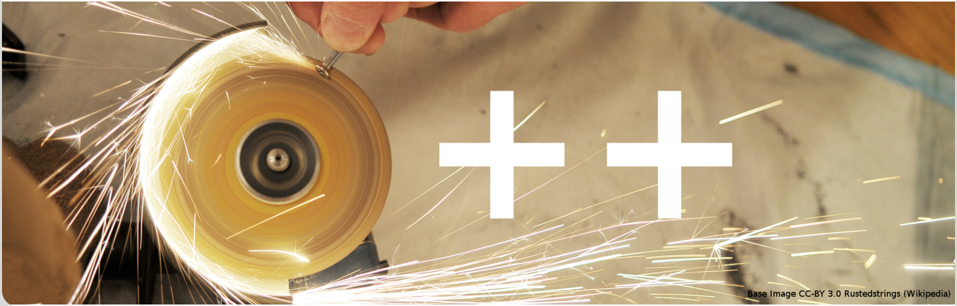
Project DALKIT (informal working title)

Michael Axtmann, Timo Bingmann, Peter Sanders, Sebastian Schlag, and Students | 2015-03-27

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS



Projektpraktikum: Verteilte Datenverarbeitung mit MapReduce

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

Institute of Theoretical Informatics – Algorithmics

Flavours of Big Data Frameworks

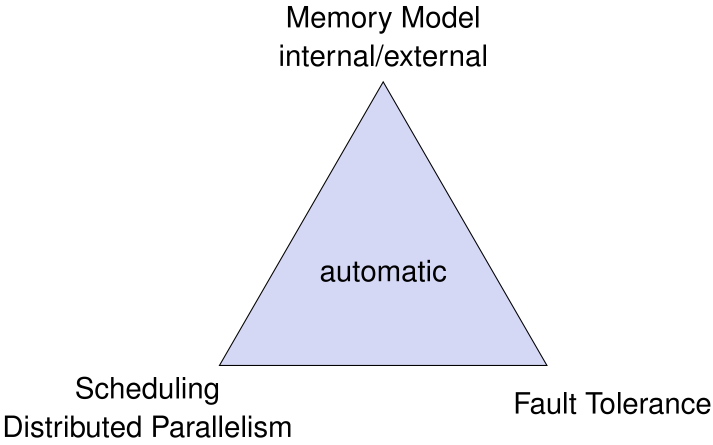
- Batch Processing
  Google's MapReduce, Hadoop MapReduce, Apache Spark, Apache Flink (Stratosphere), Google's FlumeJava.
- Real-time Stream Processing
  Apache Storm, Apache Spark Streaming, Google's MillWheel.
- Distributed Processing
  Microsoft's Dryad, Microsoft's Naiad.
- Interactive Cached Queries
  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.

Why another Big Data Framework?

	Hadoop World Record	Spark 100 TB	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	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/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Daytona Rules	Yes	Yes	No
Environment	dedicated	EC2 (i2.8xlarge)	

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

Dimensions of Batch Processing



Dimensions of Batch Processing

- User Programming Language
  Java, C++, Scala, Python, ...
- Cluster/System Model
  distributed main memory vs. distributed external memory.
- Interface Expressiveness
  simple (map / shuffle / reduce) vs. richer primitives
- Execution Model
  single step vs. complex DAGs
  iterative algorithms with host language control flow
- Item and Dataset Model
  opaque items vs. tuples/components.
  sets of items vs. arrays of items.
- Redundancy Model

Dimensions of Batch Processing

- User Programming Language
  Java, C++, Scala, Python, ...
- Cluster/System Model
  distributed main memory vs. distributed external memory.
- Interface Expressiveness
  simple (map / shuffle / reduce) vs. richer primitives
- Execution Model
  single step vs. complex DAGs
  iterative algorithms with host language control flow
- Item and Dataset Model
  opaque items vs. tuples/components.
  sets of items vs. arrays of items.
- Redundancy Model

