UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JIMMY KRAIMER MARTÍN VALVERDE
SÁNCHEZ

Distributed Data Analysis over Meteorological Datasets using the Actor Model

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Nicolas Maillard

Porto Alegre
August 2017

CIP — CATALOGING-IN-PUBLICATION

Valverde Sánchez, Jimmy Kraimer Martín

Distributed Data Analysis over Meteorological Datasets using the Actor Model / Jimmy Kraimer Martín Valverde Sánchez. – Porto Alegre: PPGC da UFRGS, 2017.

104 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Nicolas Maillard.

1. Actor model. 2. Big data. 3. Manager-Worker. 4. GRIB. 5. Akka. I. Maillard, Nicolas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

ACKNOWLEDGEMENTS

First of all, I would like to thank God, for leading me through all the difficulties and uncertain roads.

I want to express sincere my deepest sense of gratitude to my advisor Prof. Dr. Nicolas Maillard, and Prof. Dr. Philippe Navaux for the opportunity of being part of the GPPD research group. Also, I thank my colleagues and friends at the GPPD research group.

Special thanks are given to my parents and grandparents. Words cannot express my immense love and gratitude to them, for their support and company even in the distance.

Also, I would like to thank to Allynne for joining me during all this time, really many thanks. And Josefa who was as a second mother as for as long I was in Brazil. Maryorit and Sierra for their unconditional help and friendship.

Finally, I would like to thank everyone who directly or indirectly had an influence in the completion of this stage of my life.

ABSTRACT

Because of the continuous and overwhelming growth of scientific data in the last few years, data-intensive analysis on this vast amount of scientific data is very important to extract valuable scientific information. The GRIB (GRIdded Binary) scientific data format is widely used within the meteorological community and is used to store historical meteorological data and weather forecast simulation results. However, current libraries to process the GRIB files do not perform the computation in a distributed environment. This situation limits the analytical capabilities of scientists who need to perform analysis on large data sets in order to obtain information in the shortest time possible using of all available resources. In this context, this work presents an alternative to data processing in the GRIB format using the well-know Manager-Worker pattern, which was implemented with the Actor model provided by the Akka toolkit. We also compare our proposal with other mechanisms, such as the round-robin, random and an adaptive load balancing, as well as with one of the main frameworks currently existing for big data processing, Apache Spark. The methodology used considers several factors to evaluate the processing of the GRIB files. The experiments were conducted on a cluster in Microsoft Azure platform. The results show that our proposal scales well as the number of worker nodes increases. Our work reached a better performance in relation to the other mechanisms used for the comparison particularly when eight worker virtual machines were used. Thus, our proposal upon using metadata achieved a gain of 53.88%, 62.42%, 62.97%, 61.92%, 62.44% and 59.36% in relation to the mechanisms: round-robin, random, an adaptive load balancing that used CPU, JVM Heap and mix metrics, and the Apache Spark respectively, in a scenario where a search criteria is applied to select 2 of 27 total parameters found in the dataset used in the experiments.

Keywords: Actor model. Big data. Manager-Worker. GRIB. Akka.

RESUMO

Devido ao contínuo crescimento dos dados científicos nos últimos anos, a análise intensiva de dados nessas quantidades massivas de dados é muito importante para extrair informações valiosas. Por outro lado, o formato de dados científicos GRIB (GRIdded Binary) é amplamente utilizado na comunidade meteorológica para armazenar histórico de dados e previsões meteorológicas. No entanto, as ferramentas atuais disponíveis e métodos para processar arquivos neste formato não realizam o processamento em um ambiente distribuído. Essa situação limita as capacidades de análise dos cientistas que precisam realizar uma avaliação sobre grandes conjuntos de dados com o objetivo de obter informação no menor tempo possível fazendo uso de todos os recursos disponíveis. Neste contexto, este trabalho apresenta uma alternativa ao processamento de dados no formato GRIB usando o padrão Manager-Worker implementado com o modelo de atores fornecido pelo Akka toolkit. Realizamos também uma comparação da nossa proposta com outros mecanismos, como o round-robin, random, balanceamento de carga adaptativo, bem como com um dos principais frameworks para o processamento de grandes quantidades de dados tal como o Apache Spark. A metodologia utilizada considera vários fatores para avaliar o processamento dos arquivos GRIB. Os experimentos foram conduzidos em um cluster na plataforma Microsoft Azure. Os resultados mostram que nossa proposta escala bem à medida que o número de nós aumenta. Assim, nossa proposta atingiu um melhor desempenho em relação aos outros mecanismos utilizados para a comparação, particularmente quando foram utilizadas oito máquinas virtuais para executar as tarefas. Nosso trabalho com o uso de metadados alcançou um ganho de 53.88%, 62.42%, 62.97%, 61.92%, 62.44% e 59.36% em relação aos mecanismos round-robin, random, balanceamento de carga adaptativo que usou métricas CPU, JVM Heap e um combinado de métricas, e o Apache Spark, respectivamente, em um cenário onde um critério de busca é aplicado para selecionar 2 dos 27 parâmetros totais encontrados no conjunto de dados utilizado nos experimentos.

Palavras-chave: Modelo de atores, Big data, Manager-Worker, GRIB, Akka.

LIST OF ABBREVIATIONS AND ACRONYMS

aGRIB	actor GRIdded Binary
API	Application Programming Interface
ARGO	Array for Real-time Geostrophic Oceanographic
BRAMS	Brazilian Regional Atmospheric Modeling System
CDL	Common Data Language
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
CPU	Central Processing Unit
GRIB	GRIdded Binary
HDF	Hierarchical Data Format
GPFS	General Parallel File System
GTG	Grab 'em, Tag 'em, Graph 'em
HDFS	Hadoop Distributed File System
JAR	Java ARchive
JSON	JavaScript Object Notation
JM	JobManager
JVM	Java Virtual Machine
JW	JobWorker
MERRA	Modern Era Retrospective-Analysis for Research and Applications
NetCDF	Network Common Data Form
PFT	Palomar Transient Factory
PVFS	Parallel Virtual File System
RDD	Resilient Distributed Dataset
URI	Uniform Resource Identifier
WMO	World Meteorological Organization
WN	Worker Node
YARN	Yet Another Resource Negotiator

LIST OF FIGURES

Figure 2.1 GRIB2 Message Structure 22
Figure 2.2 An abstract representation of an actor 26
Figure 2.3 Actions an actor may perform 26
Figure 2.4 Akka Actor Hierarchy 29
Figure 2.5 Actor Model in Akka 29
Figure 2.6 Actor sample: Main program code 30
Figure 2.7 Actor sample: JobManager actor code 31
Figure 2.8 Actor sample: JobWorker actor code 32
Figure 2.9 Actor sample: JobManager and JobWorker messages 32
Figure 2.10 Actor sample: Terminal output 33
Figure 2.11 Manager-Worker model of communication 36
Figure 2.12 Spark components on a distributed environment 38

Figure 4.1 Initial architecture: Using the Manager-Worker pattern 45
Figure 4.2 HDFS Architecture 46
Figure 4.3 Design: Using the Manager-Worker pattern 46
Figure 4.4 Design: Using different cluster-aware routers 48
Figure 4.5 Initial results for file and message-based strategy using our proposal 50
Figure 4.6 Architecture: Using metadata stored in MongoDB 51
Figure 4.7 Sample: JSON output document from MongoDB 52

