Master in High Performance Computing

Big-Data in Climate Change Models - A novel approach with Hadoop MapReduce

Supervisor:
Graziano Giuliani

Candidate:
Juan Manuel Carmona-Loaiza

2nd edition
2015–2016

Dr. Juan Manuel Carmona-Loaiza:
Big-Data in Climate Change Models – A novel approach with Hadoop MapReduce.

SUPERVISOR:
Graziano Giuliani

SISSA - ICTP ,
Trieste, Italy; 6th of December, 2016

To Sebastián
and Valentina.

ACKNOWLEDGEMENTS

The projects presented in this Thesis were entirely developed by Juan Manuel Carmona-Loaiza as part of the Master in High Performance Computing (MHPC) of the International Center of Theoretical Physics (ICTP) and the Scuola Internazionale Superiore di Studi Avanzati (SISSA) under the supervision of Graziano Giuliani and funded by the Istituto Nazionale di Oceanografia e di Geofisica Sperimentale (OGS) and CINECA under HPC-TRES program award with number 2015-04.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION: THE PARADOX OF DATA AVAILABILITY IN CLIMATE CHANGE RESEARCH

Global Warming and Climate Change hot debate, in 1995, ignited a global effort among the scientific comunity to study the issue in a more collaborative way, the Coupled Model Intercomparison Project (CMIP) organized by theWorld Climate Research Program (WCRP). The project provided a thorough comparison between the different climate models, which was needed because of the considerable scatter that existed among model results and between simulated and observed data (Lambert & Boer 2001). The initiative succeeded and, by 2012, the 5th generation of the project (CMIP5) had more than 20 modelling groups performing CMIP5 simulations using more than 50 climate models. The major goals of which were, according to Taylor et al. (2012): 1) assessing the mechanisms responsible for model differences, 2) examining climate "predictability" and 3) determining why similarly forced models produce a range of responses.

The importance of such projects within Global Warming and Climate Change debate, lies in their ability to provide a range of simulated climate futures (decades to centuries) that can asses the impact of climate change on society. In particular, as suitable adaptation and mitigation strategies need to be taken at the national level, an effort to produce high-resolution "downscaled" climate data based on the CMIP5 simulations is incarnated by the Coordinated Regional Downscaling Experiment (CORDEX). CORDEX allows for evaluating and benchmarking model performances, encompassing the majority of land areas of the world and reaching resolutions as fine as 10km (Giorgi et al. 2009)[1].

Although scientists are the main users of the data generated by these projects, goverments and service providers have started realizing the need of having access to it, as strategic development to overcome climate change requires data-driven decisions. As stated by Overpeck et al. (2011): *"Climate data provide the backbone for billion dollar decisions. With this gravity comes the responsibility to curate climate data and share it more freely, usefully, and readily than ever before"*. In view of these needs, CMIP allows researchers from a wide range of communities to access climate model output. However, granting open access puts constrains on the variety and scale of the data and requires a clever storage and delivery strategies, which becomes even more difficult as the amount of data increases. Paradoxically, what is granted

1 As a complement to these efforts, the Geoengineering Model Intercomparison Project (GeoMIP) explores the effects of possible geoengineering approaches to mitigate climate change.

by models with such a high level of accuracy, is taken by the impossibility of analysing their outputs with current storage infrastructures.

CMIP6 is the most recent phase of the project. Simulations and analysis are planned to be running from 2015 to 2020, but analysis of model outputs will continue in the future (Meehl et al. 2014).In particular, in the CORDEX context, use will be made of CMIP6 global climate model (GCM) output to produce downscaled projected changes, which in turn will allow to study vulnerabilities, impacts and adaptation strategies. A particularly interesting case, which study will be allowed by such high resolution models, is the impact of human activities (e.g. land-use changes, urban development, the growth and spread of coastal megacities and the presence of anthropogenic origin aerosols) on local climate (Gutowski Jr. et al. 2016).

Of all the factors relevant for the study of climate change, Rain and Wind have a special role because of the dramatic influence they can have in society. As such, they need to be acuratelly described by regional climate models. Precipitation is one of the most difficult variables to simulate in current climate models. Important uncertainties in precipitation arise because of both systematic errors and large natural variability. Regional winds instead, such as the mistral and bora in the Mediterranean, are often driven by fine-scale topography and surface–atmosphere exchanges, thus requiring high resolution to accurately model them.

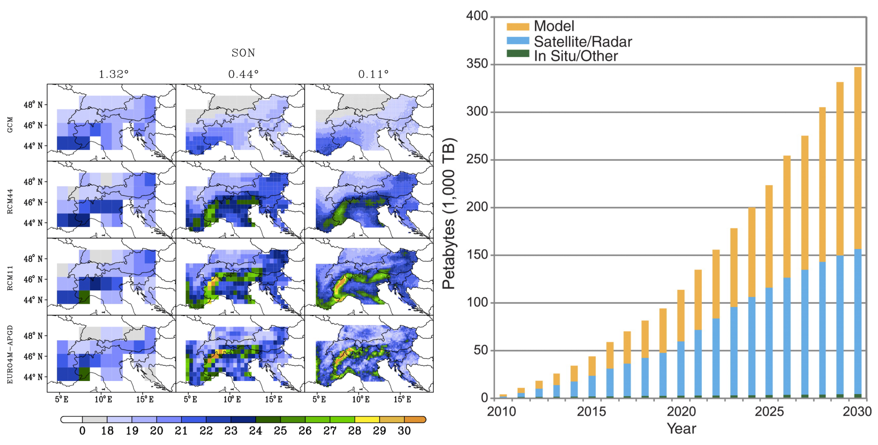

Figure 1: The paradox posed by ultra high resolution simulations: The possibility of making accurate local weather predictions, granted by ultra-high resolution models (left panel; Gutowski Jr. et al. 2016), is taken away by the impossibility of analysing their outputs with current technologies (right panel; Overpeck et al. 2011).

The paradox posed by state of the art ultra high resolution models lies in the impossibility of current technologies to efficiently store, share and analyse the outputs of the models. Coming back to the example of precipitation and wind modelling: On one hand, there is a need to characterize the uncertainty on precipitation, which requires the analysis of large ensembles of data. On the other hand there is a need of analysing ultra high resolution simulations able to accurately simulate local winds, which again translates in analysing large amounts of data. –While CMIP1 involved less than 1 TB of data, CMIP3 data amounts to 36 TB, and CMIP5 is 2.5 petabytes large (PB). If value and knowledge is going to be extracted from the ultra high resolution simulations that are running and will be running in the upcoming years, developing a clever software-hardware infrastructure which is capable of storing, sharing and analysing large ensembles of data (Petabytes) is of the utmost importance (Figure 1).

2 BIG DATA @ ICTP

Like all the Climate Research groups in the world, the Earth System Physics group at ICTP is facing the problem of storing and sharing big amounts of data produced by *Global and Regional Climate Models*. Last decade, the problem was solved by taking advantage of the network to access a big centralized storage using the Network File System (NFS). NFS is a distributed file system protocol that allows users on client computers to access files on a server as if they resided on their local machines. This is an optimum solution for *storing* and *sharing* files, even a large amount of them. However, problems arise when

i) analyses have to be performed over a large number of them, or

ii) the amount of files becomes so large and the directory tree so complicated that the main difficulty of any analysis becomes knowing where the data is actually stored.

Moreover, human mistakes like careless ordering and sticking to naming conventions can potentially result in an effective information loss, even if the data is physically there, *somewhere*[1]. With large datasets, the task of curating the data becomes extremely complicated.

Lets suppose an ideal situation in which all the users of the data –researchers, post-docs, PhD students, etc, are very careful, stick to the naming conventions and a clever ordering schema exist, so all files contain what they are supposed to contain and that there are no files that should not be there. Users of this database will not lose time on looking for the data they need but will only be concerned in writing scripts to perform the desired analysis. One of these scripts may, for example, go through several hundreds of files contained in a directory called `./hourly_samples`, opening each of them and performing some calculations. At the end of it's execution, the script may store several files to a directory called `./monthly_means`. This is a typical use of the data in the ICTP Earth System Physics group. Having several PhD students, post-docs, researchers and staff, continuously retrieving and writing large amounts of data to the single NFS server is inefficient and work-flows slow-down, as hard disk seek times are long[2] and the bandwidth of the network is limited. It is clear that the

1 In this case, to find the data, one should go through all the database, opening every file to look what is inside. This kind of task is exactly the one described in point i)

2 HDD seek times are measured in `ms`. This is comparable to the time it takes to `ping` 64 bytes to a remote server in a neighbouring city!

bottleneck of the work-flow is not the CPU or the Memory but the reading of the data from disk and the network streaming of it.

Let's take this idealisation further and suppose that there is not a bunch of users competing for disk seeks and bandwidth. Let's suppose that we are the only users and that the File System is not NFS but our local file system. Is the storage model efficient now?

In the remaining of the chapter I will address issues I) and II) above, arguing that complications in the immediate future will inevitably arise if the data-storage model is not rethought and improved.

