STXXL and Thrill:
(Distributed Big Data Batch Processing in C++)

Michael Axtmann, Timo Bingmann, Peter Sanders, Sebastian Schlag, and 6 Students | 2016-09-21
Example $T = [\text{dbadcbcccbabdc}c$]$

<table>
<thead>
<tr>
<th>$SA_i$</th>
<th>$T_{SA_i...n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>$$$</td>
</tr>
<tr>
<td>9</td>
<td>abdc$</td>
</tr>
<tr>
<td>2</td>
<td>abdc$</td>
</tr>
<tr>
<td>8</td>
<td>abdc$</td>
</tr>
<tr>
<td>1</td>
<td>abdc$</td>
</tr>
<tr>
<td>5</td>
<td>abdc$</td>
</tr>
<tr>
<td>10</td>
<td>abdc$</td>
</tr>
<tr>
<td>13</td>
<td>abdc$</td>
</tr>
<tr>
<td>7</td>
<td>abdc$</td>
</tr>
<tr>
<td>4</td>
<td>abdc$</td>
</tr>
<tr>
<td>12</td>
<td>abdc$</td>
</tr>
<tr>
<td>6</td>
<td>abdc$</td>
</tr>
<tr>
<td>0</td>
<td>abdc$</td>
</tr>
<tr>
<td>3</td>
<td>abdc$</td>
</tr>
<tr>
<td>11</td>
<td>abdc$</td>
</tr>
</tbody>
</table>
Flavours of Big Data Frameworks

- High Performance Computing (Supercomputers)
  MPI

- Batch Processing
  Google’s MapReduce, Hadoop MapReduce 🐘, Apache Spark 🌟, Apache Flink 🐍 (Stratosphere), Google’s FlumeJava.

- Real-time Stream Processing
  Apache Storm ⛄, Apache Spark Streaming, Google’s MillWheel.

- Interactive Cached Queries
  Google’s Dremel, Powerdrill and BigQuery, Apache Drill 🖋.

- Sharded (NoSQL) Databases and Data Warehouses
  MongoDB 🦛, Apache Cassandra, Apache Hive, Google BigTable, Hypertable, Amazon RedShift, FoundationDB.

- Graph Processing
  Google’s Pregel, GraphLab 🐘, Giraph 🐍, GraphChi.

- Time-based Distributed Processing
  Microsoft’s Dryad, Microsoft’s Naiad.
Big Data Batch Processing

- **Performance**
  - Low Level: Difficult
  - High Level: Simple

- **Interface**
  - Low Level: MPI
  - High Level: Apache Spark, Apache Flink, MapReduce, Hadoop

- **Speed**
  - Fast: MPI
  - Slow
Projektpraktikum:
Verteilte Datenverarbeitung mit MapReduce

Timo Bingmann, Peter Sanders und Sebastian Schlag | 21. Oktober 2014 @ PdF Vorstellung
Big Data Batch Processing

Our Requirements:
- compound primitives into complex algorithms
- overlap computation and communication,
- efficient simple data types,
- C++, and much more...

New Project: Thrill

Efficiency

Low Level
Difficult

Interface

High Level
Simple

Fast

MPI

Slow

Apache Spark
Apache Flink
MapReduce
Hadoop

Apache
Hadoop

Our Requirements:
- compound primitives into complex algorithms
- overlap computation and communication,
- efficient simple data types,
- C++, and much more...
Big Data Batch Processing

Our Requirements:
- compound primitives into complex algorithms
- overlap computation and communication,
- efficient simple data types,
- C++, and much more...

Lower Layers of Thrill

New Project: Thrill

Lower Level
Difficult

Interface

High Level
Simple

Fast

MPI

Efficiency

Apache Spark

Apache Flink

MapReduce

Hadoop

Low Level
Difficult
Thrill’s Design Goals

- A new and easier way to program distributed algorithms.
- Distributed arrays of small items (characters or integers).
- High-performance, parallelized C++ operations.
- Locality-aware, in-memory computation.
- Transparently use disk if needed ⇒ external memory algorithms.
- Avoid all unnecessary round trips of data to memory (or disk).
- Optimize chaining/pipelining of local operations.

Current Status:

Distributed Immutable Array (DIA)

- User Programmer’s View:
  - DIA\(<T>\) = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use transformations and actions.