Spark's Word Count Example: Scala

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Spark's Word Count Example: Java

```
JavaRDD<String> file = spark.textFile("hdfs://...");
JavaRDD<String> words = file.flatMap(
    new FlatMapFunction<String, String>() {
        public Iterable<String> call(String s) {
            return Arrays.asList(s.split(" ")); }
    });
JavaPairRDD<String, Integer> pairs = words.mapToPair(
    new PairFunction<String, String, Integer>() {
        public Tuple2<String, Integer> call(String s) {
            return new Tuple2<String, Integer>(s, 1); }
    });
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(
    new Function2<Integer, Integer>() {
        public Integer call(Integer a, Integer b) { return a + b; }
     });
counts.saveAsTextFile("hdfs://...");
```

Our Experimental C++ Interface

```
DIA<std::string> paragraphs = Job().ReadFromFS("file.txt",
    [](const std::string& line) { return line; });

DIA<std::string> words = paragraphs.FlatMap(
    [](const std::string& input, Emitter<std::string>& emit)
    {
       /* map lambda */
       std::istringstream iss(input); std::ostringstream oss(input);
       while (iss) {
           std::string word; iss >> word; emit(word);
       }
    });

using WordPair = std::pair<std::string, int>;
DIA<WordPair> word_counts = words.Reduce(     /* (non-associative) */
    [](const std::string& input) { return input; }, /* key extract */
    [](const std::vector<std::string>& input) {     /* reduce */
        return WordPair(input[0], input.size());
    });
```

Our Experimental C++ Interface

```
auto paragraphs = Job().ReadFromFS("file.txt",
    [](const std::string& line) { return line; });

auto words = paragraphs.FlatMap(
    [](const std::string& input, Emitter<std::string>& emit)
    {
       /* map lambda */
       std::istringstream iss(input); std::ostringstream oss(input);
       while (iss) {
           std::string word; iss >> word; emit(word);
       }
    });

using WordPair = std::pair<std::string, int>;
auto word_counts = words.Reduce(              /* (non-associative) */
    [](const std::string& input) { return input; }, /* key extract */
    [](const std::vector<std::string>& input) {     /* reduce */
        return WordPair(input[0], input.size());
    });
```

Our Experimental C++ Interface

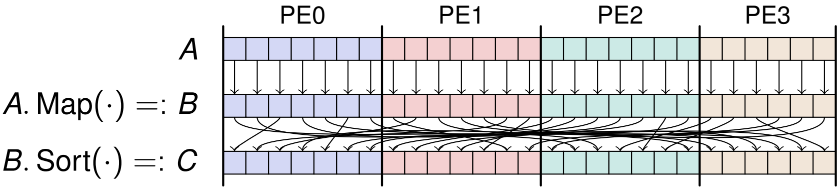
```
auto paragraphs = Job().ReadFromFS("file.txt",
    [](const std::string& line) { return line; })

    .FlatMap(
    [](const std::string& input, Emitter<std::string>& emit)
    {
       /* map lambda */
       std::istringstream iss(input); std::ostringstream oss(input);
       while (iss) {
           std::string word; iss >> word; emit(word);
       }
    })


    .Reduce(                                  /* (non-associative) */
    [](const std::string& input) { return input; }, /* key extract */
    [](const std::vector<std::string>& input) {     /* reduce */
        return std::make_pair(input[0], input.size());
    });
```

Our Experimental C++ Interface

```
auto paragraphs = Job().ReadFromFS("file.txt")
    .FlatMap(
    [](const std::string& input, Emitter<std::string>& emit)
    {
       /* map lambda */
       std::istringstream iss(input); std::ostringstream oss(input);
       while (iss) {
           std::string word; iss >> word; emit(word);
       }
    })
    .Reduce(                                        /* (non-associative) */
    [](const std::vector<std::string>& input) {      /* reduce */
        return std::make_pair(input[0], input.size());
    });
```

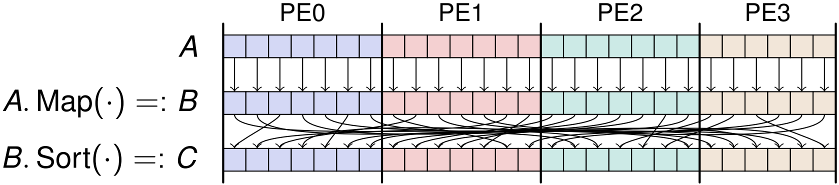
Distributed Immutable Array (DIA)

- 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 actions.



