Karlsruhe Institute of Technology

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

Timo Bingmann, Michael Axtmann, Peter Sanders, Sebastian Schlag, and 6 Students | 2016-12-06

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS

Abstract

We present on-going work on a new distributed Big Data processing framework called Thrill. It is a C++ framework consisting of a set of basic scalable algorithmic primitives like mapping, reducing, sorting, merging, joining, and additional MPI-like collectives. This set of primitives goes beyond traditional Map/Reduce and can be combined into larger more complex algorithms, such as WordCount, PageRank, k-means clustering, and suffix sorting. These complex algorithms can then be run on very large inputs using a distributed computing cluster. Among the main design goals of Thrill is to lose very little performance when composing primitives such that small data types are well supported. Thrill thus raises the questions of a) how to design algorithms using the scalable primitives, b) whether additional primitives should be added, and c) if one can improve the existing ones using new ideas to reduce communication volume and latency.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

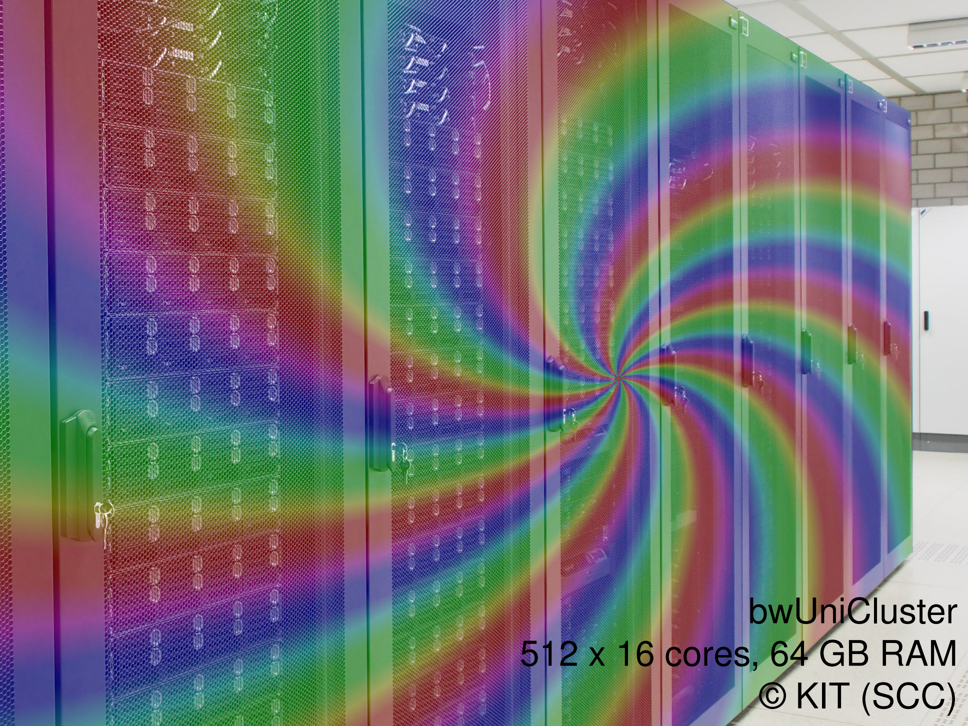
Example $T = [\texttt{dbadcbccbabdcc\$}]$

i	T_i
0	d b a d c b c c b a b d c c \$
1	b a d c b c c b a b d c c \$
2	a d c b c c b a b d c c \$
3	d c b c c b a b d c c \$
4	c b c c b a b d c c \$
5	b c c b a b d c c \$
6	c c b a b d c c \$
7	c b a b d c c \$
8	b a b d c c \$
9	a b d c c \$
10	b d c c \$
11	d c c \$
12	c c \$
13	c \$
14	\$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Example $T = [\texttt{dbadcbccbabdcc\$}]$

SA_i	LCP_i	$T_{SA_i \ldots n}$
14	-	$
9	0	a b d c c $
2	1	a d c b c c b a b d c c $
8	0	b a b d c c $
1	2	b a d c b c c b a b d c c $
5	1	b c c b a b d c c $
10	1	b d c c $
13	0	c $
7	1	c b a b d c c $
4	2	c b c c b a b d c c $
12	1	c c $
6	2	c c b a b d c c $
0	0	d b a d c b c c b a b d c c $
3	1	d c b c c b a b d c c $
11	2	d c c $