![Diagram]

- PE0 PE1 PE2 PE3
- A
- A. Map(·) =: B
- B. Sort(·) =: C
Distributed Immutable Array (DIA)

User Programmer’s View:
- DIA<T> = \textbf{result} of an operation (local or distributed).
- Model: \textit{distributed array} of items T on the cluster
- Cannot access items directly, instead use \textit{transformations} and \textit{actions}.

Framework Designer’s View:
- Goals: distribute work, optimize execution on cluster, add redundancy where applicable. \(\implies\) \textbf{build data-flow graph.}
- DIA<T> = \textit{chain of computation items}
Distributed Immutable Array (DIA)

User Programmer’s View:
- \( \text{DIA}<T> = \text{result} \) of an operation (local or distributed).
- Model: \textit{distributed array} of items \( T \) on the cluster.
- Cannot access items directly, instead use and \textit{actions}.

Framework Designer’s View:
- Goals: distribute work, optimize execution on cluster, add redundancy where applicable. \( \Rightarrow \) \textit{build data-flow graph}.
- \( \text{DIA}<T> = \text{chain of computation items} \)
List of Primitives

- Local Operations (LOp): input is one item, output $\geq 0$ items. Map(), Filter(), FlatMap().
- Distributed Operations (DOp): input is a DIA, output is a DIA.
  - Sort(): Sort a DIA using comparisons.
  - ReduceByKey(): Shuffle with Key Extractor, Hasher, and associative Reducer.
  - GroupByKey(): Like ReduceByKey, but with a general Reducer.
  - PrefixSum(): Compute (generalized) prefix sum on DIA.
  - Window$_k()$: Scan all $k$ consecutive DIA items.
  - Zip(): Combine equal sized DIAs item-wise.
  - Merge(): Merge equal typed DIAs using comparisons.
- Actions: input is a DIA, output: $\geq 0$ items on master.
  - At(), Min(), Max(), Sum(), Sample(), pretty much still open.
Example: WordCount in Thrill

```cpp
using Pair = std::pair<std::string, size_t>;
void WordCount(Context& ctx, std::string input, std::string output) {
    auto word_pairs = ReadLines(ctx, input) // DIA<std::string>
        .FlatMap<Pair>(
            // flatmap lambda: split and emit each word
            [](const std::string& line, auto emit) {
                Split(line, ' ', [&](std::string_view sv) {
                    emit(Pair(sv.to_string(), 1)); });
            }); // DIA<Pair>
    word_pairs.ReduceByKey(
        // key extractor: the word string
        [](const Pair& p) { return p.first; },
        // commutative reduction: add counters
        [](const Pair& a, const Pair& b) {
            return Pair(a.first, a.second + b.second);
        }); // DIA<Pair>
    .Map([](const Pair& p) {
        return p.first + " : " + std::to_string(p.second); })
    .WriteLines(output); // DIA<std::string>
}  
```
DOps: ReduceByKey

\[ \textbf{ReduceByKey}(k, r) : \langle A \rangle \rightarrow \langle A \rangle \]

\[ k : A \rightarrow K \quad \text{key extractor} \]

\[ r : A \times A \rightarrow A \quad \text{reduction} \]
Compile program into **one binary**, running on all hosts.

Collective coordination of work on compute hosts, like MPI.

Control flow is decided on by using C++ statements.

Runs on MPI HPC clusters and on Amazon’s EC2 cloud.
Pipelining Stages in Thrill

A := ReadLines()

B := A. Sort()

C := B. Map()

D := Zip(C, D)

E := E. WriteLines()

PE 0

A := ReadLines_{0, n/2}()

pre-op: sample, store
exchange samples
post-op: transmit and sort

C := B. Map()

pre-op: store
align arrays (exchange)
post-op: zip lambda

E := E. WriteLines_{0, ℓ/2}()

PE 1

A := ReadLines_{n/2, n}()

pre-op: sample, store
exchange samples
post-op: transmit and sort

C := B. Map()

pre-op: store
align arrays (exchange)
post-op: zip lambda

E := E. WriteLines_{ℓ/2, ℓ}()
Pipelining Stages in Thrill

A := ReadLines()

B := A. Sort()

C := B. Map()

D := C

E := Zip(C, D)

