High Scalable Scientific Computing using Hadoop

Ing. José P. Alberto Andreotti, Lic. Nicolás Wolovick, Dr. Javier Oscar Blanco.
albertoandreotti@gmail.com, {nicolasw,blanco}@famaf.unc.edu.ar

Fa.M.A.F., U.N.C., Diciembre de 2010.

Abstract

This work provides a description of our experience in trying to apply the capabilities of Hadoop [1] to scientific problems. Hadoop is a framework intended to provide open-source software for reliable, scalable, and distributed computing. It is the opensource counterpart of Google's map-reduce framework. Hadoop is often applied to data mining problems, in which large amounts of data[1] in the form of strings are processed.

On the other hand, scientific applications work mostly on binary data that is stored in linear structures such as vector and matrixes. Our intention is to leverage Hadoop capabilities such as scalability, and load balancing, in order to target scientific workloads. We describe our experience by discussing a specific example application that solves the problem of heat transfer in a squared board.

1 Introduction to Hadoop

Hadoop is a framework intended to provide open-source software for reliable, scalable, and distributed computing. Support for *reliability* comes into play with the use of commodity hardware where failures chances are high. Map-reduce is also a *linearly scalable* programming model that works on a *distributed* set of machines. The programmer writes two sequential functions, a *map function* and a *reduce function*. The framework does the rest, namely provide reliability, scalability and distribution. Furthermore, Hadoop is the opensource counterpart of Google's map-reduce framework, and is often applied to data mining problems, in which large amounts of data in the form of strings are processed. The most widely known parts of Hadoop are the map-reduce framework and the HDFS distributed filesystem. We will explain the concepts associated with them in the following sections.

1.1 Map-Reduce

The basic idea behind the map-reduce model of computation is to divide the computation into two phases: *map* and *reduce*. Input data is first processed by the map phase, whose output data will be the input to the reduce phase. The reduce phase will produce the final output. Both the input and output of the map and reduce functions is in the $(key, value)$ format.

During the map phase, multiple map functions begin execution. The number of these functions is determined by how the input data is processed. Input data is composed by a number of $(key, value)$ pairs. As it will be explained later, Hadoop provides a *serialization framework* to work with data in this format.

All values sharing the same key will be given as input to the same map function, and the number of map functions will be equal to the number of unique keys in the input data.

In a similar fashion, multiple reduce functions begin execution during the reduce phase. The mission

[1] Even reaching the order of petabytes.

of the reduce functions is to take the output of the map phase and write the final output to disk.
An example of this model can be seen in Figure 1, the input of the map functions is taken from the HDFS filesystem. Then the output of these functions is sorted, and provided as input to the reduce phase. During the reduce phase, the spawned reduce functions take the sorted values, perform the computation and write the final result to the filesystem.
In this example only two distinct keys are present at the end of the map phase, thus only two reduce functions are spawned. As we will show later, more complex computations may include a succession of multiple map-reduce cycles.

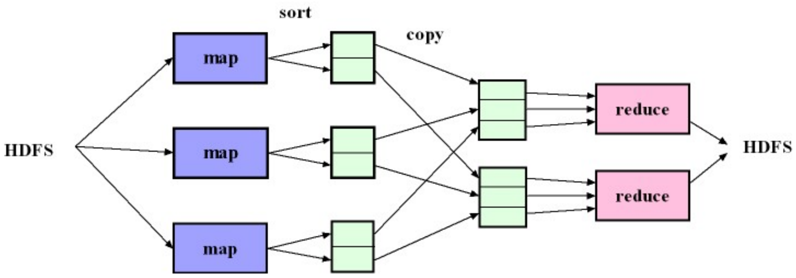


Figure 1: Example of a map-reduce job with three maps and two reduces.