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

Suffix Sorting with DC3: Example

$$\begin{array}{rccccccccccccl} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \\ T = & [& \mathtt{d} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{d} & \$ & \$ &] \quad = [\, t_i \,]_{i=0,\dots,n-1} \end{array}$$

triples	(bac,1),	(bac,4),	(bd\$,7),	(acb,2)	(acb,5),	(d\$\$,8)
sorted	(acb,2)	(acb,5),	(bac,1),	(bac,4),	(bd\$,7),	(d\$\$,8)
equal 0/1	0	0	1	0	1	1
prefix sum	0	0	1	1	2	3

$$R = \boxed{1\;\;1\;\;2}\,\boxed{0\;\;0\;\;3}\;\;\$$$

$$\mathrm{SA}_R = 3\;\;4\;\;0\;\;1\;\;2\;\;5\;\;\$ \qquad \mathrm{ISA}_R = \boxed{\overset{r_1}{2}\;\;\overset{r_4}{3}\;\;\overset{r_7}{4}}\,\boxed{\overset{r_2}{0}\;\;\overset{r_5}{1}\;\;\overset{r_8}{5}}\;\;\$$$

$$S_0 = [(\mathtt{d},\mathtt{b},2,0,0),(\mathtt{c},\mathtt{b},3,1,3),(\mathtt{c},\mathtt{b},4,5,6)] \qquad (t_i, t_{i+1}, r_{i+1}, r_{i+2}, i)$$

$$S_1 = [(2,\mathtt{b},0,1),(3,\mathtt{b},1,4),(4,\mathtt{b},5,7)] \qquad (r_{i+1}, t_{i+1}, r_{i+2}, i+1)$$

$$S_2 = [(0,\mathtt{a},\mathtt{c},3,2),(1,\mathtt{a},\mathtt{c},4,5),(5,\mathtt{d},\$,6,8)] \qquad (r_{i+2}, t_{i+2}, t'_{i+3}, r'_{i+4}, i+2)$$

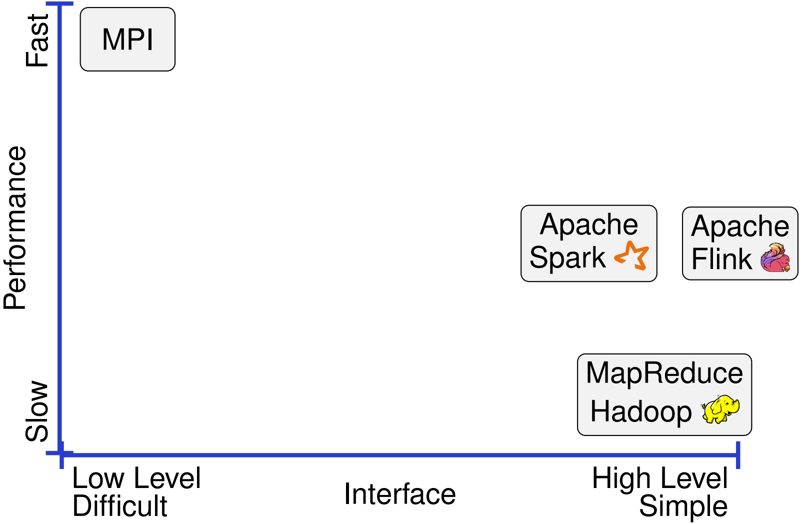
$$\mathrm{SA}_T = \mathrm{Merge}(\mathrm{Sort}(S_0), \mathrm{Sort}(S_1), \mathrm{Sort}(S_2))$$

$\boxed{\Theta(\mathrm{sort}(n))}$

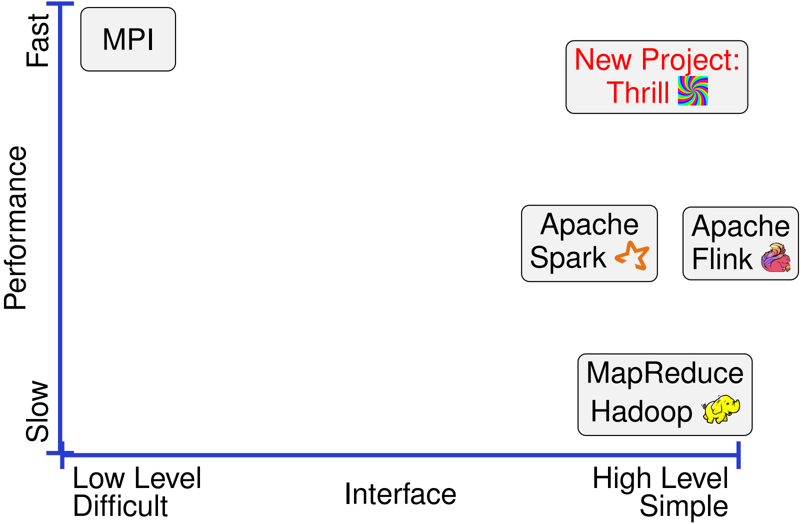
Flavours of Big Data Frameworks

- 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.

