Map Reduce Programming Model

cac

NSF

RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

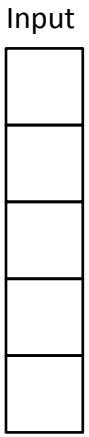
MapReduce in a Nutshell

- MapReduce
  - Programming model (and execution framework)
  - Allows expressing distributed computations on large data sets
  - MapReduce is highly scalable and can be used across many computers
  - Many small machines can be used to process jobs that normally could not be processed by a large machine
    - Designed for local clusters (locality is important!)

MapReduce in a Nutshell

- Pattern:

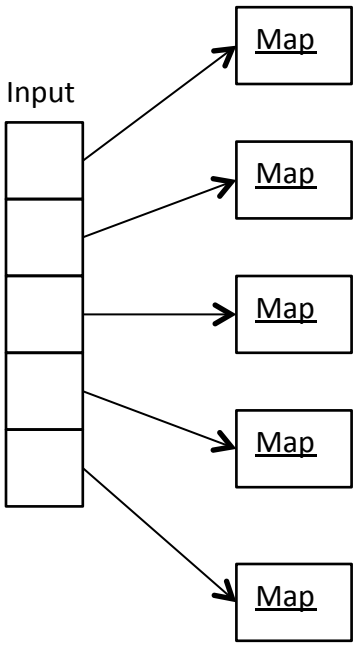
1. Application data is partitioned into smaller homogeneous pieces (chunks)



MapReduce in a Nutshell

- Pattern:

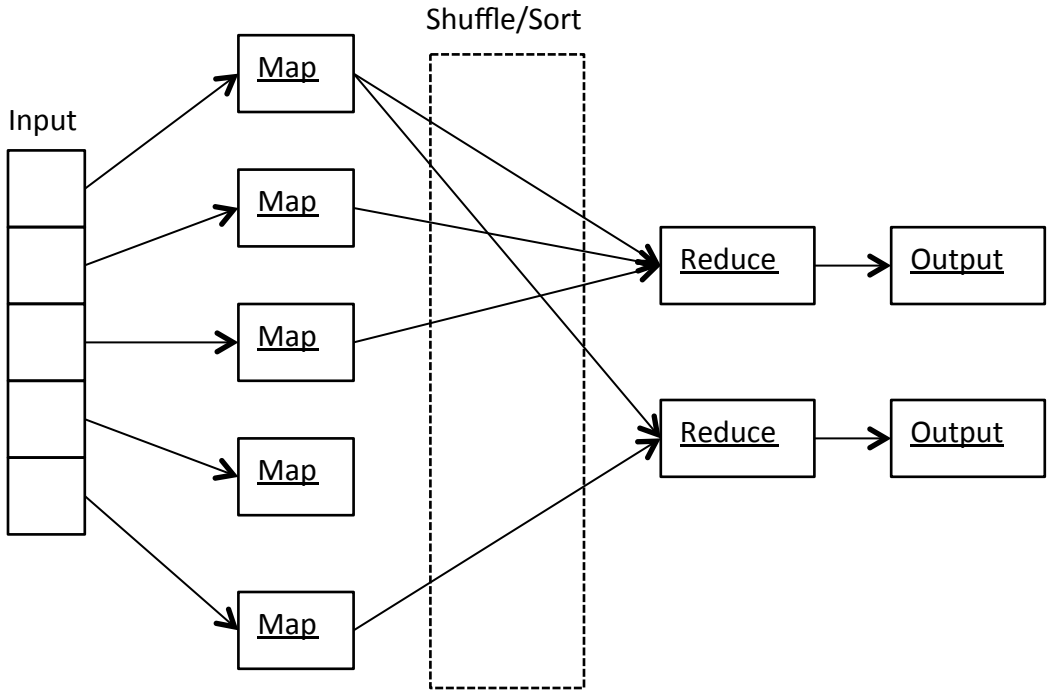
1. Application data is partitioned into smaller homogeneous pieces (chunks)
2. Performs computations on these smaller pieces



MapReduce in a Nutshell

- Pattern:

1. Application data is partitioned into smaller homogeneous pieces (chunks)
2. Performs computations on these smaller pieces
3. Aggregated the results in parallel fashion



MapReduce in a Nutshell

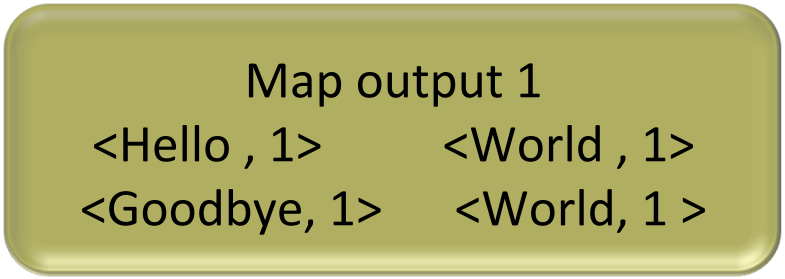
- Map abstraction
  - Inputs a key/value pair
    - Key is a reference to the input value
    - Value is the data set on which to operate
  - Evaluation
    - Function defined by user
    - Applies to every value in value input
      - Might need to parse input
  - Produces a new list of key/value pairs
    - Can be different type from input pair