Figure 5.1 Cluster in Microsoft Azure 53
Figure 5.2 First scenario: Best and worst configuration for each mechanism 58
Figure 5.3 Time spent by the JobWorkers using the `4WN-1JW` configuration 60
Figure 5.4 Time spent by the JobWorkers using the `1WN-4JW` configuration 61
Figure 5.5 Time spent by the JobWorkers using 3 different configurations and 4 VMs 62
Figure 5.6 First scenario: CPU utilization for the aGrib using 8 worker VMs 65
Figure 5.7 First scenario: CPU utilization for the round-robin using 8 worker VMs 66
Figure 5.8 First scenario: Memory utilization for the aGrib using 8 worker VMs 67
Figure 5.9 First scenario: Best and worst speedup for each mechanism 68
Figure 5.10 Second scenario: Best and worst configuration for each mechanism 70
Figure 5.11 Second scenario: Best cases using 1 worker virtual machine 71
Figure 5.12 Second scenario: Best cases using 2 worker virtual machines 72
Figure 5.13 Second scenario: Best cases using 8 worker virtual machines. 74
Figure 5.14 Second scenario: CPU utilization for the aGrib using 8 worker VMs 75
Figure 5.15 Second scenario: CPU utilization for the round-robin using 8 worker VMs76
Figure 5.16 Second scenario: Best and worst speedup for each mechanism 77
Figure 5.17 First scenario: Speedup achieved with and without metadata 78
Figure 5.18 Speedup achieved for aGrib and Apache Spark 80

Figure B.1 First scenario: Execution time using 1 worker machine 96
Figure B.2 First scenario: Execution time using 2 worker machines 97
Figure B.3 First scenario: Execution time using 4 worker machines 98
Figure B.4 First scenario: Execution time using 8 worker machines 99
Figure B.5 Second scenario: Execution time using 1 worker machine 100
Figure B.6 Second scenario: Execution time using 2 worker machines 101
Figure B.7 Second scenario: Execution time using 4 worker machines 102
Figure B.8 Second scenario: Execution time using 8 worker machines 103

Figure B.9 First scenario: Total execution times of our proposal by using metadata . 104

LIST OF TABLES

Table 2.1 GRIB2 Code Table 0.0 - Discipline 22
Table 2.2 GRIB2 Code Table 3.1 - Grid Definition Template Number 23
Table 2.3 GRIB2 Code Table 5.0 - Grid Definition Template Number 24

Table 3.1 Related work summary 41

Table 5.1 Overview of the software components 54
Table 5.2 Configurations: Total number of JobWorkers per set of VMs 55
Table 5.3 Summary: round-robin with the `4WN-1JW-22` configuration 60
Table 5.4 Summary: aGrib with the `4WN-1JW-22` configuration 60
Table 5.5 Time spent by using the `2WN-8JW-88` configuration 63
Table 5.6 First scenario: Summary of the results by using metadata. 79
Table 5.7 First scenario: Summary of the results for aGrib and Apache Spark 81
Table 5.8 Second scenario: Summary of the results for aGrib and Apache Spark 81

CONTENTS

1 INTRODUCTION 19
1.1 Document Structure 20
2 BACKGROUND 21
2.1 GRIB 21
2.1.1 GRIB Structure 22
2.2 Actor Model 24
2.3 Akka Toolkit 27
2.3.1 The actor system 28
2.3.2 Actor Messaging 29
2.3.3 Creating Actors 30
2.3.4 Sending and receiving messages 32
2.3.5 Modifying the actor behavior 33
2.3.6 Clustering 34
2.3.7 Routers 34
2.4 Manager-Worker Pattern 36
2.5 Apache Spark 37
3 RELATED WORK 39
4 AGRIB: PROCESSING GRIB DATA IN DISTRIBUTED ENVIRONMENTS WITH AKKA 43
4.1 Scala client module 43
4.2 Using the Manager-Worker pattern 44
4.3 Using cluster-aware routers 48
4.4 Using Metadata 49
4.5 Using Apache Spark 51
5 EXPERIMENTAL EVALUATION 53
5.1 Experimental Environment 53
5.2 Dataset Description 54
5.3 Experimental Methodology 54
5.3.1 Configurations 55
5.3.2 Grouping messages 56
5.3.3 Mechanisms 56
5.3.4 Varying dataset 56
5.4 Baseline 57
5.5 Experimental Results 57
5.5.1 First scenario: Using two parameters 58
5.5.2 Second scenario: Using all parameters 70
5.5.3 Using Metadata 78
5.5.4 Using Apache Spark 80
6 CONCLUSION 83
REFERENCES 85
APPENDIX A — GRIB SAMPLE 91
APPENDIX B — PERFORMANCE EVALUTION 95

1 INTRODUCTION

Currently we are living the Big Data era (CHEN; ZHANG, 2014), (DOBRE; XHAFA, 2014), which is a result of the continuous and overwhelming increase of data generated from different sources, sensors, simulations, and logs, among many others. Such is the case of the scientific community, that generates huge amounts of data day-to-day. This data growth leads to the need for a lot of resources and a better use of these resources to store and process the gathered data in order to perform simulations and data-intensive analysis.

In turn, it is necessary to have applications that can take advantage of the information contained in these large data sets. In the scientific community, it is done with the goal to understand phenomena in meteorology, seismology, and health, among other areas.

In this sense, the sequential programming model is not enough to cope with the requirements of these application types. So the parallel and distributed programming model becomes more valuable in order to process large amounts of data within a timely manner (CHEN; ZHANG, 2014), (KAMBATLA et al., 2014).

There are many efforts for processing and performing data analysis on the datasets generated by the scientific community in distributed and parallel environments. These efforts are focused mainly on scientific data formats such as NetCDF (LI et al., 2003), (ZHAO et al., 2010), (BUCK et al., 2011), (DUFFY et al., 2012), (WANG; JIANG; AGRAWAL, 2012), (PALAMUTTAM et al., 2015) and HDF5 (WANG; JIANG; AGRAWAL, 2012), (BLANAS et al., 2014).

However, there are other scientific data formats, such as the GRIB format (WMO, 2003) which is used widely within the meteorological community for exchanging and storing historical and forecast weather data. Nevertheless this format does not support random access (FORTNER, 1995), while both NetCDF and HDF5 are random access format. Thereby, the GRIB format is not easily accessible (CANDANEDO; PARADIS; STYLIANOU, 2013) and lacks research in methodology and tools for processing a large numbers of GRIB files in a reasonable amount of time.

In the Center for Weather Forecasting and Climate Research (CPTEC[1]) different models of weather forecasts are performed periodically, covering South America and adjacent oceans. The information resulting from these forecasts and the initial model condition are provided twice a day. All of this scientific data contains the state of the atmosphere and is in the GRIB format.

Thus, in the pre-processing phase of the BRAMS (Brazilian Regional Atmospheric Modeling System) execution, which is a numerical weather prediction model used to simulate atmospheric conditions, the GRIB files have to be converted into an intermediary file that contains the input data for the model (MARQUES, 2009), however this pre-

[1]http://www.cptec.inpe.br/

processing is performed using a sequential manner (CARREÑO, 2015), taking a considerable time of the overall process, which depends mainly the time period and the data packing method to be processed.

The current libraries to process the GRIB files do not perform the computation in a distributed environment and the data analysis performed is taking into account only one individual GRIB file. In this context, we propose an alternative way to process and perform data analysis on large datasets in the GRIB format over a distributed environment by implementing the Manager-Worker pattern using the Akka toolkit, which is an implementation of the Actor model (HEWITT; BISHOP; STEIGER, 1973). It is based on single kind of object called Actor. Despite the fact that the idea of this model originated in 1973, the Actor model has been gaining a significant importance during the last years in the concurrent and distributed programming (TASHAROFI; DINGES; JOHNSON, 2013), since there is no shared state between actors and the communication is exclusively through asynchronous message passing and therefore conceptually there are no problems such as critical sections, deadlocks, race conditions (Lightbend Inc, 2016). Thus, for example, two of the main Big Data processing frameworks, such as Apache Spark (ZAHARIA et al., 2010) and Apache Flink (ALEXANDROV et al., 2014) use the Akka toolkit for its distributed communication (ROSÀ; CHEN; BINDER, 2016).