Big Data Batch Processing



Big Data Batch Processing

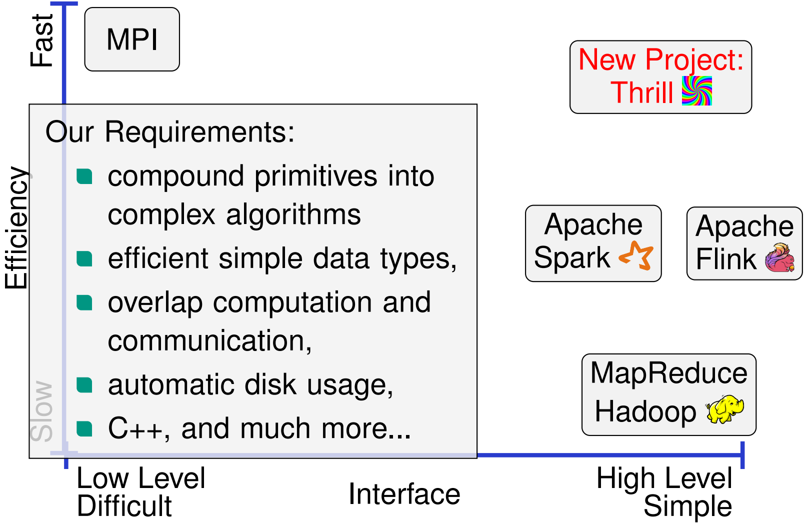


Projektpraktikum: Verteilte Datenverarbeitung mit MapReduce

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

Institute of Theoretical Informatics – Algorithmics

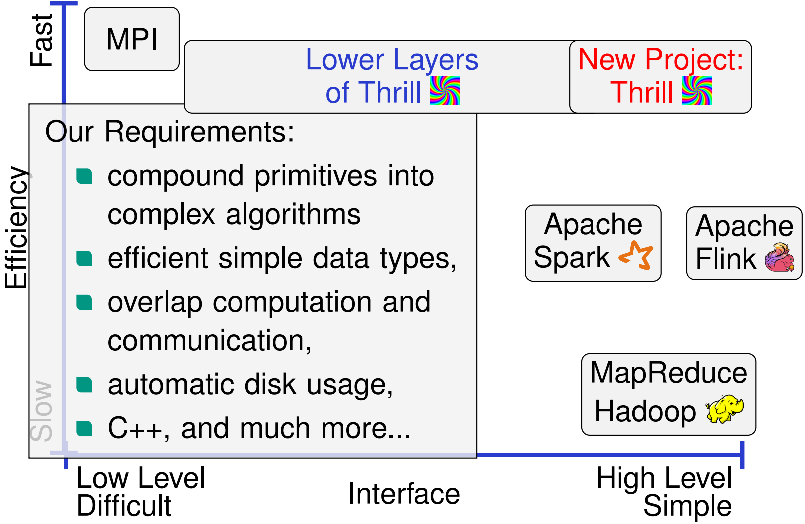
Big Data Batch Processing



Our Requirements:

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

Big Data Batch Processing



Fast

MPI

Lower Layers of Thrill

New Project: Thrill

Efficiency

Our Requirements:

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

Apache Spark

Apache Flink

MapReduce Hadoop

Slow

Low Level Difficult

Interface

High Level Simple

Why another Big Data Framework?

Sorting Records	Hadoop	Spark	Spark
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Hosts	2100	206	190
# Cores	50400	6592	6080
# Reducers	10 000	29 000	250 000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/host	11.2 MB/sec	345 MB/sec	375 MB/sec
Daytona Rules	Yes	Yes	No
Environment	dedicated	EC2 (i2.8xlarge)	

source: `http://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`

EC2 (i2.8xlarge): 32 core each, 244 GiB RAM,
8 x 800 GiB SSD (≈ 8 x 400 MB/s), 10 GBit Ethernet (≈ 800 MB/s).

Thrill's Design Goals

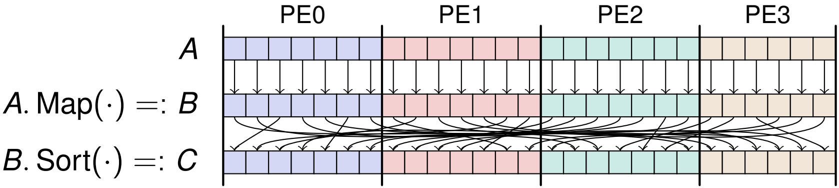
- An easy way to program distributed algorithms in C++.
- Distributed arrays of small items (characters or integers).
- High-performance, parallelized C++ operations.
- Locality-aware, in-memory computation.
- Transparently use disk if needed
  ⇒ external memory or cache-oblivious algorithms.
- Avoid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

Current Status:

- open-source prototype at `http://github.com/thrill/thrill`.