EXAMPLE OF AN IDEAL DATA FLOW

Suppose we want to analyse the temperature field of a certain region of spacetime. Such a temperature field may have been generated by some simulation with such a resolution that provides N_i temperature samples along coordinate i ($i \in \{t, x, y, z\} = \{$ time, longitude, latitude and height $\}$), with a grand total of $N_T = N_t \times N_x \times N_y \times N_z$ temperature samples[3]. This potentially huge 4D array will likely be chunked into sub-arrays and stored in separate NetCDF files to be handled. Each NetCDF file will thus contain the temperature field of a subinterval of time and a subregion of space, with $n_T = n_t \times n_x \times n_y \times n_z$ temperature values (The total number of files at the end of this chunking is $N_{\text{files}} = N_T/n_T$).

Getting concrete and easy for now, we focus on a simple case: $N_t = N_x = N_y = N_z = 4$, with chunks of size $n_t = n_x = n_y = n_z = 2$ stored to separate NetCDF files. We thus have $N_T = 256$, $n_T = 16$ and $N_{\text{files}} = 16$. A representation of the data is shown in Figure 2. If we want to take the time-and-height average of this temperature field and store the results to a single NetCDF file, we can do it easily with available tools like Iris[4]. A simple script that does exactly this is shown in Snippet 1.

For this low amount of files and resolution, our simple script works perfectly. Figure 3 shows the results of applying the script of Snippet 1 to two datasets: The first dataset is the one we have just described and the second dataset is almost like the first, with the only difference being the number of temperature samples stored in each file, $n_i = 64$, so that $N_i = 128$, for $i \in \{t, x, y, z\}$. Note that doubling the resolution requires handling $2^4 = 16$ times more data. In this case, we have increased the resolution by a factor of $64/2 = 32$, leading to a total number of temperature samples $N_T \simeq 0.26 \times 10^9$.

3 Such an array will weight $N_T \times 8$ bytes if it is composed of `doubles`.
4 Iris: A python library for Meteorology and Climatology, http://scitools.org.uk/iris/

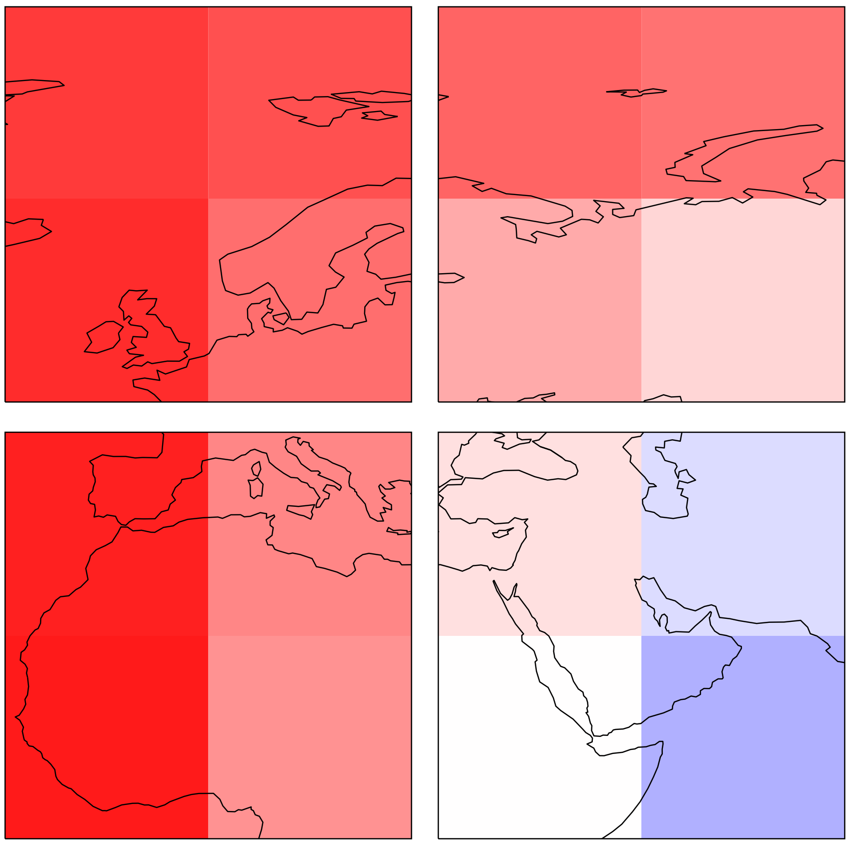

Figure 2: A temperature field is sampled. The number of samples taken for each coordinate is $N_t = N_x = N_y = N_z = 4$. The temperature field is chunked and stored in different NetCDF files so that, in each file, a chunk with $n_t = n_x = n_y = n_z = 2$ samples is found (i.e. the chunk number of samples is $\prod n_i = 16$). The complete temperature field is thus divided in $N_{\text{files}} = \prod N_i / \prod n_i = 16$ chunks. The Figure shows 4 of these chunks, corresponding to longitude-latitude slices at constant time and height. Needless to say, the colors that in a real case scenario would correspond to the values of the temperature, in this case are physically meaningless. What concerns us is that each value weights 8 bytes, i.e, is a `double`.

Snippet 1: Python script that calculates time and height averages of a temperature field distributed over several NetCDF files.

```python
#!/usr/bin/env/ python
import iris

#Load data:
cubes = iris.load("./low_res_tas/*.nc")

#Concatenate the data into a single cube:
concat = cubes.concatenate_cube()

#Take the average:
avg = concat.collapsed(['time','height'], iris.analysis.MEAN)

#Save to a new file:
iris.save(avg,"./low_res_tas/time_height_avg.nc")
```

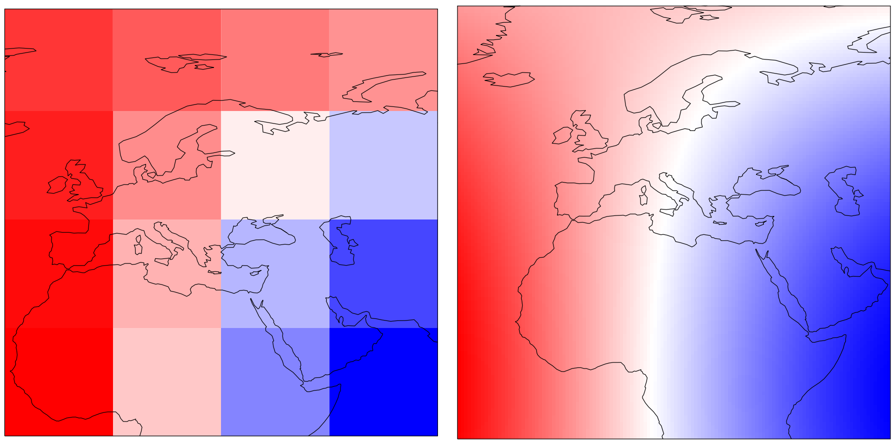
Figure 3: Time-height average of the temperature field described in the text. Both averages were computed taking the data from only 16 NetCDF files, the left subplot corresponds to low resolution data, $n_t = n_x = n_y = n_z = 2$ ($N_T = 256$), while the right subplot corresponds to higher resolution data, $n_t = n_x = n_y = n_z = 64$ ($N_T = 268435456$), though still low compared to the resolution of a production run.

This approach breaks down immediately for resolutions $n_i = 128$, even when $n_i = 2$.

MAP-REDUCE APPROACH

As we have seen, the simple script of Snippet 1 for taking temperature averages breaks as soon as $n_i \geqslant 2^7$. The memory is not enough to handle all the data and a `MemoryError` like the one shown in Snippet 2 is thrown. Having 256 total samples per dimension is not really a respectable resolution for any code claiming to tackle *state-of-the-art* problems, so a workaround must be found to handle the analysis of higher resolutions.

Snippet 2: Memory error thrown when the number of samples per file per coordinate $n_i \geqslant 128$

```
MemoryError: Failed to create the cube's data as there was
    not enough memory available.
The array shape would have been (256, 256, 256, 256) and the
    data type float64.
Consider freeing up variables or indexing the cube before
    getting its data.
```

An immediate workaround which does not require much effort is implementing a *map-reduce* approach: we first calculate partial means by loading one file at a time (we *map*) and, once we have the partial means of all files, we calculate from them the total average temperature[5] (we *reduce*). The modified version of our original script (Snippet 1) is shown in Snippet 3

Snippet 3: Map-Reduce approach as a workaround to the `MemoryError` thrown when handling high resolution files.

```python
import iris
import os

#Directory where our high resolution data is:
dir_in  = "./high_res_tas"

##   Map phase: get partial averages   ##

cubelist = iris.cube.CubeList([])
for root,dirs,files in os.walk(dir_in):
    for f in files:
        file_path = "/".join([root,f])

        #load the data
        cube  = iris.load_cube(file_path)

        #calculate the partial average
        partial_avg = cube.collapsed(['time','height'], iris.
            analysis.MEAN)

        #Store the calculated partial average
        cubelist.append(partial_avg)

##  Reduce phase: Compute global average.  ##

#Get a single cube from the partial averages:
concat_avg = cubelist.concatenate().merge_cube()

#Take the average of the new cube just obtained:
avg = concat_avg.collapsed(['time','height'], iris.analysis.
    MEAN)

#Save to a new file:
iris.save(avg,"./high_res_tas/time_height_avg.nc")
```

This approach, in principle, is able to handle any amount of data. However, keeping $n_i = 128$, it takes already an hour to analyse 81 files (i.e. close to 1 minute per file). As soon as the size of the data that needs to be analysed increases to sizes of more than 1TB, we are faced with 24-hour long analyses. Clearly this is not acceptable.