1.1 Document Structure

The remaining text of this document is organized as follows:

- Chapter 2 introduces the basics concepts for a better understanding of this work, such as the GRIB scientific data format, the Actor model, the Akka toolkit – one of the main implementations of the Actor model, the Manager-Worker model of communication.
- Chapter 3 presents relevant related work.
- Chapter 4 details our proposal for processing large amounts of GRIB files in a distributed environment. This chapter also presents other approaches used to compare our proposal.
- Chapter 5 introduces the environment used to perform the experiments and details the dataset utilized in the experiments, presents the evaluation methodology and finally, the experimental results.
- Chapter 6 presents the conclusions of this work, highlights the main contributions and outlines ideas for future work.

2 BACKGROUND

This chapter presents concepts for a better understanding of the following chapters of this dissertation. Thus, Section 2.1 introduces a global overview of the GRIB messages and their structure. Section 2.2 details the ideas behind the Actor Model. Section 2.3 discusses some basis concepts and common terminology related to the Akka Toolkit, one of the main implementations of the Actor Model. Section 2.4 describes the Manager-Worker model of communication, utilized on our proposal. Section 2.5 covers a brief introduction to the Apache Spark.

2.1 GRIB

GRIB (GRIdded Binary) is a scientific data format, created by the World Meteorological Organization (WMO)[1]. It was designed to be a general purpose, self-describing, bit-oriented data exchange format, machine-independent transmission and storage of meteorological data (WMO, 2003), (FORTNER, 1995). This scientific data format is widely utilized by the meteorological community, it is commonly used to store weather and weather forecast data (MARKIEWICZ et al., 2013), generally the data stored in this format comes from satellite images or simulations of the weather (FORTNER, 1995).

It is worth highlighting that this format is not easily accessible (CANDANEDO; PARADIS; STYLIANOU, 2013), since this format does not support a random access (FORTNER, 1995), unlike other scientific data formats such as NetCDF (REW; DAVIS, 1990) or HDF (The HFD5 Format, 2002). Nevertheless, the GRIB format as well as other scientific data formats are still used in their respective communities because since their initial adoption there has been investment in time and resources which makes it difficult to fully migrate to another data format (BUCK, 2014).

There are three versions of this format, versions 0, 1 and 2. However, the version 0 was deprecated and the version 2 was proposed to enable the representation of new parameters, which for the version 1 was no longer possible (WMO, 2003). The definition of these parameters are found as part of the standard and not as part of the GRIB message, so that, the GRIB message stores a reference to information defined on an external table. For example, Table 2.1 presents the Product disciplines available for a GRIB message, where *code* is the value found in the GRIB message, and *description* is the value specified by the standard.

Thus, the files in the GRIB format are denominated as GRIB files, in turn, these files are composed of GRIB messages, and each one of these messages may contain submessages allowing the creation of hierarchical and ordered structures (MARKIEWICZ et al., 2013).

[1] https://www.wmo.int/

Table 2.1: GRIB2 Code Table 0.0 - Discipline (NCEP WMO GRIB2 Documentation, 2016).

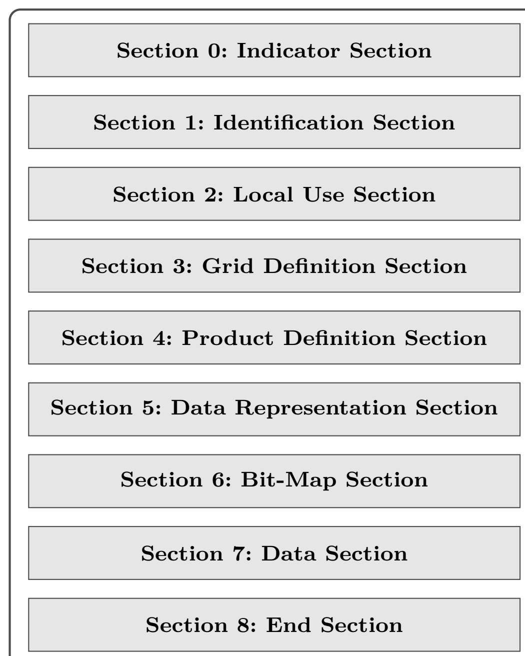
Code	Description
0	Meteorological products
1	Hydrological products
2	Land surface products
3	Space products
4-9	Reserved
10	Oceanographic products
11-191	Reserved
192-254	Reserved for local use
255	Missing

2.1.1 GRIB Structure

The GRIB message provides information necessary to decode the message without knowing the grid size and structure in advance (HIBBARD et al., 2002).

According to the standard defined by the WMO, each GRIB message contains logical divisions called *sections*. Concretely for the GRIB edition 2 there are nine types of sections, numbered 0 through 8. Figure 2.1 depicts the structure for a GRIB2 message and Appendix A shows the sample of a GRIB2 message.

Figure 2.1: GRIB2 Message Structure.



The **Indicator Section**: Marks the beginning of the message with the text “GRIB” and contains the Product Discipline (e.g., Discipline 0 corresponds to Meteorological products) of the data in the message – as detailed in Table 2.1 –, GRIB Edition number (e.g., 2) and the total length of the message in bytes.

The rest of the sections except the last, contain the length of each section in bytes at the byte location from 1 to 4 and the section number at the byte location 5.

The **Identification Section**: Contains information that apply to all stored data in the messages, such as originating centre, sub-centre, significance of reference time (e.g., Analysis, Start of forecast, etc), reference time of data, type of processed data (e.g., Analysis products, Forecast products, etc), among others.

The **Local Use Section**: This section is optional and contains information for local use by the originating meteorological centres.

The **Grid Definition Section**: Contains the number of data points, the grid type, among others. The grid definition template code is found at the byte location from 13 to 14, this code refers to a row on the Table *GRIB2 Code Table 3.1 - Grid Definition Template Number*[2]. This table contains a list of defined grid types and also has rows reserved for future grid types for general and local use. Table 2.2 shows some grid types of a 65535 total rows.

Table 2.2: GRIB2 Code Table 3.1 - Grid Definition Template Number (NCEP WMO GRIB2 Documentation, 2016).

Code	Description
0	Latitude/Longitude (Template 3.0)
1	Rotated Latitude/Longitude (Template 3.1)
2	Stretched Latitude/Longitude (Template 3.2)
3	Rotated and Stretched Latitude/Longitude (Template 3.3)
4	Variable Resolution Latitude/longitude (Template 3.4)
5	Variable Resolution Rotated Latitude/longitude (Template 3.5)
6-9	Reserved
10	Mercator (Template 3.10)

Besides, each row indicates the Template to use from the byte location 15. For example, the structure for the Template 3.0 (Latitude/Longitude)[3] would be placed at the byte location from 15 to 72, while that for the Template 3.1 (Rotated Latitude/Longitude)[4] would be placed at the byte location from 15 to 84. Each Template has a different structure.

The **Product Definition Section**: Presents the nature of the data stored in the Data Section. The product definition template code is found at the byte location from 8 to 9, this code refers to a row on the Table *GRIB2 Code Table 4.0 - Product Definition Template Number*[5], which can be Analysis or forecast at a horizontal level or in a horizontal layer at a point in time (Template 4.0)[6], and others.

