Big Data Analytics

4. Map Reduce I

Lars Schmidt-Thieme

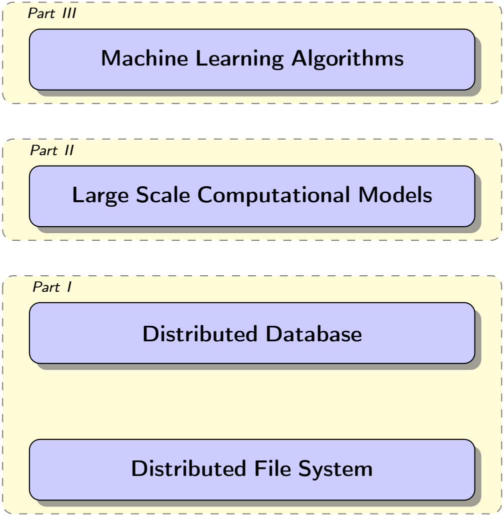
Information Systems and Machine Learning Lab (ISMLL)
Institute of Computer Science
University of Hildesheim, Germany

original slides by Lucas Rego Drumond, ISMLL

Outline

Outline

Overview


Part III
Machine Learning Algorithms
Part II
Large Scale Computational Models
Part I
Distributed Database
Distributed File System

Why do we need a Computational Model?

- Our data is nicely stored in a distributed infrastructure

Why do we need a Computational Model?

- Our data is nicely stored in a distributed infrastructure
- We have a number of computers at our disposal

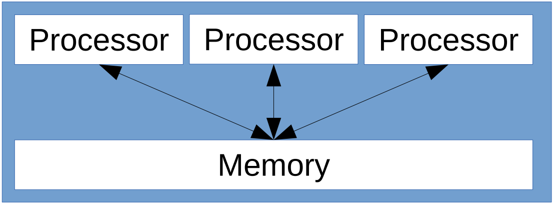
Why do we need a Computational Model?

- Our data is nicely stored in a distributed infrastructure
- We have a number of computers at our disposal
- We want our analytics software to take advantage of all this computing power

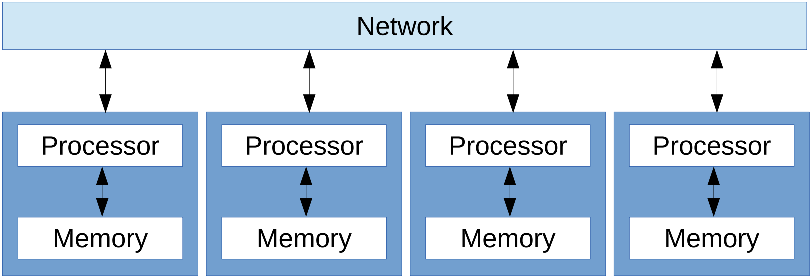
Why do we need a Computational Model?

- Our data is nicely stored in a distributed infrastructure
- We have a number of computers at our disposal
- We want our analytics software to take advantage of all this computing power
- When programming we want to focus on understanding our data and not our infrastructure

Shared Memory Infrastructure



Distributed Infrastructure



Outline

Parallel Computing principles

- We have p processors available to execute a task T

Parallel Computing principles

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed

Parallel Computing principles

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed
- Reality: synchronisation and communication costs

Parallel Computing principles

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed
- Reality: synchronisation and communication costs
- Speedup $s(T, p)$ of a task T by using p processors:

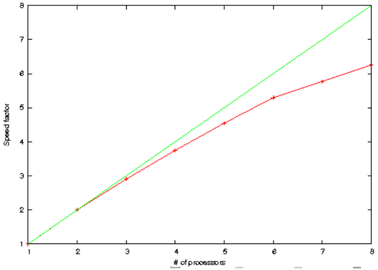
Parallel Computing principles

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed
- Reality: synchronisation and communication costs
- Speedup $s(T, p)$ of a task T by using p processors:
  - Be $t(T, p)$ the time needed to execute T using p processors