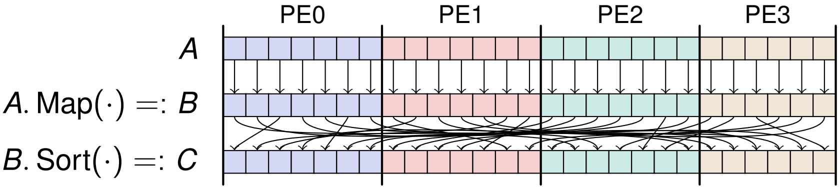
Distributed Immutable Array (DIA)

- 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.



Distributed Immutable Array (DIA)

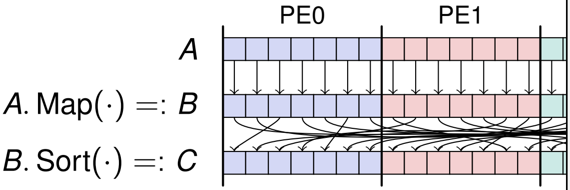
- User Programmer's View:
  - `DIA<T>` = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use transformations and actions.

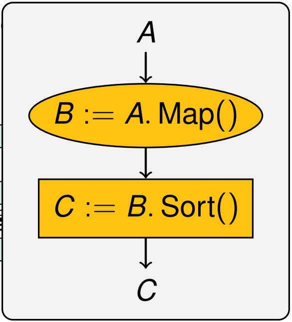


- Framework Designer's View:
  - Goals: distribute work, optimize execution on cluster, add redundancy where applicable. $\Longrightarrow$ build data-flow graph.
  - `DIA<T>` = chain of computation items
  - Let distributed operations choose "materialization".

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 (Excerpt)

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

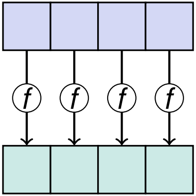
Sort()	Sort a DIA using comparisons.
ReduceBy()	Shuffle with Key Extractor, Hasher, and associative Reducer.
GroupBy()	Like ReduceBy, but with a general Reducer.
PrefixSum()	Compute (generalized) prefix sum on DIA.
Window$_k$()	Scan all k consecutive DIA items.
Zip()	Combine equal sized DIAs item-wise.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

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

Local Operations (LOps)

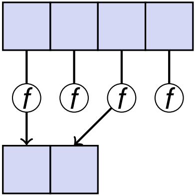
Map$(f) : \langle A \rangle \to \langle B \rangle$

$f : A \to B$



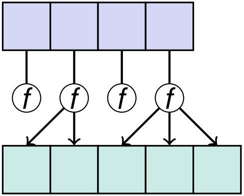
Filter$(f) : \langle A \rangle \to \langle A \rangle$

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



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

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



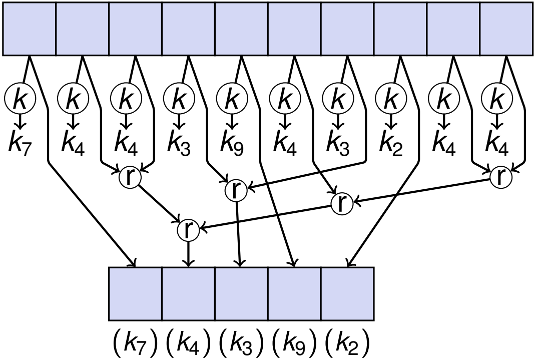
Currently: no rebalancing during LOps.

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

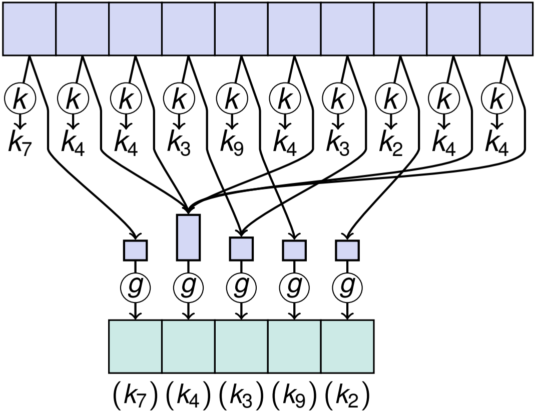


DOps: GroupByKey

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

$k : A \to K$ key extractor

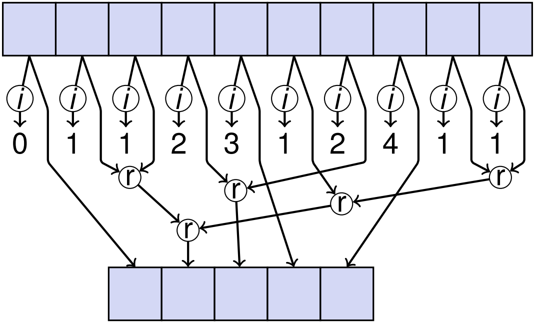
$g : \mathit{iterable}(A) \to B$ group function