The conclusion we obtain from the discussion made in this chapter is that, in order to keep up with the pace at which simulations gener-

5 Be careful: this approach must weight properly each partial mean, e.g, $\mathrm{mean}(1,2,3,4,5) \neq \mathrm{mean}(1,2) + \mathrm{mean}(3,4,5)$ but $\mathrm{mean}(1,2,3,4,5) = (2/5) \times \mathrm{mean}(1,2) + (3/5) \times \mathrm{mean}(3,4,5)$. Fortunately, `Iris` takes care of this.

ate data, a completely new approach must be provided. Switching to the new approach should, however, represent a minimum effort for the final users.

3 HADOOP, HDFS AND MAPREDUCE

Even if nowadays we have hard drives with capacities as large as 1-terabyte, transfer speeds of around 100 MB/s conspire against efficient analysis of large datasets –With these numbers, it would take more than two and a half hours only to read 1 terabyte of data. A straightforward solution (at least in theory) is to read from multiple disks at once. Reading from 100 drives, each holding one hundredth of the data and working in parallel, would take less than two minutes. Of course, most analysis tasks need to combine the data in some way, so data read from one disk may need to be combined with data from any of the other 99 disks, and correctly allowing data to be combined from multiple sources is notoriously challenging. Another challenge is met when noting that, as soon as more pieces of hardware are introduced, failure of at least one of them is more probable.

Hadoop Distributed File System (HDFS) and Hadoop MapReduce programming model provide solutions for each of these challenges:

· HDFS prevents data loss through replication: redundant copies of the data are stored in different drives so that, in event of failure, there is another copy available.

· Mapreduce abstracts the problem from disk reads and writes, casting it into a computation over sets of keys and values. Using this approach, any computation is composed of two parts –the map (reading from multiple disks at once) and the reduce (the "mixing" of the data read).

Hadoop provides a reliable, scalable platform for storage and analysis, with the additional advantage of being affordable because i) it runs on commodity hardware, ii) it is open source, and iii) one can set up a hadoop-cluster in the blink of an eye and pay only by the time one actually uses it (e.g. the service provided by Amazon).

A TOY 4-NODE HADOOP CLUSTER `@argo.ictp.it`

My first approach to the problem of setting up a hadoop cluster occurred on the 15th of June, 2016. I received root access to 4 dual-socket, 8-core, 12-GB nodes from ICTP: {hdp1, hdp2, hdp3, hdp4}. Each node with a topology like the one shown in Figure 4 and 200GB of disk space.

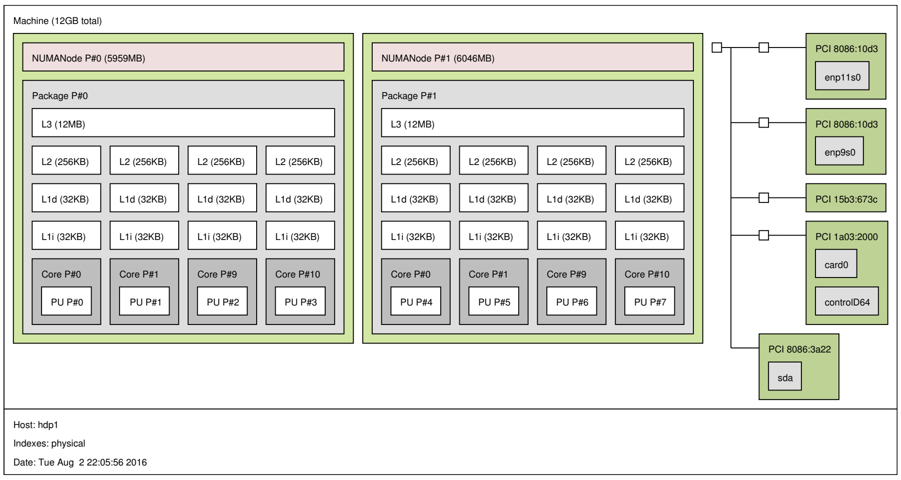

Figure 4: One of the four nodes which would host the hadoop cluster and its distributed filesystem, HDFS. All four had the same topology. The diagram was generated by running `lstopo` of the package `hwloc` on one of them, `hdp1`.

The machines were running `Ubuntu 14.04`[1]. with Oracle's `Java version "1.7.0_80"`. I downloaded the `hadoop-2.7.3` binaries and unpacked them in each machine. The rest of the installation was straightforward following hadoop's home page guide. In a nutshell, what I did to have running my first word-count, the "Hello World" of MapReduce jobs, can be summarized in 7 steps that I will explain in the following sections:

· Define some environment variables in ~/`.bashrc` (or equivalent environment script).

· Specify some configuration properties and environment variables in the hadoop configuration files.

· Format the name node.

· Start the HDFS[2], YARN[3] and JobHistory daemons.

· Create the user directory inside the distributed filesystem.

· *put* some files in it.

· Launch MapReduce word-count example, included in the binaries.

The cluster that we get at the end looks like the one shown in Figure 5 and is composed of

1 WARNING: mapreduce jobs running on `Ubuntu 16.04` machines will crash. I learned that the hard way. See the operating systems and Java versions recommended by hadoop before setting up your own cluster
2 Hadoop Distributed File System
3 Yet Another Resource Manager

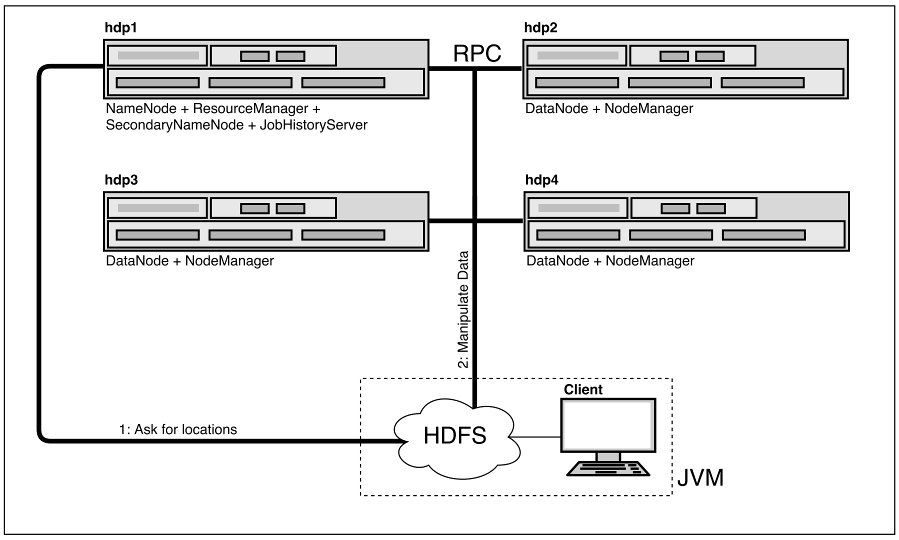

Figure 5: Anatomy of the toy 4-node Hadoop Cluster `@argo.ictp.it`. One master node runs the `NameNode`, `SecondaryNamenode`, `ResourceManager` and `JobHistoryServer` daemons. The rest of the nodes, the slaves, run the `DataNode` and `NodeManager` daemons. All the nodes and the client machines communicate using remote procedure calls (RPC). Chunks of data distributed among the slaves appear to the client machine, which communicates with the cluster using a Java Virtual Machine (JVM), as single files stored in the HDFS.

- A NAMENODE - The namenode store the physical location of each block of data in the hadoop cluster. Client machines requesting blocks of data must first contact the namenode for getting its address. It is the sole repository of the metadata and the file-to-block mapping.
- A SECONDARY NAMENODE - Despite its name, the secondary namenode does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large[4].
- A RESOURCE MANAGER - The resource manager is responsible for allocating containers. Jobs to be run in the hadoop cluster must contact the resource manager asking for resources (memory + virtual cores).
- A JOB HISTORY SERVER - The job history server will keep logs of all completed jobs. One can contact it via a web-interface to get useful information.
- SEVERAL DATANODES - The datanodes store and retrieve blocks when they are told to (by clients or the namenode), and they

4 BEWARE: The namenode is still a single point of failure (SPOF).

report back to the namenode periodically with lists of blocks that they are storing.

- SEVERAL NODE MANAGERS - Node managers send heartbeat requests to the resource manager that carry information about the node's running containers and the resources available for new containers. Each heartbeat is a potential scheduling opportunity for new containers.

From these description we can readily note some patterns: The namenode and the datanodes are responsible of *storing* information, while the resource manager and the node managers are responsible of allocating resources for jobs to *analyse* the data. So, requests for data by a client machine must be firstly addressed to the namenode, and ultimately served by the datanodes. Similarly, requests for resources must be firstly addressed to the resource manager and ultimately served by the node managers. This is the way in which hadoop logic is built, daemons are separated: the namenode and the datanodes have HDFS daemons running in them –`NameNode` and `DataNode`, while the resource manager and the node managers must run YARN daemons –`ResourceManager` and `NodeManager`. Nodes running `DataNode` + `NodeManager` daemons are called the worker nodes, or *slaves*. Nodes running `NameNode` or *ResourceManager* daemons are the *masters* (See table 1).