[2]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_table3-1.shtml
[3]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_temp3-0.shtml
[4]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_temp3-1.shtml
[5]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_table4-0.shtml
[6]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_temp4-0.shtml

The **Data Representation Section**: Describes how the data values are represented. The data representation template code is found at the byte location from 10 to 11, this code refers to a row on the Table *GRIB2 Code Table 5.0 - Data Definition Template Number*[7]. This table contains a list of defined data packing methods and also has rows reserved for future data packing methods for general and local use. Table 2.3 shows some data packing methods of a 65535 total rows.

Table 2.3: GRIB2 Code Table 5.0 - Grid Definition Template Number (NCEP WMO GRIB2 Documentation, 2016).

Code	Description
0	Grid Point Data - Simple Packing (Template 5.0)
1	Matrix Value at Grid Point - Simple Packing (Template 5.1)
2	Grid Point Data - Complex Packing (Template 5.2)
3	Grid Point Data - Complex Packing and Spatial Differencing (Template 5.3)

The **Bit-Map Section**: Indicates the presence or absence of data at each grid point.

The **Data Section**: Contains the data values for each grid point, and it is the longer of each message.

The **End Section**: Contains the code “7777” which marks the ending of each message.

It is worth mentioning, that besides the data values stored within the Data Section, the values with more significance for the GRIB messages are: *Discipline*, *Category* and *Name*, which are organized in branches, these three values allow to identify the parameter stored in the message. For instance, whether the message is defined for the Discipline 0 which is for Meteorological products, the categories for this discipline could be Category 0: Temperature, Category 1: Moisture, etc. Thus, when Category 0 is defined within the message being processed the Names available could be Temperature, Virtual Temperature, and Potential Temperature, among others.

2.2 Actor Model

The Actor Model is a mathematical theory of concurrent computation that dates back to 1973. This was proposed as an object-oriented model to be used in the Artificial Intelligence area which required a high computing power and in parallel systems, based on a single kind of object called *Actor* (KOSTER; CUTSEM; MEUTER, 2016). In this model, everything can be represented as actors and, each actor is an active computational entity, the communication between actors is performed through asynchronous message passing, besides actors have a behavior which specifies how the message should be

[7]http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_table5-0.shtml

processed (HEWITT; BISHOP; STEIGER, 1973). Even in the Actor Model, the control structures could be represented as a pattern of message passing among actors (HEWITT, 1977), the theory of the semantics for programming languages based on the actor model had been exposed in (CLINGER, 1981), conceptual implementations for a minimal actor language and the basis for a transition system for actor systems were proposed in (AGHA, 1986).

An actor can perform three operations for each message that is received (AGHA, 1986), these actions are as follows:

- Send a finite set of messages to other actors,
- Create a finite set of new actors; and,
- Define a new behavior, which specifies how the message will process the next message.

An actor only processes the tasks whose target is the mail address of this actor, a Task can be represented as a 3-tuple (AGHA, 1986), these tuple elements are as follows:

- Tag: To identify among the actors in the system,
- Target: It is the *mail address* of the actor to which the message is sent; and,
- Communication (immutable message): Contains information which is made available at the target actor. Likewise, the message is considered as a tuple of values (e.g. mail address, data types or even actors).

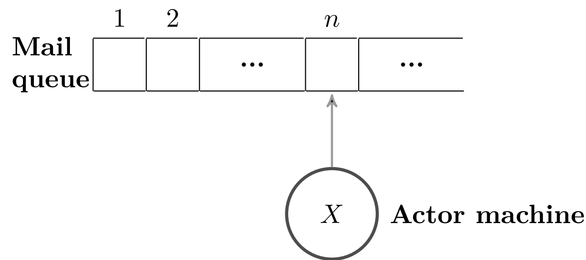
In addition, when an actor accepts a message and subsequently must send other communication, it has to know the *target* of the recipient (HEWITT; BAKER, 1977), and there are three possible ways to know this mail address (AGHA, 1986):

- The actor already knew the target before to accept the message,
- The message contains the mail address of the target; and,
- The mail address of the new actor is known by the actor that created it.

Basically, each actor has a *behavior* that indicates how to process the message received and, a *mail queue* which must be large enough to contain all the messages. The messages are sent by asynchronous message passing, and placed in the mail queue according to the order of arrival, it is also possible in this model that an actor could send messages to itself.

According to Gul Agha (AGHA, 1986), an actor can be represented abstractly as in the Figure 2.2, the actor machine contains information of the actor behavior and processes one message at a time, thus, when the actor machine accepts the message at the position n of the mail queue it can not process information from other communications.

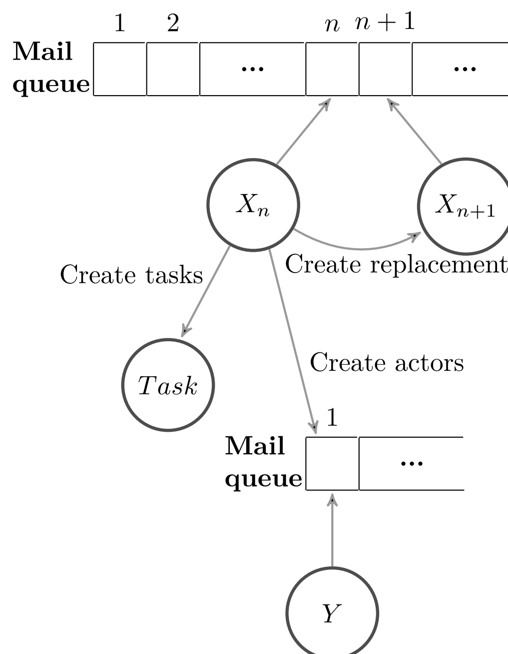
Figure 2.2: An abstract representation of an actor.



Source: Adapted from (AGHA, 1986).

As aforementioned, an actor can perform three types of actions, and they are represented in the Figure 2.3. Assuming that the mail queue has $n+1$ buffered messages, when the actor machine X_n accepts the message at the position n in the mail queue, it creates a new actor machine X_{n+1}, in turn, this new actor machine will replace the behavior of the actor, however, this "new" behavior will not be necessarily different from the previous one, since could be the same. Formally, an actor must change its behavior to accept the next message (AGHA, 1986). This new actor machine will process the message at the position $n+1$ in the mail queue, the actor machine X_n will not receive more communications neither will replace the actor behavior again, each actor machine according to its behavior can create its own actors or tasks. The actors created concurrently by an actor may know the mail addresses but do not have access to the properties of other actors (AGHA, 1986).

Figure 2.3: Actions an actor may perform.



Source: Adapted from (AGHA, 1986)

An initial configuration of an actor system may be composed of some actors and tasks unprocessed, however, there are two other types of actors, which may be part or related to the actor system. *Receptionists*, actors capable of receiving messages from outside the actor system and, *external actors*, actors that do not belong to the actor system from which they receive messages (AGHA, 1986).

It is important to mention that the actors are autonomous entities and, given that their communication is purely asynchronous, actors do not need to share memory among them, therefore conceptually in the Actor Model there are no problems related to locking and synchronization, such as critical sections, deadlocks, race conditions (Lightbend Inc, 2016).

In the ages when the concept of the Actor Model was conceived, there was no the computing power that currently exists, for instance the Intel processor 4040, introduced in 1974, barely had 3000 transistors and a maximum CPU clock speed of 750 kHz, and therefore, it was not possible to achieve the advantages of this model. However actually we can now talk about many-core architectures, clusters of computers, cloud computing, fog computing, Internet of things; thus, the Actor Model has recently gained substantial acceptance in academy and industry (TASHAROFI; DINGES; JOHNSON, 2013), (ZASADZINSKI; MUNTÉS-MULERO; SIMO, 2017). Even, Carl Hewitt mentions that the Actor model could be useful for the standardization of the Internet of Things (HEWITT, 2015). This popularity is supported for the achievements of the Erlang[8] language, which implements the Actor Model. Erlang was used in Ericson to develop the AXD301 high-performance switch, this project achieved the nine nines reliability (99.9999999%) (ARMSTRONG, 2003). This model is the basis of many programming languages, being the most popular Erlang, and the Akka Toolkit which is is strongly inspired by Erlang.