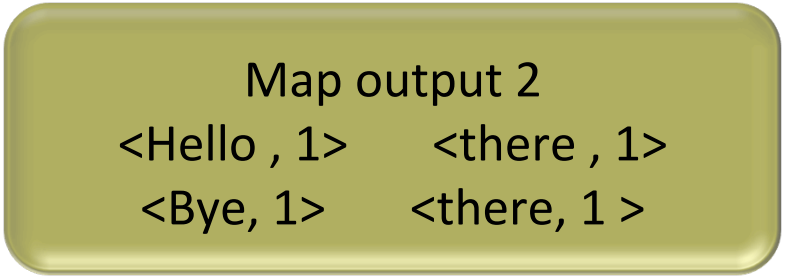
MapReduce in a Nutshell

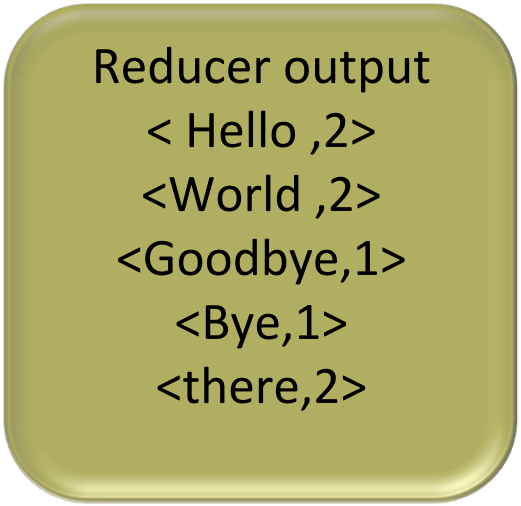
- Reduce abstraction
  - Starts with intermediate key/value pairs
  - End with finalized key/value pairs
  - Starting pairs are sorted by key
  - Iterator supplies the values for a given key to the Reduce function

MapReduce in a Nutshell

- Example: WordCount
  - Sentence 1: Hello World Goodbye World
  - Sentence 2: Hello there Bye there







MapReduce Example (WordCount)

```
public static class Map extends 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, 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);
      }
    }
  }
```

- Map reads in text and creates a {<word>,1} pair for every word read

MapReduce Example (WordCount)

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
      public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
          sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
```

- Reduce then takes all of those pairs and counts them up to produce the final count
  - If there were 20 {word,1} pairs, the final output of Reduce would be a single {word,20} pair

MapReduce Example (WordCount)