DOps: ReduceToIndex

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

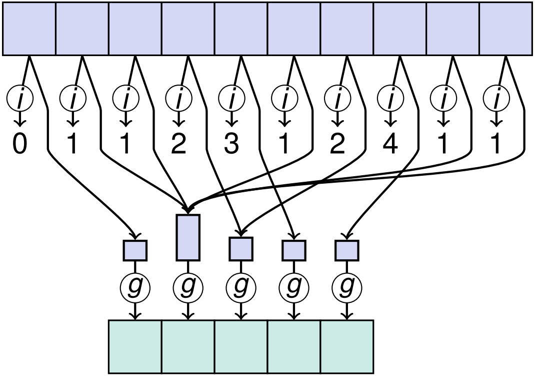
$i : A \to \{0..n-1\}$	index extractor
$n \in \mathbb{N}_0$	result size
$r : A \times A \to A$	reduction



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

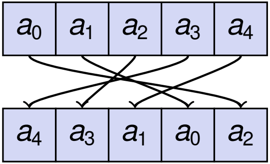


DOps: Sort and Merge

$\textbf{Sort}(o) : \langle A \rangle \to \langle A \rangle$

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

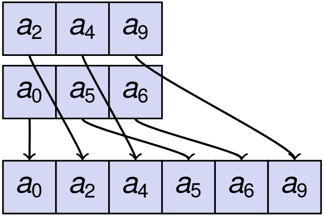
(less) order relation



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

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

(less) order relation

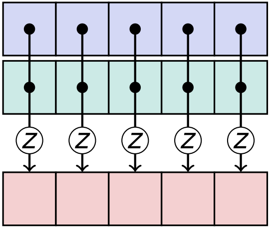


DOps: Zip and Window

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

$z : A \times B \to C$

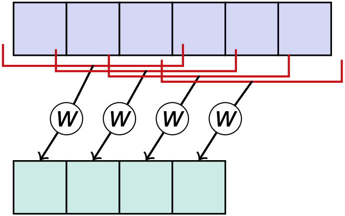
zip function



$\textbf{Window}(k, w) : \langle A \rangle \to \langle B \rangle$

$k \in \mathbb{N}$ window size

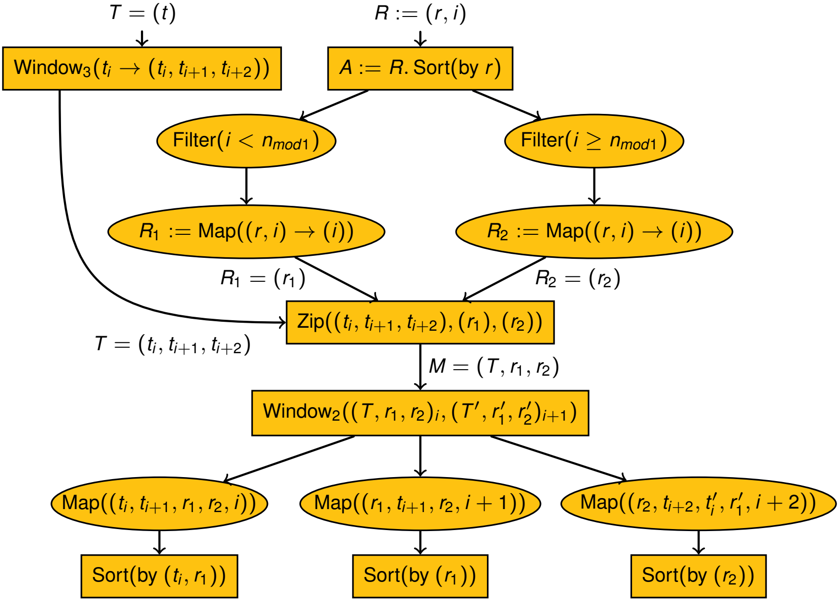
$w : A^k \to B$ window function



Example: WordCount in Thrill

```cpp
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>
}
```

