Parallel Processing
MapReduce, FlumeJava and Dryad

Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

What do we do when there is too much data to process?

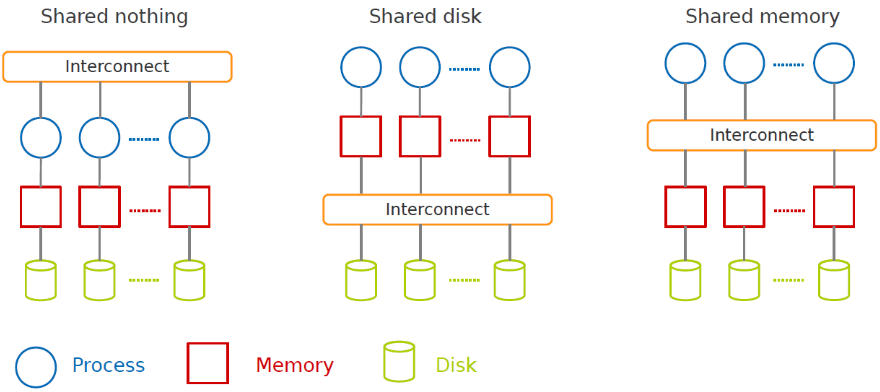
Scale Up vs. Scale Out (1/2)

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

Scale Up vs. Scale Out (2/2)

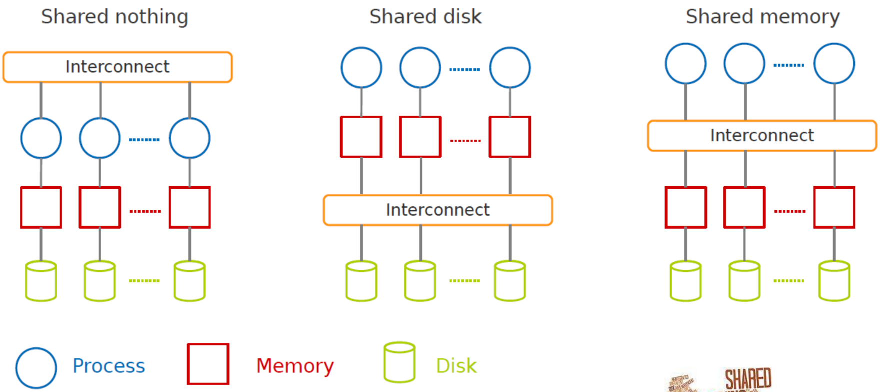
- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

MapReduce

MapReduce

- A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters of commodity hardware.

Challenges

- How to distribute computation?
- How can we make it easy to write distributed programs?
- Machines failure.

Idea

- Issue:
  - Copying data over a network takes time.

Idea

- Issue:
  - Copying data over a network takes time.

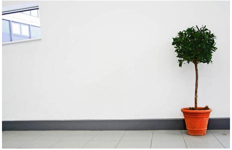
- Idea:
  - Bring computation close to the data.
  - Store files multiple times for reliability.

Simplicity

- Don't worry about parallelization, fault tolerance, data distribution, and load balancing (MapReduce takes care of these).

- Hide system-level details from programmers.

Simplicity!



MapReduce Definition

- A programming model: to batch process large data sets (inspired by functional programming).

- An execution framework: to run parallel algorithms on clusters of commodity hardware.

Programming Model

Warmup Task

- We have a huge text document.
- Count the number of times each distinct word appears in the file

Warmup Task

- We have a huge text document.
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`

Warmup Task

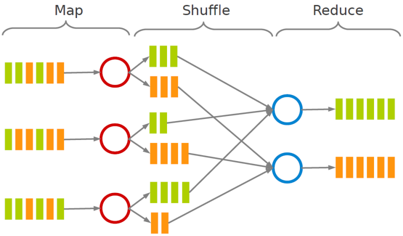
- We have a huge text document.
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`
- If not?

MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`

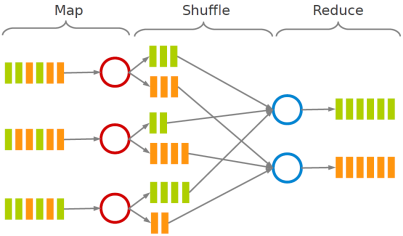
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.



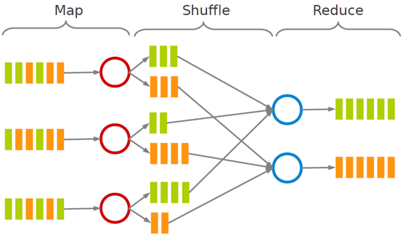
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.



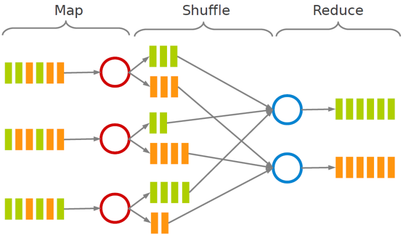
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.



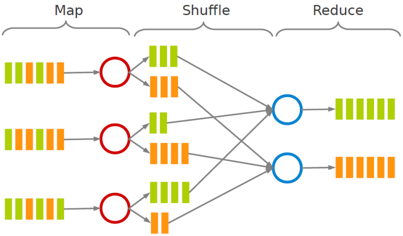
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.



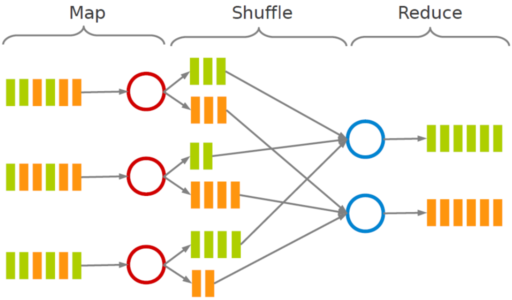
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.



MapReduce Dataflow

- map function: processes data and generates a set of intermediate key/value pairs.
- reduce function: merges all intermediate values associated with the same intermediate key.



Word Count in MapReduce

- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Word Count in MapReduce - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.
- The map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Word Count in MapReduce - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.
- The reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Word Count in MapReduce - reduce

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.
- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Combiner Function (1/2)

- In some cases, there is significant repetition in the intermediate keys produced by each map task, and the reduce function is commutative and associative.

Machine 1:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
```

Machine 2:

```
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Combiner Function (2/2)

- Users can specify an optional combiner function to merge partially data before it is sent over the network to the reduce function.
- Typically the same code is used to implement both the combiner and the reduce function.

Machine 1:

```
(Hello, 1)
(World, 2)
(Bye, 1)
```

Machine 2:

```
(Hello, 1)
(Hadoop, 2)
(Goodbye, 1)
```

Example: Word Count - map

```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

Example: Word Count - reduce

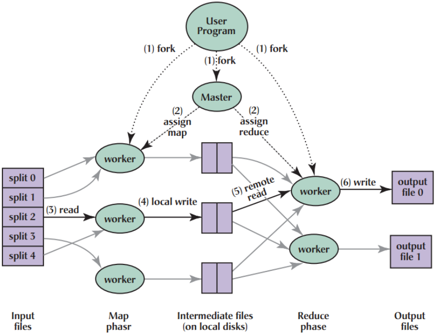
```
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