1.2 HDFS

As we have previously discussed, Hadoop is intended to handle very large amounts of data, i.e., thousands of Gigabytes. The Hadoop Distributed Filesystem (HDFS) is designed to store these data reliably, and to stream them to applications at high bandwidth. In this section, we will briefly explain the architecture of HDFS. The explanation will reproduce some of the paragraphs found in [2]. However, for the sake of brevity, much of the material found in that paper was omitted, so we encourage the interested reader to take a look at the full version for a complete description.

HDFS stores filesystem metadata and application data separately. More specifically, HDFS stores metadata on a dedicated server, called the *NameNode*. Application data is stored on other servers called *DataNodes*. All servers are fully connected and communicate with each other using TCP-based protocols. Reliability is achieved by replicating file contents on multiple DataNodes. In this way, not only data reliability such as in RAID is assured, but also this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating computation *near* the needed data.
The basic architecture consists in a NameNode, DataNodes and HDFS Clients. The NameNode stores the HDFS namespace which is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 Megabytes) and each block of the file is independently replicated at multiple DataNodes (typically three). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes.
An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the closest DataNode. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipelined fashion.
Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the blocks generation stamp. During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the *namespace id* and the software version of the DataNode. If either does not match that of the NameNode, the DataNode

automatically shuts down. The namespace id is used to identify all the nodes that belong to the filesystem instance. The consistency of software versions is important because incompatible version may cause data corruption or loss.

After the handshake, the DataNode registers with the NameNode. DataNodes persistently store their unique storage ids. The storage id is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage id is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that. During normal operation DataNodes send *heartbeats* to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The NameNode does not directly call DataNodes. Instead, it uses replies to heartbeats to send instructions to the DataNodes.

The last part we will discuss is the HDFS client. The HDFS filesystem interface is accessed by user applications by means of a library. The user references files and directories by paths in the namespace, and can perform the typical read, write and delete operations in a way that is very similar to UNIX filesystems. The user generally does not need to know that filesystem metadata and storage are on different servers, or that blocks have multiple replicas.

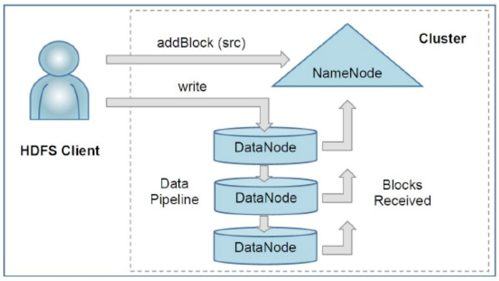


Figure 2: An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode. This figure was taken from [2].

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 2.

1.3 Jobs

A user provides a Job to the framework in order to execute useful work. A summary of the interaction of a user defined Job and the framework is provided in Table 1, which was extracted from [3].

First, the user configures a job specifying the input and its location, and ensures the input is in the expected format, and then submits the job to the framework. The job is also configured with a map and a reduce function. The framework then decomposes this job into a set of map tasks, shuffles, a sort, and a set of reduce tasks, as it was depicted in Figure 1.

After that, the framework fragments the data and distributes it among the nodes in the cluster, which can in turn be provided as input to the individual distributed tasks. Each fragment of input is called an *input split*.

Part	Handled by
Configuration of the Job.	User
Input splitting and distribution.	Hadoop framework
Start of the individual map tasks with their input split.	Hadoop framework
Map function, called once for each input key/value pair.	User
Shuffle, which partitions and sorts the per-map output.	Hadoop framework
Sort, which merge sorts the shuffle output for each partition of all map outputs.	Hadoop framework
Start of the individual reduce tasks, with their input partition.	Hadoop framework
Reduce function, which is called once for each input key, with all of the input values that share that key.	User
Collection of the output and storage in the configured job output directory, in N parts, where N is the number of reduce tasks.	Hadoop framework

Table 1: Parts of a Hadoop Job

The framework will then distribute and start the execution of the tasks, with their input splits. Job management is handled by two processes that are provided by the framework,

- TaskTracker: manages the execution of individual map and reduce tasks on a compute node in the cluster.
- JobTracker: accepts job submissions, provides job monitoring and control, and manages the distribution of tasks to the TaskTracker nodes.