Exert of DC3's Data-Flow Graph



$T = (t)$

$R := (r, i)$

$\text{Window}_3(t_i \rightarrow (t_i, t_{i+1}, t_{i+2}))$

$A := R.\,\text{Sort}(\text{by } r)$

$\text{Filter}(i < n_{mod1})$

$\text{Filter}(i \geq n_{mod1})$

$R_1 := \text{Map}((r, i) \rightarrow (i))$

$R_2 := \text{Map}((r, i) \rightarrow (i))$

$R_1 = (r_1)$

$R_2 = (r_2)$

$\text{Zip}((t_i, t_{i+1}, t_{i+2}), (r_1), (r_2))$

$T = (t_i, t_{i+1}, t_{i+2})$

$M = (T, r_1, r_2)$

$\text{Window}_2((T, r_1, r_2)_i, (T', r_1', r_2')_{i+1})$

$\text{Map}((t_i, t_{i+1}, r_1, r_2, i))$

$\text{Map}((r_1, t_{i+1}, r_2, i+1))$

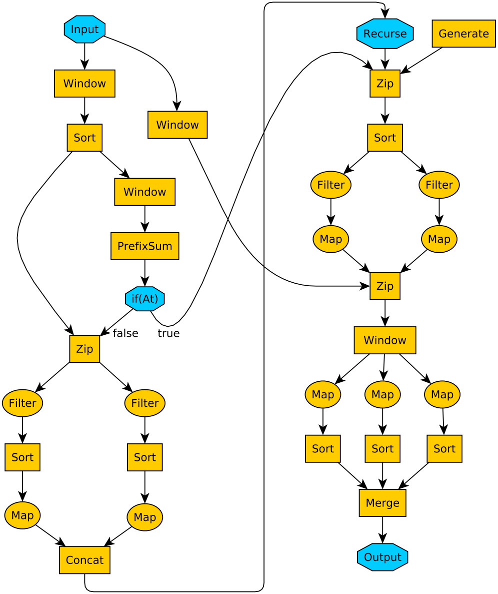
$\text{Map}((r_2, t_{i+2}, t_i', r_1', i+2))$

$\text{Sort}(\text{by } (t_i, r_1))$

$\text{Sort}(\text{by } (r_1))$

$\text{Sort}(\text{by } (r_2))$

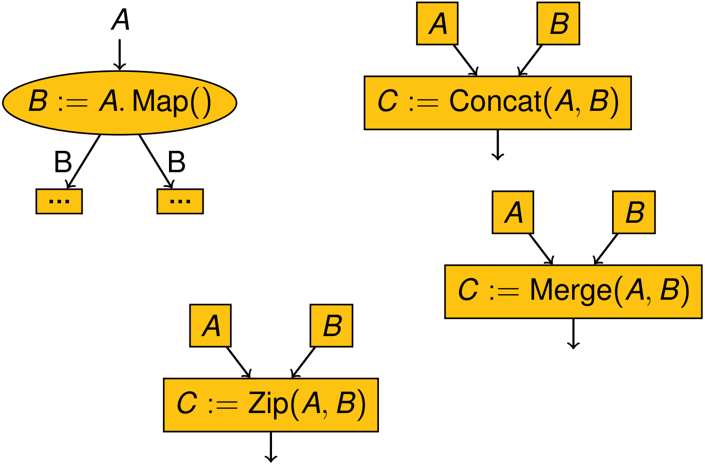
A Suffix Sorting Algorithm: DC3



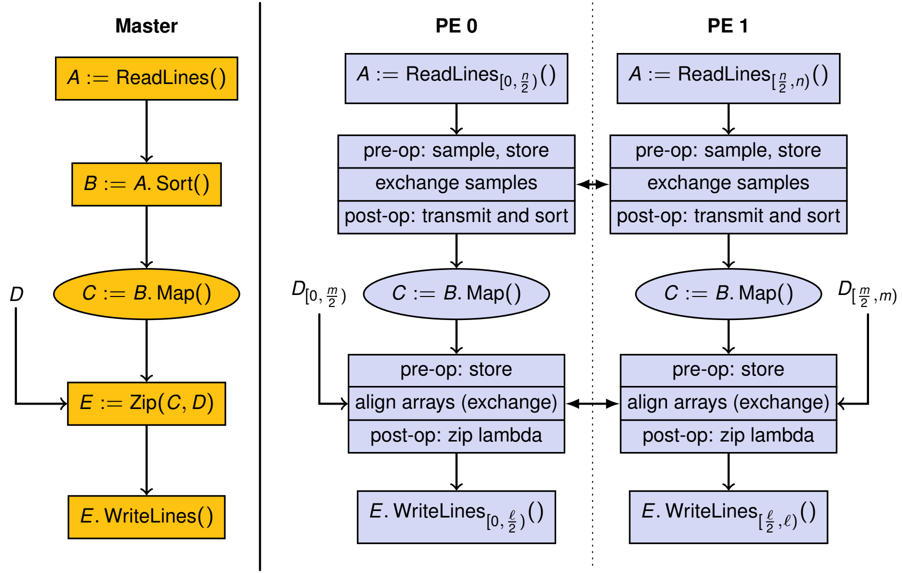
Structure of Data-Flow Graph

Any node can fork DIAs.

Combines are DOps.