Example: Word Count - driver

```java
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setCombinerClass(MyReduce.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```

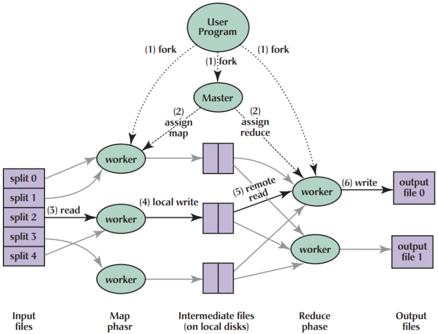
Implementation

Architecture



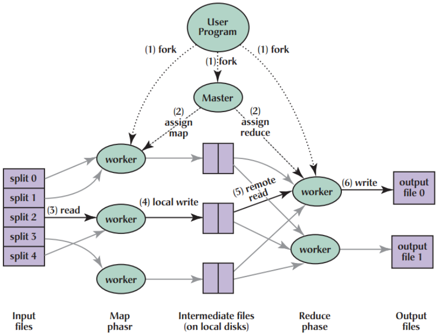
MapReduce Execution (1/7)

- The user program divides the input files into M splits.
  - A typical size of a split is the size of a HDFS block (64 MB).
  - Converts them to key/value pairs.
- It starts up many copies of the program on a cluster of machines.



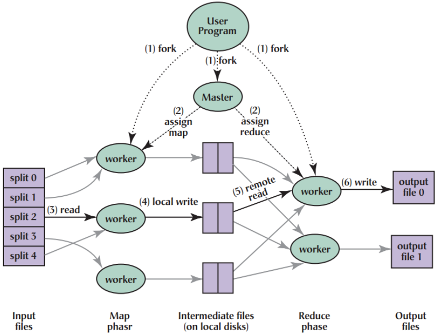
MapReduce Execution (2/7)

- One of the copies of the program is master, and the rest are workers.
- The master assigns works to the workers.
  - It picks idle workers and assigns each one a map task or a reduce task.



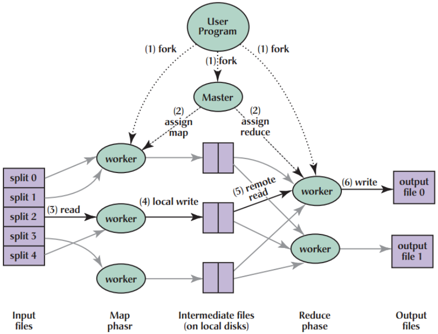
MapReduce Execution (3/7)

- A map worker reads the contents of the corresponding input splits.
- It parses key/value pairs out of the input data and passes each pair to the user defined map function.
- The intermediate key/value pairs produced by the map function are buffered in memory.



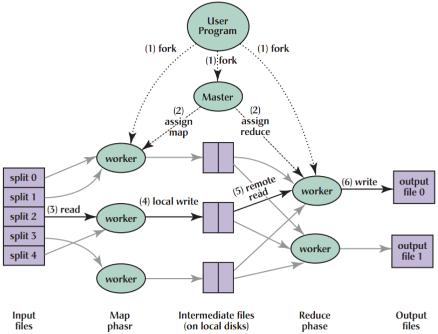
MapReduce Execution (4/7)

- The buffered pairs are periodically written to local disk.
  - They are partitioned into R regions (hash(key) mod R).
- The locations of the buffered pairs on the local disk are passed back to the master.
- The master forwards these locations to the reduce workers.



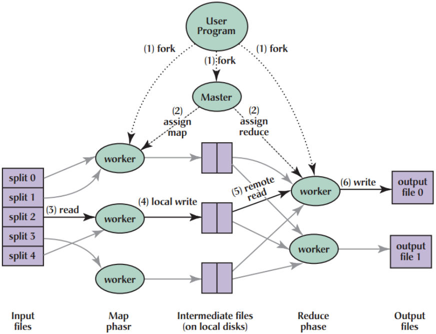
MapReduce Execution (5/7)

- A reduce worker reads the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.



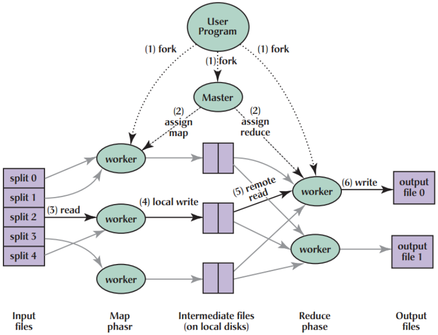
MapReduce Execution (6/7)

- The reduce worker iterates over the intermediate data.
- For each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user defined reduce function.
- The output of the reduce function is appended to a final output file for this reduce partition.

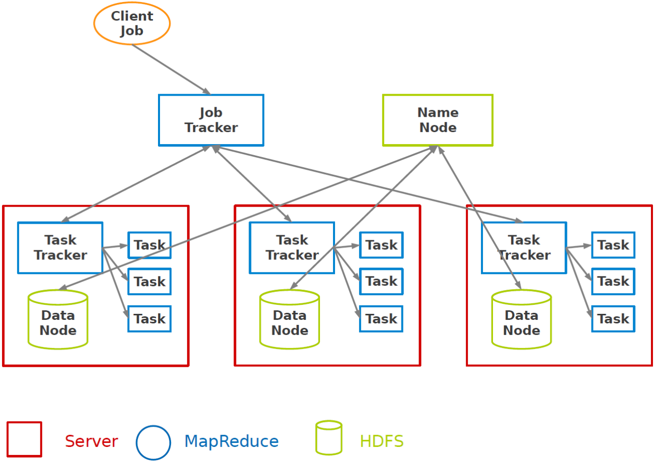


MapReduce Execution (7/7)

- When all map tasks and reduce tasks have been completed, the master wakes up the user program.



Hadoop MapReduce and HDFS



Fault Tolerance - Worker

- Detect failure via periodic heartbeats.
- Re-execute in-progress map and reduce tasks.
- Re-execute completed map tasks: their output is stored on the local disk of the failed machine and is therefore inaccessible.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global filesystem.

Fault Tolerance - Master

- State is periodically checkpointed: a new copy of master starts from the last checkpoint state.

MapReduce Limitation

- Redundant processing
- Lack of early termination
- Lack of iteration
- Lack of interactive processing
- Lack of real-time processing

FlumeJava

Motivation (1/2)

- It is easy in MapReduce:
  `words(doc.txt) | sort | uniq -c`

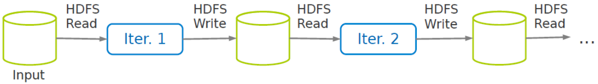
Motivation (1/2)

- It is easy in MapReduce:
  `words(doc.txt) | sort | uniq -c`

