Karlsruhe Institute of Technology

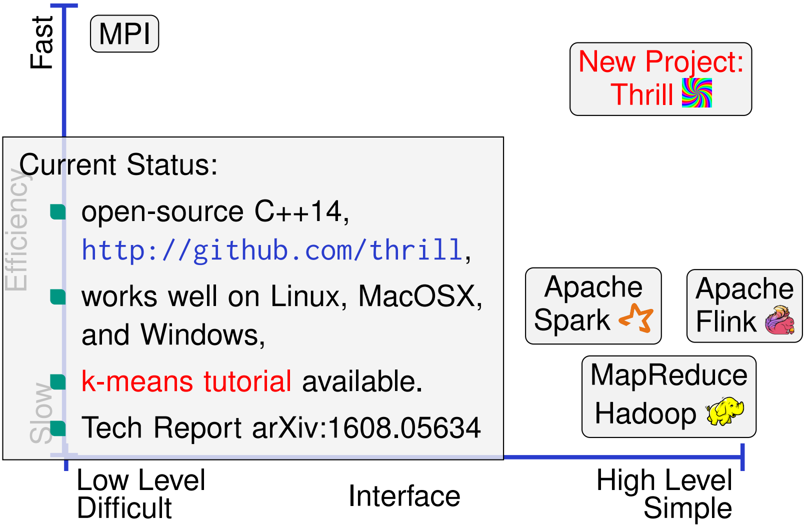
Thrill :

Distributed Big Data Batch Processing in C++

Michael Axtmann, Timo Bingmann, Peter Sanders, Sebastian Schlag, and 6 Students | 2016-10-06

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS

Algorithmic Big Data Batch Processing



Current Status:

- open-source C++14, http://github.com/thrill,
- works well on Linux, MacOSX, and Windows,
- k-means tutorial available.
- Tech Report arXiv:1608.05634

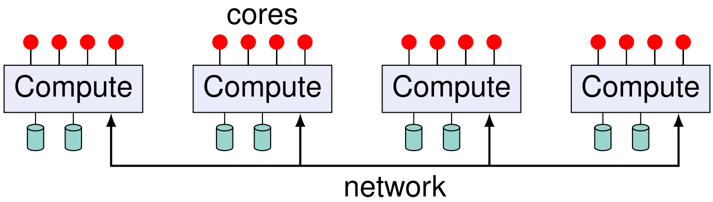
Thrill's Design Goals

- 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
  $\Rightarrow$ external memory algorithms.
- Avoid all unnecessary round trips of data to memory (or disk).
- Optimize chaining of local operations.

Current Status:

- Prototype at `http://project-thrill.org` and Github.

Execution on Cluster

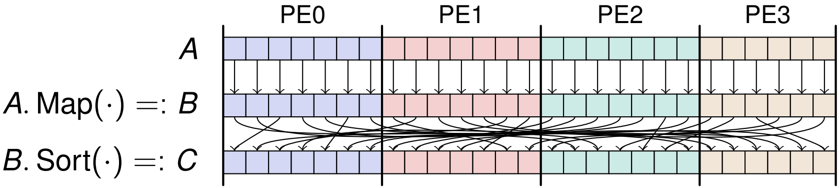


- Compile program into one binary, running on all hosts.
- Collective coordination of work on compute hosts, like MPI.
- Control flow is decided on by using C++ statements.

- Runs on MPI HPC clusters and on Amazon's EC2 cloud.

Distributed Immutable Array (DIA)

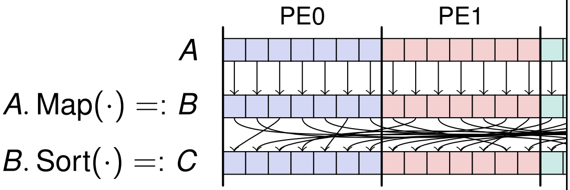
- User Programmer's View:
  - `DIA<T>` = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use transformations and actions.

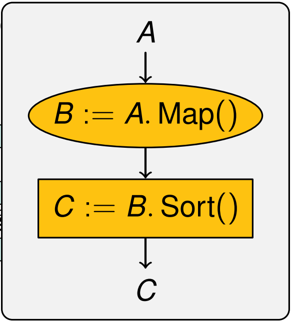


- Framework Designer's View:
  - Goals: distribute work, optimize execution on cluster, add redundancy where applicable. $\Longrightarrow$ build data-flow graph.
  - `DIA<T>` = chain of computation items
  - Let distributed operations choose "materialization".