Distributed Immutable Array (DIA)

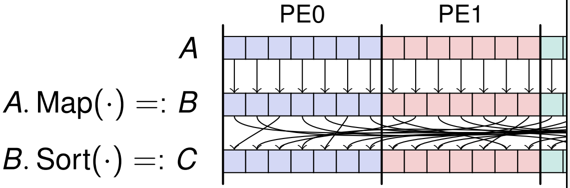
- User Programmer's View:
  - DIA<T> = result of an operation (local or distributed).
  - Model: distributed array of items T on the cluster
  - Cannot access items directly, instead use actions.

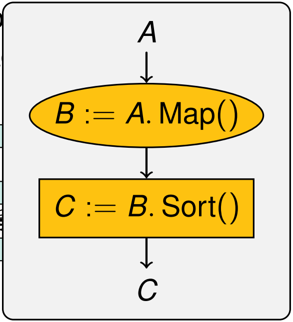


- Framework Designer's View:
  - Goals: distribute work, optimize execution on cluster, add redundancy where applicable. $\Longrightarrow$ build data-flow graph.
  - DIA<T> = chain of computation items
  - Let distributed operations choose "materialization".

Distributed Immutable Array (DIA)

- User Programmer's View:
  - DIA<T> = result of an operation (local or distributed).
  - Model: distributed array of items T o
  - Cannot access items directly, instea





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

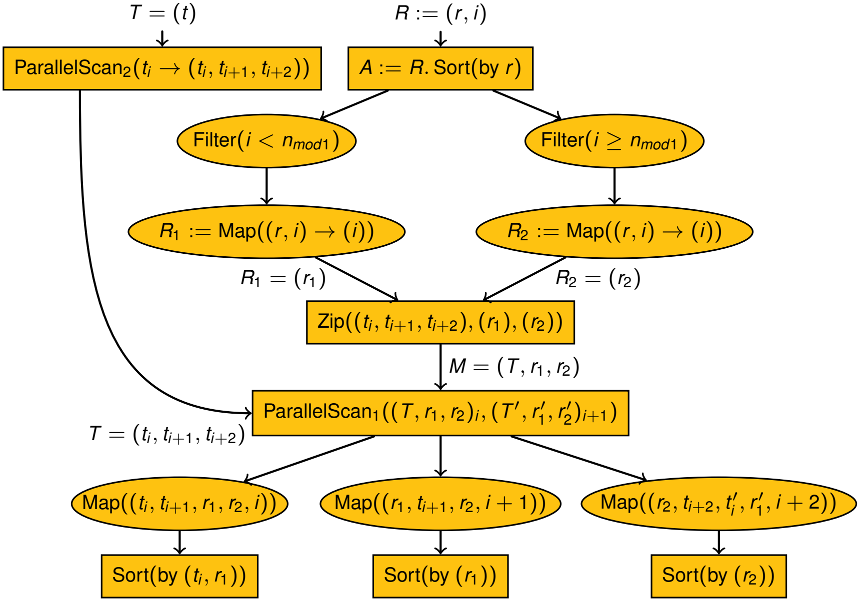
List of Primitives

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

Sort()	Sort a DIA using comparisons.
ShuffleReduce()	Shuffle with Key Extractor, Hasher, and associative Reducer.
PrefixSum()	Compute (generalized) prefix sum on DIA.
ParallelScan(k)	Scan all DIA items with backlog k.
Concat()	Concatenate two or more DIAs of equal type.
Zip()	Combine equal sized DIAs item-wise.
Merge()	Merge equal typed DIAs using comparisons.

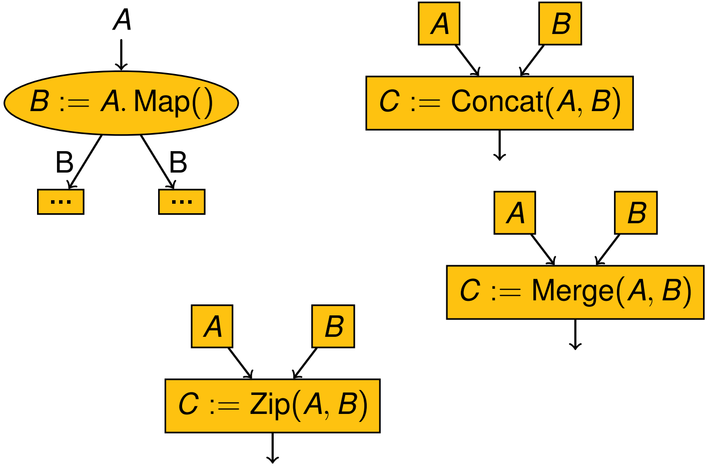
- Actions: input is a DIA, output: $\geq$ 0 items on master.
  At(), Min(), Max(), Sum(), Sample(), pretty much still open.