Typically, there is one JobTracker process per cluster and one or more TaskTracker processes per node in the cluster. The framework is responsible for distributing the job among the TaskTracker nodes of the cluster; running the map, shuffle, sort, and reduce phases; placing the output in the output directory; and informing the user of the job-completion status.

1.4 The Serialization Framework

Serialization is the process of turning structured objects into a byte stream for transmission over a network, or for writing to persistent storage. Deserialization is the reverse process of turning a byte stream back into a series of structured objects.

Serialization appears in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage. In Hadoop, interprocess communication between nodes in the system is implemented using remote procedure calls (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

The data format chosen for persistent storage would have different requirements from a serialization framework. After all, the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. As it turns out, the desirable properties of an RPCs serialization format are also crucial for a persistent storage format.

Hadoop uses its own serialization format, *Writables*, which is certainly compact and fast, but not so easy to extend or use from languages other than Java. Since Writables are central to Hadoop (most MapReduce programs use them for their key and value types), we will focus on them.

1.4.1 The Writable Interface

The Writable interface defines two methods: one for *writing* the object's state to a DataOutput binary stream, and other for *reading* its state from a DataInput binary stream:

```
package org.apache.hadoop.io;
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

Figure 3: The Writable Interface.

As an example of a writable, we will use IntWritable, a wrapper for a Java integer scalar type. We can create one and set its value using the set() method:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Objects belonging to a class that implements the Writable interface can be used to form the splits provided as input to the map and reduce functions.

2 Heat Transfer Problem

2.1 Description of the problem

The problem we are dealing with is a classical problem of heat transfer in a squared area. It consists in finding a solution for Laplace's equation,

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

Where Φ is the unknown scalar potential that represents the heat at each point. In this heat transfer model, a squared surface with side length L is divided into a mesh, and each cell within the mesh is assigned an initial temperature value. The objective is to approximate the steady-state solution for points within the mesh. The Jacobi iteration is a common iterative method used to solve Laplace's equation. We will use the Jacobi iteration to solve the equation and will base our explanation and pseudocode on the description found in [4].

In Jacobi iteration the temperature of each cell at time $t + 1$ is computed averaging the temperature of the neighbouring cells (usually called *stencil*) at time t. The zoomed area of Figure 4 explains how this division is performed, and the resulting neighbours for a sample cell h_{ij}. Figure 5 explains how this computation takes place for a sample cell.

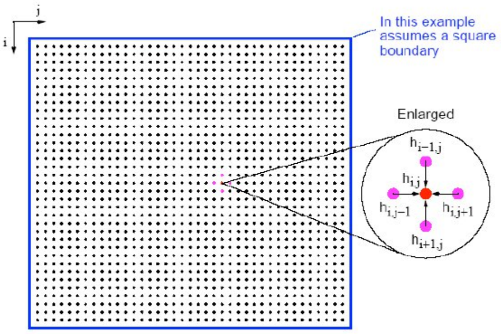
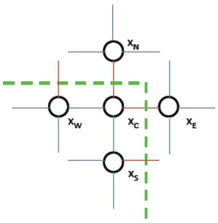


Figure 4: The squared area is divided into a mesh.



$$x_C(t+1) = \frac{x_N(t) + x_E(t) + x_S(t) + x_W(t)}{4.0}$$

Figure 5: Each cell's temperature on the next step is calculated by averaging the temperature of its neighbours.

Many variations of this problem are possible according to how we choose the boundaries to behave, and the number and location of the heat sources. First, we may fix the temperature at the boundaries so they remain the same during the whole computation, or we may update their value at every step of the computation as any other cell within the mesh. Then we can have one or multiple heat sources within the mesh and in the boundaries. For our examples, we chose to have boundaries at a fixed temperature of 0, and a single heat source.

Now, let **grid** and **next** be two $(L+2) \times (L+2)$ bidimensional matrixes. The increase in the dimensions is used to include the boundary conditions. The main computational loop of the Jacobi iteration is in Figure 6.