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, instea and actions.





- Framework Designer's View:
  - Goals: distribute work, optimize execution on cluster, add redundancy where applicable. $\Longrightarrow$ build data-flow graph.
  - DIA<T> = chain of computation items
  - Let distributed operations choose "materialization".

List of Primitives

- Local Operations (LOp): input is one item, output ≥ 0 items.
  Map(), Filter(), FlatMap().
- Distributed Operations (DOp): input is a DIA, output is a DIA.

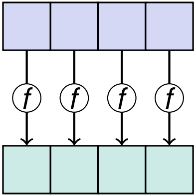
Sort()	Sort a DIA using comparisons.
ReduceBy()	Shuffle with Key Extractor, Hasher, and associative Reducer.
GroupBy()	Like ReduceBy, but with a general Reducer.
PrefixSum()	Compute (generalized) prefix sum on DIA.
Window_k()	Scan all k consecutive DIA items.
Zip()	Combine equal sized DIAs item-wise.
Merge()	Merge equal typed DIAs using comparisons.

- Actions: input is a DIA, output: ≥ 0 items on master.
  Min(), Max(), Sum(), Sample(), pretty much still open.

Local Operations (LOps)

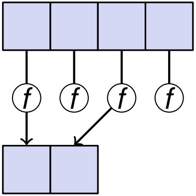
$\textbf{Map}(f) : \langle A \rangle \to \langle B \rangle$

$f : A \to B$



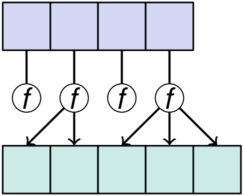
$\textbf{Filter}(f) : \langle A \rangle \to \langle A \rangle$

$f : A \to \{\mathit{false}, \mathit{true}\}$



$\textbf{FlatMap}(f) : \langle A \rangle \to \langle B \rangle$

$f : A \to \text{array}(B)$



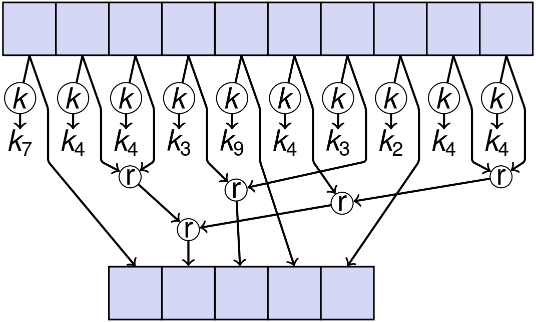
Currently: no rebalancing during LOps.

DOps: ReduceByKey

$$\textbf{ReduceByKey}(k, r) : \langle A \rangle \to \langle A \rangle$$

$k : A \to K$ key extractor

$r : A \times A \to A$ reduction

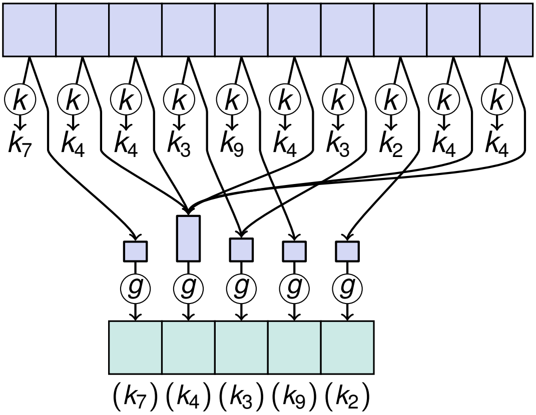


(k_7) (k_4) (k_3) (k_9) (k_2)

DOps: GroupByKey

$$\textbf{GroupByKey}(k, g) : \langle A \rangle \to \langle B \rangle$$

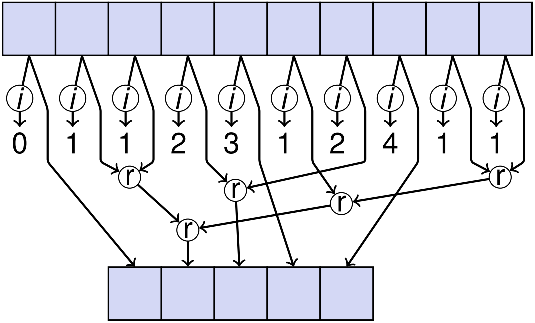
$k : A \to K$	key extractor
$g : iterable(A) \to B$	group function