Parallel Computing principles

- We have p processors available to execute a task T
- Ideally: the more processors the faster a task is executed
- Reality: synchronisation and communication costs
- Speedup $s(T, p)$ of a task T by using p processors:
  - Be $t(T, p)$ the time needed to execute T using p processors
  - **Speedup** is given by:

$$s(T, p) = \frac{t(T, 1)}{t(T, p)}$$



Parallel Computing principles

- We have p processors available to execute a task T

Parallel Computing principles

- We have p processors available to execute a task T

- **Efficiency** $e(T, p)$ of a task T by using p processors:

Parallel Computing principles

- We have p processors available to execute a task T

- **Efficiency** $e(T, p)$ of a task T by using p processors:

$$e(T, p) = \frac{t(T, 1)}{p \cdot t(T, p)}$$

Considerations

- It is not worth using a lot of processors for solving small problems

Considerations

- It is not worth using a lot of processors for solving small problems
- Algorithms should increase efficiency with problem size

Paradigms - Shared Memory

- All the processors have access to all the data $D := \{d_1, \ldots, d_n\}$

Paradigms - Shared Memory

- All the processors have access to all the data $D := \{d_1, \ldots, d_n\}$
- Pieces of data can be overwritten

Paradigms - Shared Memory

- All the processors have access to all the data $D := \{d_1, \ldots, d_n\}$
- Pieces of data can be overwritten
- Processors need to lock datapoints before using them

Paradigms - Shared Memory

- All the processors have access to all the data $D := \{d_1, \ldots, d_n\}$
- Pieces of data can be overwritten
- Processors need to lock datapoints before using them

For each processor p:

1. lock(d_i)
2. process(d_i)
3. unlock(d_i)

Word Count Example

Given a corpus of text documents

$$D := \{d_1, \ldots, d_n\}$$

Word Count Example

Given a corpus of text documents

$$D := \{d_1, \ldots, d_n\}$$

each containing a sequence of words:

$$``w_1, \ldots, w_m''$$

pooled from a set W of possible words.

Word Count Example

Given a corpus of text documents

$$D := \{d_1, \ldots, d_n\}$$

each containing a sequence of words:

$$“w_1, \ldots, w_m''$$

pooled from a set W of possible words.

the task is to generate word counts for each word in the corpus

Word Count - Shared Memory

Shared vector for word counts: $c \in \mathbb{R}^{|W|}$

$c \leftarrow \{0\}^{|W|}$

Each processor:

1. access a document $d \in D$
2. for each word w_i in document d:
   2.1 lock(c_i)
   2.2 $c_i \leftarrow c_i + 1$
   2.3 unlock(c_i)

Paradigms - Shared Memory

- Inefficient in a distributed scenario

Paradigms - Shared Memory

- Inefficient in a distributed scenario
- Results of a process can easily be overwritten

Paradigms - Shared Memory

- Inefficient in a distributed scenario
- Results of a process can easily be overwritten
- Possible long waiting times for a piece of data because of the lock mechanism

Paradigms - Message passing

- Each processor sees only one part of the data
  $\pi(D, p) := \{d_p, \ldots, d_{p+\frac{n}{p}-1}\}$

Paradigms - Message passing

- Each processor sees only one part of the data
  $\pi(D, p) := \{d_p, \ldots, d_{p+\frac{n}{p}-1}\}$

- Each processor works on its partition

Paradigms - Message passing

- Each processor sees only one part of the data
  $\pi(D, p) := \{d_p, \ldots, d_{p+\frac{n}{p}-1}\}$
- Each processor works on its partition
- Results are exchanged between processors (message passing)

Paradigms - Message passing

- Each processor sees only one part of the data
  $\pi(D, p) := \{d_p, \ldots, d_{p+\frac{n}{p}-1}\}$
- Each processor works on its partition
- Results are exchanged between processors (message passing)

For each processor p:

1. For each $d \in \pi(D, p)$
   1.1 process(d)
2. Communicate results

Word Count - Message passing

We need to define two types of processes:

1. Slave - counts the words on a subset of documents and informs the master
2. Master - gathers counts from the slaves and sums them up

