Thrill ♤: High-Performance Algorithmic Distributed Batch Data Processing in C++

Timo Bingmann, Michael Axtmann, Peter Sanders, Sebastian Schlag, and 6 Students | 2016-12-06
Example $T = [\text{dbadcbcccbabdcc}$]$

<table>
<thead>
<tr>
<th>$SA_i$</th>
<th>$LCP_i$</th>
<th>$T_{SA_i \ldots n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>-</td>
<td>$$$</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>abdc</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>adcbcccbabdcc</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>babdccc</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>badcbbcbabdc</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>cccbabdccc</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>bddcc</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>ccc</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>cbabdc</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>cbcbbcbbabdcc</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>ccc</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>cccbacddc</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>dbadcbcccbbadbdc</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>dcbbccbbabdcc</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>dccc</td>
</tr>
</tbody>
</table>
Flavours of Big Data Frameworks

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

- **High Performance Computing (Supercomputers)**
  MPI

- **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.
Our Requirements:

- Compound primitives into complex algorithms
- Efficient simple data types
- Overlap computation and communication
- Automatic disk usage
- C++, and much more...
Big Data Batch Processing

Efficiency

Fast

Low Level

Difficult

Interface

High Level

Simple

New Project: Thrill

Apache Spark

Apache Flink

MapReduce

Hadoop

MPI

Our Requirements:
compound primitives into complex algorithms

Lower Layers

of Thrill

Timo Bingmann, Michael Axtmann, Peter Sanders, Sebastian Schlag, and 6 Students – Thrill: Distributed Big Data Batch Processing in C++

Institute of Theoretical Informatics – Algorithmics

December 6th, 2016
Big Data Batch Processing

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

New Project: Thrill

Lower Layers of Thrill

Timo Bingmann, Michael Axtmann, Peter Sanders, Sebastian Schlag, and 6 Students – Thrill: Distributed Big Data Batch Processing in C++
Institute of Theoretical Informatics – Algorithmics
December 6th, 2016
Thrill’s Design Goals

- An easy way to program distributed algorithms in C++.
- Distributed arrays of small items (characters or integers).
- High-performance, parallelized C++ operations.
- Locality-aware, in-memory computation.
- Transparenlty use disk if needed → external memory or cache-oblivious algorithms.
- Avoid all unnecessary round trips of data to memory (or disk).
- Optimize chaining 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.

Framework Designer's View:
Goals: distribute work, optimize execution on cluster, add redundancy where applicable.
⇒ build data-flow graph.

DIA<T> = chain of computation items
Let distributed operations choose "materialization"
Distributed Immutable Array (DIA)

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

\begin{align*}
A & = \quad \text{PE0} & & \quad \text{PE1} & & \quad \text{PE2} & & \quad \text{PE3} \\
A. \text{Map(·)} &= \quad B & & \quad C \\
B. \text{Sort(·)} &= \quad C
\end{align*}

Framework Designer’s View:
- Goals: distribute work, optimize execution on cluster, add redundancy where applicable. \( \implies \) build data-flow graph.
- \( \text{DIA}\langle T \rangle = \text{chain of computation items} \)
- Let distributed operations choose “materialization”.
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.

A. Map(·) := B
B. Sort(·) := C

Framework Designer’s View:
- Goals: distribute work, optimize execution on cluster, add redundancy where applicable. \(\Rightarrow\) build data-flow graph.
- DIA<T> = chain of computation items
- Let distributed operations choose “materialization”.