2.3 Akka Toolkit

Akka implements the already aforementioned Actor Model, and is written in Scala[9], a programming language originated at the École Polytechnique Fédérale de Lausanne (EPFL) and that runs on the Java Virtual Machine (JVM). This toolkit comes as a library to be used with Scala, and not as part of the programming language, however, there are APIs for both Scala and Java programming language. Akka toolkit allows building highly concurrent, reactive, distributed, asynchronous, and fault tolerant applications.

There is a need to build applications that consider the parallelization of their algorithms from the beginning of development (ROSE; NAVAUX, 2003). In this sense, implementations of the Actor Model, such as Erlang or Akka, allow to implement applications that are considered parallelizable from their inception.

An Akka actor has an actor reference, state, behavior, mailbox, actor childs and a

[8]https://www.erlang.org/
[9]http://www.scala-lang.org/

supervisor strategy (Lightbend Inc, 2016).

- Actor Reference, represents an actor, thus, other actor uses this actor reference to communicate with a particular actor, in this way, the actor state is protected from other actors.
- State, refers to variables, which may maintain the state of the actor. This state can be modified for an actor only in response to a message received.
- Behavior, refers to a function, which defines the actions to be executed in response to an incoming message. An actor may change its behavior at runtime according to the computation logic.
- Mailbox, stores the incoming messages to an actor. An actor can have a unique mailbox, and the messages are processed on arrival order.
- Child actors, an actor can create new actors from its own context, so those new actors will be its children.
- Supervisor strategy, every actor has only one supervisor, and the supervisor strategy defines how the supervisor handles the failures of the children. The supervisor may resume, restart, stop the subordinate actor, or even escalate the failure to its supervisor actor. There are two strategies to which the mentioned actions are applicable: *One-For-One* and *All-For-One*, the first only applies to the actor who failed, and the second strategy means that the action applies to all subordinate actors.

2.3.1 The actor system

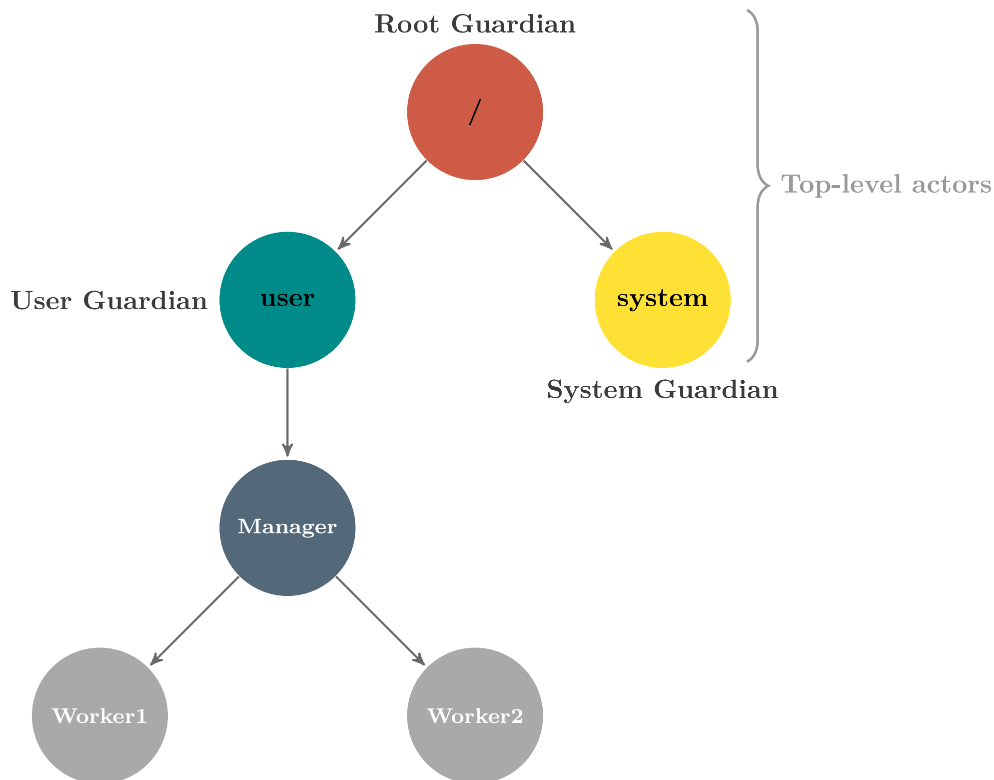
It is a heavyweight structure composed of a set of actors, that may allocate from 1 to N threads, so it is recommended to create just one per application (Lightbend Inc, 2016). The actor system houses a collection of actors, allows to create, stop, look up actors, and manages their life cycles. Actors created by an application reside only within an actor system. Likewise, an actor system has a default configuration, which may be modified to enable or disable settings of the actors, in this configuration is possible to define the actor properties, network communication protocol, mailbox type, serializer, dispatcher, router strategies, remoting, clustering, among others options. This configuration is shared for all the actors running in an actor system and may be defined in a file (*application.conf*) or in the application code.

During the creation of an actor system, three top-level actors are created, as illustrated in Figure 2.4, the *User Guardian*, also known as the *Guardian Actor*, which contains all the actors created by the application, the *System Guardian* that contains the actors created by the system (e.g. logging, dead-letters), it is shut down automatically

when the User Guardian is terminated, and the *Root Guardian*, which is the parent of the System Guardian and the User Guardian.

The actor system involves a hierarchical group of actors, where each actor created has a parent, or even a grandparent. Thus, for example, in Figure 2.4, the actor named *Manager* has two actor children, *Worker1* and *Worker2*, so the Manager actor supervises its children. Thus, the supervision model and the actor hierarchy allow Akka to be fault tolerant (VERNON, 2015).

Figure 2.4: Akka Actor Hierarchy.



2.3.2 Actor Messaging

The way how internally works the messaging in Akka is explained briefly below and illustrated in Figure 2.5. In this figure the *sender* is not necessarily an actor, however the receiver or target have to be an actor.

Figure 2.5: Actor Model in Akka.

Figure 2.6 presents the code snippet of the object class called MainProgram, where it is created an ActorSystem, a JobManager actor, and a message is sent to JobManager.

- An ActorSystem is initialized, and a name is assigned to it (Line 3).
- Through the actor system, a reference to the JobManager is created, using the *actorOf* method, which takes in *Props* the name of the actor class to be created. This reference in the sample was called jobManager (Line 4).
- The JobRequest message is sent to the jobManager ActorRef, using the bang operator (!) (Line 7).
- The ActorRef delegates responsibility for handling the messages to the Dispatcher.
- The Dispatcher places the messages in the mailbox of the target actor.
- The mailbox dequeues the message and deliveries the message to actor.
- The actor accepts the message on the receive method according to its behavior, and processes the message.

Figure 2.6: Actor sample: Main program code.

```
1  object MainProgram extends App {
2    println("Main Program")
3    val system = ActorSystem("sample")
4    val jobManager = system.actorOf(Props[JobManager], name="jobManager")
5    val files: List[String] = List("file1", "file2", "file3")
6    println(s"jobManager: ${jobManager.path}")
7    jobManager ! JobManager.JobRequest(files)
8  }
```

2.3.3 Creating Actors

Using Scala, an actor class is defined by extending the Actor[10] base trait and implementing the abstract method *receive*, in this method are declared all the message types that an actor can handle by its default behavior. The definition of an actor class is illustrated in Figure 2.7, where the *default* behavior (receive method) handles the JobRequest message, and the *processing* behavior handles the WorkerResult message.