- What about this one?
  `words(doc.txt) | grep | sed | sort | awk | perl`

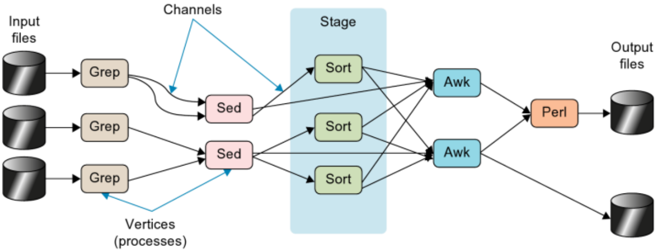
Motivation (2/2)

- Big jobs in MapReduce run in more than one Map-Reduce stages.
- Reducers of each stage write to replicated storage, e.g., HDFS.



FlumeJava

- FlumeJava is a library provided by Google to simply the creation of pipelined MapReduce tasks.



Parallel Collections

- A few classes that represent parallel collections and abstract away the details of how data is represented.

Parallel Collections

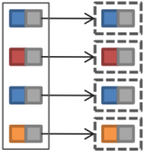
- A few classes that represent parallel collections and abstract away the details of how data is represented.
- `PCollection<T>`: an immutable bag of elements of type `T`.
- `PTable<K, V>`: an immutable multi-map with keys of type `K` and values of type `V`.

Parallel Collections

- A few classes that represent parallel collections and abstract away the details of how data is represented.
- `PCollection<T>`: an immutable bag of elements of type `T`.
- `PTable<K, V>`: an immutable multi-map with keys of type `K` and values of type `V`.
- The main way to manipulate these collections is to invoke a data-parallel operation (transform) on them.

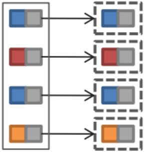
Transforms (1/2)

- parallelDo(): elementwise computation over an input PCollection<T> to produce a new output PCollection<S>.

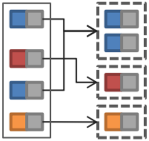


Transforms (1/2)

- **parallelDo()**: elementwise computation over an input `PCollection<T>` to produce a new output `PCollection<S>`.



- **groupByKey()**: converts a multi-map of type `PTable<K, V>` into a uni-map of type `PTable<K, Collection<V>>`.



Transforms (2/2)

- `combineValues()`: takes an input `PTable<K, Collection<V>>` and an associative combining function on `V`s, and returns a `PTable<K, V>`, where each input collection of values has been combined into a single output value.

Transforms (2/2)

- `combineValues()`: takes an input `PTable<K, Collection<V>>` and an associative combining function on `V`s, and returns a `PTable<K, V>`, where each input collection of values has been combined into a single output value.

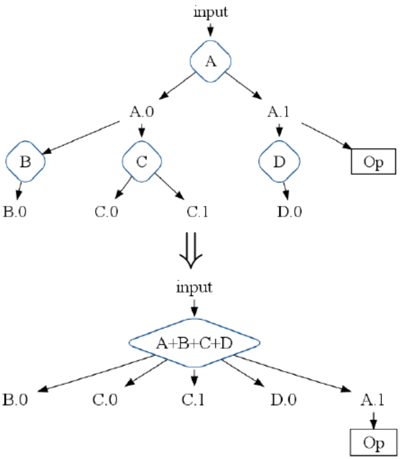
- `flatten()`: takes a list of `PCollection<T>`s and returns a single `PCollection<T>` that contains all the elements of the input `PCollections`.

Word Count in FlumeJava

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    Pipeline pipeline = new MRPipeline(WordCount.class);

    PCollection<String> lines = pipeline.readTextFile(args[0]);

    PCollection<String> words = lines.parallelDo(new DoFn<String, String>() {
      public void process(String line, Emitter<String> emitter) {
        for (String word : line.split("\\s+")) {
          emitter.emit(word);
        }
      }
    }, Writables.strings());

    PTable<String, Long> counts = Aggregate.count(words);

    pipeline.writeTextFile(counts, args[1]);

    pipeline.done();
  }
}
```