```
while(true) {
    // compute new values from all interior points
    for(int i=1;i<=L;i++){
        for(int j=1;j<=L;j++){
            next[i,j] = (grid[i-1,j] + grid[i+1,j] +
                         grid[i,j-1] + grid[i,j+1])/4;
        }
    }
    iters++;
    //Compute the maximum difference
    maxdiff = 0.0f;
    for(int i=1;i<=L;i++){
        for(int j=1;j<=L;j++){
            maxdiff = max(maxdiff, abs(next(i,j)-grid(i,j)));
        }
    }

    //Check for termination
    if(maxdiff<C)
        break;

    //Copy the new grid to prepare for next updates
    for(int i=1;i<=L;i++){
        for(int j=1;j<=L;j++){
            grid(i,j) = next(i,j);
        }
    }
}
```

Figure 6: Pseudocode for the Jacobi iteration adapted from [4].

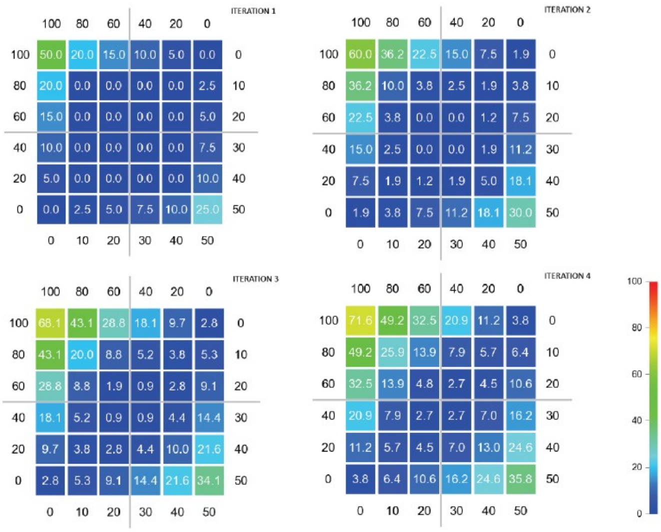
The maxdiff variable is a float used to store the maximum difference between cells during each step of the computation. In the above code, the computation of the maximum difference takes place on every iteration of the loop. However, as it will be explained in the following section, it is more convenient to make this calculation only every certain amount of iterations. An example showing four iterations can be seen in Figure 7, in this case the heat sources are in the boundaries.

2.1.1 Convergence

The calculation of temperatures based on the previous state of the neighbouring cells takes place until certain convergence criteria is met. To explain this, suppose that an initial state of the surface included a heat source in certain cell. The temperature of this cell will remain the same during the whole computation.

As one may suspect, during the first steps of the iteration the cells within the mesh will change their values abruptly. However, it may come a point in time where all the cells surrounding the heat source will be at about the same temperature of the heat source, and eventually the whole surface will be at the same temperature of the heat source.

This is what will match our physical intuition, and we will expect the model to behave in this way, but we are really more interested in studying how the heat flows in a point at time that is way before that obvious limit.



ITERATION 1

	100	80	60	40	20	0	
100	50.0	20.0	15.0	10.0	5.0	0.0	0
80	20.0	0.0	0.0	0.0	0.0	2.5	10
60	15.0	0.0	0.0	0.0	0.0	5.0	20
40	10.0	0.0	0.0	0.0	0.0	7.5	30
20	5.0	0.0	0.0	0.0	0.0	10.0	40
0	0.0	2.5	5.0	7.5	10.0	25.0	50
	0	10	20	30	40	50	

ITERATION 2

	100	80	60	40	20	0	
100	60.0	36.2	22.5	15.0	7.5	1.9	0
80	36.2	10.0	3.8	2.5	1.9	3.8	10
60	22.5	3.8	0.0	0.0	1.2	7.5	20
40	15.0	2.5	0.0	0.0	1.9	11.2	30
20	7.5	1.9	1.2	1.9	5.0	18.1	40
0	1.9	3.8	7.5	11.2	18.1	30.0	50
	0	10	20	30	40	50	

ITERATION 3

	100	80	60	40	20	0	
100	68.1	43.1	28.8	18.1	9.7	2.8	0
80	43.1	20.0	8.8	5.2	3.8	5.3	10
60	28.8	8.8	1.9	0.9	2.8	9.1	20
40	18.1	5.2	0.9	0.9	4.4	14.4	30
20	9.7	3.8	2.8	4.4	10.0	21.6	40
0	2.8	5.3	9.1	14.4	21.6	34.1	50
	0	10	20	30	40	50	

ITERATION 4

	100	80	60	40	20	0	
100	71.6	49.2	32.5	20.9	11.2	3.8	0
80	49.2	25.9	13.9	7.9	5.7	6.4	10
60	32.5	13.9	4.8	2.7	4.5	10.6	20
40	20.9	7.9	2.7	2.7	7.0	16.2	30
20	11.2	5.7	4.5	7.0	13.0	24.6	40
0	3.8	6.4	10.6	16.2	24.6	35.8	50
	0	10	20	30	40	50	

Figure 7: Four iterations of the stencil aproximation are shown. The values next to the borders are heat sources at fixed temperatures.