Diagram:
- PE0 and PE1 represent processing elements.
- Data flow from A to B and then to C.
List of Primitives (Excerpt)

- 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.
  - ReduceBy() Shuffle with Key Extractor, Hasher, and associative Reducer.
  - GroupBy() Like ReduceBy, 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.
  - Union() Combine equal typed DIAs in arbitrary order.
  - Merge() Merge equal typed sorted DIAs.
- Actions: input is a DIA, output: $\geq 0$ items on every worker.
  - At(), Min(), Max(), Sum(), Sample(), pretty much still open.
Local Operations (LOps)

Map\( (f) : \langle A \rangle \rightarrow \langle B \rangle \)
\( f : A \rightarrow B \)

Filter\( (f) : \langle A \rangle \rightarrow \langle A \rangle \)
\( f : A \rightarrow \{ \text{false, true} \} \)

FlatMap\( (f) : \langle A \rangle \rightarrow \langle B \rangle \)
\( f : A \rightarrow \text{array}(B) \)

Currently: no rebalancing during LOps.
DOps: ReduceByKey

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

\[ k : A \rightarrow K \quad \text{key extractor} \]
\[ r : A \times A \rightarrow A \quad \text{reduction} \]
DOps: ReduceToIndex

\textbf{ReduceToIndex}(i, n, r) : \langle A \rangle \rightarrow \langle A \rangle

\begin{align*}
i : A &\rightarrow \{0..n - 1\} \quad \text{index extractor} \\
n &\in \mathbb{N}_0 \quad \text{result size} \\
r : A \times A &\rightarrow A \quad \text{reduction}
\end{align*}
DOps: Sort and Merge

**Sort** \((o) : \langle A \rangle \rightarrow \langle A \rangle\)

\(o : A \times A \rightarrow \{false, true\}\)

(less) order relation

**Merge** \((o) : \langle A \rangle \times \langle A \rangle \cdots \rightarrow \langle A \rangle\)

\(o : A \times A \rightarrow \{false, true\}\)

(less) order relation

\[\begin{array}{cccccc}
a_0 & a_1 & a_2 & a_3 & a_4 \\
a_4 & a_3 & a_1 & a_0 & a_2 \\
\end{array}\]

\[\begin{array}{cccccc}
a_2 & a_4 & a_9 \\
a_0 & a_5 & a_6 \\
\end{array}\]

\[\begin{array}{cccccc}
a_0 & a_2 & a_4 & a_5 & a_6 & a_9 \\
\end{array}\]
**DOps: Zip and Window**

**Zip** \( z \): \( \langle A \rangle \times \langle B \rangle \cdots \to \langle C \rangle \)

\( z : A \times B \to C \)

zip function

**Window** \( (k, w) \): \( \langle A \rangle \to \langle B \rangle \)

\( k \in \mathbb{N} \) window size

\( w : A^k \to B \) window function
Example: WordCount in Thrill

```cpp
typedef std::pair<std::string, size_t> Pair;

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);
    });  // DIA<std::string>
    .WriteLines(output);  // DIA<std::string>
}
```
Mapping Data-Flow Nodes to Cluster

**Master**

- \( A := \text{ReadLines}() \)
- \( B := A.\text{Sort}() \)
- \( C := B.\text{Map}() \)
- \( D \)
- \( E := \text{Zip}(C, D) \)
- \( E.\text{WriteLines}() \)

**PE 0**

- \( A := \text{ReadLines}_{[0, \frac{n}{2}}() \)
- pre-op: sample, store
- exchange samples
- post-op: transmit and sort
- \( D_{[0, \frac{m}{2}}) \)
- \( C := B.\text{Map}() \)
- pre-op: store
- align arrays (exchange)
- post-op: zip lambda
- \( E.\text{WriteLines}_{[0, \frac{\ell}{2}}() \)

**PE 1**

- \( A := \text{ReadLines}_{[\frac{n}{2}, n}}() \)
- pre-op: sample, store
- exchange samples
- post-op: transmit and sort
- \( D_{\frac{m}{2}, m} \)
- \( C := B.\text{Map}() \)
- pre-op: store
- align arrays (exchange)
- post-op: zip lambda
- \( E.\text{WriteLines}_{[\frac{\ell}{2}, \ell}}() \)
Mapping Data-Flow Nodes to Cluster

