Karlsruhe Institute of Technology

Thrill Tutorial: High-Performance Algorithmic Distributed Computing with C++

Timo Bingmann · 2020-06-01 @ Online Tutorial Recording

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS

Abstract

In this tutorial we present our 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 can be combined into larger more complex algorithms, such as WordCount, PageRank, and suffix sorting. Such compounded algorithms can then be run on very large inputs using a distributed computing cluster with external memory.

After introducing the audience to Thrill we guide participants through the initial steps of downloading and compiling the software package. The tutorial then continues to give an overview of the challenges of programming real distributed machines and models and frameworks for achieving this goal. With these foundations, Thrill's DIA programming model is introduced with an extensive listing of DIA operations and how to actually use them. The participants are then given a set of small example tasks to gain hands-on experience with DIAs.

After the hands-on session, the tutorial continues with more details on how to run Thrill programs on clusters and how to generate execution profiles. Then, deeper details of Thrill's internal software layers are discussed to advance the participants' mental model of how Thrill executes DIA operations. The final hands-on tutorial is designed as a concerted group effort to implement K-means clustering for 2D points.

Table of Contents

1. Thrill Motivation Pitch
   - Benchmarks and Introduction
   - Tutorial: Clone, Compile, and Run Hello World
2. Introduction to Parallel Machines
   - The Real Deal: Examples of Machines
   - Networks: Types and Measurements
   - Models
   - Implementations and Frameworks
3. The Thrill Framework
   - Thrill’s DIA Abstraction and List of Operations
   - Tutorial: Playing with DIA Operations
   - Execution of Collective Operations in Thrill
   - Tutorial: Running Thrill on a Cluster
   - Tutorial: Logging and Profiling
   - Going Deeper into Thrill
   - Tutorial: First Steps towards k-Means
   - Conclusion

1 Thrill Motivation Pitch

- **Benchmarks and Introduction**
- Tutorial: Clone, Compile, and Run Hello World

Weak-Scaling Benchmarks

WordCountCC – $h \cdot 49\,\text{GiB}$ 222 lines

- Reduce text files from CommonCrawl web corpus.

PageRank – $h \cdot 2.7\,\text{GiB}$, $|E| \approx h \cdot 158\,\text{M}$ 410 lines

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

TeraSort – $h \cdot 16\,\text{GiB}$ 141 lines

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

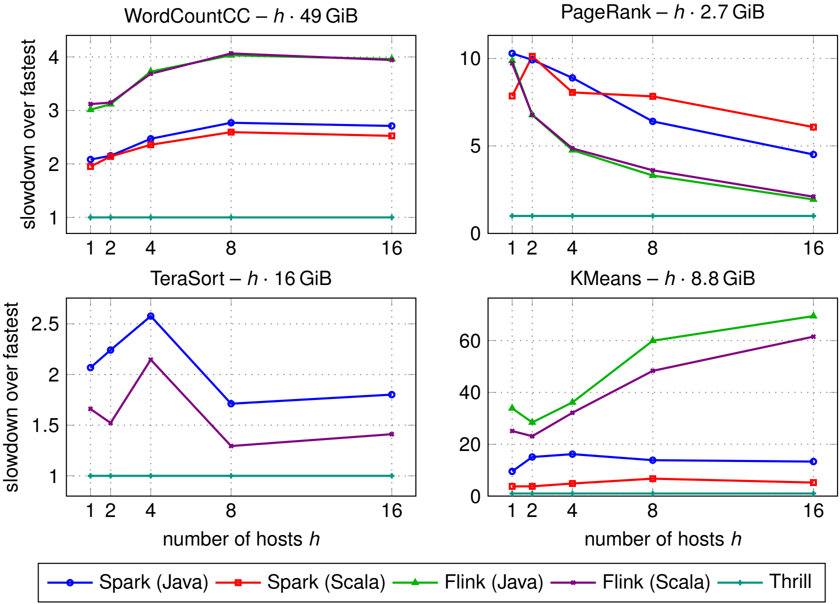
K-Means – $h \cdot 8.8\,\text{GiB}$ 357 lines

- Calculate K-Means clustering with 10 iterations.

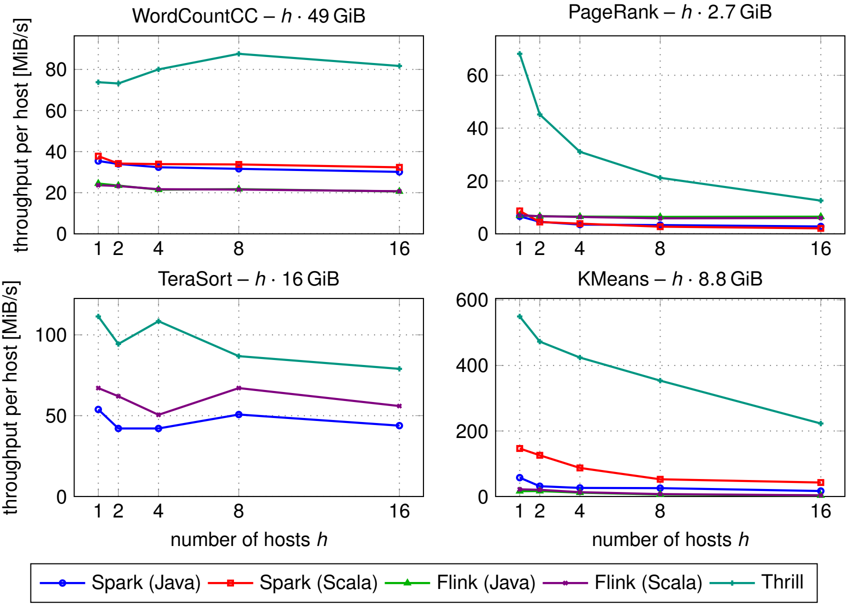
Platform: $h \times$ r3.8xlarge systems on Amazon EC2 Cloud

- 32 cores, Intel Xeon E5-2670v2, 2.5 GHz clock, 244 GiB RAM, 2 x 320 GB local SSD disk, ≈ 400 MiB/s read/write Ethernet network ≈ 1000 MiB/s throughput, Ubuntu 16.04.

Experimental Results: Slowdowns



Experimental Results: Throughput



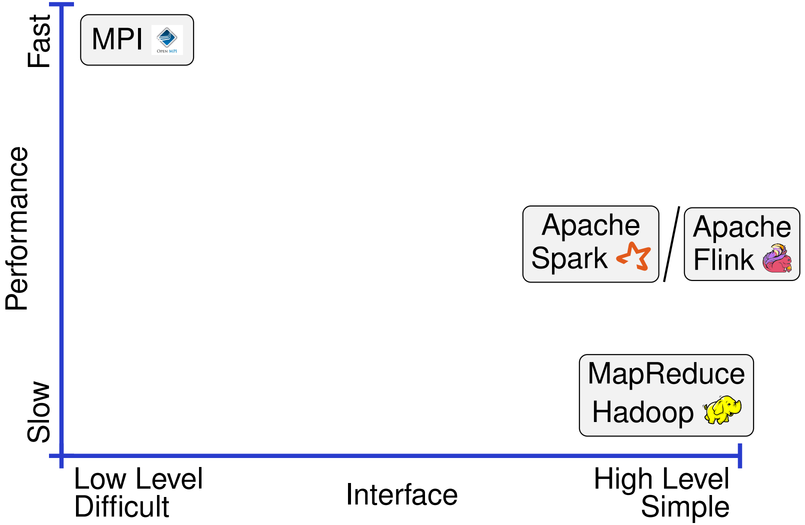
0 1 2 3 4 5 6 7 8 9 10 11 12 13

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

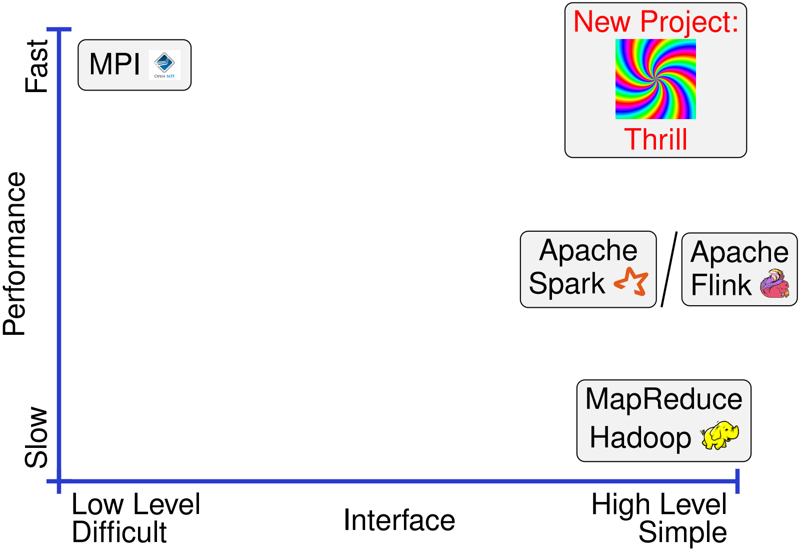
SA_i	LCP_i	$T_{SA_i \ldots n}$
13	-	$
11	0	be$
2	2	beornottobe$
12	0	e$
3	1	eornottobe$
6	0	nottobe$
10	0	obe$
1	3	obeornottobe$
4	1	ornottobe$
7	1	ottobe$
5	0	rnottobe$
9	4	tobe$
0	1	tobeornottobe$
8	0	ttobe$

bwUniCluster
KIT (SCC)

Big Data Batch Processing



Big Data Batch Processing



Fast

Performance

Slow

MPI

New Project:

Thrill

Apache Spark / Apache Flink

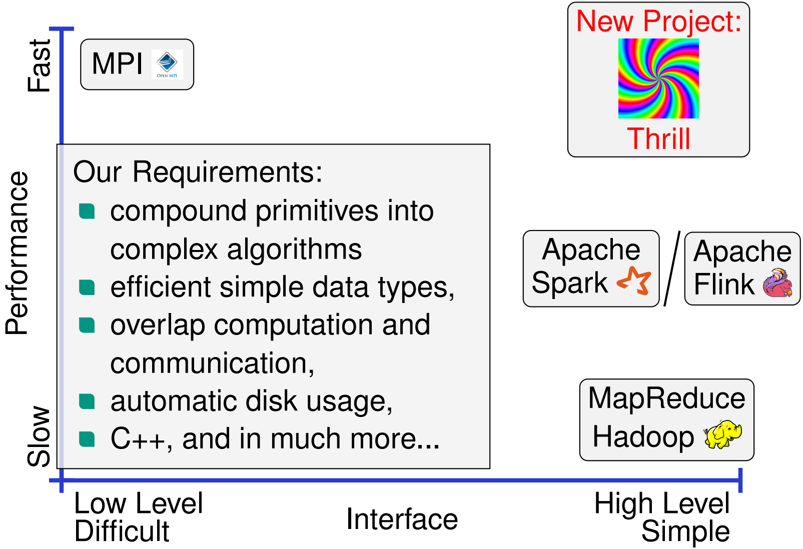
MapReduce Hadoop

Low Level Difficult

Interface

High Level Simple

Big Data Batch Processing



Fast

Performance

Slow

MPI

New Project:

Thrill

Our Requirements:

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

Apache Spark / Apache Flink

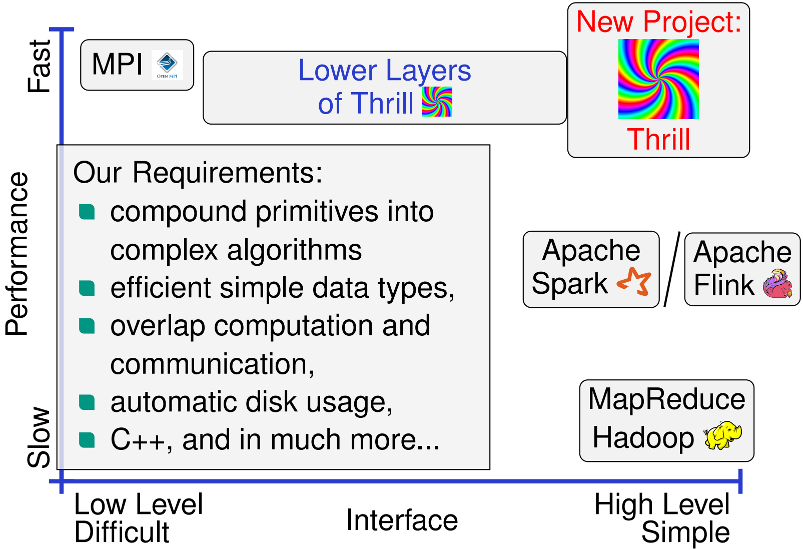
MapReduce Hadoop

Low Level
Difficult

Interface

High Level
Simple

Big Data Batch Processing



Our Requirements:

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

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

Thrill is a moving target,
this tutorial is for the version in June 2020.

Thrill's Goal and Current Status

An easy way to program distributed external algorithms in C++.

Current Status:

- Open-source prototype at http://github.com/thrill/thrill.
- $\approx 60\,$K lines of C++14 code, written by ≥ 12 contributors.
- Published at IEEE Conference on Big Data [B, et al. '16]
- Faster than Apache Spark and Flink on five micro benchmarks: WordCount1000/CC, PageRank, TeraSort, and K-Means.

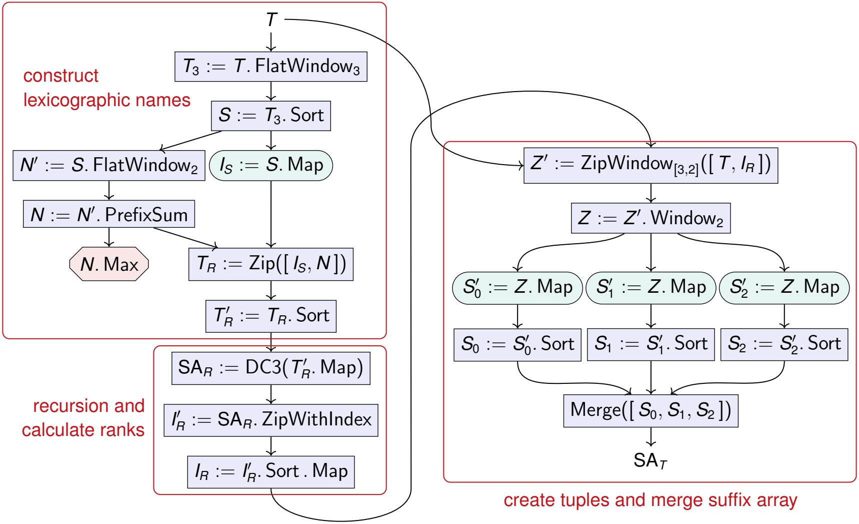
Case Studies:

- Five suffix sorting algorithms [B, Gog, Kurpicz, BigData'18]
- Louvain graph clustering algorithm [Hamann et al. Euro-Par'18]
- Process scientific data on HPC (poster) [Karabin et al. SC'18]
- More: stochastic gradient descent, triangle counting, etc.
- Future: fault tolerance, scalability, predictability, and more.