Firstly, to create actors, it is necessary to have an actor system in which the actors can reside. Actors are not created directly, instead the actor system is used to create them, however, this does not return the instance of an actor, instead, it returns a reference to the actor created, and allows to interact with the actor. Thus, an actor is created using the *actorOf* method of the ActorSystem[11] class, which returns an ActorRef[12].

[10]http://doc.akka.io/api/akka/current/index.html#akka.actor.Actor
[11]http://doc.akka.io/api/akka/current/index.html#akka.actor.ActorSystem
[12]http://doc.akka.io/api/akka/current/index.html#akka.actor.ActorRef

It is important to point out that in Figure 2.6 in the line 4, the created actor will be placed under the User Guardian, due to this actor was created using the ActorSystem. Besides, as aforementioned, an actor may create other actors, the manner of creating an actor as child of an existing actor is shown in the line 7 of the Figure 2.7, where the JobManager actor creates three JobWorker actors. The JobWorker class is defined in Figure 2.8. For this other case, the actors are created using the *actorOf* method of the ActorContext[13] class, which represents the context of an actor.

Figure 2.7: Actor sample: JobManager actor code.

```
1  class JobManager extends Actor {
2
3    def receive = {
4      case JobManager.JobRequest(files) => {
5        println(s"JobRequest: ${files}")
6        for(i <- 0 until files.length) {
7          val jobWorker = context.actorOf(Props[JobWorker], name=s"
              jobWorker${i+1}")
8          jobWorker ! JobWorker.ProcessTask(files(i))
9        }
10       context.become(processing)
11     }
12   }
13
14   def processing: Receive = {
15     case JobManager.WorkerResult(filename, worker) => {
16       println(s"${filename} was processed by ${worker.path}")
17     }
18     case _ => println("Unknown message" )
19   }
20 }
```

Every actor has a name and also belongs to a hierarchical collection of actors, in this sense, each actor has a path, which represents the sequence of actor names since the User Guardian towards the actor itself including the ActorSystem name. For example, the jobManager's path is `akka://sample/user/jobManager`, and the jobWorker1 which is the jobManager's child, has the path `akka://sample/user/jobManager/jobWorker1`. Besides, when the system involves remote actors, the path also contains the communication network protocol, hostname and port used, thus, for example the path for a remote actor could be: `akka.tcp://sample@hostname:port/user/remoteJobWorker`. This fashion to identify an actor is very useful in Akka as it provides a location transparency for actors.

[13] http://doc.akka.io/api/akka/current/index.html#akka.actor.ActorContext

Figure 2.8: Actor sample: JobWorker actor code.

```
1  class JobWorker extends Actor {
2
3    def receive = {
4      case JobWorker.ProcessTask(filename) => {
5        println(s"ProcessTask: ${filename}")
6        sender ! JobManager.WorkerResult(filename, self)
7      }
8    }
9  }
```

2.3.4 Sending and receiving messages

The actor communication is performed through asynchronous message passing, in Akka, messages may be any kind of object, however it should be an immutable object. For this, reason, when Akka is used with Scala, it is preferable to use the *case classes* or *case objects* which are immutable data-holding objects and may be pattern matched. Thus, for example, in Figure 2.9 are defined the messages that can be processed for the JobManager actor, JobRequest and WorkerResult (Lines 2-3), and the messages that can be processed for JobWorker actor, Processtask (Line 7).

Figure 2.9: Actor sample: JobManager and JobWorker messages.

```
1  object JobManager {
2    case class JobRequest(files: List[String])
3    case class WorkerResult(filename: String, worker: ActorRef)
4  }
5
6  object JobWorker {
7    case class ProcessTask(filename: String)
8  }
```

The foremost way to send a message is using the bang operator (`!`), also referred to as *tell*, it follows the philosophy "fire and forget", send a message and return immediately, in other words, there is no blocking, the sender actor does not wait for the message to be processed from the recipient actor, the Tell method barely enqueues the message in the target's mailbox. Every time a message is sent, Akka implicitly attaches the sender reference within the message, it is useful when the target actor after processing the message requires to send back a message to the sender. Line 7, 8 and 6 in the Figure 2.6, 2.7 and 2.8 respectively illustrate the manner to send messages in Akka. In this first case due to *MainProgram* is not an actor, there is no sender itself and implicitly within the message an *ActorRef.noSender* is passed. The second case, *JobManager* is effectively an actor, thus, when a message is sent to a *JobWorker* actor, the *context.self* reference implicitly is passed together with the message. For the latter case, the *JobWorker* actor sends back a message to the sender (JobManager actor).

A second way to send messages in Akka is using the *forward* method, there is a mediator in this case, which may be a router, a load balancer, or a replicator, it means that the target actor does not recognize to the mediator as sender but rather to the original sender (Lightbend Inc, 2016).

A third way is using the ask operator (?), it sends a message to the target actor and waits for a response as a future message.

2.3.5 Modifying the actor behavior

At this point, it is necessary to explain briefly the codes shown above, to illustrate some basic concepts related to the creation of actors and the message sending between actors in Akka. Thus, the code snippet in Figure 2.6 creates an actor system, next the JobManager actor, and at the last a JobRequest message is sent to JobManager, this message contains a list of three strings. In the JobManager actor, code snippet in the Figure 2.7, receives the JobRequest message at the initial behavior, iterates the list of strings, and for each one creates a JobWorker actor, and subsequently sends a Processtask message to the created actors. After that, the JobManager changes its behavior from default to processing, using the *context.become* method, that takes the name of the new behavior, it basically means that the next messages received for the JobManager actor will be attended for the processing method. Next, the ProcessTask message is received for the default behavior of the JobWorker actor, code snippet in the Figure 2.8, here could be performed some type of processing, and subsequently a WorkerResult message is sent back to the JobManager.

Figure 2.10: Actor sample: Terminal output.

```
1  Main Program
2  jobManager: akka://sample/user/jobManager
3  JobRequest: List(file1, file2, file3)
4  ProcessTask: file1
5  ProcessTask: file3
6  file1 was processed by akka://sample/user/jobManager/jobWorker1
7  file3 was processed by akka://sample/user/jobManager/jobWorker3
8  ProcessTask: file2
9  file2 was processed by akka://sample/user/jobManager/jobWorker2
```

In the JobManager actor, which now has active the processing behavior, receives the WorkerResult messsage and prints on the terminal the string and the path of actor that processed the message. Any type of message received for the JobManager actor that does not match with the WorkerResult message will be ignored, although it could be handled if required, as illustrated in the line 18 of Figure 2.7, where a message *Unknown message* is printed on the terminal. Finally, Figure 2.10 shows the output result in the terminal after to compile and execute the sample utilized.

2.3.6 Clustering

Akka cluster nodes may have different types of roles, it is useful to group actors that perform a specific work, these roles are defined in a configuration file, besides it is also possible to define other parameters related to the Akka cluster, such as the minimum number of nodes of a specific role necessaries to start an application, the maximum number of actors allowed per node, the maximum allowed number of actors that can run in the cluster, a list of seed nodes, among others.

The seed nodes are initial reference points for new nodes joining the cluster, the first seed node declared in the configuration file is in charge of forming the cluster. This first seed node will be the leader of the cluster, but if this node fails, it will be removed, and any other node will become the leader, taking into account that only one leader can exist at a time (ROESTENBURG; BAKKER; WILLIAMS, 2016).