Example Data-Flow Graph

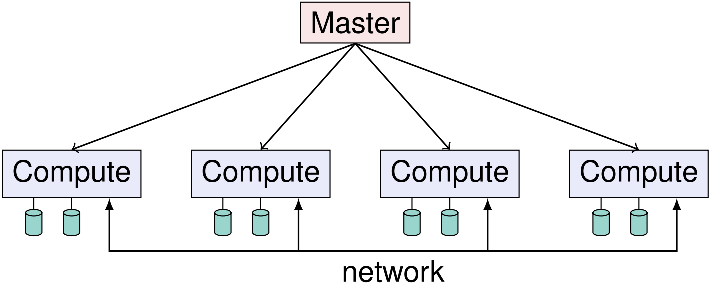


Structure of Data-Flow Graph

Any node can fork DIAs. Combines are DOps.

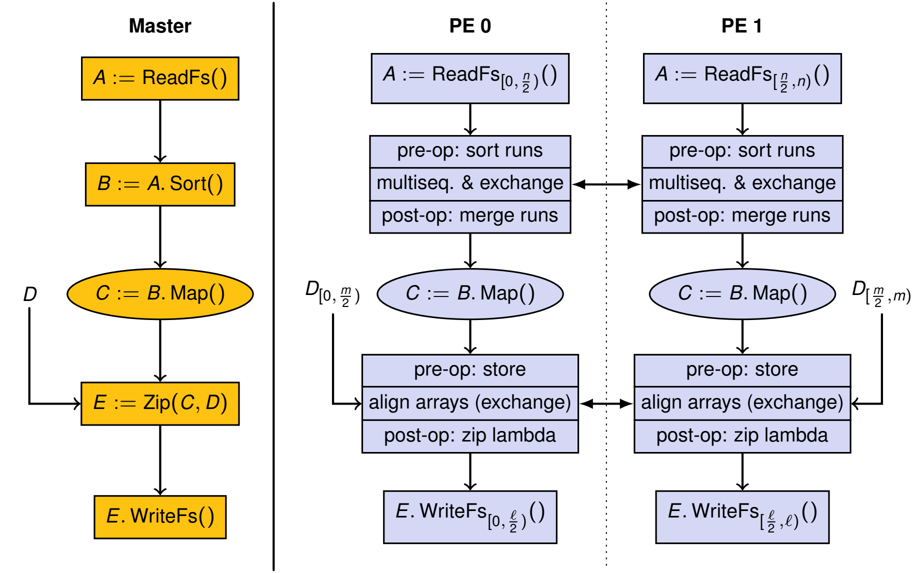


Execution on Cluster

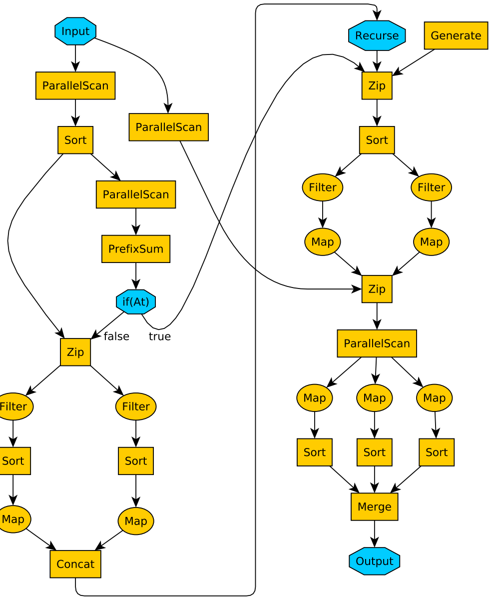


- Compile program into one binary, running on all nodes.
- Must isolate user code from framework using fork().
- Master coordinates work on compute nodes.
- Control flow is decided on by the master using C++ statements.