Dataflow Optimization (1/2)

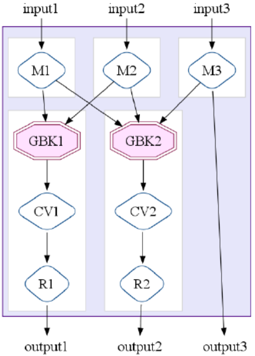
- ParallelDo fusion



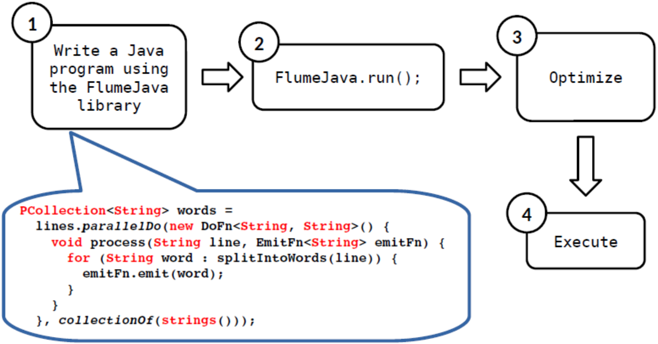
Dataflow Optimization (2/2)

- MapShuffleCombineReduce (MSCR): combining `ParallelDo`, `GroupByKey`, `CombineValues`, and `Flatten` into single MapReduces.