```
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(Map.class);
    job.setReducerClass(Reduce.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);
  }
}
```

MAPREDUCE-COMETCLOUD

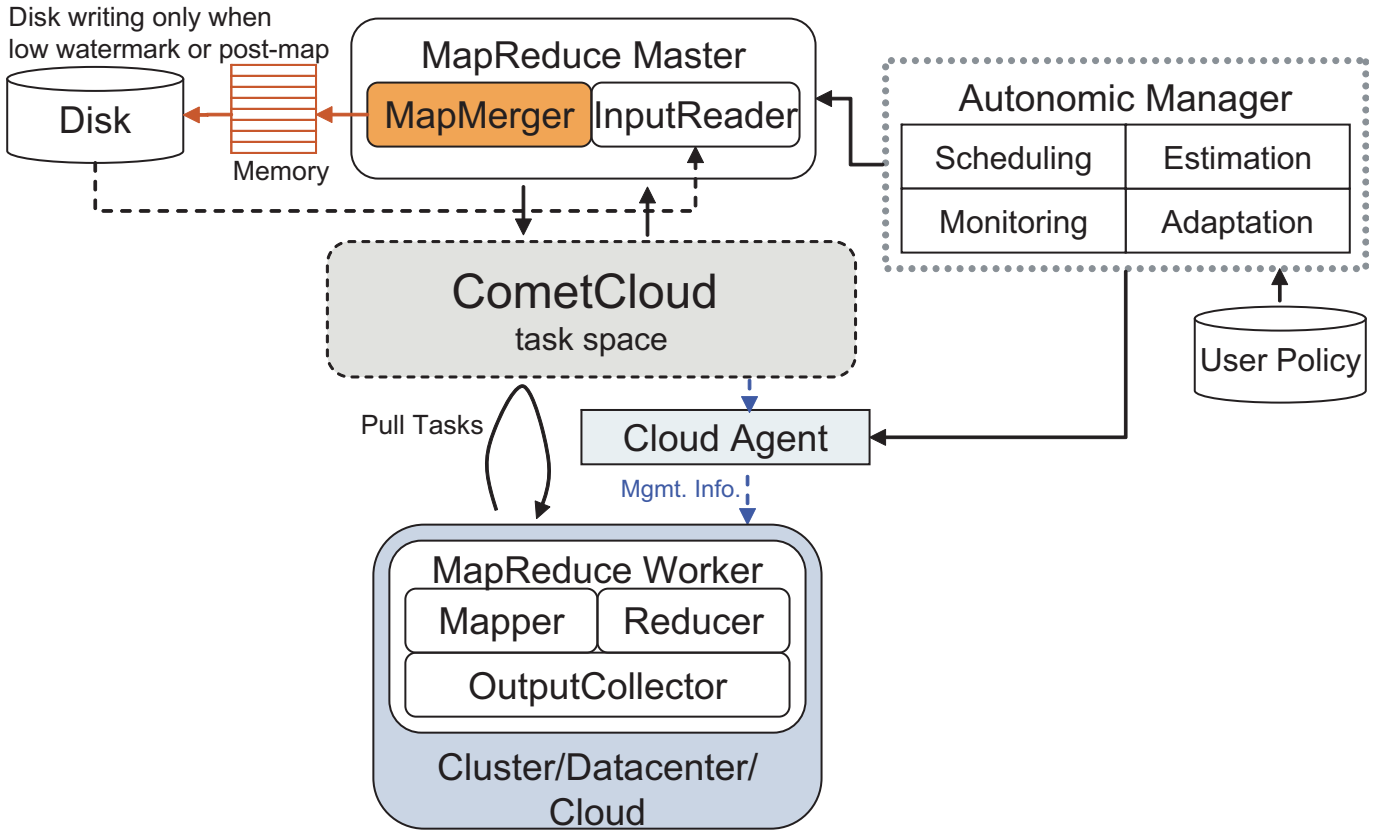
Motivation

- Performance limitations of MapReduce-Hadoop
  - MapReduce-Hadoop distributes data chunks to nodes and then pushes computational tasks to the nodes
  - Designed for single datacenters
    - Support required for multiple Clouds
- Data of varying size complex scientific workflows that can consist of a large number of small files
  - Input data size for each MapReduce task can vary
    - The heuristic comparison of task completion times to that of an average task is not feasible

Goals

- Improve the performance of MapReduce
  - By using memory instead of disk file systems
- Introduce load balancing
  - To enhance performance in heterogeneous computing environments when processing heterogeneous datasets
- Enable cloud-bursting to public clouds
  - Accelerate the deployment of MapReduce workflows
  - Meet user requirements (e.g., deadline or budget constraints)

Architecture Overview



MapReduce Dataflow

- The Input Reader
  - Can run on NFS/Master/File server is configurable
- Comet Master
  - Creates Map tasks for each of the input files in the Input Directory and puts them in the shared space
- Comet Worker – Mapper
  - Pick up map jobs from the space initially and run the user supplied mapper implementation and collects the generated key value pair output
  - The collection of map result is a set of Key and Values, is sent to the master

MapReduce Dataflow

- Output Collector in Master
  - The master periodically merges the different map results (partial aggregation is done)

    <K1, [V1, V2, V3…]>, <K2, [V1, V2, V3…]> …

  - Once all Map tasks are done the reduce tasks are put into space
- Comet Worker – Reducer
  - Pick up reduce jobs from the space initially and run the user supplied reducer implementation to get final aggregation of Key, Value
- Disk autonomics comes into play when application hits a memory threshold

Required properties files

- mapreduce.properties
  - **InputDataFile**: This is the directory path in which the input files are present. If it's a single file then it will be the complete absolute path of that file. This should be a shared folder which all machines can access.
  - **OutputDir**: The Directory where the intermediate results and in process data is stored. This should be a shared folder which all machines can access.
  - **InputReaderClass**: The complete class/path of the class file which has the user implementation of the input reader. *tassl.automate.application.mapreduce.wordcount.WordCountInputReader*
  - **MapperClass**: The complete class/path of the class file which has the user implementation of the mapper function to be executed. *tassl.automate.application.mapreduce.wordcount.WordCountMapper*
  - **ReducerClass**: The complete class/path of the class file which has the user implementation of the reducer function to be executed. *tassl.automate.application.mapreduce.wordcount.WordCountReducer*

Required properties files

- **NumMapTasks**: This is an optional parameter. This gives the number of map tasks to be run. If this value is commented out from the file the default number of tasks would be the same as the number of files in the directory mentioned in the InputDataFile option.
- **NumReduceTasks**: This is an optional parameter. This gives the number of map tasks to be run. If this value is commented out from the file the default number of tasks would be the total final keys obtained at the end of Map tasks.
- **hpcs**: This parameter tells whether the machines used to execution use linux or windows OS, If hpcs =1 ⇔ Windows OS any other value or commenting out the parameter would by default assume it to be Linux/ Unix based OS
- **ec2**: This parameter is used in case of cloud infrastructure being used or if the data is on a remote file server (then its value will be ec2=1). Commenting it out would mean the run is on the

Questions?