E. WriteLines()

PE 0

A := ReadLines_{0, n/2}()

pre-op: sample, store
exchange samples
post-op: transmit and sort

C := B. Map()

pre-op: store
align arrays (exchange)
post-op: zip lambda

E. WriteLines_[0, ℓ/2]()

PE 1

A := ReadLines_{n/2, n}()

pre-op: sample, store
exchange samples
post-op: transmit and sort

C := B. Map()

pre-op: store
align arrays (exchange)
post-op: zip lambda

E. WriteLines_[ℓ/2, ℓ]()}
# Layers of Thrill

<table>
<thead>
<tr>
<th>api: High-level User Interface</th>
<th>core: Internal Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIA&lt;T&gt;, Map, FlatMap, Filter, Reduce, Sort, Merge, ...</td>
<td>reducing hash tables (bucket and linear probing), multiway merge, stage executor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data: Data Layer</th>
<th>net: Network Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block, File, BlockQueue, Reader, Writer, Multiplexer, Streams, BlockPool (paging)</td>
<td>(Binomial Tree) Broadcast, Reduce, AllReduce, Async-Send/Recv, Dispatcher</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>io: Async File I/O</th>
<th>mem: Memory Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>borrowed from STXXL</td>
<td>Allocators, Counting</td>
</tr>
</tbody>
</table>
Layers of STXXL

Applications

STL Interface
- Containers: vector, stack, set, map, priority_queue, matrix
- Algorithms: sort, for_each, merge

Pipelining
- Pipelined sorting, zero-I/O scanning

Block Management
- typed block, block manager, buffered streams, block prefetcher, buffered block writer

Asynchronous I/O Primitives
- files, I/O requests, disk queues, completion handlers

Operating System

STXXL

Michael Axtmann, Timo Bingmann, Peter Sanders, Sebastian Schlag, and 6 Students – STXXL and Thrill
Institute of Theoretical Informatics – Algorithmics
September 21st, 2016
### STXXL and Thrill

<table>
<thead>
<tr>
<th></th>
<th>STXXL</th>
<th>Thrill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td>external</td>
<td>distributed external</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td>partially parallelized</td>
<td>inherently parallel</td>
</tr>
<tr>
<td><strong>External Memory</strong></td>
<td>explicit</td>
<td>via swapping</td>
</tr>
<tr>
<td><strong>Items</strong></td>
<td>fixed size</td>
<td>variable length</td>
</tr>
<tr>
<td><strong>High-level API</strong></td>
<td>containers, streams</td>
<td>DIA operations</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>mix of imperative and functional parts</td>
<td></td>
</tr>
<tr>
<td><strong>Pipelining via</strong></td>
<td>nested template streams</td>
<td>functional templating,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>consume in pipeline</td>
</tr>
<tr>
<td><strong>RAM Limit</strong></td>
<td>manual (parameters)</td>
<td>automatic (stages)</td>
</tr>
<tr>
<td><strong>Code Base</strong></td>
<td>C++98</td>
<td>very modern C++14</td>
</tr>
</tbody>
</table>
STXXL and Thrill

**STXXL API:** vector (Paging), sorter (Sort), sequence (Scan), map (B-Tree), unordered_map (Hash), priority_queue (PQ), parallel_priority_queue (PPQ), matrix (Block Matrix), Stream/Pipelining: stream::sort, stream::runs_creator, stream::runs_merger, ...

**Lower Layers:** BID, typed_block, block_manager, read_write_pool, buf_istream, buf_ostream, async I/O impl.

**Thrill API:** DIA.Generate, DIA.Map, DIA.ReduceByKey, DIA.Window, DIA.Sort, DIA.Union, DIA.Zip, DIA.ReadBinary, ...

**Lower Layers:** Block, ByteBlock, File, BlockQueue, BlockReader, BlockWriter, BlockPool (Paging), STXXL’s BlockManager and async I/O implementations.
File and Blocks in Thrill

Block

ByteBlock

Item 1 Item 2 Item 3 Item 4 Item 5

begin first end

num_items=2

Block

ByteBlock

Item 1 Item 2 Item 3 Item 4 Item 5 Item 6

begin first end

num_items=1

Block

ByteBlock
Future of STXXL and Thrill?

On STXXL:
- Pre-C++11 old-style code. But good architecture.
- Many simplifications possible with C++11.
- Shared memory parallelism is critically important.
- Missing some important convenience features: variable length items, memory management.

Thrill and STXXL:
- Share much in common code layers, esp. C++ tooling.
- Thrill is distributed and inherently parallel, but not the API itself.

Thank you for your attention!
Questions? ... Discussion?