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) {
            tlx::split_view(' ', line, [&](tlx::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>
}
```

DC3 Data-Flow Graph with Recursion



construct lexicographic names

T

$T_3 := T.\,\mathrm{FlatWindow}_3$

$S := T_3.\,\mathrm{Sort}$

$N' := S.\,\mathrm{FlatWindow}_2$

$I_S := S.\,\mathrm{Map}$

$N := N'.\,\mathrm{PrefixSum}$

$N.\,\mathrm{Max}$

$T_R := \mathrm{Zip}([\,I_S, N\,])$

$T'_R := T_R.\,\mathrm{Sort}$

recursion and calculate ranks

$\mathrm{SA}_R := \mathrm{DC3}(T'_R.\,\mathrm{Map})$

$I'_R := \mathrm{SA}_R.\,\mathrm{ZipWithIndex}$

$I_R := I'_R.\,\mathrm{Sort}\,.\,\mathrm{Map}$

$Z' := \mathrm{ZipWindow}_{[3,2]}([\,T, I_R\,])$

$Z := Z'.\,\mathrm{Window}_2$

$S'_0 := Z.\,\mathrm{Map}$

$S'_1 := Z.\,\mathrm{Map}$

$S'_2 := Z.\,\mathrm{Map}$

$S_0 := S'_0.\,\mathrm{Sort}$

$S_1 := S'_1.\,\mathrm{Sort}$

$S_2 := S'_2.\,\mathrm{Sort}$

$\mathrm{Merge}([\,S_0, S_1, S_2\,])$

SA_T

create tuples and merge suffix array

1 Thrill Motivation Pitch

- Benchmarks and Introduction
- Tutorial: Clone, Compile, and Run Hello World

Tutorial: Clone, Compile, and Run

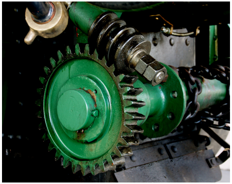
This tutorial focuses on Linux and similar systems. Windows/Visual C++ is supported using CMake, but needs some extra steps.

- Clone the tutorial example repository:

```
git clone --recursive https://github.com/thrill/tutorial-project.git
```

- Compile with auto-detected C++14 GCC compiler:

```
$ cd tutorial-project
$ ./compile.sh
        -DTHRILL_BUILD_EXAMPLES=ON
```



- Run simple example:

```
$ cd build
$ ./simple
```

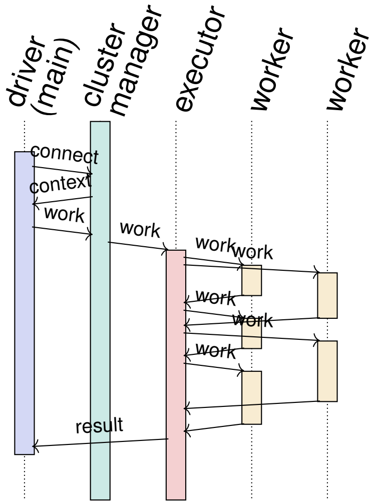
Tutorial: Run Hello World

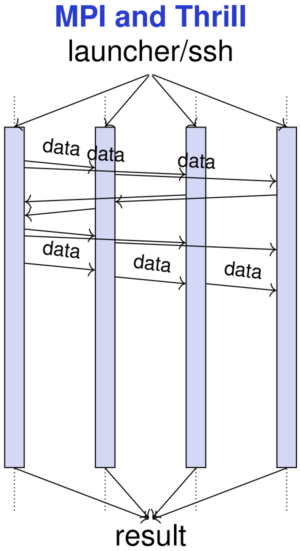
```
#include <thrill/thrill.hpp>
#include <iostream>

void program(thrill::Context& ctx) {
    std::cout << "Hello World, I am "
              << ctx.my_rank() << std::endl;
}

int main(int argc, char* argv[]) {
    return thrill::Run(program);
}
```

Control Model: Spark vs. MPI/Thrill





Tutorial: Hello World Output