Instead, we will consider that the simulation has *converged* when the difference between each of the h_{ij} cells in two successive steps of the computation is less than certain value C, which we will choose conveniently.

2.2 Translating into a map-reduce application

The challenge when writing a map-reduce application is to figure out how the computation will be organized in a set of map and reduce tasks, and how these tasks are fed with the input data. We will explore two alternative approaches to achieve this in the two following subsections.
In our exposition we will use cold boundaries (i.e., borders at a fixed temperature of 0° Celsius), and a cell that will act as the heat source, remaining constant during the whole computation.

2.2.1 First Alternative

The first option we will consider is to have a map function that distributes each individual element of the matrix to all its neighbours. Thus, if after the execution of all the map functions, the values of all the neighbours of any element i are sent to reduce function i, the calculation of the average can take place on the reduce function.
In this way, the information that is output from the map phase is *four times* the size of the original input information. This implies, that we will have a replication of data that is four times the size of the input data. Furthermore, we will have LxL map functions and LxL reduce functions. Moving that amount of data, and performing that amount of method calls can be a source of performance degradation. In the next section we will explore an alternative solution that tries to reduce the overhead from both sources. We will have a performance comparison in section 3.

In order to achieve the required distribution of data, the input data must be stored in a way by which every single element in the matrix is given as input to a single unique map function. We do this by storing each matrix element with its own key, being the key the linear position of the element in the matrix. This is shown in Figure 8.

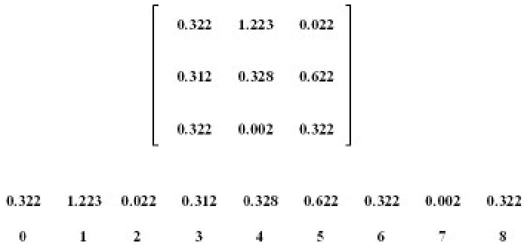


Figure 8: The matrix is stored as a succession of $(key, value)$ pairs, being each key an IntWritable and each value a FloatWritable.

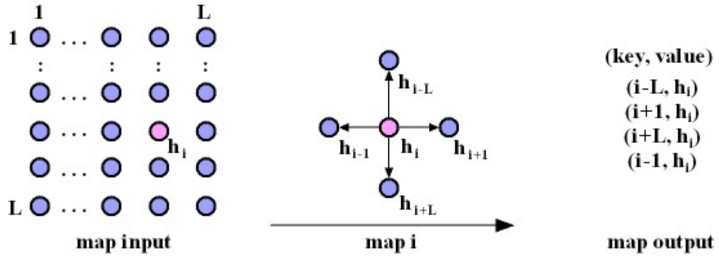


Figure 9: Map function for our first solution to the heat transfer problem.

The resulting map function that process this input data can be seen in Figure 9. On the left we have the input data. In the middle, we have the map function that distributes each element to all its neighbouring cells. The right part shows the output of the map function, which is a set of $(key, value)$ pairs.

The matching reduce function is in Figure 10. On the left, there is the input data, in the $(key, value)$ form. In the middle, we see how each reduce function receives as its input the temperature values of its neighbouring cells at the previous step. On the right side of the figure is the output data. Each reduce function contributes to a single element in the output data.

The listings for the map and reduce functions, can be seen in Figure 11, and Figure 12, respectively.

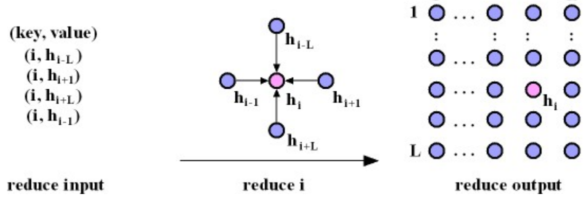


Figure 10: Reduce function for our first alternative solution of the heat transfer problem.