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

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

$D_{[0,\frac{m}{2})}$

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

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

$D_{[\frac{m}{2},m)}$

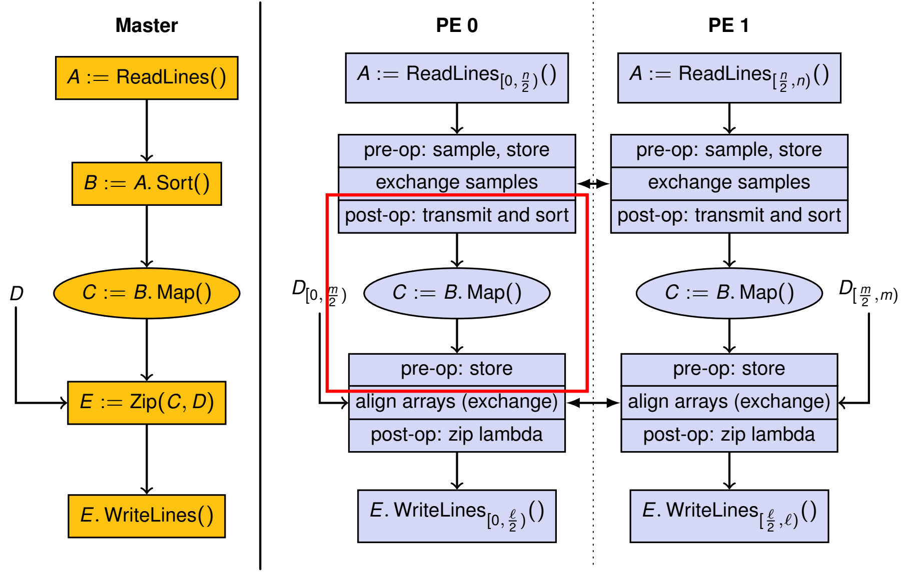
pre-op: store

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

$E := \text{Zip}(C, D)$

$E.\text{WriteLines}()$

PE 0

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

pre-op: sample, store

exchange samples

post-op: transmit and sort

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

$D_{[0,\frac{m}{2})}$

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

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

$D_{[\frac{m}{2},m)}$

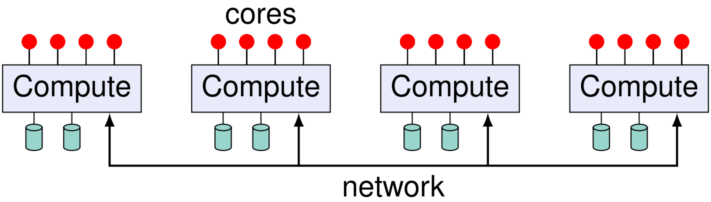
pre-op: store

align arrays (exchange)

post-op: zip lambda

$E.\text{WriteLines}_{[\frac{\ell}{2},\ell)}()$

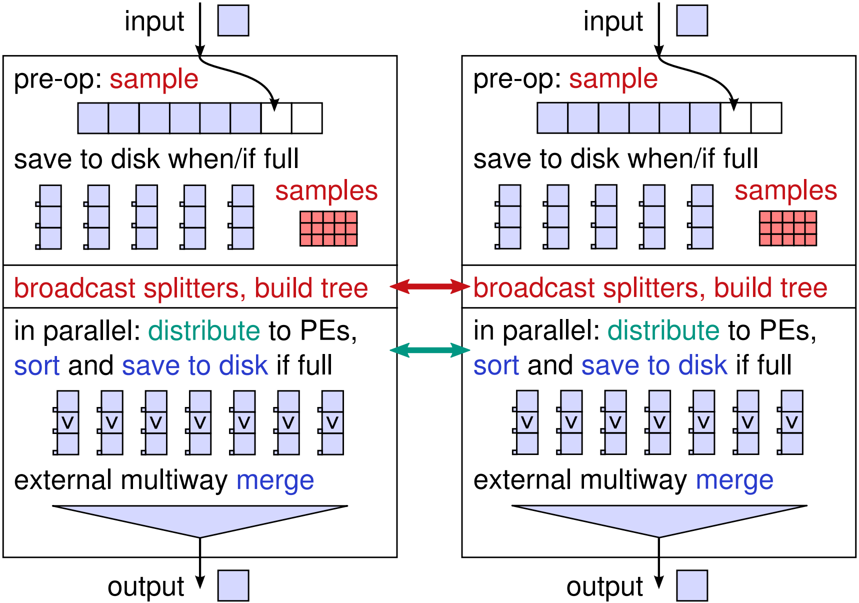
Execution on Cluster



- 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.

Sorting DOp



input

pre-op: sample

save to disk when/if full

samples

broadcast splitters, build tree

in parallel: distribute to PEs,
sort and save to disk if full

external multiway merge

output

input

pre-op: sample

save to disk when/if full

samples

broadcast splitters, build tree

in parallel: distribute to PEs,
sort and save to disk if full

external multiway merge

output

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.