```
Thrill: using 7.709 GiB RAM total, BlockPool=2.570 GiB,
    workers=657.877 MiB, floating=2.570 GiB.
Thrill: running locally with 2 test hosts and 4 workers per host
    in a local tcp network.
Thrill: using 7.709 GiB RAM total, BlockPool=2.570 GiB,
    workers=657.877 MiB, floating=2.570 GiB.
Thrill: no THRILL_LOG was found, so no json log is written.
[main 000000] FOXXLL v1.4.99 (prerelease/Release)
    (git a4a8aeee64743f845c5851e8b089965ea1c219d7)
[main 000001] foxxll: Using default disk configuration.
[main 000002] foxxll: Disk '/var/tmp/thrill.30713.tmp' is allocated,
    space: 1000 MiB, I/O implementation: syscall queue=0 devid=0 unlink_on_open
Hello World, I am 0
Hello World, I am 1
Hello World, I am 2
Hello World, I am 7
Hello World, I am 3
Hello World, I am 6
Hello World, I am 4
Hello World, I am 5
Thrill: ran 6.7e-05s with max 0.000 B in DIA Blocks, 0.000 B network traffic,
    0.000 B disk I/O, and 0.000 B max disk use.
malloc_tracker ### exiting, total: 1163264, peak: 1163264,
    current: 0 / 65536, allocs: 71, unfreed: 4
```

2 Introduction to Parallel Machines

- **The Real Deal: Examples of Machines**
- Networks: Types and Measurements
- Models
- Implementations and Frameworks

The Real Deal: HPC Supercomputers

Summit at Oak Ridge National Laboratory (ORNL)
#1 in TOP500 list since June 2018

CC BY Oak Ridge Leadership Computing Facility at ORNL

4 356 nodes with two 22-core Power9 CPUs and six NVIDIA Tesla V100 GPUs each. That are 202 752 physical CPU cores plus 2 211 840 GPU SMs reaching 148.6 petaflops. The nodes are connected with a Mellanox dual-rail EDR InfiniBand network. 2×800 GB non-volatile RAM per node.

The Real Deal: HPC Supercomputers

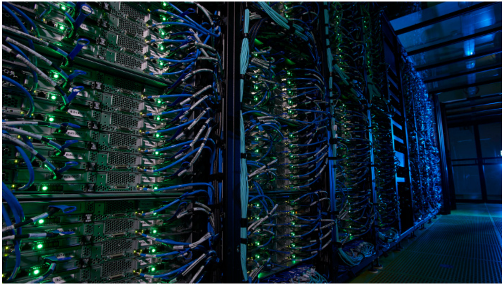
SuperMUC-NG at Leibniz Rechenzentrum (LRZ) in Munich
#9 in TOP500 list from June 2019

Picture: Veronika Hohenegger, LRZ

6 336 nodes with (24+24)-core Intel Xeon 8174 CPUs with 96 GiB RAM. The nodes are connected with an Intel Omni-Path 100 GB/s. In total 152 064 physical cores reaching 19.5 petaflops. No local disks.

The Real Deal: HPC Supercomputers

ForHLR II at Steinbuch Centre for Computing (SCC) at KIT



Close-up of ForHLR II, Andreas Drollinger, KIT (SCC)

1 152 nodes with two (10+10)-core Intel Xeon E5-2660 v3 with 64 GiB RAM. The nodes are connected with a Mellanox FDR adapter to an InfiniBand 4X EDR interconnect. In total 23 040 physical cores reaching about 1 petaflop. One 480 GB local SSD per node.

The Real Deal: Cloud Computing

amazon web services

Microsoft Azure

Google Cloud

Not much is public about their size, infrastructure, or even location.

Delivers virtualized computer, disk, and network resources.

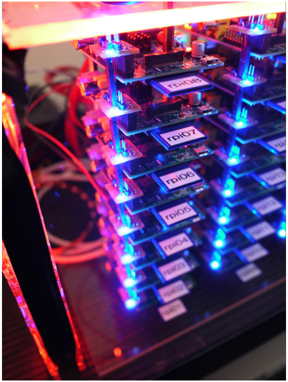
Probably built on commodity hardware, such as Intel processors, with some proprietary customizations and a virtualization stack.

Examples of AWS instances:

- m5.12xlarge has 48 vCPUs with 192 GB RAM and 10 Gb/s network, and costs $2.31 per hour
- i3.8xlarge has 32 vCPUs with 244 GB RAM, 10 Gb/s network, 4 × 1.9 TB NVMe SSDs, and costs $2.50 per hour

The Real Deal: Custom Local Clusters

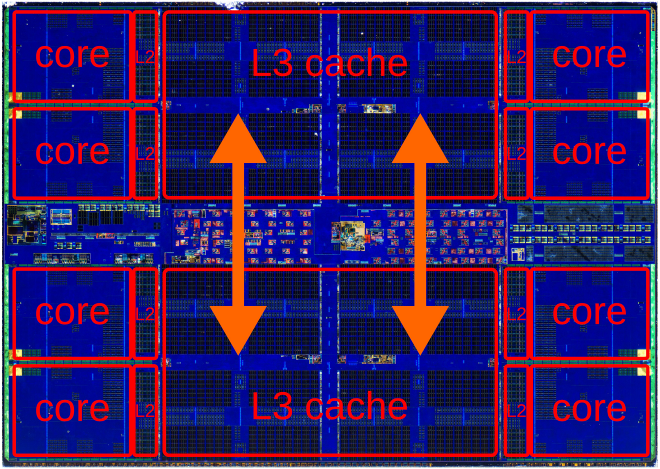
heterogeneous
server installations



Raspberry Pi clusters

photo and report by Joshua Kiepert,
see *Joshua Kiepert, "Creating a Raspberry Pi-based Beowulf Cluster." Technical Report, Boise State University (2013).*

The Real Deal: Shared Memory



AMD Ryzen 5 3600, 6 cores, 3.60 GHz, 7 nm, 32 MiB L3 cache,
die photo from https://www.flickr.com/photos/130561288@N04/albums, modified

The Real Deal: GPUs

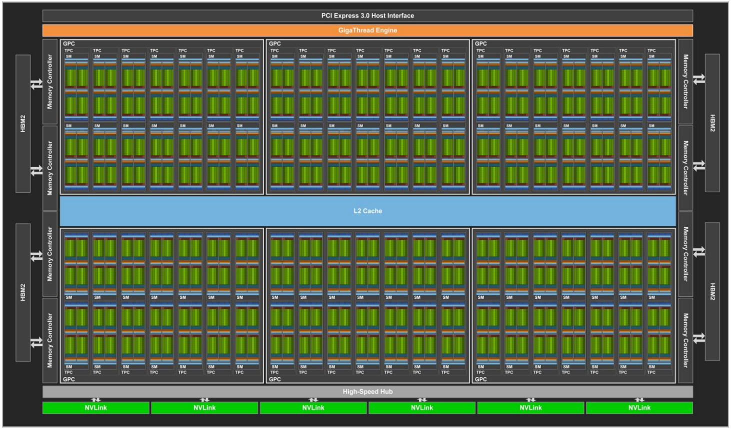


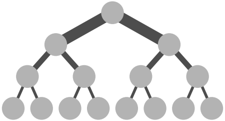
diagram from NVIDIA Tesla V100 GPU architecture whitepaper

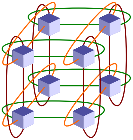
NVIDIA Tesla V100 with 80 streaming multiprocessors (SMs), each containing 64 CUDA cores, in total of 5 120 cores and up to 32 GB RAM.

2 Introduction to Parallel Machines

- The Real Deal: Examples of Machines
- **Networks: Types and Measurements**
- Models
- Implementations and Frameworks

Types of Networks

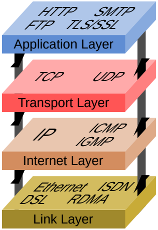




- HPC supercomputers:
  - remote direct memory access (RDMA)
  - different network topologies: fat trees, *k*D-torus, islands.

- cloud computing and local Ethernet clusters:
  - TCP/UDP/IP stack
  - switched 100 Mb/s, 1 Gb/s, 10 Gb/s, or more

- shared-memory many-core and GPU systems
  - implicit communication via cache coherence



HTTP *SMTP* *FTP* *TLS/SSL*

Application Layer

TCP *UDP*

Transport Layer

IP *ICMP* *IGMP*

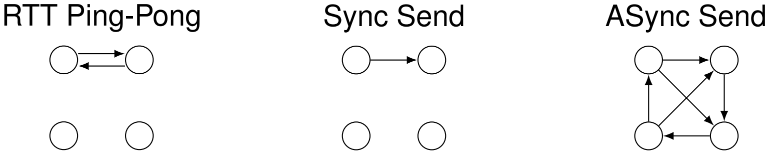
Internet Layer

Ethernet *ISDN* *DSL* *RDMA*

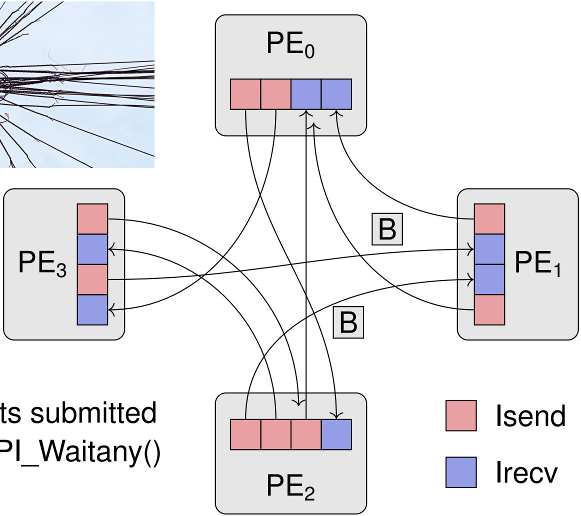
Link Layer

Round Trip Time (RTT) and Bandwidth

- 2 hosts in LAN at our institute at KIT 2019-08-08
  RTT: 140 µs, bandwidth sync: 941 MiB/s
- 4 × r3.8xlarge AWS instances with 10 Gb/s net 2016-07-14
  RTT: 100 µs, bandwidth sync: 389 MiB/s
- 4 × i3.4xlarge AWS instances with 10 Gb/s net 2017-12-17
  RTT: 81 µs, bandwidth sync: 1 144 MiB/s, async: 4 278 MiB/s
- 4 × ForHLR II hosts with RDMA/4X EDR Infiniband 2019-08-08
  RTT: 10.4 µs, bandwidth sync: 5 935 MiB/s, async: 5 554 MiB/s



MPI Random Async Block Benchmark



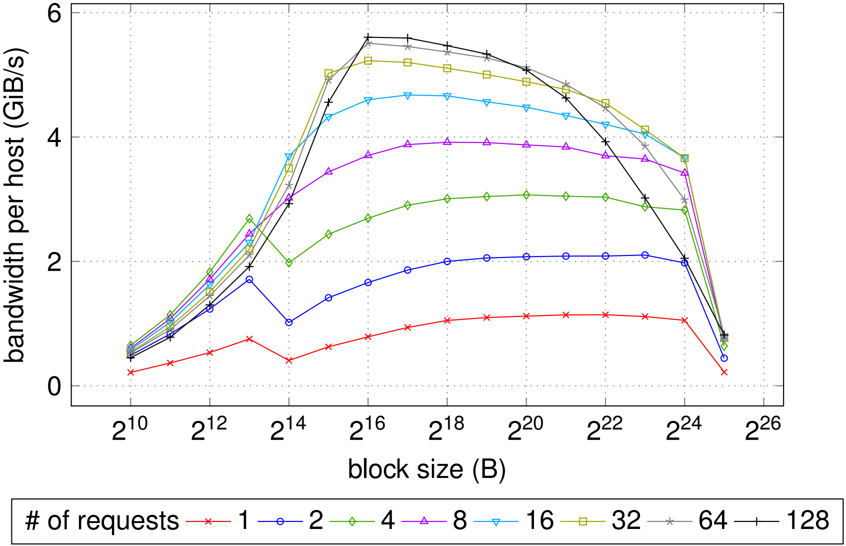
requests submitted
with MPI_Waitany()

🟥 Isend

🟦 Irecv

more: https://github.com/bingmann/mpi-random-block-test

Random Blocks on ForHLR II, 8 Hosts



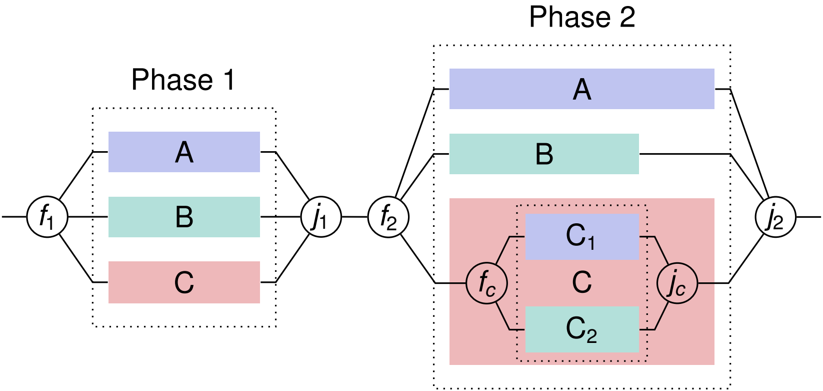
Variety of Parallel Computing Hosts

- cluster types: homogeneous or heterogeneous
- host types: commodity hardware, virtual instances on cloud computing platforms, shared-memory many-core systems, GPUs, or HPC systems with RDMA.
- storage devices:
  - no local storage
  - local storage: rotational disks, SSD, or NVMe devices
  - transparent distributed storage
- network interconnect:
  - implicit communication protocols
  - explicit communication: Ethernet, virtual networking, RDMA/Infiniband, etc.

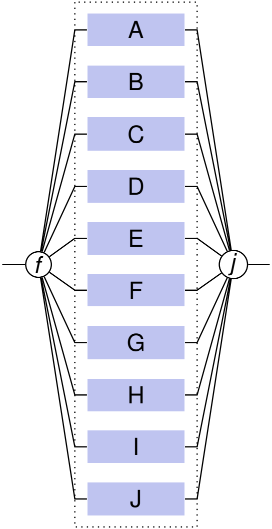
2 Introduction to Parallel Machines

- The Real Deal: Examples of Machines
- Networks: Types and Measurements
- **Models**
- Implementations and Frameworks

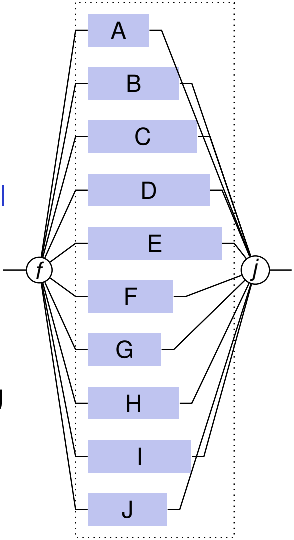
Control Model: Fork-Join



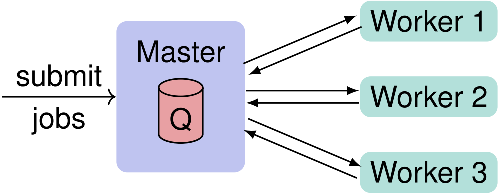
Trivially Parallelizable Work

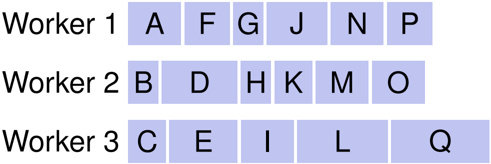


- many workloads are trivially parallelizable, also called embarrassingly parallel
- only one phase, no synchronization needed between tasks
- easy to schedule using batch processing systems



Control Model: Master-Worker





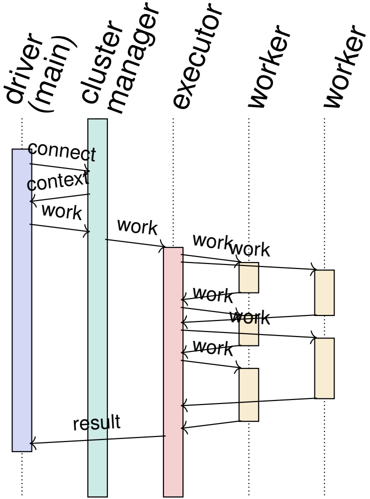
- master controls jobs on workers
- easy to add or replace workers
- implicit dynamic load balancing

- used by Apache Spark and Apache Flink

- single point of failure!
- not truly scalable!

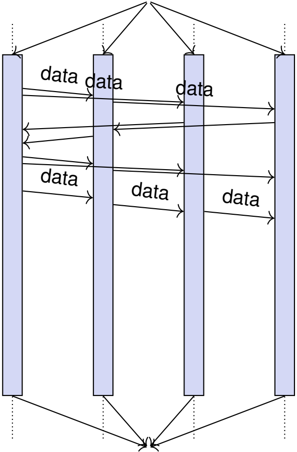
Control Model: Spark vs. MPI/Thrill

Apache Spark



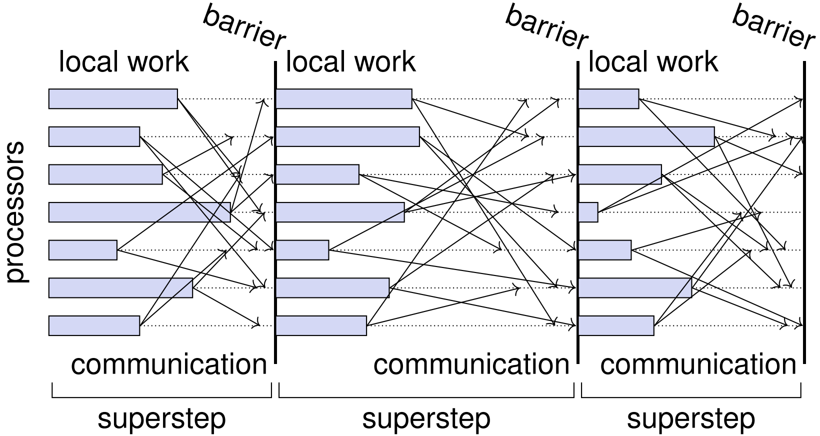
MPI and Thrill

launcher/ssh



result

Bulk Synchronous Parallel (BSP)



2 Introduction to Parallel Machines

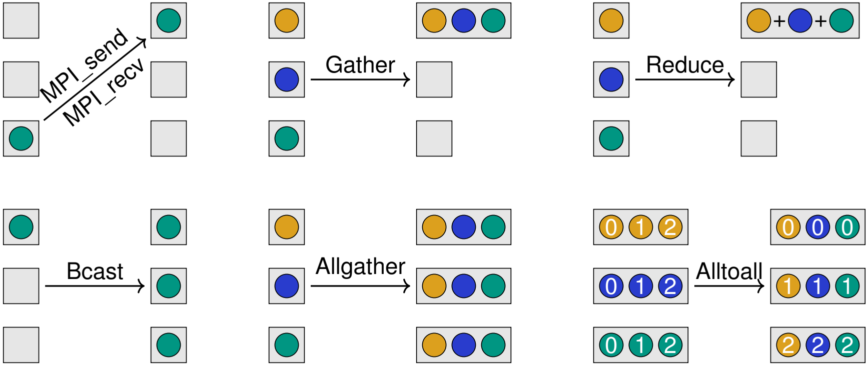
- The Real Deal: Examples of Machines
- Networks: Types and Measurements
- Models
- **Implementations and Frameworks**

MPI (Message Passing Interface)

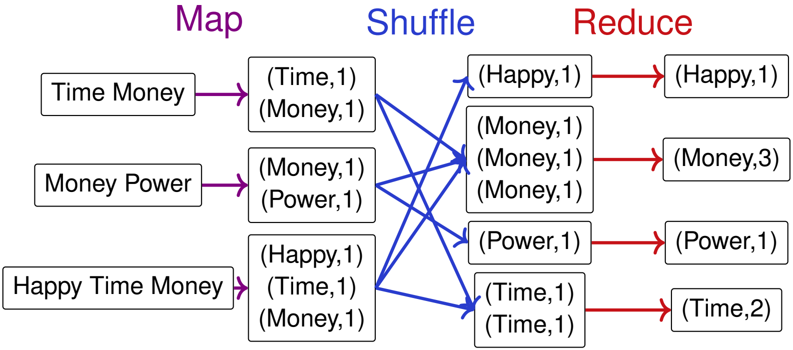
History:

- Version 1.0 from 1994 for C, C++, and Fortran.
- Still most used interface on supercomputers.

Collective Operations:



Map/Reduce Model



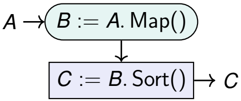
Computation model popularized in 2004
by Google with the name MapReduce.

Map/Reduce Framework

- Changes the perspective from the number of processors to how data is processed.
- A simple algorithmic and programming abstraction with
  - automatic parallelization of independent operations (map) and aggregation (reduce),
  - automatic distribution and balancing of data and work,
  - automatic fault tolerance versus hardware errors.

⇒ **all provided by MapReduce framework**

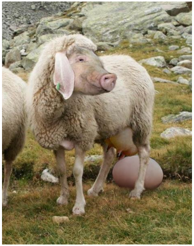
Apache Spark and Apache Flink



- New **post-**Map/Reduce frameworks use data-flow functional-style programming.
- Apache Spark started in 2009 in Berkeley.
  - central data structure: resilient distributed data sets (RDDs)
  - operations broken down into stages executed on cluster
  - driver initiates and controls execution of stages
- Apache Flink started as Stratosphere at TU Berlin.
  - first version (2010): “PACTs” and Nephele engine.
  - uses host language to construct data-flow graphs
  - optimizer and scheduler decide how to run them

Flavours of Big Data Frameworks

- Batch Processing
  Google's MapReduce, Hadoop MapReduce, Apache Spark, Apache Flink (Stratosphere).
- High Performance Computing (Supercomputers)
  MPI
- Real-time Stream Processing
  Apache Storm, Apache Spark Streaming.
- Interactive Cached Queries
  Google's Dremel, Powerdrill and BigQuery, Apache Drill.
- Sharded (NoSQL) Databases and Data Warehouses
  MongoDB, Apache Cassandra, Google BigTable, Amazon RedShift.
- Graph Processing
  Google's Pregel, GraphLab, Giraph, GraphChi.
- Machine Learning Frameworks and Libraries
  Tensorflow, Keras, scikit-learn, Microsoft Cognitive Toolkit.



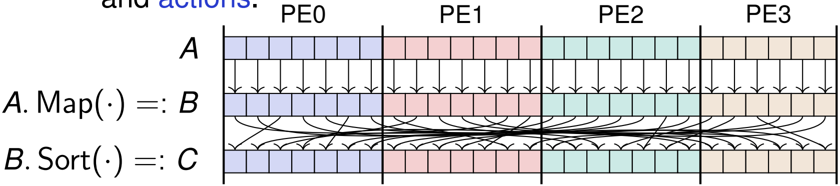
eierlegende Wollmilchsau
CC BY-SA Georg Mittenecker

3 The Thrill Framework

- **Thrill's DIA Abstraction and List of Operations**
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- Conclusion

Distributed Immutable Array (DIA)

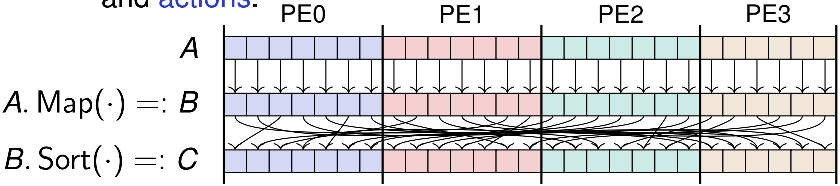
- User Programmer's View:
  - `DIA<T>` = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use transformations and actions.



PE0
PE1
PE2
PE3
A
$A.\,\mathrm{Map}(\cdot) =: B$
$B.\,\mathrm{Sort}(\cdot) =: C$

Distributed Immutable Array (DIA)

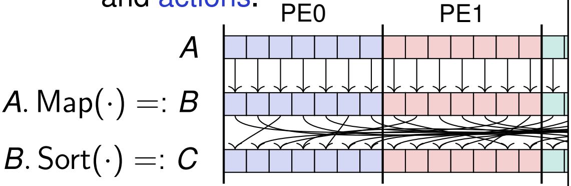
- User Programmer's View:
  - `DIA<T>` = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use transformations and actions.

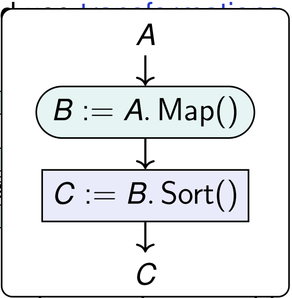


- Framework Designer's View:
  - Goals: distribute work, optimize execution on cluster, add redundancy where applicable. $\Longrightarrow$ build data-flow graph.
  - `DIA<T>` = chain of computation items
  - Let distributed operations choose "materialization".

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, 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".

List of Primitives (Excerpt)

- Local Operations (LOp): input is one item, output $\geq$ 0 items.
  Map(), Filter(), FlatMap(), BernoulliSample(), etc.
- Distributed Operations (DOp): input is a DIA, output is a DIA.

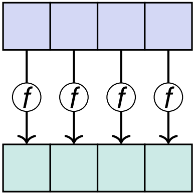
Sort()	Sort a DIA using comparisons.
ReduceBy()	Shuffle with Key Extractor, Hasher, and associative Reducer.
GroupBy()	Like ReduceBy, but with a general Reducer.
PrefixSum()	Compute (generalized) prefix sum on DIA.
Window$_k$()	Scan all k consecutive DIA items.
Zip()	Combine equal sized DIAs index-wise.

- Sources: read external data and start a DIA chain.
  Generate(), ReadLines(), ReadBinary(), etc.
- Actions: input is a DIA, output: $\geq$ 0 items on every worker.
  Sum(), Min(), WriteLines(), WriteBinary(), etc.

Local Operations (LOps)

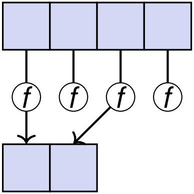
Map$(f) : \langle A \rangle \to \langle B \rangle$

$f : A \to B$



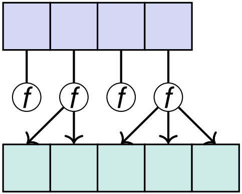
Filter$(f) : \langle A \rangle \to \langle A \rangle$

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



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

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

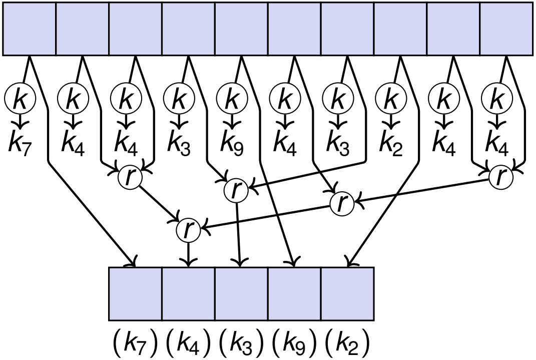


DOps: ReduceByKey

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

$k : A \to K$ key extractor

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

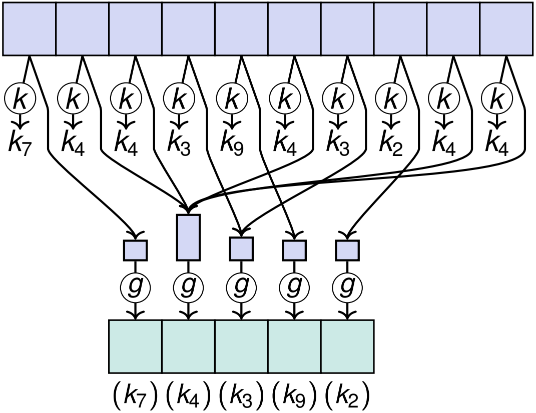


DOps: GroupByKey

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

$k : A \to K$ key extractor

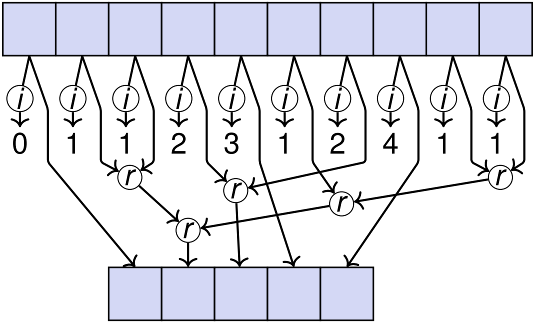
$g : \mathit{iterable}(A) \to B$ group function



DOps: ReduceToIndex

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

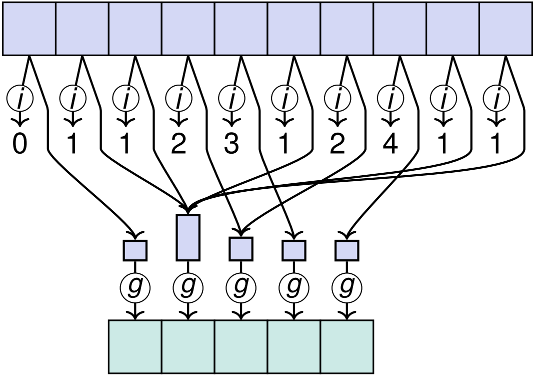
$i : A \to \{0..n-1\}$	index extractor
$r : A \times A \to A$	reduction
$n \in \mathbb{N}_0$	result size



DOps: GroupToIndex

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

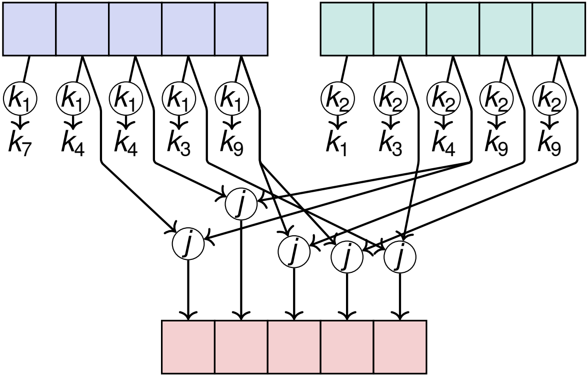
$i : A \to \{0..n-1\}$	index extractor
$g : \mathit{iterable}(A) \to B$	group function
$n \in \mathbb{N}_0$	result size



DOps: InnerJoin

$$\textbf{InnerJoin}(k_1, k_2, j) : \langle A \rangle \times \langle B \rangle \to \langle C \rangle$$

$k_1 : A \to K$	key extractor for A
$k_2 : B \to K$	key extractor for B
$j : A \times B \to C$	join function

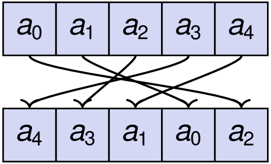


DOps: Sort and Merge

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

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

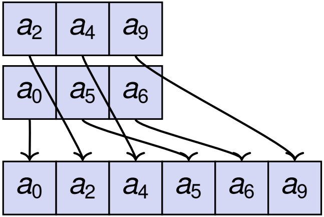
(less) order relation



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

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

(less) order relation



DOps: PrefixSum and ExPrefixSum

PrefixSum$(s, v) : \langle A \rangle \to \langle A \rangle$

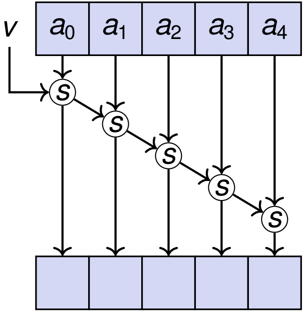
$s : A \times A \to A$ sum function

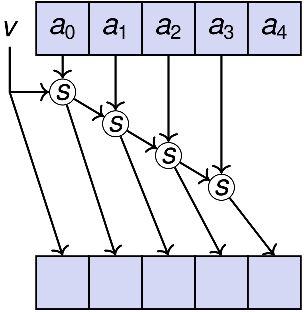
$v : A$ initial value

ExPrefixSum$(s, v) : \langle A \rangle \to \langle A \rangle$

$s : A \times A \to A$ sum function

$v : A$ initial value

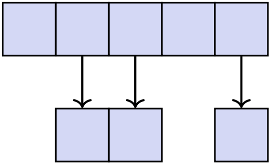




Sample (DOp), BernoulliSample (LOp)

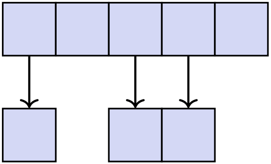
Sample$(k) : \langle A \rangle \to \langle A \rangle$

$k \in \mathbb{N}_0$ result size



BernoulliSample$(p) : \langle A \rangle \to \langle A \rangle$

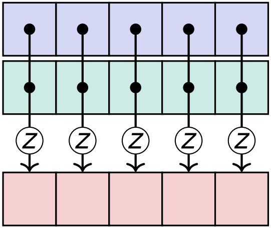
$p \in [0, 1]$ probability



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



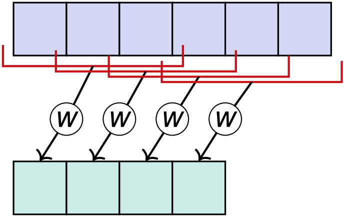
$$\textbf{ZipWithIndex}(z) : \langle A \rangle \to \langle C \rangle$$

$z : A \times \mathbb{N}_0 \to C$ zip function

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

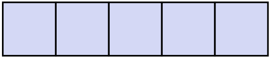
$k \in \mathbb{N}$ window size

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



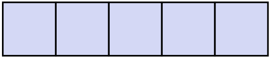
Auxiliary Ops: Cache and Collapse

Cache() : $\langle A \rangle \to \langle A \rangle$



Materializes a DIA, needed e.g. for caching or random data generation.

Collapse() : $\langle A, f_1, f_2 \rangle \to \langle A \rangle$



Folds local operation lambdas f_1, f_2 into a DIA, needed for iterations.

Source DOps: Generate, -ToDIA

Generate$(n, g) : \langle A \rangle$

$n \in \mathbb{N}_0$ result size

$g : \{0..n-1\} \to A$ generator

$g(0)$	$g(1)$	$g(2)$	$g(3)$	$g(4)$

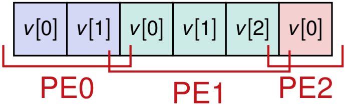
Generate$(n) : \langle \mathbb{N}_0 \rangle$

$n \in \mathbb{N}_0$ result size

0	1	2	3	4

ConcatToDIA$(v) : \langle A \rangle$

$v : vector(A)$ input data

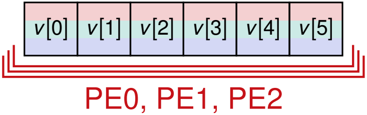


$v[0]$	$v[1]$	$v[0]$	$v[1]$	$v[2]$	$v[0]$

PE0 PE1 PE2

EqualToDIA$(v) : \langle A \rangle$

$v : vector(A)$ input data



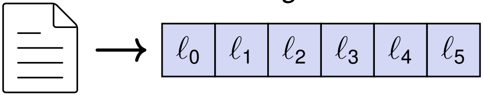
$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$

PE0, PE1, PE2

Source DOps: ReadLines, ReadBinary

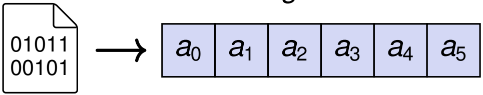
ReadLines(f) : $\langle$`std::string`$\rangle$

f : *string* list of files



ReadBinary$\langle A\rangle(f)$: $\langle A\rangle$

f : *string* list of files



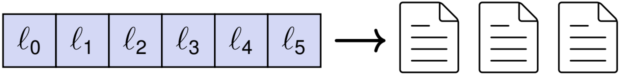
Items A are serialized in Thrill's binary representation.

Both either read from a common distributed file system (DFS), or concatenate from all PEs with the "local-storage" flag.

Actions: WriteLines, WriteBinary

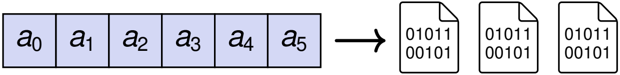
WriteLines(f) : $\langle$`std::string`$\rangle \to \mathit{void}$

f : *string* path/file pattern



WriteBinary(f) : $\langle A \rangle \to \mathit{void}$

f : *string* path/file pattern



Items A are serialized with Thrill's binary representation.

Each PE writes one or more files to the DFS or local disk.

Actions: Size, Print, and more

$\textbf{Size}() : \langle A \rangle \to \mathbb{N}_0$

$\boxed{}\boxed{}\boxed{}\boxed{} \longrightarrow 4$

$\textbf{Print}(t) : \langle A \rangle \to \textit{void}$

$t : \textit{string}$ variable name

$\textbf{Execute}() : \langle A \rangle \to \textit{void}$

$\textbf{AllGather}() : \langle A \rangle \to \textit{vector}(A)$

$\boxed{a_0}\boxed{a_1}\boxed{a_2}\boxed{a_3}$

$\longrightarrow [\, a_0, a_1, a_2, a_3 \,]$

$\textbf{Gather}(t) : \langle A \rangle \to \textit{vector}(A)$

$t \in \mathbb{N}_0$ target worker

$\textbf{Sum}(s) : \langle A \rangle \to A$

$s : A \times A \to A$ sum function

$\boxed{a_0}\boxed{a_1}\boxed{a_2}\boxed{a_3} \longrightarrow \sum_{i=0}^{3} a_i$

also: **Min**() and **Max**().

$\textbf{AllReduce}(s) : \langle A \rangle \to \textit{vector}(A)$

$s : A \times A \to A$ sum function

$\boxed{a_0}\boxed{a_1}\boxed{a_2}\boxed{a_3}$

$\longrightarrow s(s(s(a_0, a_1), a_2), a_3)$

3 The Thrill Framework

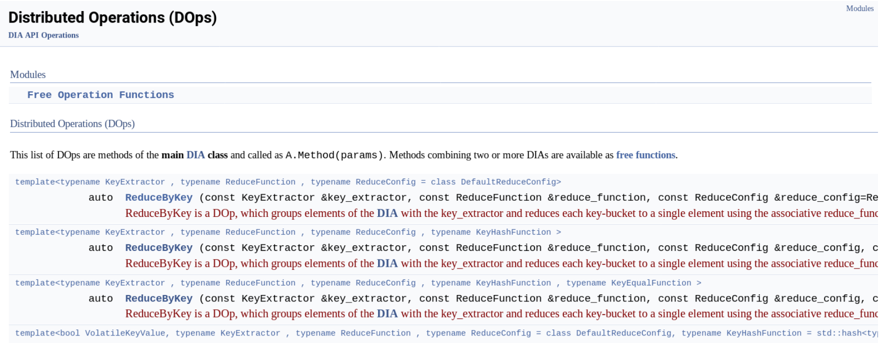
- Thrill’s DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- Conclusion

Playing with DIA Operations

How to get from the illustrated DIA operation to C++ code:

- Many operations have multiple variants and more parameters.
- The Doxygen documentation contains a very technical but complete list of DIA operations:

  https://project-thrill.org/docs/master/group___dia___api.html



Modules

Distributed Operations (DOps)

DIA API Operations

Modules

Free Operation Functions

Distributed Operations (DOps)

This list of DOps are methods of the **main DIA class** and called as `A.Method(params)`. Methods combining two or more DIAs are available as **free functions**.