- Generalizes MapReduce
  - Multiple reducers/combiners
  - Multiple output per reducer
  - Pass-through outputs



FlumeJava Workflow



Dryad

Motivation (1/2)

- It is easy in MapReduce:
  `words(doc.txt) | sort | uniq -c`

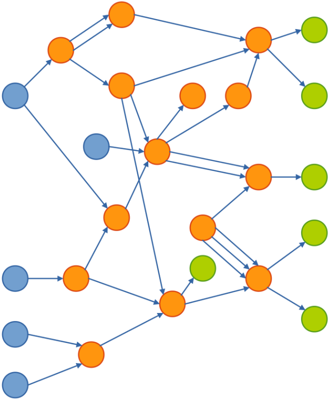
Motivation (1/2)

- It is easy in MapReduce:
  `words(doc.txt) | sort | uniq -c`

- What about this one?
  `words(doc.txt) | grep | sed | sort | awk | perl`

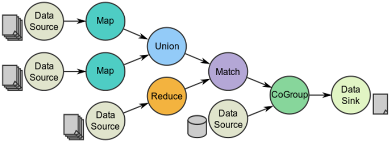
Motivation (1/3)

- In Dryad, each job is represented with a DAG.
- Intermediate vertices write to channels.
- More operation than `map` and `reduce`, e.g., `join` and `distribute`.



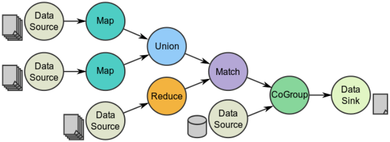
Motivation (3/3)

- Dataflow is a popular abstraction for parallel programming.



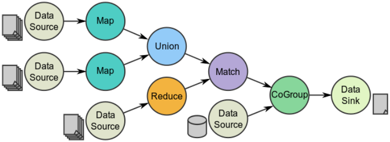
Motivation (3/3)

- Dataflow is a popular abstraction for parallel programming.
- Don't worry about the global state of a system: just write simple vertices that maintain local state and communicate with other vertices through edges.



Motivation (3/3)

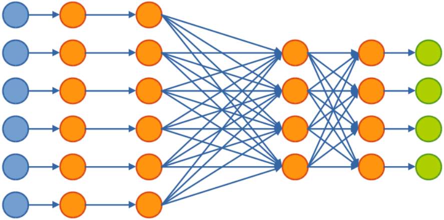
- Dataflow is a popular abstraction for parallel programming.
- Don't worry about the global state of a system: just write simple vertices that maintain local state and communicate with other vertices through edges.
- MapReduce is a simple form of dataflow, with two types vertices: the mapper and the reducer



Programming Model

Programming Model (1/2)

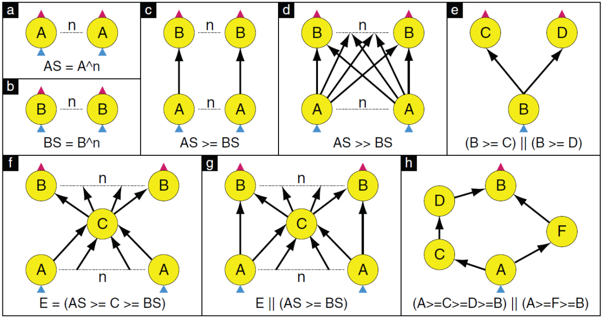
- Jobs are expressed as a Directed Acyclic Graph (DAG): dataflow
- Vertices are computations.
- Edges are communication channels.



Programming Model (2/2)

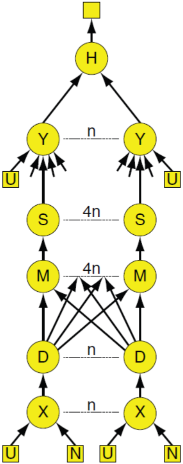
- Each vertex can have several input and output channels.
- Each vertex runs one or more times.
- Stop when all vertices have completed their execution at least once.