Akka uses the Gossip protocol and an automatic failure detector to provide a membership service based on the Amazon Dynamo model (DECANDIA et al., 2007), which is fault-tolerant descentralized peer-to-peer and without a single point of failure (Lightbend Inc, 2016). The cluster state is communicated to all nodes of the cluster using a Gossip protocol, where each node gossips to other randomly chosen nodes its current state and the state of other nodes that it has observed. It allows to the nodes to know about the state of the other nodes in the cluster, achieving a gossip convergence at a node at certain points in time. Besides, Akka uses the Phi Accrual Failure Detector (HAYASHIBARA et al., 2004) to detect unreachable nodes.

2.3.7 Routers

Akka provides several built-in router strategies to send the messages among a group of actors which can be local or remote actors, the target actors are denominated as *routees*. This distribution aims balance the load over the different actors involved in a job, a router in Akka, is also an actor which encapsulates a routing logic and defines the settings to be used for the router, this settings can be declared in the code or loaded from a configuration file. Some of the available router strategies in Akka are the following:

- **RoundRobin** sends the received messages to its routees in a round-robin fashion.

- **Random** sends the received messages to its routees in a randomly order.

- **Broadcast** sends the received messages to all its routees.

- **AdaptiveLoadBalancing** uses the latest cluster metrics collected of the cluster nodes to decide which node is most suitable to receive messages. Based on the cluster metrics, the router sends the messages to the nodes with a lower weight,

since a lower weight means a high probability of available resources (Lightbend Inc, 2016). There are four type of metrics:

- **heap**, it is based on the remaining JVM heap. The formula for calculating the weights is:

  $(max - used)/max$

  Where:

  max: Maximum JVM heap memory.

  $used$: Used JVM heap memory.

- **cpu**, it is based on the CPU utilization percentage. The formula for calculating the weights is:

  $1 - (User + Sys + Nice + Wait)$

  Where:

  $User$: Percentage of CPU utilization that occurred while executing at the user level (application).

  $System$: Percentage of CPU utilization that occurred while executing at the system level (kernel).

  $Nice$: Percentage of CPU utilization that occurred while executing at the user level with nice priority.

  $Wait$: Percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.

- **load**, it is based on the Unix system load average. The formula for calculating the weights is:

  $1 - (load/processors)$

  Where:

  $load$: Unix system load average.

  $processors$: Number of processors.

- **mix**, it is based on the combination of the heap, cpu and load metrics.

Likewise, there are two categories of routers, both share the router logic (e.g. round-robin logic), the difference lies in the management of the routees.

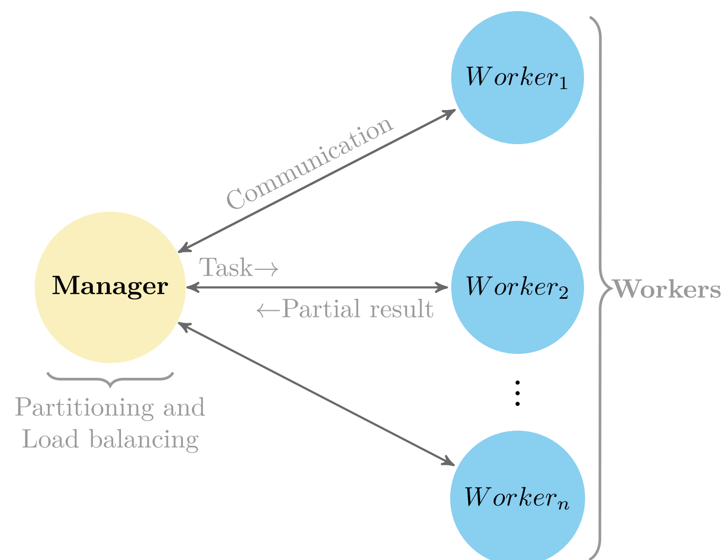
- **Group** uses routees created previously on the nodes. The path of the all routees must be declared on the router's configuration before the router is initialized. It is not responsible for the life cycle of the routees, besides the routees may be shared with other router or routers.

- **Pool** creates its own routees as children, it means, a Pool router manages the life cycle of the routees, and routees are not shared among other routers.

2.4 Manager-Worker Pattern

This model of communication is based on a dynamic task distribution, composed of a manager and a set of workers (CHANDY; TAYLOR, 1992), the Figure 2.11 illustrates the design of this parallel programming model. The manager, or also named master process (BALAJI, 2015), is responsible for partitioning in sub-problems the problem to solve, these sub-problems are the tasks which are distributed among a group of workers. The workers are the responsible for processing the received tasks and returning the results to the manager.

Figure 2.11: Manager-Worker model of communication.



Source: Adapted from (CHANDY; TAYLOR, 1992)

Thus, the manager partitions the work into small tasks, stores a record of those tasks, and initiates the execution of the workers, each worker send a request asking a task to be processed, the manager will choose a task to be sent to the worker in response to the request. The workers execute the tasks independently and return the partial result to the manager, at this point, the worker also requests a new task from the manager. This cycle continues until there are no more pending tasks on the manager, when this happens the manager sends a message for each worker's request, notifying that all tasks were processed. Thereafter, the manager computes the final result from the partial results sent from the workers (CHANDY; TAYLOR, 1992). Finally, when all processing is done, the workers could be removed followed by the manager (JANSSEN; NIELSEN, 2008).

It is worth noting that this model may be used both for uniform and for non-uniform tasks, besides the main advantage is that provides implicitly a load balancing (BALAJI, 2015).

The Manager-Worker model is derived of the well-known Master-Slave pattern (ORTEGA-ARJONA, 2004), the difference lies in that for the Master-Slave scheme, the

manager assigns the task to the workers instances, and those workers no perform continuous requests for new tasks to the manager (BUSCHMANN et al., 1996).

2.5 Apache Spark

Spark was originated at the UC Berkeley RAD Lab and later was included as a open source top-level project of the Apache Software Foundation. Internally, it is composed of different projects, such as Spark SQL, Spark Streaming, MLib for machine learning, GraphX for graphs processing, and the Spark Core, which is responsible for scheduling, distributing and monitoring tasks across the cluster (KARAU et al., 2015). It emerged with the purpose of attending specially workloads that required to work with iterative algorithms, interactive queries and stream processing, which were not suitable for working with Apache Hadoop. Spark is written in Scala, and runs on the Java Virtual Machine, however provides APIs to work with Java, Python and R.

Spark introduces the Resilient Distributed Dataset (RDD), which is an immutable collection of elements partitioned among the worker nodes, besides this collection is persisted in memory (ZAHARIA et al., 2012). There are two ways to create a RDD, the first is parallelizing a collection in the driver program, and the second is loading datasets from an external source, such as Hadoop Distributed File System (HDFS), Cassandra, or any data source supported by the Hadoop API. Apache Spark allows to perform two types of operations over a RDD, actions and transformations. The first type returns a value as result of a calculation, while that the second returns a new RDD after to perform an operation over a RDD, besides this type is lazy evaluation (ZAHARIA et al., 2010), it means that the RDDs are computed only when an action is performed over them, The combination of RDDs on a transformation is called as lineage, in this sense, the fault tolerance is efficiently supported by Spark, since the lineage allows recompute a RDD partition lost, it is for that reason that a RDD is resilient.