	HDFS	YARN
master	NameNode	ResourceManager
slave	DataNode	NodeManager

Table 1: Relations between the different components of a hadoop cluster.

HADOOP'S ENVIRONMENT

Hadoop's HDFS, YARN and MapReduce offer command-line interfaces to perform tasks such as *put*ting and *get*ting files in HDFS, launching, killing and monitoring jobs. For the client machine to run these command-line tools, it is necessary to set some environment variables. In particular, `$HADOOP_HOME` should point to the directory where the hadoop binaries were unpacked (in my particular case, `$HADOOP_HOME=/usr/local/hadoop/`). Then we can make available the hadoop executables in our environment:

```
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

As it is likely that we, as administrators, will be using often the hadoop configuration directory to set and change the parameters, it is a good idea to set the environment variable `$HADOOP_CONF_DIR` to point to where the configuration files reside. Additionally, setting it will make Hadoop, Yarn and MapReduce daemons to read their configuration from that location. Otherwise they will try to read it from `etc/hadoop` subdirectory under `$HADOOP_HOME`.

Hadoop configuration is passed to all tools composing the framework mainly via xml files, and, for setting a hadoop cluster with minimal effort, it is at least necessary to edit `core-site.xml`, `hdfs-site.xml`, `yarn-site.xml` and `mapred-site.xml`. There we can specify which nodes will be running each daemon needed to the hadoop cluster. Ideally, in large clusters, the `NameNode`, the `SecondaryNameNode`, the `ResourceManager` and the `JobHistoryServer` should run in dedicated nodes, however, for the toy cluster we are setting up, the best thing to do is to group all the non-*slave* daemons into a single *master* node to free the maximum amount of resources for the worker nodes where the *slave* daemons will run[5].

STORING DATA INSIDE HDFS

After setting up all the required parameters, we can start the hadoop cluster by running the following commands in a terminal[6]:

```
$ ssh namenode $HADOOP_HOME/bin/hdfs namenode -format
$ ssh namenode $HADOOP_HOME/sbin/start-dfs.sh
$ ssh resourcemanager $HADOOP_HOME/sbin/start-yarn.sh
$ ssh jobhistoryserver $HADOOP_HOME/sbin/start-history-server.sh
$ hdfs dfs -mkdir /user
$ hdfs dfs -mkdir /user/hadoop_user
```

The first command will format the namenode (it will set the arena for storing all the metadata in the node we designed as namenode in the configuration files), the second command will start the HDFS daemons (`NameNode`, `SecondaryNameNode` and `DataNodes`), the third command will start the YARN daemons (`ResourceManager` and `NodeManagers`), and the fourth command will start the only MapReduce daemon, the `JobHistoryServer`. The last two command will create the default directory for the user `hadoop_user`. Communication among the nodes and client machines occurs using remote procedure calls (RPC). In general, for a client machine to communicate with the cluster, it is necessary to start a Java Virtual Machine (JVM).

After successfully reaching this point, we are able to store our files on HDFS, e.g,

5 After trying alternative configurations this was indeed the best one

6 These scripts rely on SSH to perform cluster-wide operations. In order to work, SSH needs to be set up to allow passwordless login for the user running the commands from machines in the cluster. The simplest way to achieve this is to generate a public/private key pair and place it in an NFS location that is shared across the cluster.

```
$ hdfs dfs -Ddfs.blocksize=256m -Ddfs.replication=3  -put
    some_file some_existing_hdfs_directory
```

Running this command in a terminal of a machine with access to the hadoop cluster starts a Java Virtual Machine which establishes a communication with the HDFS daemons, instructing them to take `some_file` in the local machine, store it in the hadoop cluster in chunks of size 256 MB, and replicate it 3 times. Data is thus distributed in this way among slaves. However, to the client machine, it will appear as single files stored in the HDFS:

```
$ hdfs dfs -ls
$ some_existing_hdfs_directory/some_file
```

We can also `put` entire directories at once:

```
$ hdfs dfs -Ddfs.blocksize=256m -Ddfs.replication=3  -put
    some_directory some/location/in/hdfs/
```

HDFS command line interface (CLI) has lots of handy commands available to manage and explore stored files, e.g. `cat`, `grep`, `ls`, `mkdir`, `rm`, ... When in doubt, ask Google.

A TYPICAL MAPREDUCE JOB

Having set up the Hadoop cluster, and having stored some data in it, what I needed to do for approaching my first MapReduce job is what every programmer needs to do when learning a new programming language: Write a *"Hello World"* program, which, in MapReduce means running a word-count job. To my surprise, Hadoop installation not only came with a word-count program, but also with a random text generator which itself is a MapReduce job (with only the map part) which writes 10 GB of random text to every node. So, running a first MapReduce job was a piece of cake. Though useless from the pedagogical point of view, it allowed me to assert the correct installation of the hadoop cluster and its suitability to run my own MapReduce jobs in the future[7]. The command for launching a typical MapReduce job looks like this:

```
$ hadoop jar [generic options] MyMapReduce.jar job_main_class [
    arguments relevant to the job]
```

That command will start on the client a Java Virtual Machine (JVM) to communicate with the `ResourceManager` and submit the application. The `ResourceManager` will then ask an available node's `NodeManager` to launch a `MRAppMaster`. At this point, the `MRAppMaster` will continue to orchestrate the job, spawning `MapTasks` and `ReduceTask` among the

7 My first jobs were dying randomly with no clear reason. After around one month I got to the root of the problem: Ubuntu 16.04. See footnote 1

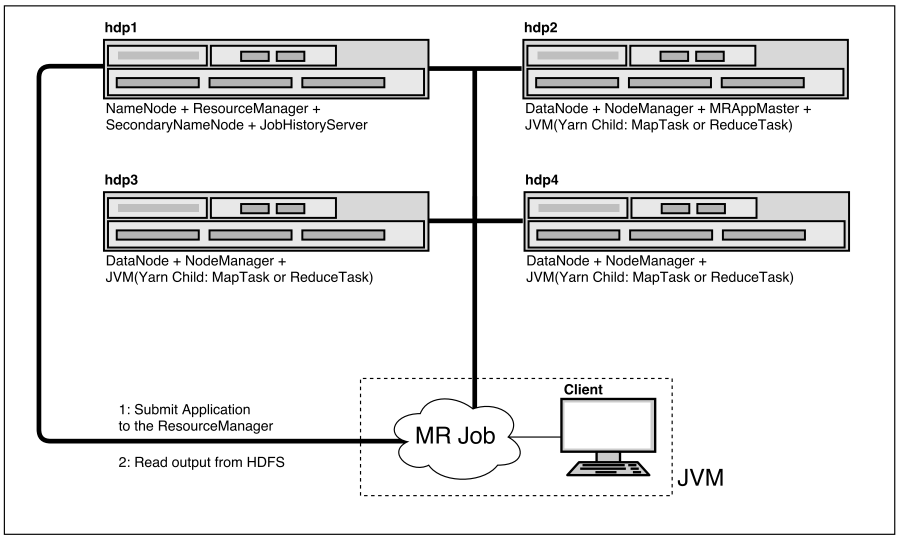
hdp1
NameNode + ResourceManager +
SecondaryNameNode + JobHistoryServer
hdp2
DataNode + NodeManager + MRAppMaster +
JVM(Yarn Child: MapTask or ReduceTask)
hdp3
DataNode + NodeManager +
JVM(Yarn Child: MapTask or ReduceTask)
hdp4
DataNode + NodeManager +
JVM(Yarn Child: MapTask or ReduceTask)
1: Submit Application
to the ResourceManager
2: Read output from HDFS
MR Job
Client
JVM

Figure 6: Anatomy of a Map Reduce Job running in the toy 4-node Hadoop Cluster `@argo.ictp.it`. A Java Virtual Machine (JVM) is started to communicate with the `ResourceManager`, which will ask an available node's `NodeManager` to launch a `MRAppMaster`. The `MRAppMaster` then asks the `NodeManager` of the slave nodes to allocate containers for starting a `YarnChild` JVM inside which a `MapTask` or a `ReduceTask` will be ran.

slave nodes. `NodeManagers` are asked by the `MRAppMaster` to create containers if they have available resources. These containers must provide enough resources to host a JVM inside which a `YarnChild` will run a `MapTask` or a `ReduceTask`. See Figure 6. If a `Task` uses more resources than those allocated, the `ResourceManager` will kill it. If a `Task` fails more than a certain number of times (defaults to 4), the complete job is declared as Failed.

Map Tasks will always be placed as close as possible to the node where data resides to avoid wasting bandwidth by moving data around the cluster: i) If the Map Task is runs in a node which hosts the input split of the data on which it is working, the task is called Data-Local. ii) In the case that all nodes having the particular input split needed by a certain Map Task are busy, the Map Task is launched in another node of the same rack. Finally, iii) if no nodes are available in the whole rack where the data input split is located, the Map Task is launched in another rack. This makes MapReduce a very powerful programming model.

The output (stderr) that the user sees when submitting a MapReduce job looks like the one shown in Snippet 10 is if the run is successful.

Snippet 4: Typical output of a successful MapReduce job