```
template<typename KeyExtractor , typename ReduceFunction , typename ReduceConfig = class DefaultReduceConfig>
          auto  ReduceByKey (const KeyExtractor &key_extractor, const ReduceFunction &reduce_function, const ReduceConfig &reduce_config=Re
```
ReduceByKey is a DOp, which groups elements of the DIA with the key_extractor and reduces each key-bucket to a single element using the associative reduce_func

```
template<typename KeyExtractor , typename ReduceFunction , typename ReduceConfig , typename KeyHashFunction >
          auto  ReduceByKey (const KeyExtractor &key_extractor, const ReduceFunction &reduce_function, const ReduceConfig &reduce_config, c
```
ReduceByKey is a DOp, which groups elements of the DIA with the key_extractor and reduces each key-bucket to a single element using the associative reduce_func

```
template<typename KeyExtractor , typename ReduceFunction , typename ReduceConfig , typename KeyHashFunction , typename KeyEqualFunction >
          auto  ReduceByKey (const KeyExtractor &key_extractor, const ReduceFunction &reduce_function, const ReduceConfig &reduce_config, c
```
ReduceByKey is a DOp, which groups elements of the DIA with the key_extractor and reduces each key-bucket to a single element using the associative reduce_func

```
template<bool VolatileKeyValue, typename KeyExtractor , typename ReduceFunction , typename ReduceConfig = class DefaultReduceConfig, typename KeyHashFunction = std::hash<ty
```

Playing with DIA Operations

How to get your first DIA object:

- Use `thrill::Run()` to auto-detect the cluster setup and launch worker threads.
- Initial DIAs are created from Source operations. These require the `thrill::Context` as first parameter.

```
#include <thrill/thrill.hpp>

void program(thrill::Context& ctx) {
    auto lines = ReadLines(ctx, "/etc/hosts");
    lines.Print("lines");
}
int main(int argc, char* argv[]) {
    return thrill::Run(program);
}
```

Playing with DIA Operations

Applying operations to DIA objects:

- DIA objects have many operations like `.Sum()` as methods, but there are also free functions like `Zip()` and `ReadLines()`.
- Generally use auto instead of `DIA<T>`:

```cpp
void program(thrill::Context& ctx) {
    auto lines = ReadLines(ctx, "/etc/hosts");
    std::cout << "lines: " << lines.Size() << std::endl;
}
```

- Or use chaining of operations:

```cpp
void program(thrill::Context& ctx) {
    size_t num_lines = ReadLines(ctx, "/etc/hosts").Size();
    std::cout << "lines: " << num_lines << std::endl;
}
```

Playing with DIA Operations

More advanced uses of DIA objects:

- DIA<T> objects are only handles to actual graphs nodes in the DIA data-flow. This means they are copied as references.
- It is straight-forward to have functions with DIAs as parameters and return type. Again, prefer templates and the auto keyword.

```cpp
template <typename InputDIA>
auto LinesToLower(const InputDIA& input_dia) {
    return input_dia.Map(
        [](const std::string& line) {
            return tlx::to_lower(line);
        });
}
void program(thrill::Context& ctx) {
    auto lines = ReadLines(ctx, "/etc/hosts");
    std::cout << "lines: " << LinesToLower(lines).Size() << "\n";
}
```

Playing with DIA Operations

- Use C++11 lambdas for functor parameters.

```
using Pair = std::pair<std::string, size_t>;
void program(thrill::Context& ctx) {
    ReadLines(ctx, "/etc/hosts")
    .FlatMap<Pair>(
        // flatmap lambda: split and emit each word
        [](const std::string& line, auto emit) {
            tlx::split_view(' ', line, [&](tlx::string_view sv) {
                emit(Pair(sv.to_string(), 1)); });
    })
    .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);
    })
    .Execute();
}
```

Context Methods for Synchronization

The `Context` object also has many useful methods:

- `ctx.my_rank()` – rank of current worker thread.