Word Count - Message passing

Slave:
Local memory:

subset of documents: $\pi(D, p) := \{d_p, \ldots, d_{p+\frac{n}{p}-1}\}$

address of the master: addr_master

local word counts: $c \in \mathbb{R}^{|W|}$

1. $c \leftarrow \{0\}^{|W|}$
2. for each document $d \in \pi(D, p)$
   for each word w_i in document d:
   $c_i \leftarrow c_i + 1$
3. **Send message** send(addr_master, c)

Word Count - Message passing

Master:

Local memory:

1. **Global word counts:** $c^{\text{global}} \in \mathbb{R}^{|W|}$
2. **List of slaves:** S

$c^{\text{global}} \leftarrow \{0\}^{|W|}$

$s \leftarrow \{0\}^{|S|}$

For each received message (p, c^p)

1. $c^{\text{global}} \leftarrow c^{\text{global}} + c^p$
2. $s_p \leftarrow 1$
3. if $||s||_1 = |S|$ return c^{global}

Paradigms - Message passing

- We need to manually assign master and slave roles for each processor

Paradigms - Message passing

- We need to manually assign master and slave roles for each processor
- Partition of the data needs to be done manually

Paradigms - Message passing

- We need to manually assign master and slave roles for each processor
- Partition of the data needs to be done manually
- Implementations like OpenMPI only provide services to exchange messages

Outline

Map-Reduce

- Builds on the distributed message passing paradigm

Map-Reduce

- Builds on the distributed message passing paradigm
- Considers the data is partitioned over the nodes

Map-Reduce

- Builds on the distributed message passing paradigm
- Considers the data is partitioned over the nodes
- Pipelined procedure:

Map-Reduce

- Builds on the distributed message passing paradigm
- Considers the data is partitioned over the nodes
- Pipelined procedure:
  1. Map phase

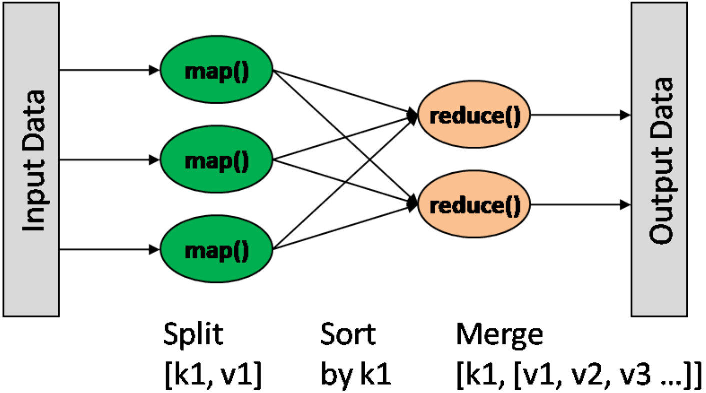
Map-Reduce

- Builds on the distributed message passing paradigm
- Considers the data is partitioned over the nodes
- Pipelined procedure:
  1. Map phase
  2. Reduce phase

Map-Reduce

- Builds on the distributed message passing paradigm
- Considers the data is partitioned over the nodes
- Pipelined procedure:
  1. Map phase
  2. Reduce phase
- High level abstraction: programmer only specifies a *map* and a *reduce* routine

Map-Reduce



- No need to worry about how many processors are available
- No need to specify which ones will be mappers and which ones will be reducers

Key-Value input data

- Map-Reduce requires the data to be stored in a key-value format

Key-Value input data

- Map-Reduce requires the data to be stored in a key-value format
- Natural if one works with column databases

Key-Value input data

- Map-Reduce requires the data to be stored in a key-value format
- Natural if one works with column databases
- Examples:

Key-Value input data

- Map-Reduce requires the data to be stored in a key-value format
- Natural if one works with column databases
- Examples:

Key	Value
document	array of words
document	word
user	movies
user	friends
user	tweet

The Paradigm - Formally

Given

- A set of input keys I
- A set of output keys O
- A set of input values X
- A set of intermediate values V
- A set of output values Y