```
16/12/01 12:55:42 INFO client.RMProxy: Connecting to ResourceManager at resourcemanager/140.105.32.201:8032
16/12/01 12:55:43 INFO input.FileInputFormat: Total input paths to process : 16
16/12/01 12:55:43 INFO mapreduce.JobSubmitter: number of splits:16
16/12/01 12:55:43 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1479296429870_0032
16/12/01 12:55:43 INFO impl.YarnClientImpl: Submitted application application_1479296429870_0032
16/12/01 12:55:43 INFO mapreduce.Job: The url to track the job: http://hdp1:9046/proxy/application_1479296429870_0032/
16/12/01 12:55:43 INFO mapreduce.Job: Running job: job_1479296429870_0032
16/12/01 12:55:50 INFO mapreduce.Job: Job job_1479296429870_0032 running in uber mode : false
16/12/01 12:55:50 INFO mapreduce.Job:  map 0% reduce 0%
[...]
16/12/01 12:56:24 INFO mapreduce.Job:  map 100% reduce 100%
16/12/01 12:56:25 INFO mapreduce.Job: Job job_1479296429870_0032 completed successfully
16/12/01 12:56:25 INFO mapreduce.Job: Counters: 49
        File System Counters
        [...]
        Job Counters
        [...]
        Map-Reduce Framework
        [...]
        Shuffle Errors
        [...]
        File Input Format Counters
        [...]
        File Output Format Counters
        [...]
```

4 THE DRAWBACKS AND ADVANTAGES OF SOME AVAILABLE HADOOP UTILITIES

To perform even a simple analysis over a *Big* amount of *Data* in a reasonable amount of time, it is desirable to have some framework able to distribute the reads and writes –the *bottleneck of Big-Data analysis*, in the way MPI[1] distributes the floating point operations –The *bottleneck of High Performance Computing*. A framework that does exactly this is Hadoop, together with its distributed file system (HDFS) and its implementation of the Map-Reduce programming model (MapReduce[2]) which brings the computation to the places where data is stored. Unfortunately, Hadoop has some drawbacks that prevent NetCDF data storage and analysis out-of-the-box:

i) Hadoop is designed and highly optimized to read and write text data. Weather and Climate scientists work with binary data in the form of NetCDF files.

ii) HDFS stores the data by splitting every file in blocks and distributing those blocks among different nodes. NetCDF files cannot be split, since they are binary and splitting would represent data corruption.

iii) MapReduce programs must be written in Java. Weather and Climate scientists codes are written in every programming language available since the dawn of computer age but Java.

iv) MapReduce is a batch processing system, not suitable for interactive analysis. Weather and Climate scientists need interaction with their data.

In the rest of this chapter I will describe some of the tools already available in the market, extracting the salient aspects of them that could be used to tackle the problem at hand.

Hadoop Streaming

Hadoop-Streaming is a utility that comes with Hadoop and allows the user to launch MapReduce jobs without writing a single line of Java code. This sounds promising for solving issue number iii) at the

1 Message Passing Interface
2 The introduction of YARN, a cluster resource management system, in Hadoop 2 has allowed any distributed program (not just MapReduce) to run on data stored in a Hadoop cluster.

beginning of the chapter. The vanilla command to start a Hadoop-Streaming job is shown in Snippet 5:

Snippet 5: Vanilla command for starting a Hadoop-Streaming Job

```
$ $HADOOP_HOME/bin/hadoop jar hadoop-streaming.jar \
  -files mapper_script,reducer_script,
  -input input1,input2,input3... \
  -output hdfs_output_dir \
  -mapper mapper_script \
  -reducer reducer_script
```

Hadoop-Streaming will take each of the files `input1,input2,input3...`, and will feed the contents of them to the `mapper_script` via standard input. If any of the inputs happens to be a directory, Hadoop-Streaming will also take as inputs all the files inside the directory. `mapper_script` must be an executable that takes the standard input and translates it into a set of `(key,value)` pairs written to standard output. Hadoop-Streaming will then take this `(key,value)` pairs, shuffle-sort them by key, and feed them to the `reducer_script` via standard input as new `(key,value)` pairs. After processing the set of key value pairs, the `reducer_script` must write to standard output a new set of `(key,value)` pairs, which Hadoop-Streaming will save to `hdfs_output_dir` as the result of the MapReduce job.

The "Hello World!" program of any Map-Reduce implementation is the word-count. Snippets 6 and 7 show an example input and the output expected from wuch a job, and a couple of python scripts to achieve that, `wc_mapper.py` and `wc_reducer.py`, are shown in Snippets 8 and 9. Indeed, there is no need to write a single line of Java code! [3]

Hadoop-Streaming works by linking the chain "`input | mapper | reducer | output`" via streams of bytes (preferably text). Clearly, this is a huge drawback to solve our problem. However, there is something appealing in the logic behind it, if not in the specifics of the implementation:

Provide a Java utility that makes data stored in HDFS available to executable scripts written in arbitrary languages. Provide also a set of rules that the executable scripts have to follow in their implementation in order to build the chain "`input | mapper | reducer | output`".

3 For this simple example, the exact same thing can be achieved by using standard shell tools instead of Hadoop-Streaming, e.g,

`$ echo -e "one two three four \n two three four \n three four \n four" | ./mapper.py | sort | ./reducer.py`.

In general, it is a good practice to test locally the mapper and reducer scripts before launching a Hadoop-Streaming job. The beauty of this one-liner resides in the simplicity in which it casts the logic of Hadoop-Streming.

Snippet 6: Example input for a Hadoop-Streming word-count job

```
one two three four
two three four
three four
four
```

Snippet 7: Example output of a Hadoop-Streming word-count job

```
four 4
one 1
three 3
two 2
```

Snippet 8: A Python mapper script to implement a word-count MapReduce job using Hadoop-Streaming

```
#!/usr/bin/env python
#wc_mapper.py

import sys

# Read each line from stdin
for line in sys.stdin:
    # Get the words in each line
    words = line.split()

    # Write the (key,value) pairs to stdout: (word,1)
    for word in words:

        # By default, Hadoop-Streaming takes as key
        # anything before the first tab character.
        print '{0}\t{1}'.format(word.lower(), 1)
```

Snippet 9: A Python reducer script to implement a word-count MapReduce job using Hadoop-Streaming

```
#!/usr/bin/env python
#wc_reducer.py

import sys
curr_word = None
curr_count = 0

# Process each key-value pair from the mapper
for line in sys.stdin:
    # Get the key and value from the current line
    word, count = line.split('\t')
    count = int(count)

    # increment counter for the current word...
    if word == curr_word:
        curr_count += count

    # or print to stdout if the word changed...
    else:
        #Remeber that key values are separated by tabs:
        if curr_word:
            print '{0}\t{1}'.format(curr_word, curr_count)

        curr_word = word
        curr_count = count

# Output the count for the last word
if curr_word == word:
    print '{0}\t{1}'.format(curr_word, curr_count)
```

Luigi and mrjob

Luigi[4] and **mrjob**[5] are Python MapReduce libraries, created by Yelp[6] and Spotify[7] respectively, that wrap Hadoop streaming, allowing multi-step MapReduce jobs to be written in pure Python. One of the strengths of these libraries is that they offer the possibility of writing MapReduce jobs in a single class, avoiding the nuisance of having to write separate programs for the mapper and the reducer. Another advantage these libraries offer is that applications can be executed and tested without having Hadoop installed, enabling development and testing before deploying to a Hadoop cluster. To use these libraries, all dependencies (e.g.`numpy` or `iris`) must be either available on the task nodes, or uploaded to the cluster when jobs are submitted.

Although mrjob scope –simplicity, differs from Luigi's –ability to chain long pipelines of jobs, the philosophy behind them is the same:

Wrap an already available Java utility that allows for more general classes of mappers and reducers, and allow the user to deploy MapReduce jobs to a Hadoop cluster, not only without writing a single line of Java code, but also without explicitly launching the underlying Java utility.

To show the simplicity of a code written for this kind of libraries, I reproduce on Snippet 10 a word-count example using **mrjob**.

These kind of libraries seem to be just what we need to solve the problem of running MapReduce jobs over large datasets of NetCDF files: They are completely written in other language than Java and, the fact that they are written in python, promises some possibility of tweaking them to be interactive, thus solving problems iii) and iv) at the beginning of the chapter.

These libraries, however, are still designed to operate over text files divided in blocks inside HDFS, eaving issues i) and ii) still waiting for a solution. For the moment, let's keep the philosophy of **Luigi** and **mrjob** and move on to tackle the remaining issues.

4 `https://luigi.readthedocs.io`
5 `https://mrjob.readthedocs.io`
6 `https://www.yelp.com`
7 `https://www.spotify.com`

Snippet 10: Python implementation of a word-count MapReduce job using mrjob library

```python
#!/usr/bin/env python
#wc_mrjob.py
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordCount(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
        ]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordCount.run()
```

Running this example in a hadoop cluster is as simple as typing on a client machine:

```
$ python wc_mrjob.py -r hadoop hdfs://path/to/input/input.txt
```

RootOnHadoop

A simple approach to the problem of file-splitting by HDFS, was taken in 2012 by a then-student of the *Università degli Studi di Udine*, Stefano Russo. His goal was to efficiently analyse High Energy Physics data generated by the Atlas experiment in the Large Hadron Collider (LHC); the problem to overcome being that the data analysis was (and already is) analysed using ROOT, a software with a very large community of users that is written in C++, python and R. Because of this issues, similar to the ones we have at hand now, developing a more efficient strategy to analyse the data using Hadoop + MapReduce was not straightforward.

The solution developed by Russo, a Java package called `RootOnHadoop` , allowed him and his colleagues to run ROOT without any modification and store the data to HDFS in its original format without splitting it. This kind of approach is what we need to solve points i) and ii) at the beginning of the chapter. However, directly using `RootOnHadoop` presents some drawbacks:

- Jobs need to be submitted in a classic, batch-fashioned manner.
- The overhead it adds for our particular problem is huge.

Both of these issues conspire against developing a interactive and efficient application to launch MapReduce jobs targeted to analyse NetCDF files. Snippet 11 shows how to sumbit a `RootOnHadoop` job (batch fashioned manner; very similar to launching Hadoop-Streaming).

Snippet 11: RootOnHadoop runs in batch fashioned manner; the command to launch a RootOnHadoop job is similar to that for launching a Hadoop-Streaming job

```
$ $HADOOP_HOME/bin/hadoop jar RootOnHadoop.jar RootOnHadoop \
  -map mapper_script \
  -reduce reducer_script \
  -in input_file_or_dir \
  -out output_hdfs_dir \
```

To test the applicability and performance of `RootOnHadoop` as a solution, I created a simple job that, for each NetCDF file found on a database containing ~ 1300 files, creates some lines (~ 10) of text describing what is inside (i.e, a catalogue entry). The results of this test are shown in Figure 7, which not only compares the timings when executing the job serially and when using `RootOnHadoop`, but also show the timing corresponding to pure overhead (i.e. when the mapper script and the reducer script do not even read the NetCDF files).

There may be several reasons for the tremendous overhead shown in Figure 7, but the one that is immediately evident by looking at the code, is that there are several system calls to handle files inside the HDFS. Making such kind of calls is not only expensive because calling the kernel, but also because almost all of the calls require to start a new Java Virtual Machine. In the end, any kind of analysis done to a single file will add more than a couple of seconds to the total overhead[8] (See snippet 12).

8 Notice that numbers add up in a rough calculation: Making around 3 2-second system calls for each one of the ~ 1300 files, divided in 3 nodes, gives just around 40 minutes, comparable to the obtained overhead.

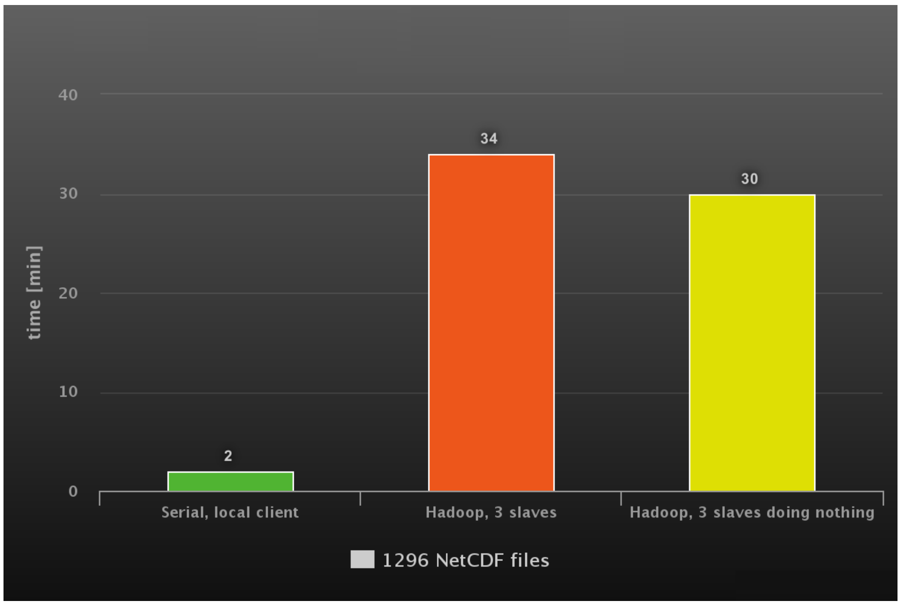

Figure 7: Catalogue creation test timings for `RootOnHadoop` (middle column): for each NetCDF file found on a database containing ~ 1300 files, a `mapper_script` creates a catalogue entry (~ 10 lines of text). `RootOnHadoop` takes around 17 times more than running the script locally in a serial way (left bar). The rightmost column corresponds to pure overhead –the `mapper_script` for this overhead test was not even opening the NetCDF files.

Snippet 12: RootOnHadoop relies on several system calls to HDFS to operate on a single file. The creation of JVMs by these calls, when dealing with thousands of files this can result in tremendous overhead

```
$ time for i in {1..10}; do hdfs dfs -ls > /dev/null; done
real    0m26.112s
user    0m37.352s
sys     0m1.452s
```

The strength of `RootOnHadoop`, however, lies not on its implementation, but in the use it makes of the Hadoop + MapReduce logic to operate on any kind of data with any kind of executable:

`MapReduce` *ensures that every* `MapTask` *will most likely run on machines where data resides. Use this fact to bypass the HDFS framework and run any desired executable* **on local data,** *just as it normally would do.*

Almost evil.

5 BIG NETCDF DATA ANALYSED... FASTER: PIPISTRELLO & TINA

Of all of the tools available on the market, not a single one is able to distribute and analyse NetCDF files efficiently. Summarizing, a framework is needed that i) preserves the data structure and ii) is simple to implement and understand, using the already available state-of-the-art analysis tools. What is needed is not a completely different tool for the users to learn; what is needed is a framework that allows for the data analysis to safely take advantage of bleeding-edge technologies without users noticing a change in their work-flows.

Some attempts that have been done to efficiently analyse scientific data and their drawbacks are the following:

1. NCO[12] performs all computations on a single node and data is read serially from the file system.
2. The SWAMP project ([18]) parallelizes the execution of NCO queries but requires computations to be expressed using procedural scripts.
3. ([20])'s method enable processing NetCDF data with MapReduce, but this solution requires to transform data into a text-based format.
4. SciHadoop allows to efficiently execute queries as map/reduce programs defined over the logical data model but it has to take care of multiple and complicated repartitions of the data and, to my knowledge, does not interface with current available tools.

[References taken from: "SciHadoop: Array-based Query Processing in Hadoop"]

In the rest of the chapter I will introduce my two cents: `Pipistrello` and `Tina`. A Java package and a Python library which, combined, allow users to deploy MapReduce jobs over Scientific datasets in a transparent way. Three of the Four Pillars on which these pieces of software stand take inspiration from available tools described in Chapter 4. The fourth one instead is a completely original contribution of this Thesis.

1. *Provide a Java utility that makes data stored in HDFS available to executable scripts written in arbitrary languages. Provide also a set of rules that the executable scripts have to follow in their implementation in order to build the chain* "`input | mapper | reducer | output`".

2. *Wrap an already available Java utility (1) that allows for more general classes of mappers and reducers, and allow the user to deploy MapReduce jobs to a Hadoop cluster, not only without writing a single line of Java code, but also without explicitly launching the underlying Java utility.*

3. `MapReduce` *ensures that every* `MapTask` *will most likely run on machines where data resides. Use this fact to bypass the HDFS framework and run any desired executable* **on local data,** *just as it normally would do.*

4. *Provide an interactive mode and let the users query for data properties (such as locations in the directory tree, bounds or coordinates) without knowing a-priori the structure of the directory tree. Let them also refine previous queries and test the validity of a MapReduce job before deploying it.*

PIPISTRELLO

`Pipistrello`[1] is a Java utility very similar in philosophy to Hadoop-Streaming that allows users to run MapReduce jobs over any kind of binary files. Just as Hadoop-Streaming, Pipistrello requires mappers and reducers to follow a set of conventions in order for the chain "`input | mapper | reducer | output`" to be correctly linked. Fist of all, let's see the typical command to invoke Pipistrello (much like Hadoop-Streaming; Snippet 13).

Snippet 13: Invoking Pipistrello feels like invoking Hadoop-Streaming.