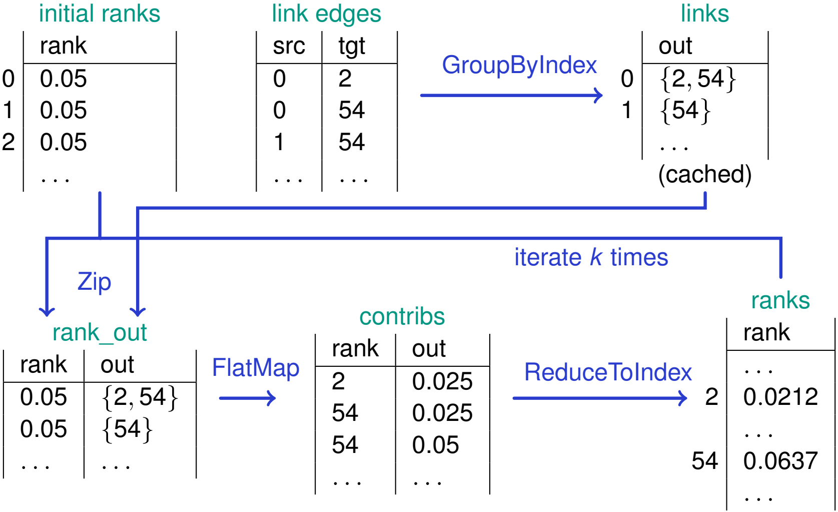
K-Means

- 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.

PageRank in Thrill



initial ranks

	rank
0	0.05
1	0.05
2	0.05
	. . .

link edges

src	tgt
0	2
0	54
1	54
. . .	. . .

GroupByIndex →

links

	out
0	$\{2, 54\}$
1	$\{54\}$
	. . .

(cached)

Zip

iterate k times

rank_out

rank	out
0.05	$\{2, 54\}$
0.05	$\{54\}$
. . .	. . .

FlatMap →

contribs

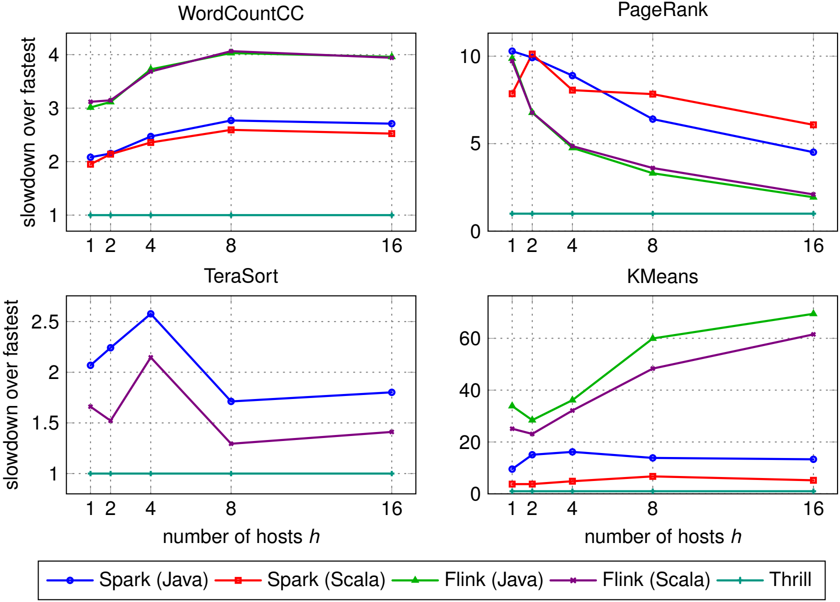
rank	out
2	0.025
54	0.025
54	0.05
. . .	. . .

ReduceToIndex →

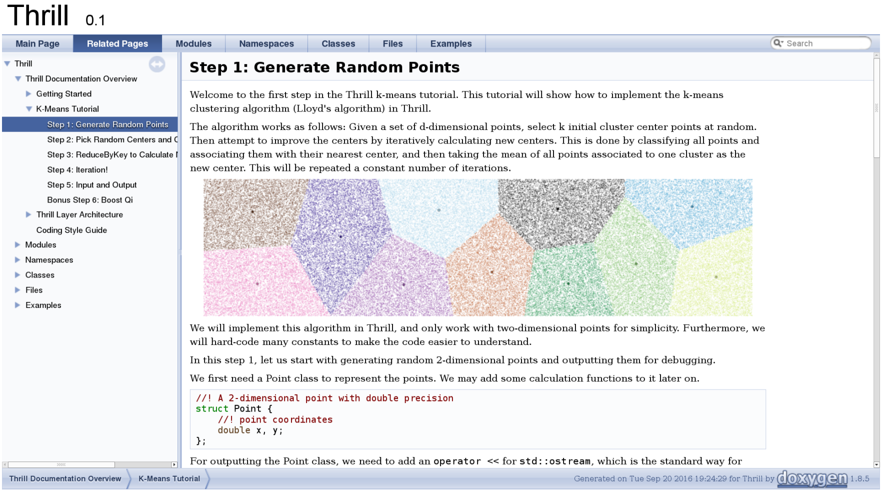
ranks

	rank
	. . .
2	0.0212
	. . .
54	0.0637
	. . .

Experimental Results: Slowdowns



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.

```
//! 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` 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?