DOps: ReduceToIndex

ReduceToIndex$(i, n, r) : \langle A \rangle \to \langle A \rangle$

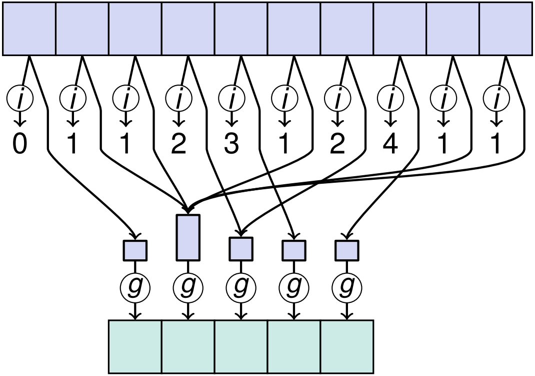
$i : A \to \{0..n-1\}$	index extractor
$n \in \mathbb{N}_0$	result size
$r : A \times A \to A$	reduction



DOps: GroupToIndex

$$\textbf{GroupToIndex}(i, n, g) : \langle A \rangle \to \langle B \rangle$$

$i : A \to \{0..n-1\}$	index extractor
$n \in \mathbb{N}_0$	result size
$g : \mathit{iterable}(A) \to B$	group function

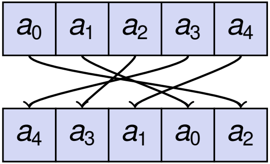


DOps: Sort and Merge

Sort$(o) : \langle A \rangle \to \langle A \rangle$

$o : A \times A \to \{\mathit{false}, \mathit{true}\}$

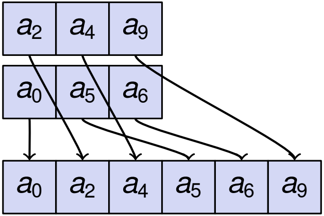
(less) order relation



Merge$(o) : \langle A \rangle \times \langle A \rangle \cdots \to \langle A \rangle$

$o : A \times A \to \{\mathit{false}, \mathit{true}\}$

(less) order relation

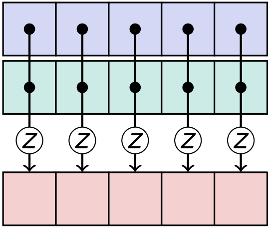


DOps: Zip and Window

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

$z : A \times B \to C$

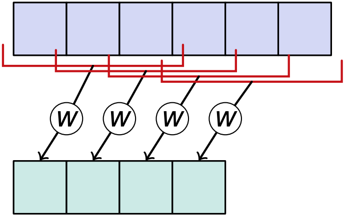
zip function



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

```
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>
}
```

Benchmarks

Karlsruhe Institute of Technology

WordCount

- Reduce random text files containing only 1000 words.

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.