Master

\[ A := \text{ReadLines()} \]

\[ B := A.\text{Sort()} \]

\[ C := B.\text{Map()} \]

\[ D := \text{Zip}(C, D) \]

\[ E := \text{WriteLines()} \]

PE 0

\[ A := \text{ReadLines}_{[0, \frac{n}{2})}() \]

\[ \text{pre-op: sample, store} \]

\[ \text{exchange samples} \]

\[ \text{post-op: transmit and sort} \]

\[ C := B.\text{Map()} \]

\[ D := \text{Zip}(C, D) \]

\[ \text{pre-op: store} \]

\[ \text{align arrays (exchange)} \]

\[ \text{post-op: zip lambda} \]

\[ E := \text{WriteLines}_{[0, \frac{\ell}{2})}() \]

PE 1

\[ A := \text{ReadLines}_{[\frac{n}{2}, n)}() \]

\[ \text{pre-op: sample, store} \]

\[ \text{exchange samples} \]

\[ \text{post-op: transmit and sort} \]

\[ C := B.\text{Map()} \]

\[ D := \text{Zip}(C, D) \]

\[ \text{pre-op: store} \]

\[ \text{align arrays (exchange)} \]

\[ \text{post-op: zip lambda} \]

\[ E := \text{WriteLines}_{[\frac{\ell}{2}, \ell)}() \]
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.
Benchmarks

WordCountCC
- Reduce text files from CommonCrawl web corpus.

PageRank
- Calculate PageRank using join of current ranks with outgoing links and reduce by contributions. 10 iterations.

TeraSort
- Distributed (external) sorting of 100 byte random records.

K-Means
- Calculate K-Means clustering with 10 iterations.

Platform: \( h \times r3.8xlarge \) systems on Amazon EC2 Cloud
- 32 cores, Intel Xeon E5-2670v2, 2.5 GHz clock, 244 GiB RAM, 2 x 320 GB local SSD disk, \( \approx 400 \) MiB/s bandwidth Ethernet network \( \approx 1000 \) MiB/s network, Ubuntu 16.04.
Experimental Results: Slowdowns

WordCountCC

PageRank

TeraSort

KMeans

Slowdown over fastest

number of hosts $h$

Spark (Java) — Blue
Spark (Scala) — Red
Flink (Java) — Green
Flink (Scala) — Purple
Thrill — Cyan

number of hosts $h$
K-Means Tutorial

Step 1: Generate Random Points

Welcome to the first step in the Thrill k-means tutorial. This tutorial will show how to implement the k-means clustering algorithm (Lloyd's algorithm) in Thrill.

The algorithm works as follows: Given a set of d-dimensional points, select k initial cluster center points at random. Then attempt to improve the centers by iteratively calculating new centers. This is done by classifying all points and associating them with their nearest center, and then taking the mean of all points associated to one cluster as the new center. This will be repeated a constant number of iterations.

We will implement this algorithm in Thrill, and only work with two-dimensional points for simplicity. Furthermore, we will hard-code many constants to make the code easier to understand.

In this step 1, let us start with generating random 2-dimensional points and outputting them for debugging.

We first need a Point class to represent the points. We may add some calculation functions to it later on.

```cpp
#include <iostream>

//: A 2-dimensional point with double precision

struct Point
{
  //: point coordinates
  double x, y;
};
```

For outputting the Point class, we need to add an operator `<<` for `std::ostream`, which is the standard way for...
Current and Future Work

- Open-Source at [http://project-thrill.org](http://project-thrill.org) and Github.
- High quality, very modern C++14 code.

Ideas for Future Work:

- Distributed rank()/select() and wavelet tree construction.
- Beyond DIA<T>? Graph<V,E>? DenseMatrix<T>?
- Fault tolerance? Go from $p$ to $p - 1$ workers?
- Communication efficient distributed operations for Thrill.
- Distributed functional programming language on top of Thrill.

Thank you for your attention!

Questions?