```
public class HeatTransfer {

public static class HeatTransferMapper
    extends Mapper<IntWritable, FloatWritable, IntWritable, FloatWritable>{

    public void map(IntWritable linearPos, FloatWritable heat, Context context
                            ) throws IOException, InterruptedException {
       int myLinearPos = linearPos.get();

       //Distribute my value to the previous and the next
       linearPos.set(myLinearPos - 1);
       context.write(linearPos, heat);
       linearPos.set(myLinearPos + 1);
       context.write(linearPos, heat);

       //Distribute my value to the cells above and below
       linearPos.set(myLinearPos - MatrixData.Length());
       context.write(linearPos, heat);
       linearPos.set(myLinearPos + MatrixData.Length());
       context.write(linearPos, heat);

    }//end map
  }//HeatTransferMapper
}//HeatTransfer
```

Figure 11: Map function implementing the behaviour of Figure 9. In this listing, the method MatrixData.Length() correspond to the length of the matrix, L.

The MatrixData class implements a constant data manager. It is used to retrieve the length and linear size of the matrix. It also stores the heat source location and value.

```
public class HeatTransfer {

    public void reduce(IntWritable linearPos, Iterable<FloatWritable> heats,
                       Context context) throws IOException, InterruptedException {

       //Handle first and last "cold" boundaries
       if(linearPos.get()<0 || linearPos.get()>MatrixData.LinearSize()){
          return;
       }

       if(linearPos.get()==MatrixData.HeatSourceLinearPos()){
          context.write(linearPos, new FloatWritable(MatrixData.HeatSourceTemperature()));
          return;
       }

       float result = 0.0f;
       //Add all the values
       for(FloatWritable heat : fwValues) {
          result += heat.get();
       }

      context.write(linearPos, new FloatWritable(result/4.0) );

    }//end reduce
}//end HeatTransfer
```

Figure 12: Reduce function implementing the behaviour of Figure 10.

2.3 Second Alternative

The next option we will explore, distributes data with a coarser granularity pattern. Instead of distributing each individual element, the map functions will now distribute entire rows of the matrix. The key to be used will be the row number within the matrix. The resulting map function is depicted in Figure 13. Similarly, the reduce function now will take three rows as input. This is shown in Figure 14. The code for both functions is found in Figures 15 and 16.

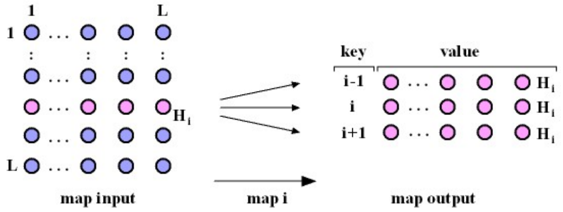


Figure 13: Map function for our second alternative solution of the heat transfer problem.

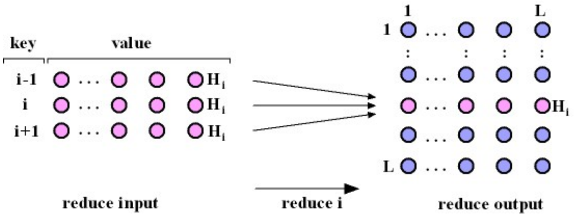


Figure 14: Reduce function for our second alternative solution of the heat transfer problem.