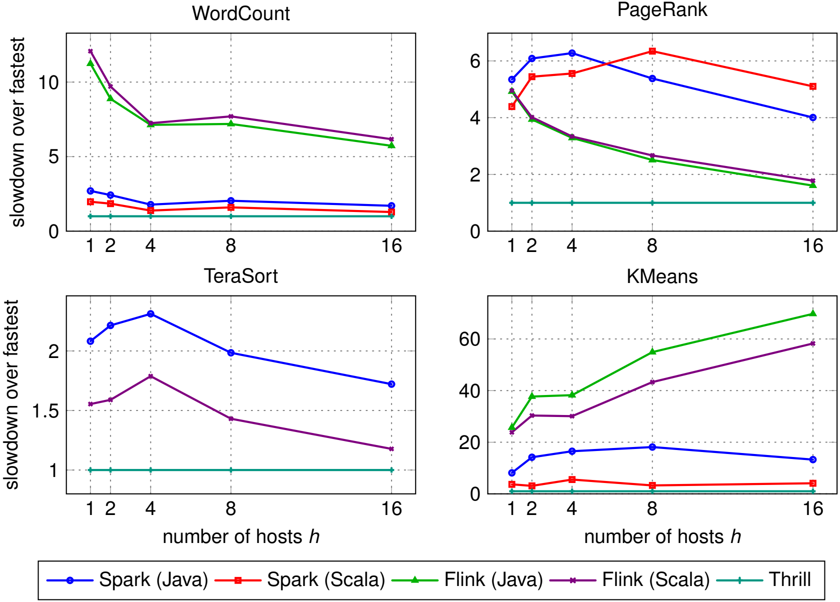
K-Means

- 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



K-Means Tutorial

Thrill 0.1

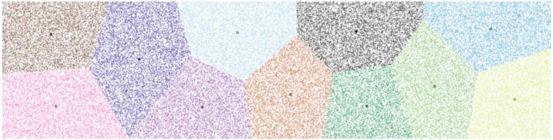
Main Page | Related Pages | Modules | Namespaces | Classes | Files | Examples | Search

- Thrill
  - Thrill Documentation Overview
    - Getting Started
    - K-Means Tutorial
      - Step 1: Generate Random Points
      - Step 2: Pick Random Centers and C
      - Step 3: ReduceByKey to Calculate I
      - Step 4: Iteration!
      - Step 5: Input and Output
      - Bonus Step 6: Boost Qi
    - Thrill Layer Architecture
    - Coding Style Guide
  - Modules
  - Namespaces
  - Classes
  - Files
  - Examples

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.

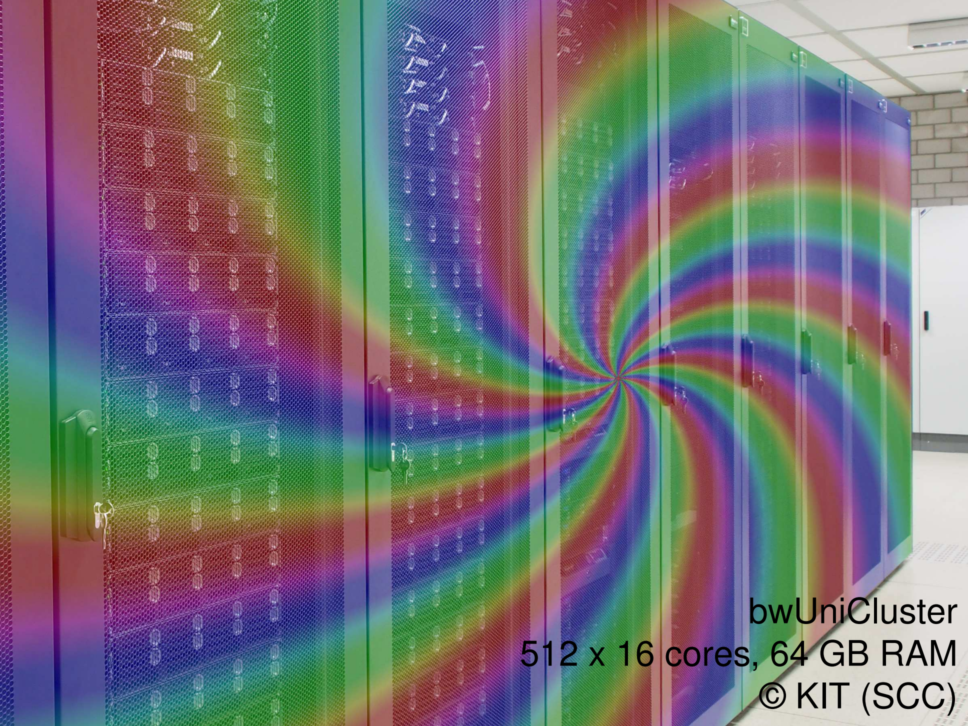
In this step 1, let us start with generating random 2-dimensional points and outputting them for debugging.

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

```
//! 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

Thrill Documentation Overview › K-Means Tutorial — Generated on Tue Sep 20 2016 19:24:29 for Thrill by doxygen 1.8.5


bwUniCluster
512 x 16 cores, 64 GB RAM
© KIT (SCC)

Current and Future Work

Karlsruhe Institute of Technology

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

Ideas for Future Work:

- Native Infiniband Support
- Distributed rank()/select() and wavelet tree construction.
- Beyond `DIA<T>`? `Graph<V,E>`? `Matrix<T>`?
- Communication efficient distributed operations for Thrill.

Thank you for your attention!

Questions?