  ```
        if (ctx.my_rank() == 0)
            std::cout << "lines: " << num_lines << '\n';
  ```

  also: `host_rank()`, `num_hosts()`, `num_workers()`.
- `y = ctx.net.Broadcast(x, 0);`
  MPI-style broadcast of x from worker 0 as y on all.
- `y = ctx.net.PrefixSum(x);`
  MPI-style prefix-sum of x with result y. also: `ExPrefixSum`.
- `y = ctx.net.AllReduce(x);`
  MPI-style all-reduce of x with result y. also: `Reduce`.
- `ctx.net.Barrier();`
  MPI-style synchronization barrier

Serializing Objects in DIAs

Thrill needs serialization methods for objects in DIAs.

- automatically supported are:
  - All plain old data types (PODs) (except pointers), which are plain integers, characters, doubles, and fixed-length structs containing such.
  - `std::string`, `std::pair`, `std::tuple`, `std::vector`, and `std::array`, if the contained type is serializable.
- otherwise, add a `serialize()` method:
  (see also cereal's docs: `https://uscilab.github.io/cereal/`)

```
#include <thrill/data/serialization_cereal.hpp>
struct Item {
    std::string string;
    size_t      value;
    template <typename Archive>
    void serialize(Archive& ar) {
        ar(string, value);
    }
};
```

Warning: Collective Execution!

Thrill programs are built from parallel, collectively synchronized operations.

- All distributed operations must be performed by all workers in the same order!
  Thrill's implicit synchronized collective execution depends on it!

OBLIGATORIO CASCO
OBLIGATORIO AURICULARES
OBLIGATORIO GAFAS
OBLIGATORIO BOTAS
OBLIGATORIO MASCARILLA
OBLIGATORIO ROPA IGNÍFUGA
PELIGRO CARGAS SUSPENDIDAS
PELIGRO PASO DE CARRETILLAS
PELIGRO CAÍDA DE OBJETOS
PELIGRO DE ATRAPAMIENTO EN MECANISMOS
PELIGRO ALTA TEMPERATURA
¡ATENCIÓN! RIESGO DE TROPEZAR
PROHIBIDO EL PASO A TODA PERSONA NO AUTORIZADA
PROHIBIDO PERMANECER BAJO CARGA
PROHIBIDO TRANSPORTAR PERSONAS

- The following **does not work** (why?):

```
auto lines = ReadLines(ctx, "/etc/hosts");
if (ctx.my_rank() == 0)
    std::cout << "lines: " << lines.Size() << std::endl;
```

Tutorial: Playing with DIAs

Hands-on Tutorial Part

Objective:
Write and run some simple programs using DIA operations.

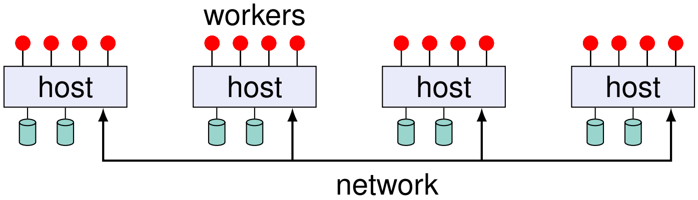
Some Ideas/Tasks:

- Read a text file, sort the lines, and write the result.
- Read a text file, transform all lines to lower case, and write them.
- Read a text file and calculate the average line length.
- Read a binary file as characters and count how many of each character occurs. Tip: use ReduceToIndex.
- Calculate the top 100 words in a text file and output all lines in which they occur.

3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- **Execution of Collective Operations in Thrill**
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- Conclusion

Execution on Cluster



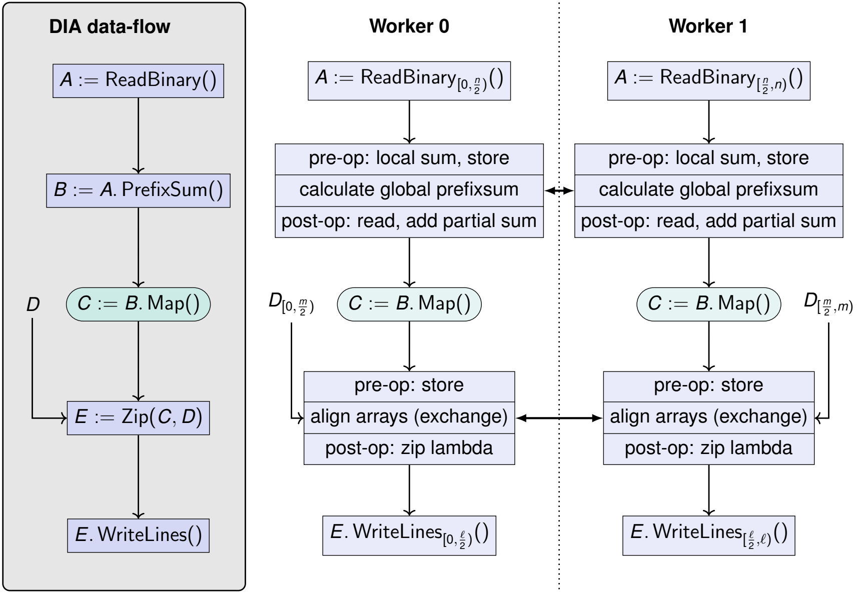
- Compile program into one binary, running on all hosts.
- Collective coordination of work on compute hosts, like MPI.
- Control flow is decided on by using C++ statements.

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

Example: WordCount in Thrill

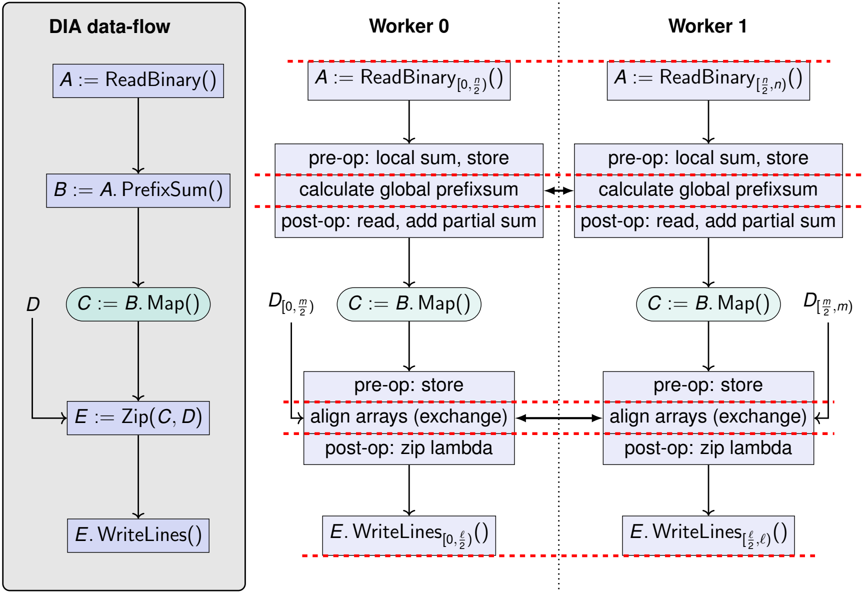
```cpp
using Pair = std::pair<std::string, size_t>;
void WordCount(Context& ctx, std::string input, std::string output) {
    auto word_pairs = ReadLines(ctx, input)     // DIA<std::string>
    .FlatMap<Pair>(
        // flatmap lambda: split and emit each word
        [](const std::string& line, auto emit) {
            tlx::split_view(' ', line, [&](tlx::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>
}
```

Mapping Data-Flow Nodes to Cluster



DIA data-flow
$A := \text{ReadBinary}()$
$B := A.\text{PrefixSum}()$
D
$C := B.\text{Map}()$
$E := \text{Zip}(C, D)$
$E.\text{WriteLines}()$
Worker 0
$A := \text{ReadBinary}_{[0,\frac{n}{2})}()$
pre-op: local sum, store
calculate global prefixsum
post-op: read, add partial sum
$D_{[0,\frac{m}{2})}$
$C := B.\text{Map}()$
pre-op: store
align arrays (exchange)
post-op: zip lambda
$E.\text{WriteLines}_{[0,\frac{\ell}{2})}()$
Worker 1
$A := \text{ReadBinary}_{[\frac{n}{2},n)}()$
pre-op: local sum, store
calculate global prefixsum
post-op: read, add partial sum
$C := B.\text{Map}()$
$D_{[\frac{m}{2},m)}$
pre-op: store
align arrays (exchange)
post-op: zip lambda
$E.\text{WriteLines}_{[\frac{\ell}{2},\ell)}()$

Mapping Data-Flow Nodes to Cluster



DIA data-flow

$A :=$ ReadBinary()

$B := A.$PrefixSum()

$C := B.$Map()

D

$E :=$ Zip(C, D)

$E.$WriteLines()

Worker 0

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

pre-op: local sum, store

calculate global prefixsum

post-op: read, add partial sum

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

$C := B.$Map()

pre-op: store

align arrays (exchange)

post-op: zip lambda

$E.$WriteLines$_{[0,\frac{\ell}{2})}$()

Worker 1

$A :=$ ReadBinary$_{[\frac{n}{2},n)}$()

pre-op: local sum, store

calculate global prefixsum

post-op: read, add partial sum

$C := B.$Map()

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

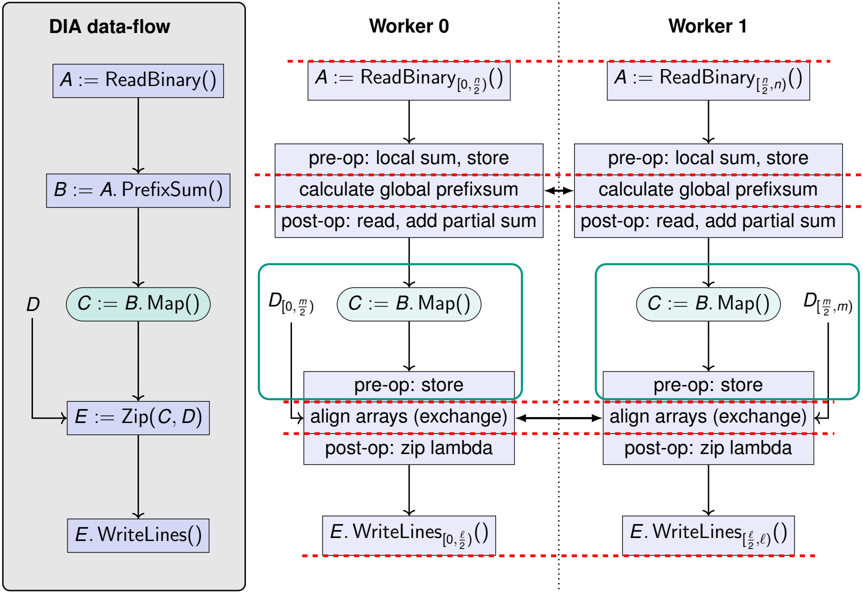
pre-op: store

align arrays (exchange)

post-op: zip lambda

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

Mapping Data-Flow Nodes to Cluster


DIA data-flow
$A := \mathrm{ReadBinary}()$
$B := A.\mathrm{PrefixSum}()$
D
$C := B.\mathrm{Map}()$
$E := \mathrm{Zip}(C, D)$
$E.\mathrm{WriteLines}()$
Worker 0
$A := \mathrm{ReadBinary}_{[0,\frac{n}{2})}()$
pre-op: local sum, store
calculate global prefixsum
post-op: read, add partial sum
$D_{[0,\frac{m}{2})}$
$C := B.\mathrm{Map}()$
pre-op: store
align arrays (exchange)
post-op: zip lambda
$E.\mathrm{WriteLines}_{[0,\frac{\ell}{2})}()$
Worker 1
$A := \mathrm{ReadBinary}_{[\frac{n}{2},n)}()$
pre-op: local sum, store
calculate global prefixsum
post-op: read, add partial sum
$C := B.\mathrm{Map}()$
$D_{[\frac{m}{2},m)}$
pre-op: store
align arrays (exchange)
post-op: zip lambda
$E.\mathrm{WriteLines}_{[\frac{\ell}{2},\ell)}()$

3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- **Tutorial: Running Thrill on a Cluster**
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- Conclusion

Tutorial: Running Thrill on a Cluster

Supported Network Systems and Launchers:

- single multi-core machine
- cluster with ssh access and TCP/IP network
- MPI as startup system and transport network

Goal is to launch a Thrill binary on all hosts and pass information on how to contact the others.

Thrill reads environment variables for configuration.
(Configuration files would have to be copied to all hosts.)

Tutorial: On One Multi-Core Machine

- This is the default startup mode for easy development.
  You have already used it:

```
Thrill: running locally with 2 test hosts and 4 workers per host
    in a local tcp network.
```

- Default local settings are to split the cores on the machine into two virtual hosts, which communicate via local TCP sockets.
- Options to change the default settings:
  - THRILL_LOCAL: number of virtual hosts
  - THRILL_WORKERS_PER_HOST: workers per host

Tutorial: Running via ssh

- Mode for plain Linux machines connected via TCP/IP.
- a) Install ssh keys on all machines for password-less login.
- b) use `thrill/run/ssh/invoke.sh` script with
  - `-h "host1 host2 host3"` (host list)
  - `-u u1234` (optional: remote user)
  - `thrill-binary` (binary and arguments)
- two setups:
  - with a common file system (NFS, ceph, Lustre, etc)
    ⇒ simply call the binary
  - without common file system (stand-alone machines).
    ⇒ add `-c` to copy the binary to all hosts.

Tutorial: Running via MPI

- For running on HPC clusters, Thrill can use MPI directly. MPI is auto-detected, no configuration is needed.

- Check that cmake finds the MPI libraries when compiling:

```
-- Found MPI_C: /usr/lib64/libmpi.so (found version ”3.1”)
-- Found MPI_CXX: /usr/lib64/libmpi_cxx.so (found version ”3.1”)
-- Found MPI: TRUE (found version ”3.1”)
```

- Run with mpirun:

```
mpirun -H ”host1,host2” thrill-binary
```

- On HPC clusters: use SLURM to launch with MPI, use only one task per host.

Tutorial: Environment Variables

- `THRILL_RAM` e.g. `=16GiB`
  override the maximum amount of RAM used by Thrill
- `THRILL_WORKERS_PER_HOST`
  override the number of workers per host
- `THRILL_LOG` e.g. `=out` (see next section)
  write log and profile to JSON file, e.g. “`out-host-123.json`”.

Environment variables can be set

- directly: “`THRILL_RAM=16GiB program`”
- with invoke.sh: “`THRILL_RAM=16GiB invoke.sh program`”
- or by mpirun: “`mpirun -x THRILL_RAM=16GiB program`”

3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- **Tutorial: Logging and Profiling**
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- Conclusion

Tutorial: Logging and Profiling

- Thrill contains a built-in logging and profiling mechanism.
- To activate: set the environment variable `THRILL_LOG=abc`.
- Thrill writes logs to `abc-host0.json` in a JSON format.
- Use the included tool `json2profile` to generate HTML graphs.

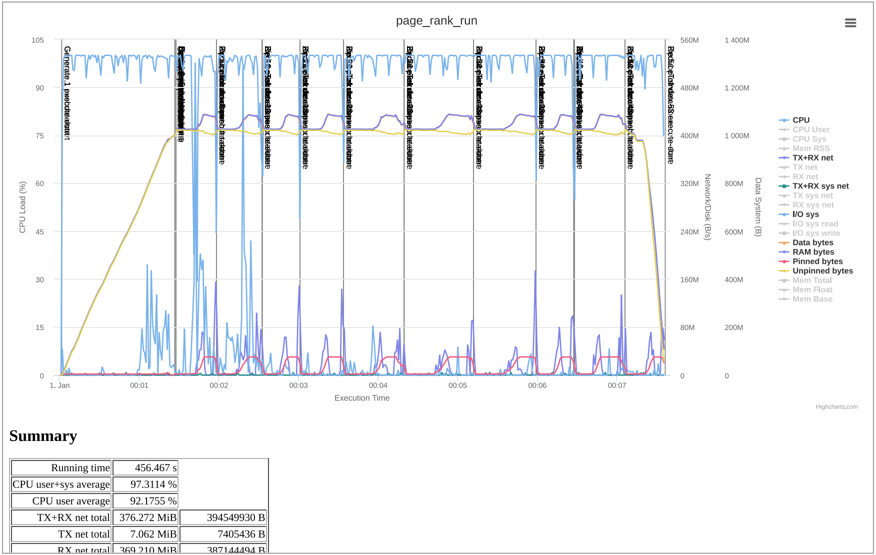
For example[1]:

```
$ cd ~/thrill/build/examples/page_rank/
$ THRILL_LOG=ourlog ./page_rank_run --generate 10000000
$ ls -la ourlog*
(this should show ourlog-host0.json and ourlog-host1.json)
$ ~/thrill/build/misc/json2profile ourlog*.json > profile.html
```

And then visit `profile.html` with a browser.

[1] (adapt paths if in tutorial-project)

Tutorial: Example Profile



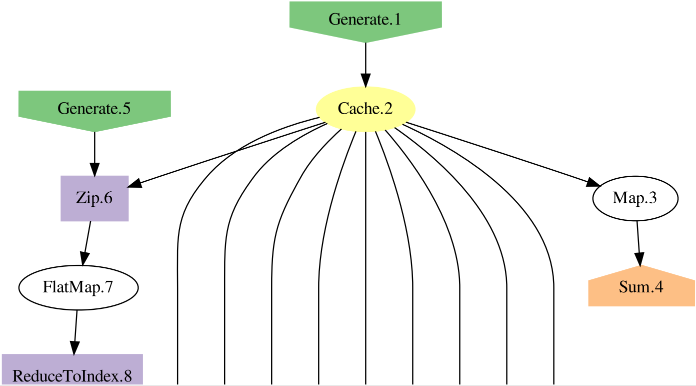
Summary

Running time	456.467 s	
CPU user+sys average	97.3114 %	
CPU user average	92.1755 %	
TX+RX net total	376.272 MiB	394549930 B
TX net total	7.062 MiB	7405436 B
RX net total	369.210 MiB	387144494 B

Tutorial: Output DIA Data-Flow Graph

- The DIA data-flow graph can also be extracted and automatically drawn with `dot` from the JSON log file:

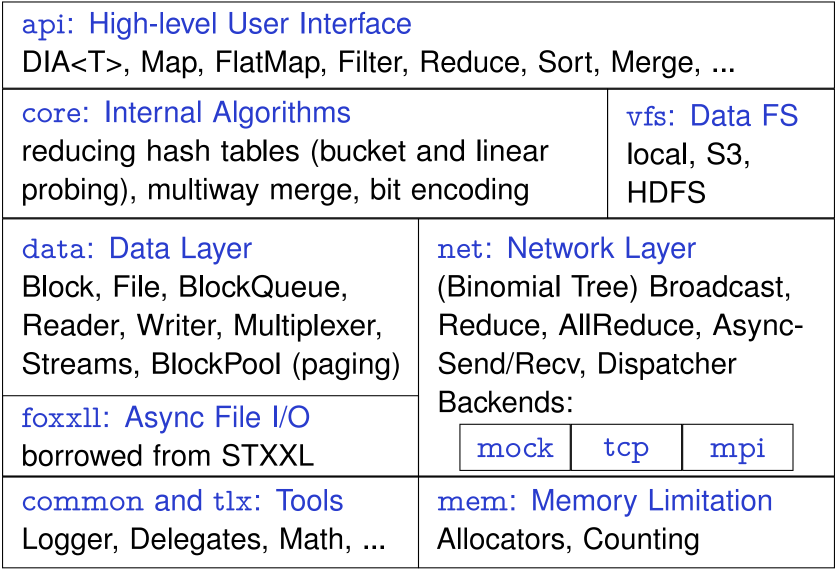
```
$ ~/thrill/misc/json2graphviz.py ourlog-host-0.json > page_rank.dot
$ dot -Tps -o page_rank.ps page_rank.dot
or
$ dot -Tsvg -o page_rank.svg page_rank.dot
```



3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- **Going Deeper into Thrill**
- Tutorial: First Steps towards k-Means
- Conclusion

Layers of Thrill



api: High-level User Interface

DIA<T>, Map, FlatMap, Filter, Reduce, Sort, Merge, ...

core: Internal Algorithms

reducing hash tables (bucket and linear probing), multiway merge, bit encoding

vfs: Data FS

local, S3, HDFS

data: Data Layer

Block, File, BlockQueue, Reader, Writer, Multiplexer, Streams, BlockPool (paging)

foxxll: Async File I/O

borrowed from STXXL

net: Network Layer

(Binomial Tree) Broadcast, Reduce, AllReduce, Async-Send/Recv, Dispatcher

Backends:

mock | tcp | mpi

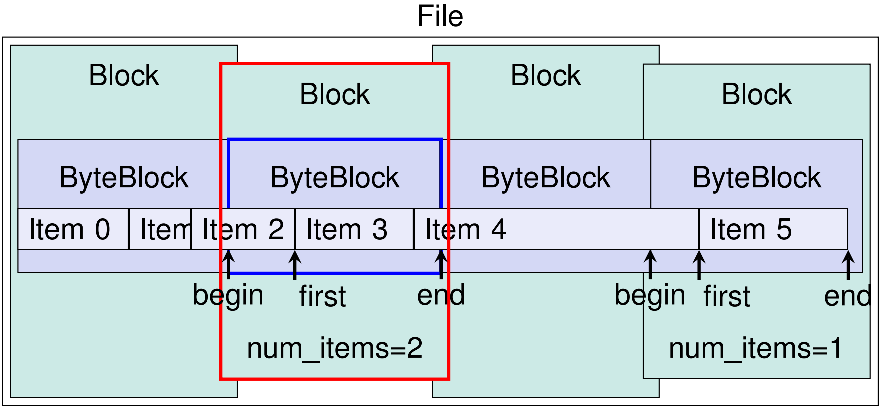
common and tlx: Tools

Logger, Delegates, Math, ...

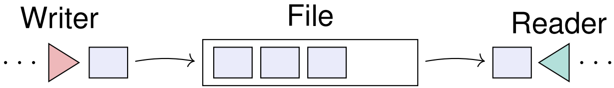
mem: Memory Limitation

Allocators, Counting

File – Variable-Length C++ Item Store



Readers and Writers

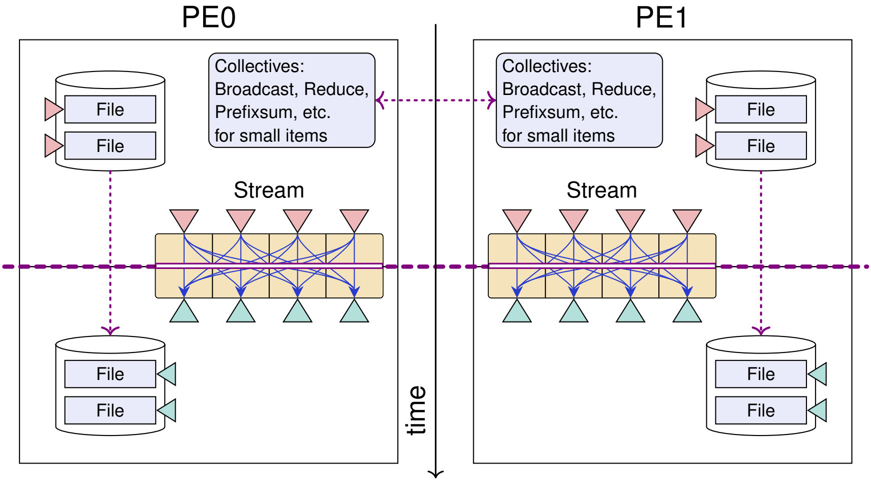


- Writers fill Blocks with items and push them into Sinks.
- Readers load Blocks from Sources and deserialize items.

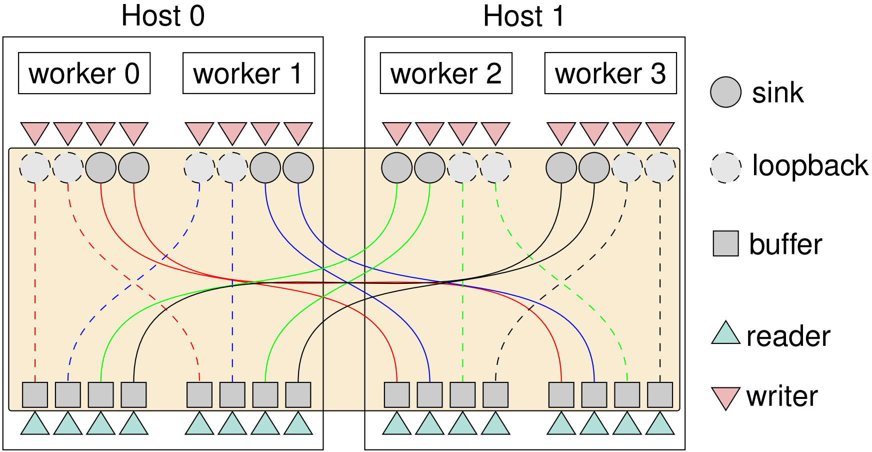
Example Code:

```
data::File file = ctx.GetFile();
auto writer = file.GetWriter();
writer.Put<Type>(type);
writer.Close();
auto reader = file.GetReader(/* consume */ false);
while (reader.HasNext())
    std::cout << reader.Next<Type>();
```

Thrill's Communication Abstraction

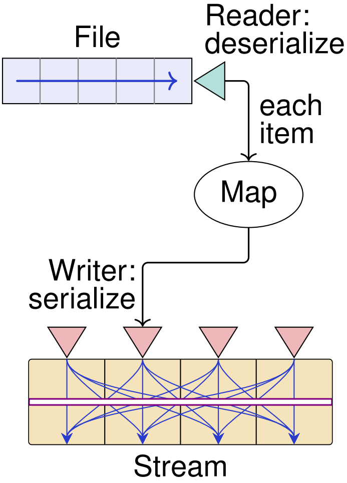


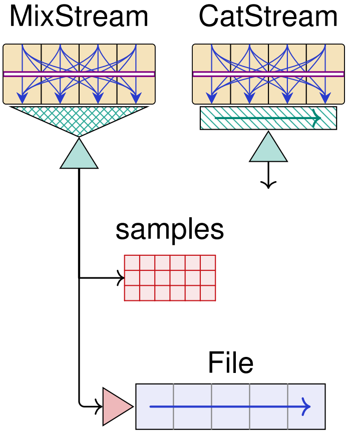
Stream – Async Big Data All-to-All



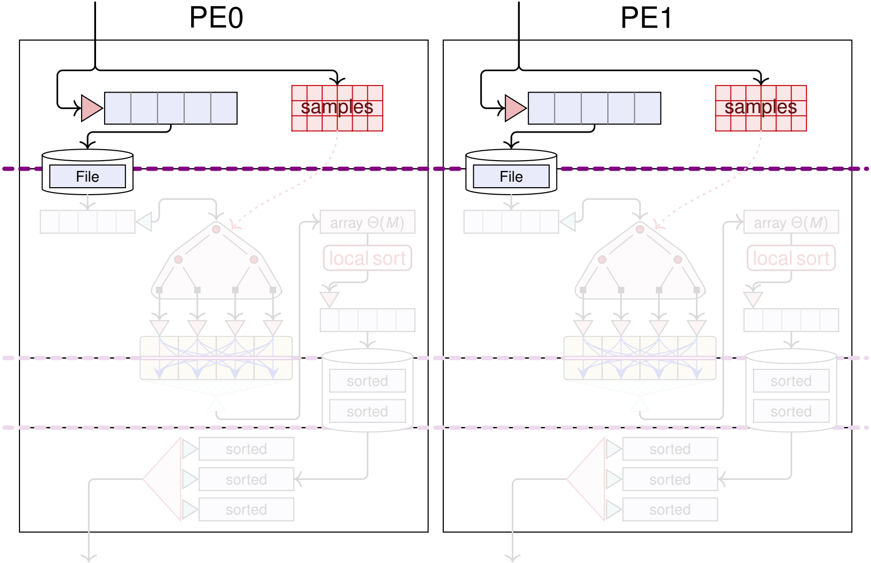
Streams are matched across hosts by ids in allocation order.

Thrill’s Data Processing Pipelines

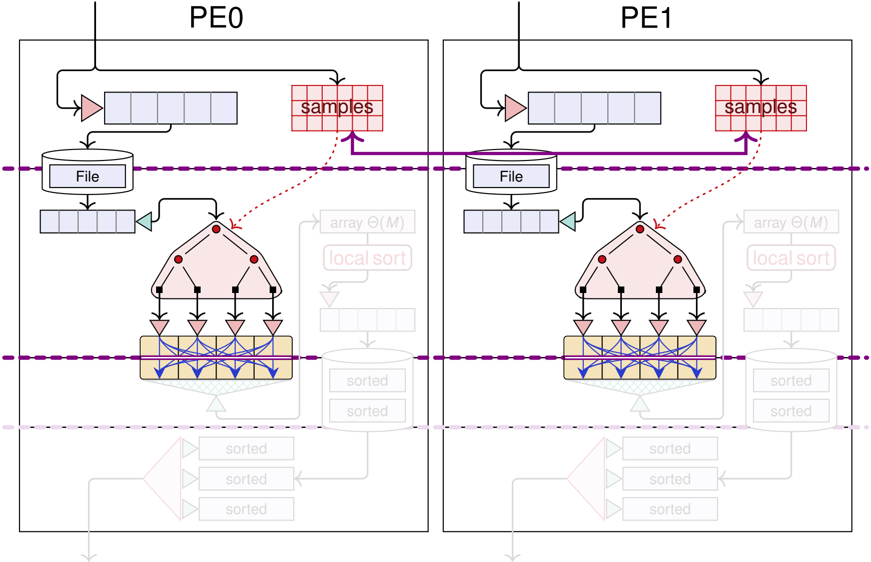




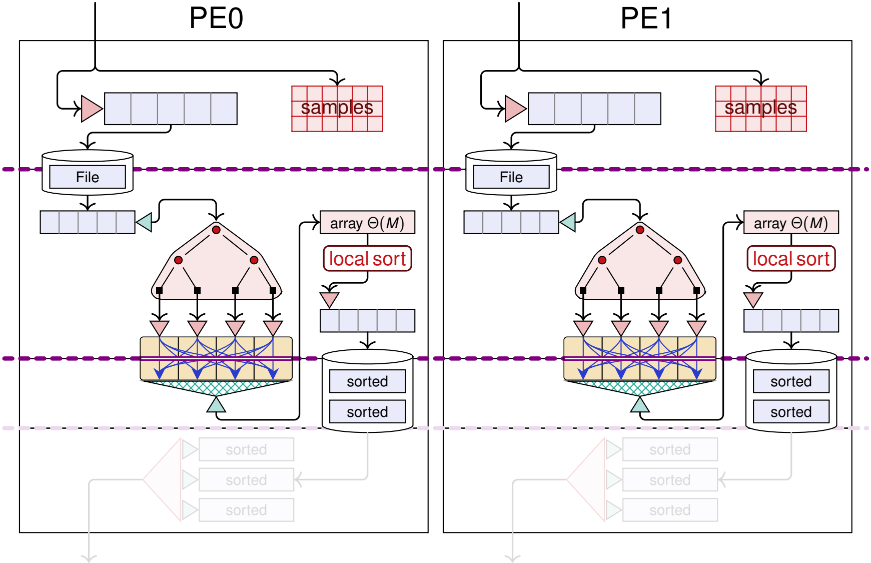
Thrill’s Current Sample Sort



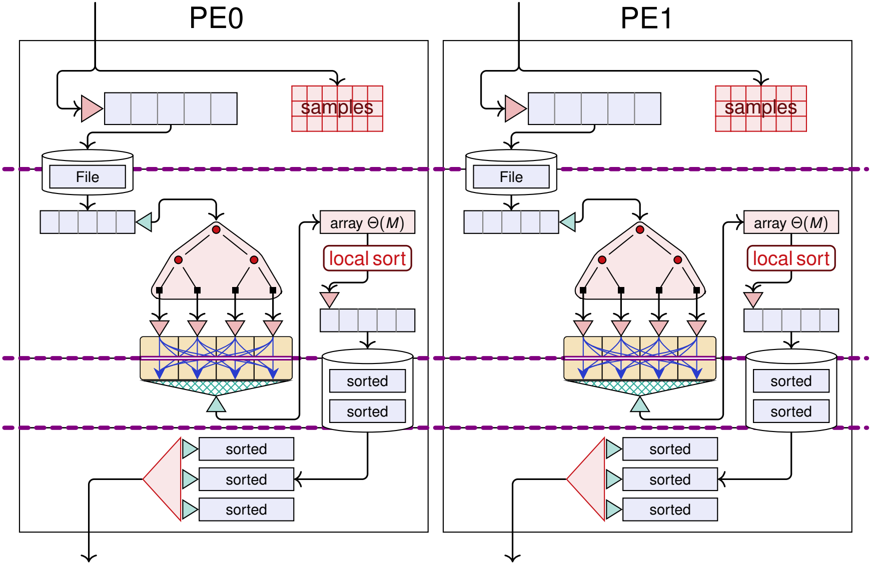
Thrill's Current Sample Sort



Thrill’s Current Sample Sort



Thrill’s Current Sample Sort

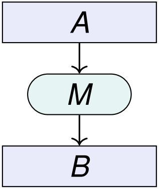


Optimization: Consume and Keep

Goal: Optimize peak DIA memory usage.

When can the data of a `DIA` be safely freed?

1. When all handles `DIA<T>` go out of scope.
2. By manually calling `.Dispose()` on a handle.
3. While processing operations using consumption and `.Keep()`:



- For example: the contents of `DIA` A is consumed during execution of the $A \rightarrow M \rightarrow B$ data processing path.
- Advantage: reduces the maximum required `DIA` data memory to about $N + \mathcal{O}(B)$ items (depending on the operation).

Optimization: Consume and Keep

Step 1: Enable DIA consumption.

- Default setting: never automatically consume DIA contents. (makes it easier for new users and when writing algorithms)
- To enable consumption: `ctx.enable_consume();`
  ⇒ all `DIA`s assume to be read at most once,
  hence are consumed by the first executed operation.

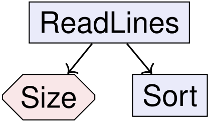
Step 2: Add `.Keep()` where needed.

- Each DIA object has a consumption counter, initially 1. To increment the counter call `.Keep()` before an operation.
- Also available: `.KeepForever()`.
- To find where `.Keep()` is needed, you can also simply run the Thrill program. It will print error messages when DIA operations are executed but the required data is already consumed.

Example: Consume and Keep

Example Code:

```
auto lines = ReadLines(ctx, "/etc/hosts");
size_t line_count = lines.Size();
auto sorted_lines = lines.Sort();
```

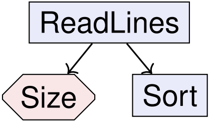


- DIA `lines` is used twice: first by `Size()`, and then by `Sort()`.
- When executing the action `Size()`, the DIA node for `Sort()` has not been added yet.
- ⇒ add `.Keep()` before the `Size()`.

Example: Consume and Keep

Example Code:

```
auto lines = ReadLines(ctx, "/etc/hosts");
size_t line_count = lines.Keep().Size();
auto sorted_lines = lines.Sort();
```

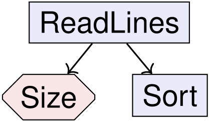


- DIA `lines` is used twice: first by `Size()`, and then by `Sort()`.
- When executing the action `Size()`, the DIA node for `Sort()` has not been added yet.
- ⇒ add `.Keep()` before the `Size()`.

Example: Consume and Keep

Example Code:

```
auto lines = ReadLines(ctx, "/etc/hosts");
size_t line_count = lines.Keep().Size();
auto sorted_lines = lines.Sort();
```

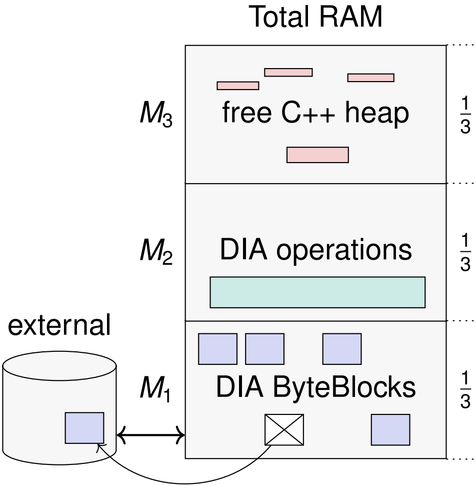


- DIA `lines` is used twice: first by `Size()`, and then by `Sort()`.
- When executing the action `Size()`, the DIA node for `Sort()` has not been added yet.
- ⇒ add `.Keep()` before the `Size()`.

Further Improvement: Action Futures