```
public void map(IntWritable key, FloatArrayWritable value, Context context
                        ) throws IOException, InterruptedException {

      int myKey = key.get();
      int zBasedWidth = MatrixData.Width()-1;

      key.set(myKey-1);
      context.write(key, value);
      key.set(myKey+1);
      context.write(key, value);

  key.set(myKey);
  float tmp1, tmp2;

  tmp1=value.floatAt(0);
  value.set(0, value.floatAt(1));
  for(int i=1; i<zBasedWidth; i++) {
    tmp2=value.floatAt(i);
    value.set(i, tmp1+value.floatAt(i+1));
    tmp1=tmp2;
  }
  value.set(zBasedWidth,tmp1);

  context.write(key, value);
}
```

Figure 15: Map function implementing the behaviour of Figure 13. In this listing, the method MatrixData.Length() corresponds to the length of the matrix, L.

```
public void reduce(IntWritable key, Iterable<FloatArrayWritable> values,
                   Context context) throws IOException, InterruptedException {

  FloatArrayWritable result = new FloatArrayWritable();
  FloatWritable[] FloatArray = new FloatWritable[MatrixData.Width()];

  //Initialize the result
  for(int i=0; i<(MatrixData.Width()); i++){
     FloatArray[i] = new FloatWritable(0f);
  }

  //Keys for which no output is produced
  if(key.get()<0||key.get()>(MatrixData.Height()-1))
    return;

  int zBasedWidth = MatrixData.Width()-1;

  //Add the rows
  for(FloatArrayWritable faw : values) {
      float tmp = FloatArray[0].get() + faw.floatAt(0);
  FloatArray[0].set(tmp);

  tmp = FloatArray[zBasedWidth].get() + faw.floatAt(zBasedWidth);
  FloatArray[zBasedWidth].set(tmp);

      for(int i=1; i<zBasedWidth; i++){
    tmp = FloatArray[i].get() + faw.floatAt(i);
    FloatArray[i].set(tmp);
    }
  }

  //Make the division and write the result
  for(int i=0; i<(MatrixData.Width()); i++){
      FloatArray[i].set( FloatArray[i].get()/4);
  }

  //Set heat source
  if(key.get()==MatrixData.HeatSourceY()){
      FloatArray[MatrixData.HeatSourceX()].set(MatrixData.HeatSourceTemperature());
  }

  result.set(FloatArray);
  context.write(key, result);