Graph Description Operators (1/2)



Graph Description Operators (2/2)

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;
GraphBuilder XInputs = (ugriz1 >= XSet) ||
                       (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;
GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i) {
  XToY = XToY ||
  (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}
GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;
GraphBuilder final = XInputs || YInputs ||
                     XToY || YToH || HOutputs;
```



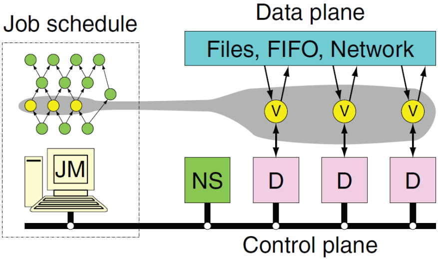
Word Count in DryadLINQ

```
public class WordCount {
  public static void WordCountExample() {
    var config = new DryadLinqContext(1);
    var lines = new LineRecord[] { new LineRecord("This is a dummy line") };
    var input = config.FromEnumerable(lines);

    var words = input.SelectMany(x => x.Line.Split(' '));
    var groups = words.GroupBy(x => x);
    var counts = groups.Select(x =>
      new KeyValuePair<string, int>(x.Key, x.Count()));
    var toOutput = counts.Select(x =>
      new LineRecord(String.Format("{0}: {1}", x.Key, x.Value)));