```
auto lines = ReadLines(ctx, "/etc/hosts");
auto size_future = lines.SizeFuture();
auto sorted_lines = lines.Sort();
size_t line_count = size_future.get();
```

Memory Allocation Areas in Thrill

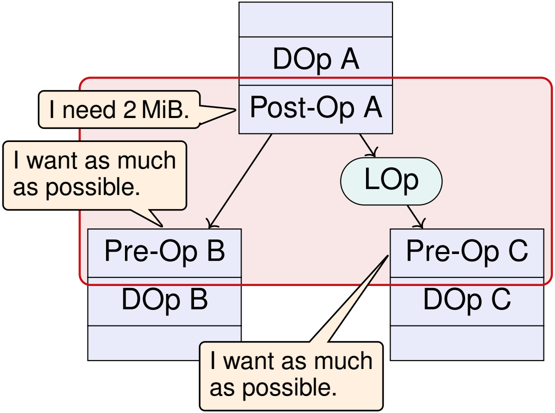


- Default heap allocations for user and Thrill internals.
- e.g.: `std::string` buffers

- Sort buffers or hash tables of DIA operations.

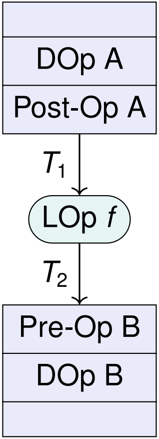
- DIA data serialized into ByteBuffers
- Pin ByteBlocks in memory.

Memory Distribution in Stages



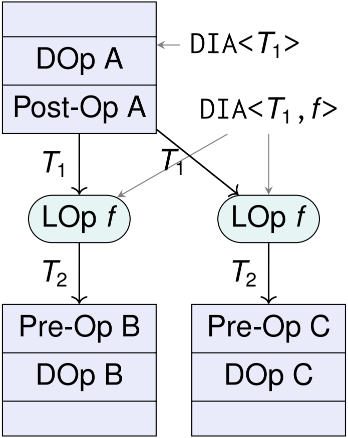
- The Stage Executor queries all DOp's participating parts for their memory requirements.
- It then distributes M_2 memory fairly, e.g. Post-Op A gets 2 MiB, and Post-Op B and C each get $(M_2 - 2\,\text{MiB})/2$.

Pipelined Data Flow Processing



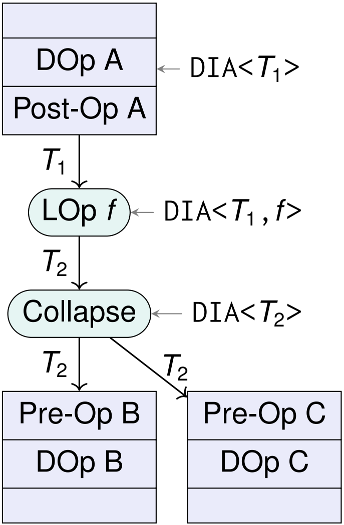
- Stages are processed by “pipelining” or “chaining” steps.
- Post-Op A generates items of type T_1 by reading from a File or Stream, on-the-fly, etc.
- DOps states: New, Executed, Disposed.
- When “executed” a DOp can emit a steam of items to new children nodes: “PushData()”. $\Rightarrow$ can dynamically attach function chains to DOps at run time.
- A stage includes all processing possible by streaming data out of a DOp’s Post-Op.

Pipelined Data Flow Construction



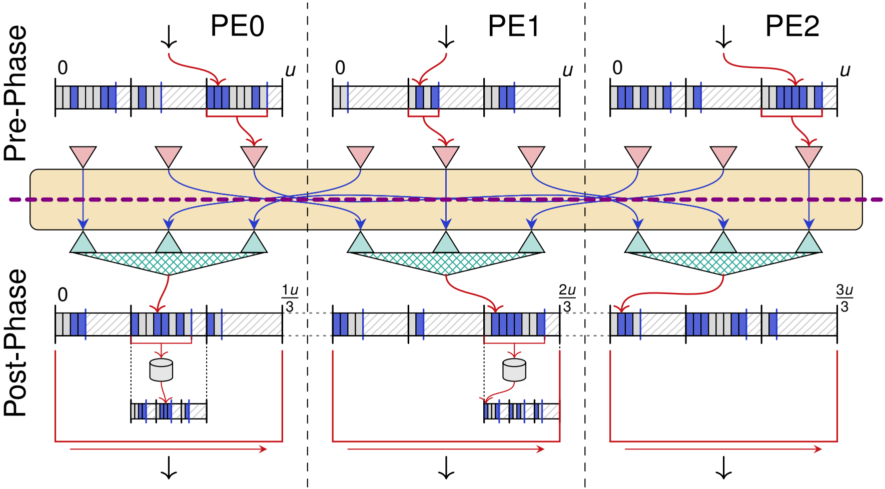
- The handle of a DOp which returns T_1 is of type DIA<T_1>.
- LOps are stored using template parameters: adding f returns a handle DIA<T_1, f>.
- The chain is closed by adding the Pre-Op of a DOp or Action.
- This function chain is folded and added to a DOp as a child.

Data Flow Construction: Collapse



- The function chain can be folded explicitly, by adding a Collapse() node.
- This is rarely required, e.g. to avoid running f twice, to return a DIA<T_2> from a function, or in iterative loops.
- Collapse is a special auxiliary node type. Others are Cache, and Union.

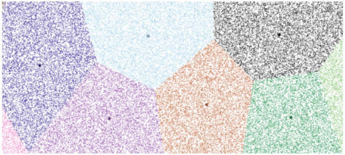
ReduceByKey Implementation



3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- **Tutorial: First Steps towards k-Means**
- Conclusion

Tutorial: First Steps towards *k*-Means

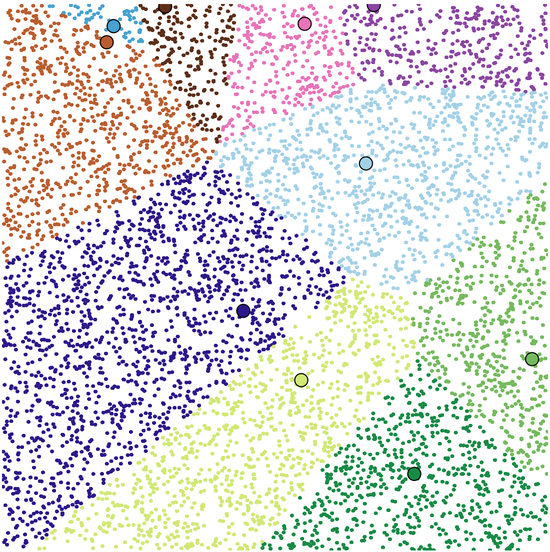


Goal of this tutorial part is to implement the *k*-means clustering algorithm.

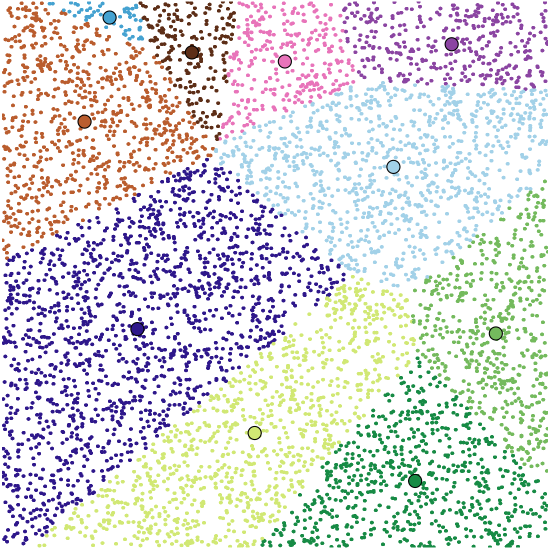
The algorithm works as follows:

1. Given are a set of *d*-dimensional points and a target number of clusters *k*.
2. Select *k* initial cluster center points at random.
3. 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.
4. This will be repeated a constant number of iterations.

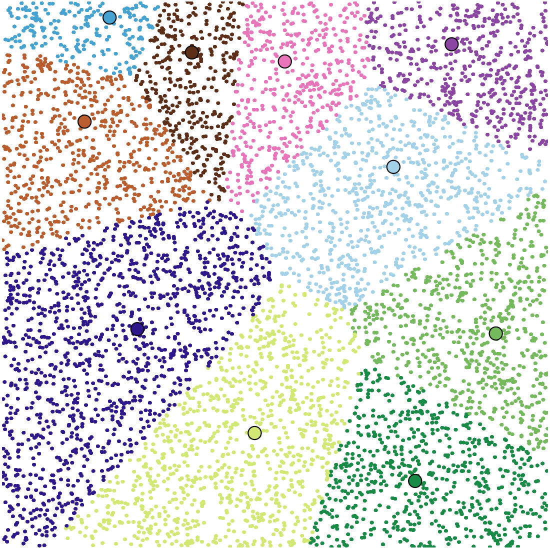
Tutorial: *k*-Means Iterations (pre 1)



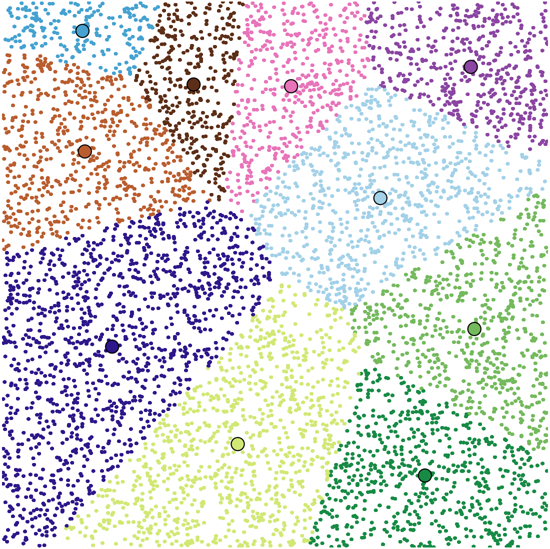
Tutorial: *k*-Means Iterations (post 1)



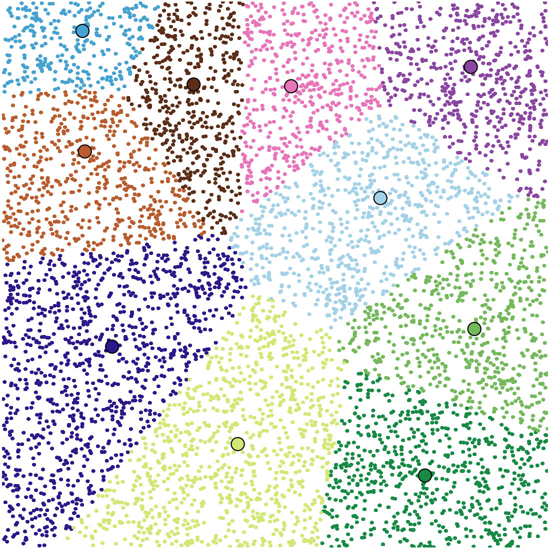
Tutorial: *k*-Means Iterations (pre 2)



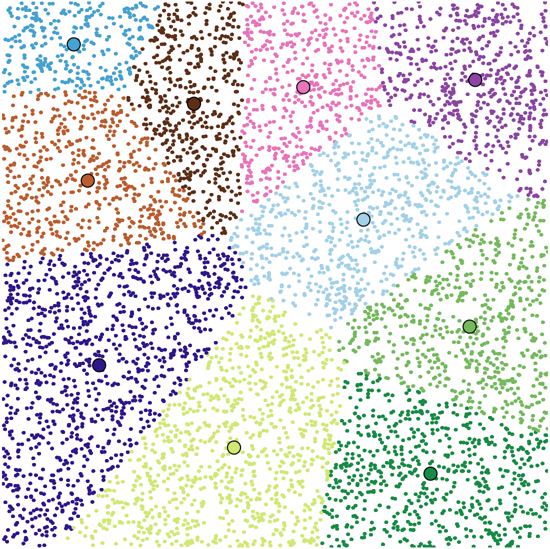
Tutorial: *k*-Means Iterations (post 2)



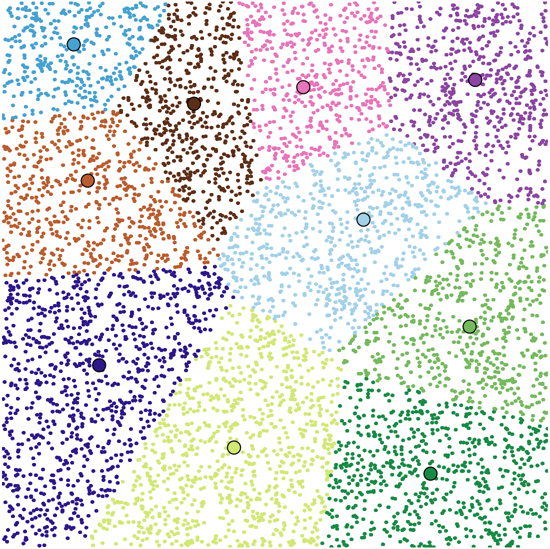
Tutorial: *k*-Means Iterations (pre 3)



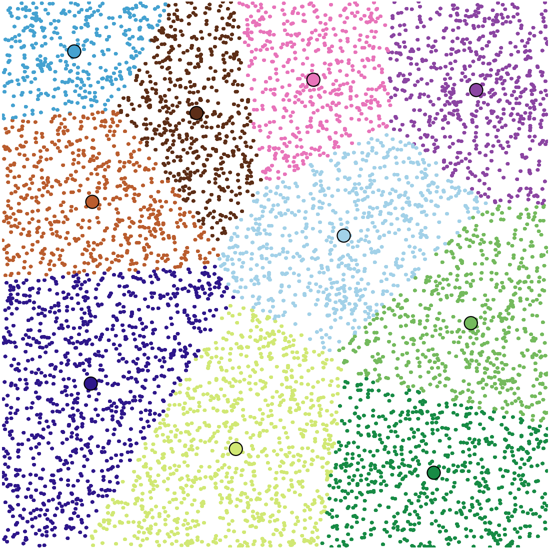
Tutorial: *k*-Means Iterations (post 3)



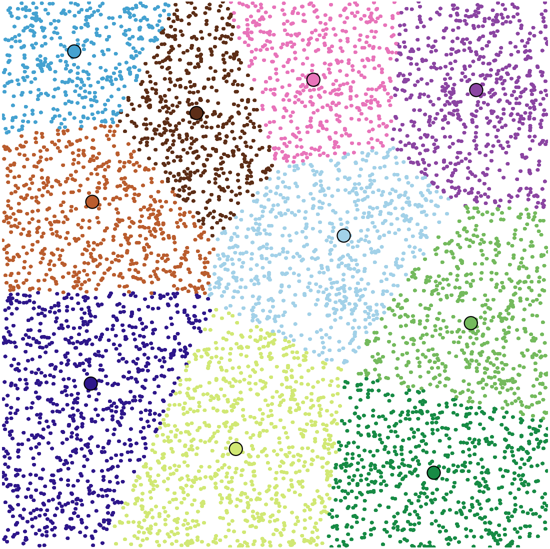
Tutorial: *k*-Means Iterations (pre 4)



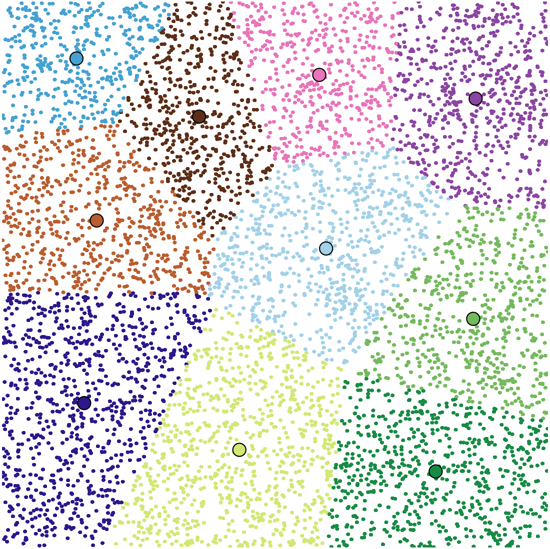
Tutorial: *k*-Means Iterations (post 4)



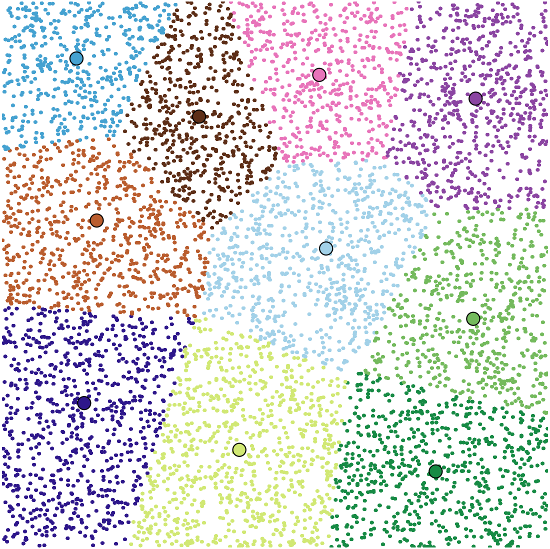
Tutorial: *k*-Means Iterations (pre 5)



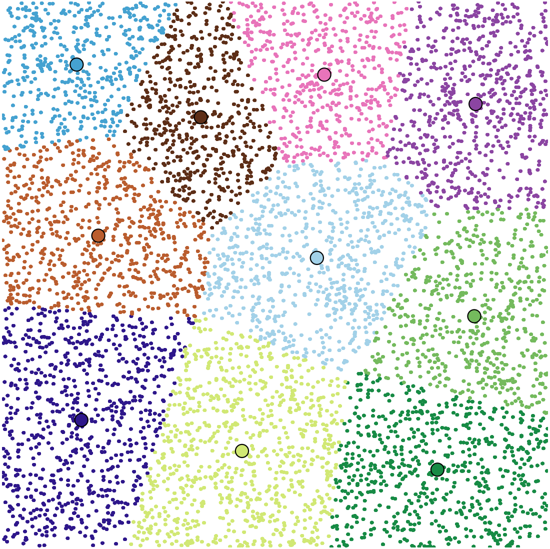
Tutorial: *k*-Means Iterations (post 5)



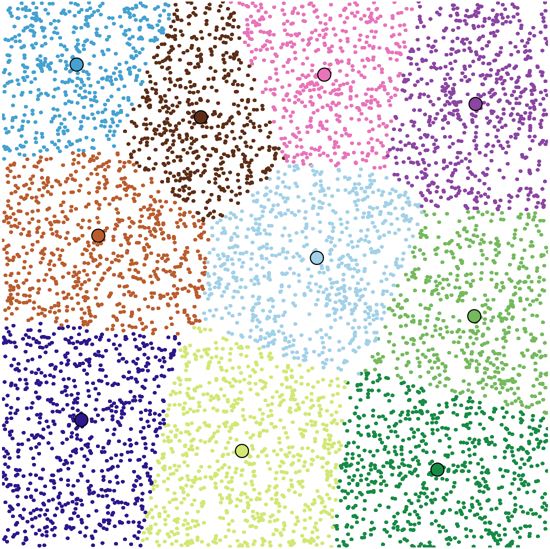
Tutorial: *k*-Means Iterations (pre 6)



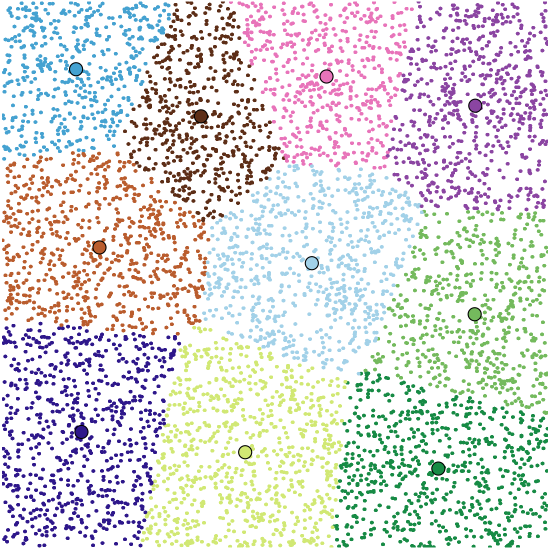
Tutorial: *k*-Means Iterations (post 6)



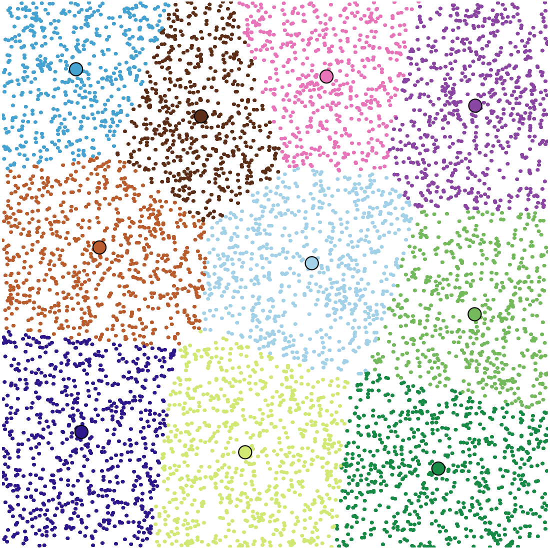
Tutorial: *k*-Means Iterations (pre 7)



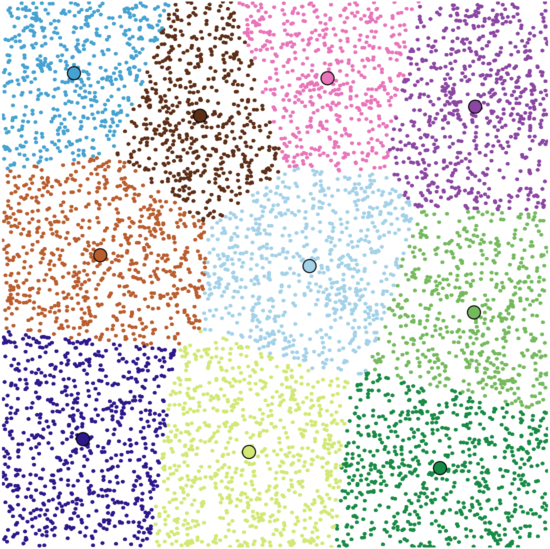
Tutorial: *k*-Means Iterations (post 7)



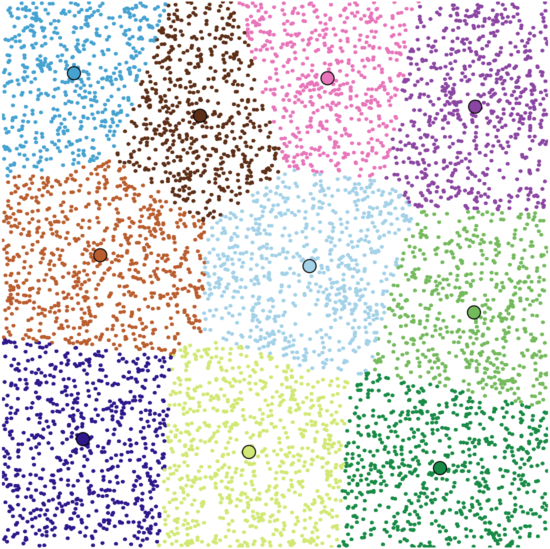
Tutorial: *k*-Means Iterations (pre 8)



Tutorial: *k*-Means Iterations (post 8)



Tutorial: *k*-Means Iterations (stop)



k-Means: Printable 2D-Points

Step 1: Make a 2D struct “Point” and generate random points.

- Use the following `Point` struct with `ostream` operator:

```cpp
//! A 2-dimensional point with double precision
struct Point {
    //! point coordinates
    double x, y;
};
//! make ostream-able for Print()
std::ostream& operator << (std::ostream& os, const Point& p) {
    return os << '(' << p.x << ',' << p.y << ')';
}
```

- Use `Generate` to make random points, `Print`, and `Cache` them.
- Use the script `points2svg.py` to display the “`(x,y)`” lines.

k-Means: Map to Random Centers

Step 2: Map points to randomly selected centers.

- Use `Sample` to select random initial centers, `Print` them.
- `Map` each Point to its closest center.
- Maybe add a `distance` method to your Point and refactor.
- What should the `Map` output for the next step?
  What is the next step?

k-Means: Calculate Better Centers

Step 3: Calculate better centers by reducing all points.

- Next step is to use `ReduceByKey` or `ReduceToIndex` to calculate the mean of all points associated with a center.
- Key idea: make a second struct `PointTarget` containing Point and new target center id.
- Reduce all structs with same target center id and calculate the vector sum and the number of points associated.
- To do this, create a third struct `PointSumCount` containing Point, vector sum, and a counter.
- Maybe add add and scalar multiplication operators to Point.

k-Means: Iterate!

Step 4: Iterate the process 10 times.

- Collect the new centers on all hosts with `AllGather`.
- Add a `for` loop for iteration.

Bonus Step 5: Add input and output to/from text files.

Bonus Step 6: Instead of 10 iterations, calculate the distance that centers moved and break if below a threshold.

Bonus Step 7: Calculate the “error” of the centers, which is the total distance of all points to their cluster center.

Bonus Step 7: Run your program on the cluster with a large dataset.

3 The Thrill Framework

- Thrill's DIA Abstraction and List of Operations
- Tutorial: Playing with DIA Operations
- Execution of Collective Operations in Thrill
- Tutorial: Running Thrill on a Cluster
- Tutorial: Logging and Profiling
- Going Deeper into Thrill
- Tutorial: First Steps towards k-Means
- **Conclusion**

Thoughts on the Architecture

Thrill’s Sweet Spot

- C++ toolkit for implementing distributed algorithms quickly.
- Platform to engineer and evaluate distributed primitives.
- Efficient processing of small items and pipelining of primitives.
- Platform for implementing on-the-fly compiled queries?

Open Questions

- Compile-time optimization only – no run-time algorithm selection or (statistical) knowledge about the data.
- Assumes *h* identical hosts constantly running, (the old MPI/HPC way, Hadoop/Spark do block-level scheduling).
- Memory management
- Malleability, predictability, and scalability to 1 million cores

Future Work and Ideas

Ideas for Future Work:

- Beyond `DIA<T>`? `Graph<V,E>`? `DenseMatrix<T>`?
- Distributed rank()/select() and other stringology algorithms.
- Malleability and fault tolerance.
- Predictability of algorithm execution on platforms.
- Communication efficient distributed operations for Thrill.
- Distributed functional programming language on top of Thrill.

Thank you for your attention!

More Information at https://project-thrill.org
and https://panthema.net/thrill