```
$ $HADOOP_HOME/bin/hadoop jar Pipistrello.jar \
  -${generic_hadoop_mapreduce_options} \
  -files file1,file2,file3... \
  -input input1,input2,input3... \
  -output hdfs_output_dir \
  -mapper mapper_script \
  -reducer reducer_script
```

The set of conventions that `mapper_script` and `reducer_script` must follow are not exactly the same:

1 https://pipistrello.readthedocs.io

`i | m` First of all, the inputs must have been stored in HDFS using **only one block per file**[2]. The `mapper_script` should expect to receive two arguments: The **first argument** will represent an hdfs fiename (corresponding to the file on which the mapper will work). This first argument, for all practical purposes is just a dummy name inside the `mapper_script`, though it is very convenient to know it. The **second argument** will be an actual file path, local to the machine in which the map task is running. This is the file containing the data needed by the `mapper_script` to carry out it's job[3].

`m | r` The `mapper_script` should write to the local directory where it is running an output file of any kind, writing its file-name to `stdout` without blank spaces or line breaks. The `reducer_script` must be ready to receive only one argument: the name of a file containing lines of text.

`r | o` From the file provided as an argument, the `reducer_script` must read lines of text, each of which will be a file path local to where the reduce task is running. These are the partial outputs generated by the mappers, which the reducer will combine in some way to i) generate an output file of any kind to the local directory and ii) write its file-name to `stdout` with no spaces or line breaks. After the job has finished, the user will be able to find this file inside the HDFS `output` directory.

Pipistrello under the hood

Pipistrello works in a very similar way to Hadoop-Streaming. From the command line, it is given the `mapper_script` and the `reducer_script`, which it packs and distributes among the nodes hosting the data and map tasks will run.

The first difference between Hadoop-Streaming and Pipistrello is that, while Hadoop streaming implements a couple of Java `classes` (a `RecordReader` and an `InputFormat`) to read the records in a specific format and stream them to the `mapper_script` and the `reducer_script`, Pipistrello's only purpose of implementing a `RecordReader` and an `InputFormat` is to play by the rules of the Hadoop-MapReduce framework[4].

All the magic of Pipistrello lies inside its implementation of the `class Mapper` and its `class HdfsToLocalTranslator`. The `class Mapper` uses the `class HdfsToLocalTranslator` to get the local path of the

2 e.g. `hdfs dfs -Ddfs.blocksize=10g -put some_file some_hdfs_dir`

3 known bug here: this argument is fed to the mapper with no extension! Trying to open a file without an extension could be confusing for some scripts.

4 Although these classes are not needed by the current design, a more clever way of designing Pipistrello could use these classes to get the information that is needed during the map phase

data corresponding to the input split the `MapTask` needs to process. The `class HdfsToLocalTranslator` is in charge of asking directly to the `NameNode` –without starting any JVM or doing a system call– the name of the local file corresponding to a given hdfs file. Once the name is "leaked" inside the `Mapper` class, it can be given to the `mapper_script` as an argument[5]. After the `mapper_script` is done, the `Mapper class` will upload its output (a file) to the HDFS. It will also read the `stdout` of the `mapper_script` (a filename) and set it as the `value` of the `(key,value )` pair that the MapReduce framework will send to the reducer[6].

MapReduce will start a `ReduceTask` in any available node with enough resources to run it. At this point, `(key,value)` pairs are going to be fed to the `Reduce class`. The `Reducer class` will open a text file `files_to_reduce.txt` and write all the `values` (filenames), one for each line. At the same time, the `Reducer class` will download these files from the HDFS to make them available locally[7]. Finally, the `Reducer Class` will launch the `reducer_script`, giving it as an argument the file `files_to_reduce.txt` and its output (a file) is uploaded to HDFS. All in all, Pipistrello is composed of 7 Java classes:

· `HdfsToLocalTranslator.java`

· `PipistrelloInputFormat.java`

· `PipistrelloJob.java`

· `PipistrelloMapper.java`

· `PipistrelloRecordReader.java`

· `PipistrelloReducer.java`

· `PipistrelloMapReduce.java`

The core ones have already been described and the rest of them are wrappers that parse command line arguments and launch the Pipistrello MapReduce job. Pillars 1 and 3 are in place.

5 If the data is not available in the local node, the map task will fail. The MapReduce framework will relaunch the task in a new node, highly likely where data is. This is faster than moving the data –a potentially big file– around the network (or isn't it?)

6 What is the `key` set to? Currently it is set to a generic value, always the same. There's ground to play in this area for further improvements.

7 We are making the assumption that these files, the outcomes of the map phase, are orders of magnitude smaller than the input files. This operation should not represent any substantial cost.

TINA

Tina is a very light Python library that sits on top of:

- Iris to handle NetCDF data.
- Snakebite to efficiently communicate with the Hadoop Filesystem.
- Pipistrello to launch MapReduce jobs over a Scientific dataset.

Loading Iris cubes

Loading from the dataset Iris CubeLists containing all the cubes corresponding to a certain region of spacetime, is as simple as it is in Iris, though the call is not the same, as shown in Snippet 14:

Snippet 14: Loading cubes from HDFS using Tina is as easy as loading cubes from NFS using Iris

```python
#Load all cubes from files inside hdfs:///user/username/high_res_sim with
#temperature higher than 20
#time between 24 and 48
#longitude lower than 40
cube_list = tina.load("high_res_sim","20<air_temperature",["24<time<48","10<latitude<20","longitude<40"])
```

At this point, having an CubeList loaded, we are able to run any analysis using all the well known and well tested functionality offered by Iris.

MapReduce Big Data using Tina records

However, if we are using Tina, it is highly likely that we are actually dealing with a huge amount of data and a direct analysis is not possible. In that case, we need to i) either take the local mapreduce approach (slower) or ii) distribute the computation in the Hadoop cluster (faster). To this end, we need to:

i. Load the Tina records we want to analyse,
ii. Choose the mapper and the reducer,
iii. Call the mapreduce function.

Just like the basic objects in Iris are the cubes, the basic objects in Tina are the records. These objects, upon request, are read on the fly from HDFS and contain information about all the available cubes. Each record will contain the information of only one cube:

- The file containing the cube[8].
- The minimum and maximum values of the data in the cube.
- The bounds on each of the coordinates of the cube.

The power of Tina resides on the simplicity of its logic[9]. With Tina, launching the time-height-average MapReduce job (**??**), is as easy as writing a script like the one shown in Snippet 15. The interaction with the HDFS to load the Iris cubes or the Tina records, the call to Pipistrello to launch MapReduce jobs and the getting the result in the form of an iris cube, all of that happens under the hood.

Snippet 15: Time-height average MapReduce job using Tina

```
#!/usr/bin/env python
import tina
import iris

#A selection of mappers and reducers comes packed with Tina.
#One can also use custom mappers and reducers.
mapper="./time_height_mean_mapper.py"
reducer="./time_height_mean_reducer.py"

#Get all records in the WHOLE dataset with
#latitudes between 20 and 30
#(handling units not implemented yet).
record_list = tina.filter_records( "10<temperature<20",
                                   ["30<latitude<40",
                                    "10<longitude<30"] )

#Launches a MapReduce job to the Hadoop cluster.
#Returns an Iris cube.
avg = tina.mapreduce(reclist,mapper,reducer)

#Save the result locally using Iris.
iris.save(avg,"/home/output/filename.nc")

#Upload to the hdfs using Tina.
tina.put("/home/output/filename.nc","some_hdfs_dir")

#Simple as saying "We're done"
```

Please take this with a grain of salt: even if Tina comes with a handful of mappers and reducers, for specific cases, the mapper and the reducer still need to be written by the user (More on this on Section 6).

8 the file size: to be implemented
9 Certainly not in its robustness... yet.

Local testing with Tina fake cubes

It may be the case that the selected data is not suitable to be MapReduced, or that there is a bug in either the mapper script or the reducer script. Launching a MapReduce job in these cases could potentially make us lose hours waiting for a job that was ill from the very beginning. Fortunately, to prevent these cases, Tina comes with a handy functionality: from each Tina record a small fake cube can be created on the fly. This will allow us to do a preliminary local analysis of our data in the blink of an eye. If this fake analysis succeeds, it is likely –though not certain, that a MapReduce job will succeed. Conversely, if doing a fake local analysis on a certain set of records does not give the results expected, it is certain that a MapReduce job on the same set of records will fail.

Snippet 16: Use fake cubes to quickly simulate an analysis before attempting a MapReduce job

```
#!/usr/bin/env python
import tina
import iris

#Load Tina records:
reclist = tina.filter_records("high_res_data","30<temperature
    ")

#Create a cube list of fake cubes:
cl = iris.cube.CubeList([])
        for r in reclist:
                cl.append(r.fake_cube())

## Do something with the cube list ##
#       ...do what mapper_script would do
#       ...do what the reducer_script would do
#       result = ...
## Is the result as expected? In that case:

result = tina.mapred(reclist,mapper_script,reducer_script)
```

Tina is interactive

Tina, like most python packages, is suitable for working interactively. Be it the command line or a `jupyter notebook`, Tina works there. In particular, taking a look at the contents of a directory inside HDFS, which might be handy when working interactively, is as simple as typing `tina.ls()`. Snippet 17 shows an example interactive workflow using Tina, and Figure 8 shows a result of a local test job using Tina fake cubes together with the true results after running the complete

MapReduce job. As shown in the Figure, fake cubes are not accurate nor smooth, they are intended only as an aid to know whether the dataset is suitable to be MapReduced in the Hadoop cluster by a particular couple of mapper and the reducer scripts.

Snippet 17: Interactive workflow using Tina

```
>>> import tina
>>> import iris
>>> tina.ls()
/user/hadoop_user/high_res_sim
>>>tina.ls("high_res_sim")
/user/hadoop_user/high_res_sim/file_0001.nc
/user/hadoop_user/high_res_sim/file_0001.tina
[...]
/user/hadoop_user/high_res_sim/file_00081.nc
/user/hadoop_user/high_res_sim/file_00081.tina
>>> reclist = tina.filter_records("high_res_sim")
>>> mapper = "/path/to/mapred_scripts/mapper.py"
>>> reducer = "/path/to/mapred_scripts/mapper.py"
>>> fcubelist = iris.cube.CubeList([r.fake_cube() for r in reclist ])
>>> print(fcubelist)
[...]
78: air_temperature / (celsius)         (time: 2; longitude: 2; latitude: 2; height: 2)
79: air_temperature / (celsius)         (time: 2; longitude: 2; latitude: 2; height: 2)
80: air_temperature / (celsius)         (time: 2; longitude: 2; latitude: 2; height: 2)
>>> fake_result = fcubelist.concatenate_cube().collapsed(["time","height"],iris.analysis.MEAN)
>>> print(fake_result)
air_temperature / (celsius)         (longitude: 6; latitude: 6)
     Dimension coordinates:
          longitude                           x            -
          latitude                            -            x
     Scalar coordinates:
          height: 50.1302083332 metres, bound=(0.130208333333, 100.130208333) metres
          time: 2000-01-02 08:05:00, bound=(2000-01-01 00:05:00, 2000-01-03 16:05:00)
     Cell methods:
          mean: time, height
>>> true_result = tina.mapreduce(reclist,"./tina/mean_mapper.py","./tina/mean_reducer.py")
Mapreducing 81 files, this make take a while...