    foreach (LineRecord line in toOutput) {
      Console.WriteLine(line.Line);
    }
  }
}
```

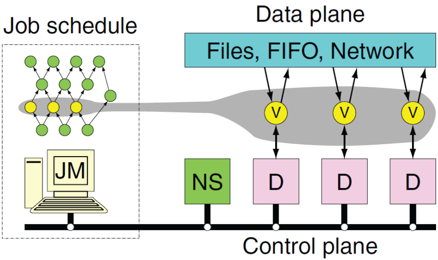
Implementation

Dryad Architecture

- Job manager (JM)
- Vertices (V)
- Daemon (D)
- Name server (NS)

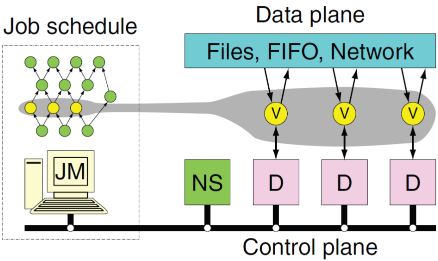


Job Manager



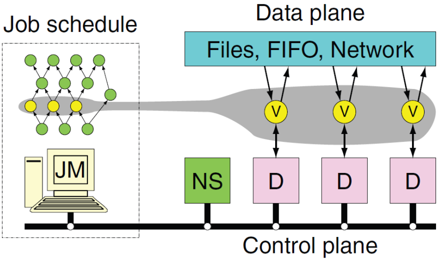
- Constructs the job's DAG.
- Schedule the work across the available resources in the cluster
- Dynamic graph refinements.

Vertices and Channels



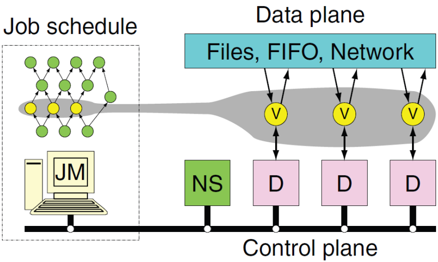
- **Vertex**: arbitrary binary application code.
  - The binary code will be sent to the corresponding node directly from the JM.
- **Channels**: transport a finite sequence of structured items between vertices.
  - Files, TCP pipes, or shared memory (FIFO)

Daemons



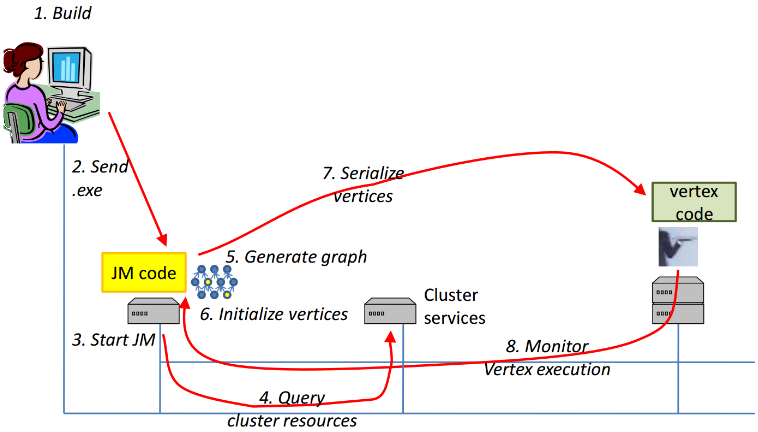
- Running on each computer in the cluster.
- Create processes on behalf of the JM.
- As a proxy that so that the JM can communicate with the remote vertices.

Name Server



- Enumerate all the available computers in the cluster.
- Exposes the position of each computer within the network topology: locality.

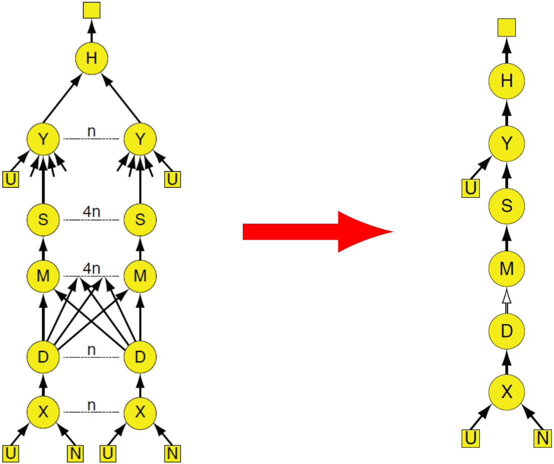
Dryad Execution (1/2)



Dryad Execution (2/2)

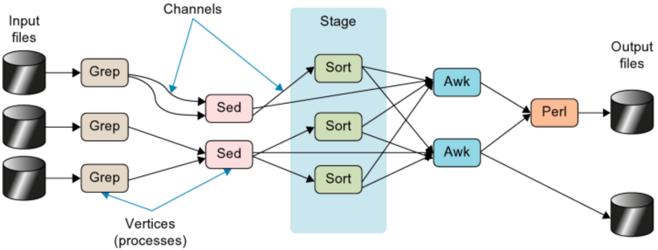
- Dataflow is mapped on a set of computation engines.
- During the runtime the JM monitors the states of the vertices through the daemons.
- When all input channels of a vertex become ready a new execution record is created for the vertex and placed in a scheduling queue.
- Prefer executing a vertex near its inputs.
- If every vertex eventually completes then the job is deemed to have completed successfully.

Job Stages and Scalability (1/2)



Job Stages and Scalability (2/2)

- Stage manager
  - Locality
  - Replicated stages to avoid straggler problem
- `words(doc.txt) | grep | sed | sort | awk | perl`



Fault Tolerance

- JM fails
  - Computation fails.

- Vertex computation fails
  - Restart vertex with different version number.
  - Previous instance of vertex may run in parallel with new instances.

Summary

Summary

- Scaling out: shared nothing architecture
- MapReduce
  - Programming model: Map and Reduce
  - Execution framework
- FlumeJava
  - Dataflow DAG
  - Parallel collection: `PCollection` and `PTable`
  - Transforms: `ParallelDo`, `GroupByKey`, `CombineValues`, `Flatten`
- Dryad
  - Dataflow DAG
  - Job manage, vertices and channels, name server

Questions?