}
```

Figure 16: Reduce function implementing the behaviour of Figure 14.

This data distribution causes a substantial change when compared to the previous example. As explained above, now every reduce function will have three rows as input, and these will be used to perform the computation of a whole output row. This leads to a data replication that is only three times the size of the input data.
This will result in fewer map functions with longer execution times. As we will show on the next section the outcome is a substantial reduction of the overall execution time.

3 Performance & Scalability

Now that we have an interesting problem in place with two alternative solutions, we would like to explore how the framework is doing regarding performance. For doing this, we run our study case in a different number of nodes.

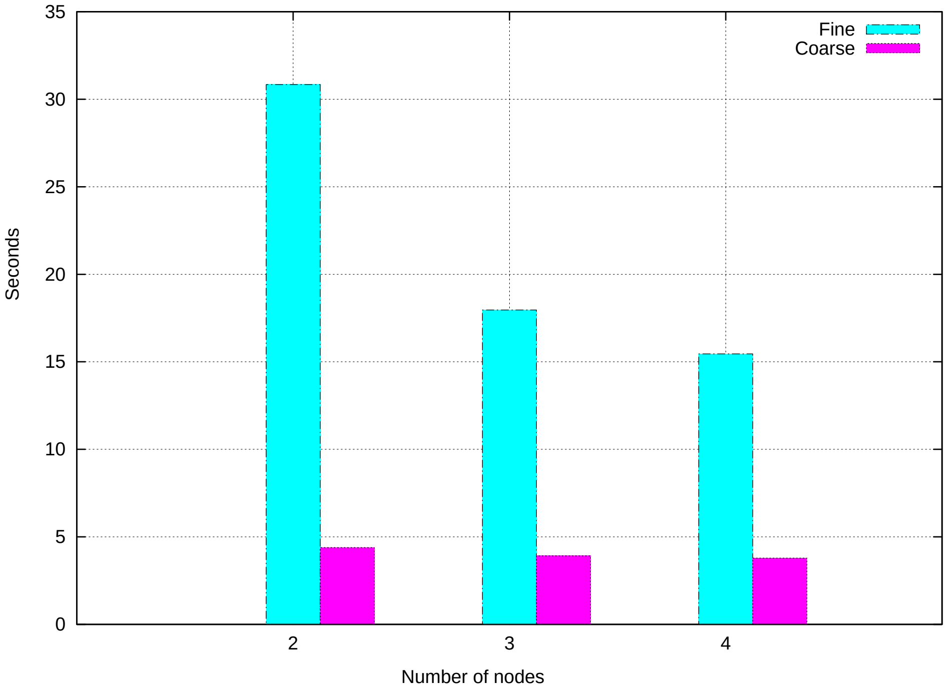


Figure 17: This graph depicts how both implementations perform for a fixed size problem of $N = 8500$. The application was run using one, two, and three Datanodes plus one node running the JobTracker.

We decided to initially run our implementations with an 8500x8500 matrix, the results are shown in Figure 17. There is a marked difference between both implementations, being the coarse grained faster by a factor ranging from 4 using three nodes to 7 when only one node is used.
As it was explained in the previous sections, this is primarily due to the overhead to spawn the map and reduce functions that takes place in the fine grained version compared to the coarse grained one; it is proportional to LxL in the first one, and proportional to L in the second one. In general, fewer map and reduce functions will result in better performance.
Furthermore, the amount of data that is moved arround is different in both versions. For every element of the matrix, *four* outputs are produced in the fine grained version. On the contrary, for the coarse grained version, for every row, the map produces only *three* output rows. Consequently, not only less data has to be moved, but also fewer keys have to be sorted in the coarse grained version.
Finally, it is noticeable from Figure 17 that the coarse grained version, although more performant for every number of processors, does not scale linearly as promissed by Hadoop. This may be due to a number of factors such as the heap size of the JVMs, number of maps and reduces, and their excecution times, and file system block size, just to mention a few. An exhaustive study of how all these factors impact the scalability of the computational load we are currently dealing with goes beyond the scope of this work.

3.1 Comparison with other computing models

Assessing Hadoop's performance compared to other computational models such as MPI or OpenMP is not a trivial task. Typical implementations for this problem using MPI/ OpenMP do not provide Hadoop's distinctive features such as a distributed filesytem, load balancing, and fault tolerance.
Furthermore, the execution model of Hadoop involves many phases that form a complex graph of activities. Detaching the strictly computational part of the execution in order to provide an acceptable measure for comparison is a challenging task. Thus, we won't provide such a comparison in this work.

4 Acknowledgements

I would like to thank Gustavo Wolfmann from LIDES at National University of Córdoba for kindly providing me access to the University cluster, and for his valuable guidance during much of the configuration that had to be done in order to obtain the results presented in this work.

References

[1] Hadoop website http://hadoop.apache.org/.

[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert ChanslerThe Hadoop Distributed filesystem, (Proceedings of MSST2010, May 2010, available at http://storageconference.org/2010/Papers/MSST/Shvachko.pdf).

[3] Jason Venner, Pro Hadoop, Apress 2009.

[4] Gregory R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2001.