16/12/06 12:12:46 INFO client.RMProxy: Connecting to ResourceManager at resourcemanager
      /140.105.32.201:8032
16/12/06 12:12:47 INFO input.FileInputFormat: Total input paths to process : 81
16/12/06 12:12:47 INFO mapreduce.JobSubmitter: number of splits:81
16/12/06 12:12:47 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1479296429870_0090
16/12/06 12:12:47 INFO impl.YarnClientImpl: Submitted application application_1479296429870_0090
16/12/06 12:12:47 INFO mapreduce.Job: The url to track the job: http://hdp1:9046/proxy/
      application_1479296429870_0090/
16/12/06 12:12:47 INFO mapreduce.Job: Running job: job_1479296429870_0090
16/12/06 12:12:56 INFO mapreduce.Job: Job job_1479296429870_0090 running in uber mode : false
16/12/06 12:12:56 INFO mapreduce.Job:  map 0% reduce 0%
16/12/06 12:36:37 INFO mapreduce.Job:  map 100% reduce 100%
16/12/06 12:36:40 INFO mapreduce.Job: Job job_1479296429870_0090 completed successfully
16/12/06 12:36:40 INFO mapreduce.Job: Counters: 50
        File System Counters
[...]
        Job Counters
[...]
        Map-Reduce Framework
[...]
        Shuffle Errors
[...]
        File Input Format Counters
[...]
        File Output Format Counters
[...]
16/12/06 12:36:40 INFO streamlike.MyJob: Output directory: _PIPISTRELLO_20161206121243
16/12/06 12:36:40 INFO streamlike.MyJob: Time ellapsed: 1434 seconds.

INFO: Local output directory /tmp/_PIPISTRELLO_//20161206100119 created.
INFO: Loaded cubes from:
INFO:   /tmp/_PIPISTRELLO_/20161206100119/_PIPISTRELLO_20161206121243/reduce_0.nc
INFO: /user/hadoop_user/_PIPISTRELLO_20161206121243 deleted.
>>>
```

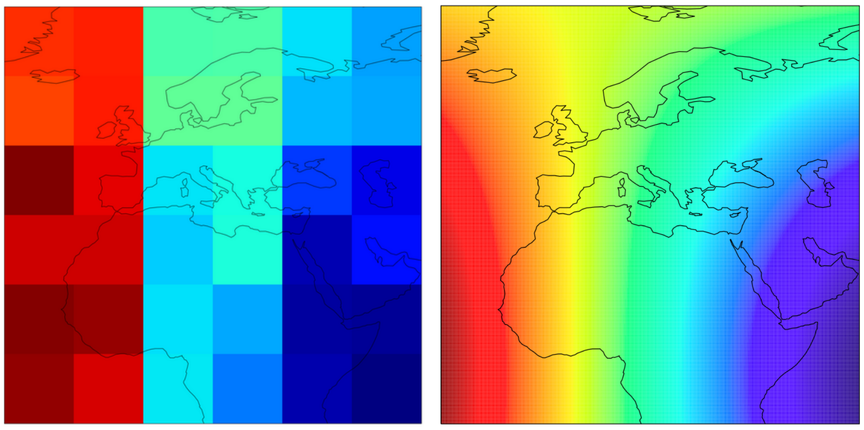

Figure 8: The left panel shows results of local analyses using Tina fake cubes. Results obtained from Tina fake cubes are not smooth nor accurate, as fake cubes are only meant to be used for testing the suitability of analyses to be deployed as MapReduce jobs to the Hadoop cluster. The right panel shows the results obtained after completing the full MapReduce job of the real data in the Hadoop cluster. The creation and analysis of fake cubes takes a couple of seconds, while, for this particular case, the MapReduce of the real data took almost half an hour.

Last but not least: `tina.put()`

For all the features already described, it is necessary to load every single file to the Hadoop cluster using the function `tina.put()`, be it in a script or in interactive mode:

```
tina.put("./some_local_file.nc","some_hdfs_dir")
```

PIPISTRELLO AND TINA ENABLE A FASTER DATA ANALYSIS

Tests were performed by generating 4-D arrays of artificial data (doubles representing temperature values) with different sizes. The arrays were then distributed among several NetCDF files in chunks of 128^4 (which amounts to file sizes of $\sim$ 2 GB). Due to lack of resources[10], only two different array sizes were tested: $(128 \times 2)^4$ distributed in 16 files and $(128 \times 3)^4$ distributed in 81 files.

The analysis made to test the performance of the Hadoop Cluster, running `Tina` and `Pipistrello` MapReduce jobs, consisted on computing the time-height-average temperature of each of the whole arrays. Figure 9 shows timings for the analysis of these arrays. By comparing these timings it is evident that, for the array sizes tested and the chunking of the data, `Pipistrelo` and `Tina` do an excellent job. The MapReduce analysis done for the smaller array takes $1/3$ of the time

10 The storage space per node was only 200GB, which amounts to an effective size of 200GB for a Hadoop cluster of 3 nodes and a replication factor of 3

the best local approach (a local Map-Reduce approach, as the big amount of data does not fit in Memory), while, for the larger array, it takes only 1/2.5 of the time.

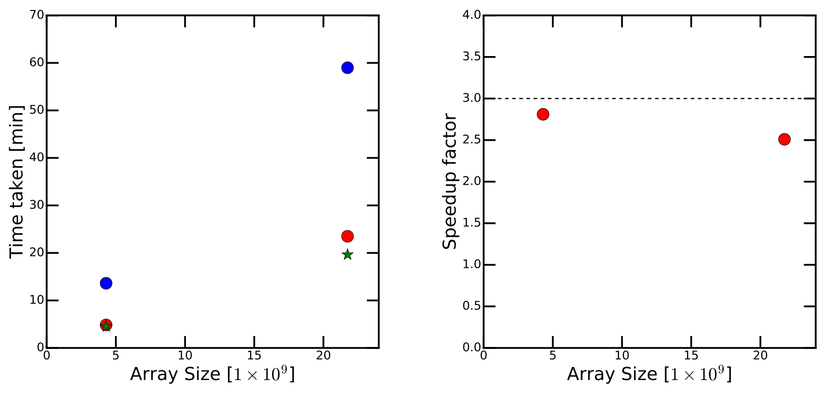

Figure 9: The left panel shows the time it takes to analyse each dataset by using the best local approach (blue circles) and by deploying a MapReduce job in the Hadoop Cluster using `Tina` and `Pipistrello` (red dots). The time it would take supposing ideal speedup is also shown (green stars). The right panel shows the speedup reached for each case (red points) and the ideal speedup (3, for our 3-node cluster) is marked by the dotted line.

6 CONCLUSIONS, WARNINGS AND FUTURE PERSPECTIVES

Needless to say, `Tina` and `Pipistrello` are experimental software. They were born out of the need for handling and analysing the data generated by the Earth System Physics group at ICTP. As experimental software, they are completely new tools not tested by anyone but the developer (me in this case). They may contain bugs and may lack desired functionality. In particular, the fact that the user has to write the mapper and the reducer as separate scripts is something that annoys me. A little step is still need to be taken to provide the same kind of functionality that `Luigi` and `mrjob` provide. Like this one, and `Tina`'s inability to handle units, there are other imperfections of which I am aware of. And I am sure that there are many more of which I am not aware of. The bright side, however, is that we now have traced the path and paved the road to future development.

In this thesis, software has been developed that not only **enables** to store data to a **Hadoop Cluster** without modifying a single bit of it, but also to **MapReduce** it with minimum overhead. All of this is achieved requiring an almost non-existent effort for the users. With larger clusters, larger storage space and willing people to carry out the "dirty work" of curating the data, **analysing** hundreds of Terabytes of **Scientific Big Data** will be **faster** than saying: "Are we done yet?".

BIBLIOGRAPHY

Giorgi, F., Jones, C., & Asrar, G. R. 2009, World Meteorological Organization (WMO) Bulletin, 58, 175

Gutowski Jr., W. J., Giorgi, F., Timbal, B., et al. 2016, Geoscientific Model Development, 9, 4087

Lambert, S. J., & Boer, G. J. 2001, Climate Dynamics, 17, 83

Meehl, G. A., Moss, R., Taylor, K. E., & Eyring, V. 2014, Eos Trans. AGU, 95, 77–84

Overpeck, J. T., Meehl, G. A., Bony, S., & Easterling, D. R. 2011, Science, 331, 700

Taylor, K. E., Stouffer, R. J., & Meehl, G. A. 2012, Bulletin of the American Meteorological Society, 93, 485

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LYX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`