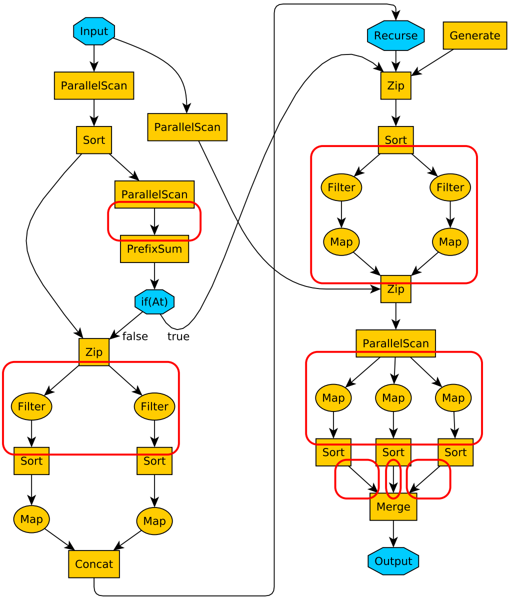
Mapping Data-Flow Nodes to Cluster



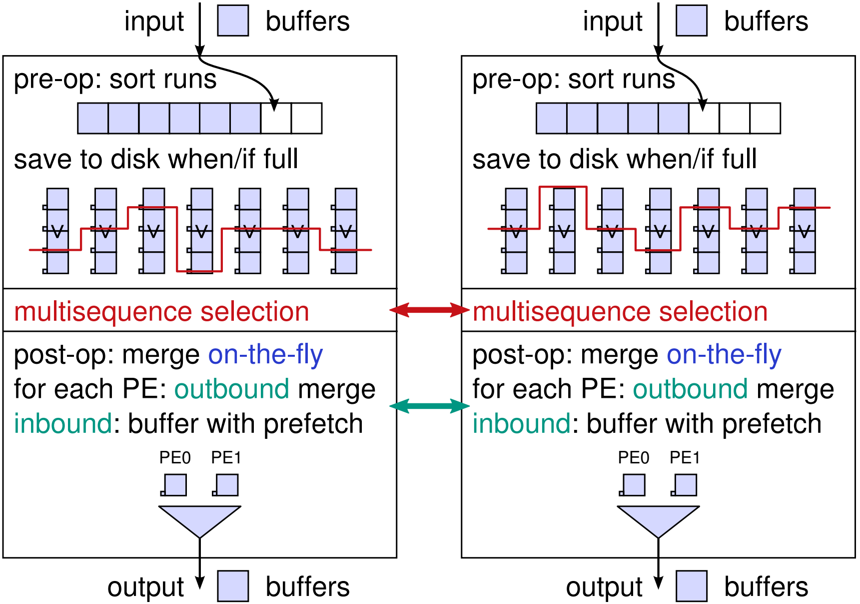
Decomposing Data-Flow into Stages



Decomposing Data-Flow into Stages



Sorting DOp


input
buffers
pre-op: sort runs
save to disk when/if full
multisequence selection
post-op: merge on-the-fly
for each PE: outbound merge
inbound: buffer with prefetch
PE0
PE1
output
buffers
input
buffers
pre-op: sort runs
save to disk when/if full
multisequence selection
post-op: merge on-the-fly
for each PE: outbound merge
inbound: buffer with prefetch
PE0
PE1
output
buffers

Redundancy Model

Karlsruhe Institute of Technology

- The Master does not fail.
- As DIAs contain their computation chain, only to checkpoint "expensive" operations (or by user request).
- DOps must expose a checkpointable recoverable state.
- Simple approach: use `fork()` and write state of DOp to fault-tolerant distributed file system.
- Future research work: make fault resilient DOps.

Key Points in Batch Processing

Dimension	DALKIT (Our Project)	Spark	(Hadoop) MapReduce
Languages	C++, ...	Java, Scala, Python	Java, ...
Memory	external / internal	internal	external
Interface	DAGs of rich primitives		Map+Reduce
Item Model	opaque items	opaque items / key-value items	
Data	arrays	RDDs (sets)	sets
Redundancy	opportunistic	via Tachyon	after each round

Project DALKIT (informal working title)

Michael Axtmann, Timo Bingmann, Peter Sanders, Sebastian Schlag, and Students | 2015-03-27

INSTITUTE OF THEORETICAL INFORMATICS – ALGORITHMICS