We can define:

$$\text{map} : I \times X \rightarrow \mathcal{P}(O \times V)$$

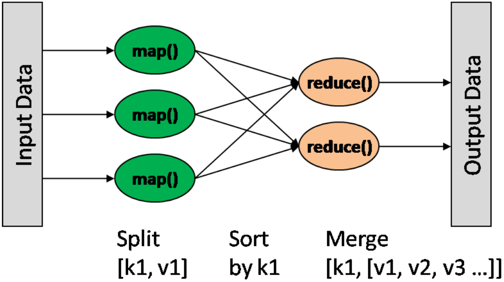
and

$$\text{reduce} : O \times \mathcal{P}(V) \rightarrow O \times Y$$

where $\mathcal{P}$ denotes the powerset

The Paradigm - Informally

1. Each mapper transforms some key-value pairs into a set of pairs of an output key and an intermediate value
2. all intermediate values are grouped according to their output keys
3. each reducer receives some pairs of a key and all its intermediate values
4. each reducer for each key aggregates all its intermediate values to one final value



Word Count Example

Map:

- Input: document-word list pairs
- Output: word-count pairs

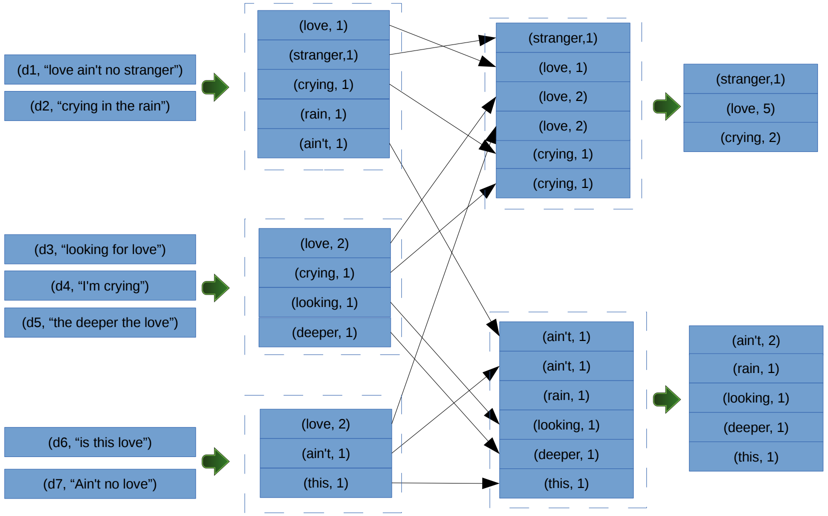
$$(d_k, \text{``}w_1, \ldots, w_m'') \mapsto [(w_i, c_i)]$$

Reduce:

- Input: word-(count list) pairs
- Output: word-count pairs

$$(w_i, [c_i]) \mapsto (w_i, \sum_{c \in [c_i]} c)$$

Word Count Example



Mappers Reducers

Map

```
public static class Map
    extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter)
        throws IOException {

        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

Reduce

```
public static class Reduce
    extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter)
        throws IOException {

        int sum = 0;
        while (values.hasNext())
            sum += values.next().get();

        output.collect(key, new IntWritable(sum));
    }
}
```

Execution snippet

```
public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

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

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

Considerations

- Maps are executed in parallel
- Reduces are executed in parallel
- Bottleneck: Reducers can only execute after all the mappers are finished

Fault tolerance

When the master node detects node failures:

- Re-executes completed and in-progress map()
- Re-executes in-progress reduce tasks

Fault tolerance

When the master node detects node failures:

- Re-executes completed and in-progress map()
- Re-executes in-progress reduce tasks

When the master node detects particular key-value pairs that cause mappers to crash:

- Problematic pairs are skipped in the execution

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T, 1) = wD$

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T, 1) = wD$
- Time for producing intermediate data σD after the *map* phase: $t(T^{\text{inter}}, 1) = \sigma D$

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T, 1) = wD$
- Time for producing intermediate data σD after the *map* phase: $t(T^{\text{inter}}, 1) = \sigma D$
- Overheads:

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T, 1) = wD$
- Time for producing intermediate data σD after the *map* phase: $t(T^{\text{inter}}, 1) = \sigma D$
- Overheads:
  - intermediate data per mapper: $\frac{\sigma D}{p}$

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T,1) = wD$
- Time for producing intermediate data σD after the *map* phase: $t(T^{\text{inter}},1) = \sigma D$
- Overheads:
  - intermediate data per mapper: $\frac{\sigma D}{p}$
  - each of the p reducers needs to read one p-th of the data written by each of the p mappers:

$$\frac{\sigma D}{p}\frac{1}{p}p = \frac{\sigma D}{p}$$

Parallel Efficiency of Map-Reduce

- We have p processors for performing *map* and *reduce* operations
- Time to perform a task T on data D: $t(T, 1) = wD$
- Time for producing intermediate data σD after the *map* phase: $t(T^{\text{inter}}, 1) = \sigma D$
- Overheads:
  - intermediate data per mapper: $\frac{\sigma D}{p}$
  - each of the p reducers needs to read one p-th of the data written by each of the p mappers:

$$\frac{\sigma D}{p}\frac{1}{p}p = \frac{\sigma D}{p}$$

- Time for performing the task with Map-reduce:

$$t_{MR}(T, p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

K - constant for representing the overhead of IO operations (reading and writing data to disk)

Parallel Efficiency of Map-Reduce

- Time for performing the task in one processor: wD

Parallel Efficiency of Map-Reduce

- Time for performing the task in one processor: wD
- Time for performing the task with p processors on Map-reduce:

$$t_{MR}(T, p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

Parallel Efficiency of Map-Reduce

- Time for performing the task in one processor: wD
- Time for performing the task with p processors on Map-reduce:

$$t_{MR}(T, p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

- Efficiency equation:

$$e(T, p) = \frac{t(T, 1)}{p \cdot t(T, p)}$$

Parallel Efficiency of Map-Reduce

- Time for performing the task in one processor: wD
- Time for performing the task with p processors on Map-reduce:

$$t_{MR}(T,p) = \frac{wD}{p} + 2K\frac{\sigma D}{p}$$

- Efficiency equation:

$$e(T,p) = \frac{t(T,1)}{p \cdot t(T,p)}$$

- Efficiency of Map-Reduce:

$$e_{MR}(T,p) = \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})}$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T,p) = \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})}$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T,p) = \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})}$$

$$= \frac{wD}{wD + 2K\sigma D}$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T,p) = \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})}$$

$$= \frac{wD}{wD + 2K\sigma D}$$

$$= \frac{wD\frac{1}{wD}}{wD\frac{1}{wD} + 2K\sigma D\frac{1}{wD}}$$

Parallel Efficiency of Map-Reduce

$$
\begin{aligned}
e_{MR}(T,p) &= \frac{wD}{p(\frac{wD}{p} + 2K\frac{\sigma D}{p})} \\
&= \frac{wD}{wD + 2K\sigma D} \\
&= \frac{wD\frac{1}{wD}}{wD\frac{1}{wD} + 2K\sigma D\frac{1}{wD}} \\
&= \frac{1}{1 + 2K\frac{\sigma}{w}}
\end{aligned}
$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T,p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

Parallel Efficiency of Map-Reduce

$$e_{MR}(T, p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

- Apparently the efficiency is independent of p

Parallel Efficiency of Map-Reduce

$$e_{MR}(T, p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

- Apparently the efficiency is independent of p
- High speedups can be achieved with large number of processors

Parallel Efficiency of Map-Reduce

$$e_{MR}(T, p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

- Apparently the efficiency is independent of p
- High speedups can be achieved with large number of processors
- If σ is high (too much intermediate data) the efficiency deteriorates

Parallel Efficiency of Map-Reduce

$$e_{MR}(T, p) = \frac{1}{1 + 2K\frac{\sigma}{w}}$$

- Apparently the efficiency is independent of p
- High speedups can be achieved with large number of processors
- If σ is high (too much intermediate data) the efficiency deteriorates
- In many cases σ depends on p