetMap.



en

# OSRM

Start: Karlsruhe, Germany

End: Beryozovaya Street, Эстосадок

Car (fastest)

### Route Description

(A 6 - M-4)

Distance: 3501 km  [GPX File]

Duration: 39 h 43 min

Head east	0.21 km
Turn right onto Kronenstrasse	66 m
Turn left onto Zirkel	85 m
Turn right onto Waldhornstrasse	0.24 km
Continue onto Fritz-Erler-Strasse	0.38 km
Turn left onto B 10	3.36 km
Continue	0.92 km
Continue onto A 5	32.7 km
Continue	0.36 km
Continue onto A 6	319 km
Continue onto D5	151 km
Continue onto I	0.56 km
Continue onto R1	29.2 km
Continue	0.55 km
Continue onto D1	218 km

Routing by [Project OSRM](#) - Geocoder by [Haminatum](#) - OSRM hosting by [KIT](#)

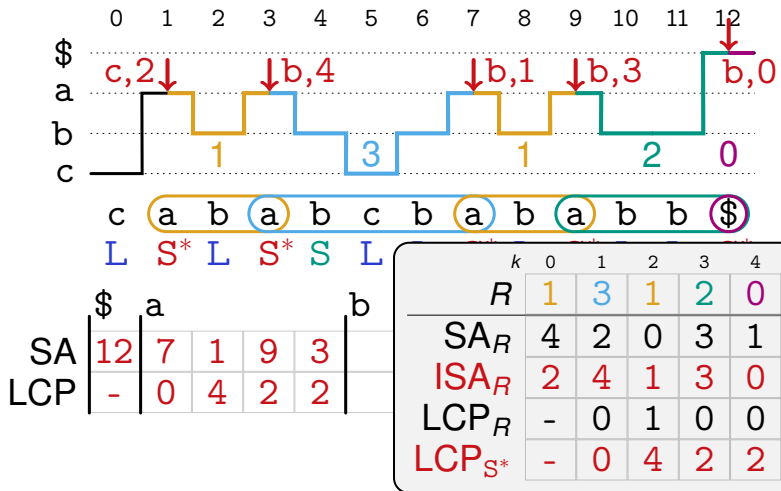
300 km

Powered by [Leaflet](#) — Data © [OpenStreetMap](#) contributors (ODL), [Imagery](#) © [MapBox](#)

<http://project-osrm.org>

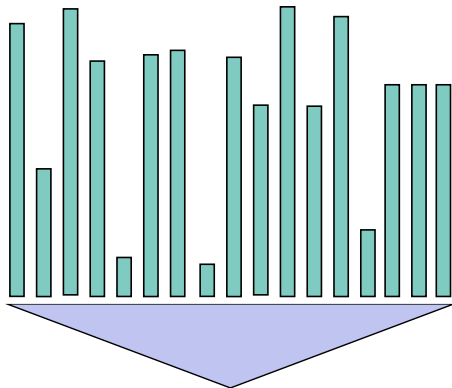
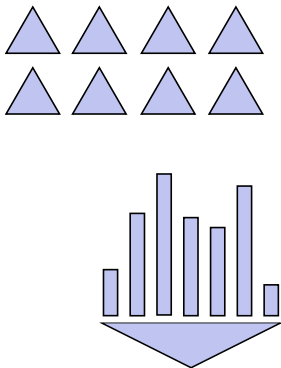
# Present and Future of STXXL

- eSAIS – induced suffix and LCP construction for text indexes.



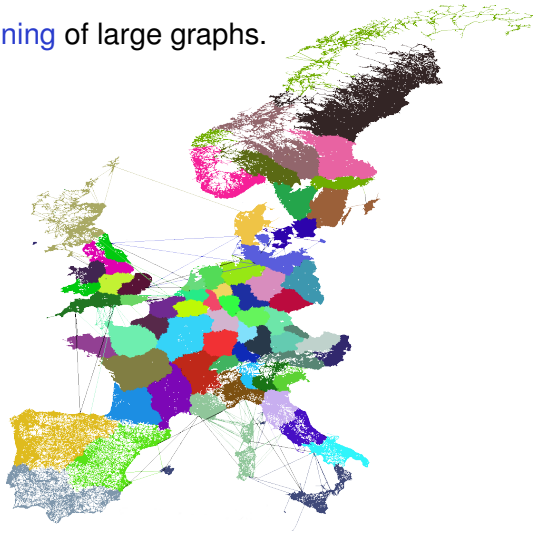
# Present and Future of STXXL

- Bulk-Parallel priority queue for fast multi-core algorithms.



# Present and Future of STXXL

- EM **graph partitioning** of large graphs.





## Present and Future:

- C++11 containers and additional functions.
- Direct support for Linux asynchronous I/O interface.
- Integration of EM hash\_map and async pipelining branches.

Show documentation.

Thank you for your attention!