Apache Spark may be deployed on top of Apache YARN[14], Apache Mesos[15] or its own cluster manager. The Figure 2.12 shows the components of Spark, the Driver program contains a SparkContext, which represents the connection to a Spark cluster and primarily serves to build RDDs. While the cluster manager starts and manages the executors on the worker nodes, these executors run tasks scheduled by the driver program, store results in memory or on disk. Apache Spark distributes the application code among the worker nodes, and the SparkContext defines the RDDs in the cluster through transformations, and sends the tasks to be performed for the executors when an action must be executed. In this way, Apache Spark distributes the data stored in RDDs and parallelizes the operations to perform over the RDDs.

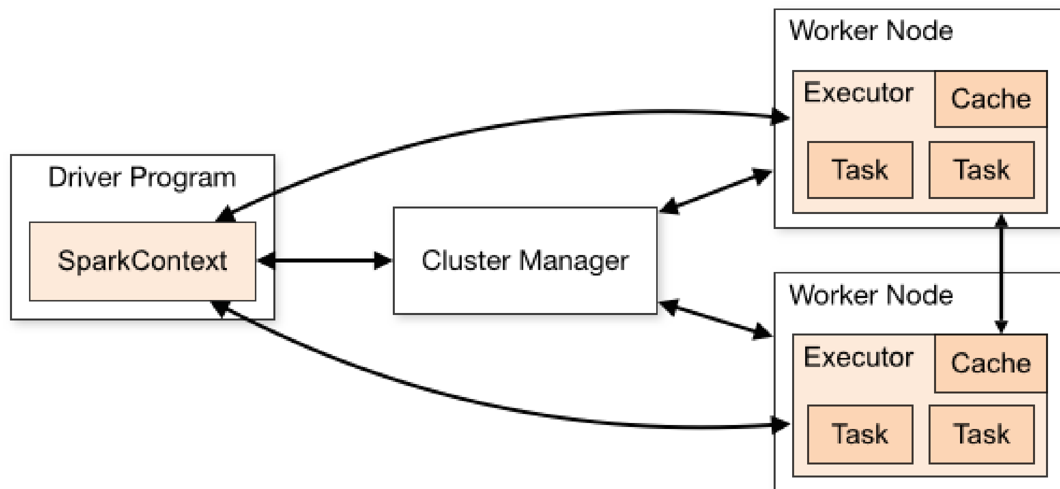
However, limitations arise when the datasets largely exceed the memory capabili-

---

[14]https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[15]http://mesos.apache.org/

ties, when it happens in Apache Spark, it is necessary to store the RDDs on disk, causing a downgrade in the performance or throwing JVM exceptions (GU; LI, 2013).

Figure 2.12: Spark components on a distributed environment.



Source: Adopted from (Spark Documentation, 2016).

3 RELATED WORK

In relation to the GRIB scientific data format, there are libraries such as GRIB API[1], wgrib2[2], Climate Data Operators (CDO)[3] that excel in the processing of GRIB files and allow reading and writing GRIB messages (MARKIEWICZ et al., 2013). The GRIB API and wgrib present support for pthreads and OpenMP, respectively, however both have to be compiled with their respective flag to enable that support. Both libraries to allow processing GRIB files in parallel in a compute node but the data packing and unpacking methods are not parallelized, and the operations performed are over a single GRIB file and not in a set of GRIB files.

On the other hand, there are many efforts to process and perform scientific data analysis over scientific data formats in a parallel and distributed environment, mainly for scientific data formats, such as NetCDF (REW; DAVIS, 1990) and HDF5 (The HFD5 Format, 2002) formats.

Thus, Li et al. (LI et al., 2003) proposed a parallel interface for reading and writing NetCDF files. Modifications to the serial NetCDF interface to support parallel data access were performed in that work. The parallel data access was based on MPI-IO. They evaluated the performance and scalability of their proposal against the serial NetCDF, and used a synthetic benchmark for reading and writing 64 MB and 1 GB of NetCDF data. Their experimental environment was composed of 144 computer nodes, IBM SP-2, each node had 4GB of memory, and 12 I/O nodes that used the General Parallel File System (GPFS).

In Zhao et al. (ZHAO et al., 2010) is argued that storing terabytes of data to search specific data in those large datasets subsequently is a key point in geoscience and climate areas. In this way, they proposed a method to store and access over massive NetCDF datasets based on MapReduce using Apache Hadoop version 0.19.1. However, at an early stage the datasets need to be parsed and transformed into CDF (Common Data Language) file – a text notation for NetCDF objects and data – to be stored in the HDFS, thereby causing an overhead mainly in storage. They used subsets of 128 MB, 512 MB, 1.86 GB, 8.15 GB and 31.1 GB from the Argo (Array for Real-time Geostrophic Oceanographic) data to evaluate the performance with different data scales, and subsequently with different number of reduce tasks. The performed job consisted of obtaining the values of the pressure parameter. Their experimental environment was composed of 1 master node and 8 data nodes, each node had 8 cores, the master node had 4 GB of memory whereas the data nodes had 8 GB of memory. Likewise HDFS was configured with the data block size of 64 MB and a replication factor of 2.

Buck et al. (BUCK et al., 2011) implemented SciHadoop, a plugin over Apache Hadoop version 0.21 that allows processing NetCDF files version 3 and executing queries

[1]https://software.ecmwf.int/wiki/display/GRIB/Home
[2]http://www.cpc.ncep.noaa.gov/products/wesley/wgrib2/index.html
[3]https://code.zmaw.de/projects/cdo

over the logical data model. Furthermore, they extended the NetCDF-Java[4] library from Unidata to work with HDFS, and detailed five optimizations to be performed over the NetCDF data stored (partitioned) in HDFS with the goal to reduce remote reads, total data transfers and unnecessary reads. They used a synthetic dataset which had a total size of 132 GB. One of the queries applied in that work was computing the median in a time-range, in an area-range and in an elevation-range for the air pressure parameter. Their experimental environment was composed of a cluster of 1 master node and 30 compute nodes, each node had 2 cores, 8 GB of memory and four 250 GB local disks of which three were dedicated for HDFS, which was configured with the data block size of 64 MB and a replication factor of 3, and one was dedicated for the operating system and temporary storage.

In Duffy et al. (DUFFY et al., 2012) was carried out an evaluation of MapReduce applied in climate data analysis in order to improve the workflow for the analysis utilized in the Soil Moisture and Ocean Salinity (SMOS) where datasets were analyzed using MATLAB with the goal of finding locations on the earth which do not change considerably over time and that serve to validate and calibrate their instruments. In that work, they calculate the average of the surface temperature over 8 years of data using the NPANA 3D data – a subset of the Modern Era Retrospective-Analysis for Research and Applications (MERRA) data – whose total size was more than 300 GB. They used Apache Hadoop version 1.0 and their experimental environment was composed of 2 head nodes which were utilized for the Namenode and JobTracker, and 8 data nodes with two 1 TB file systems using HDFS with a data block size of 640 MB and a replication factor of 3. Each node had 4 cores and 8 GB of memory.

The SciMate Framework proposed by Wang et al. (WANG; JIANG; AGRAWAL, 2012) allows to process the scientific data formats, NetCDF and HDF5, as well as plain text files, and may be extended to other scientific data formats by implementing an adapter for data loading and processing. That framework was built on top of the Mate system which was implemented in MPI and that allows to process data using the MapReduce-style programming. The datasets utilized were of 16 and 8 GB, which were stored in the Parallel Virtual File System (PVFS). Three data analysis algorithms were used: the K-means clustering, the principal component analysis and the K-nearest neighbor search. Furthermore, their experiments evaluated the scalability of their proposal on a full and partial read scenario, since not all the variables of a dataset are always used in the observation centers, but rather only a subset of those. Their experimental environment was composed of a cluster with 16 nodes, each node with 8 cores and 8 GB of memory.

Blanas et al. (BLANAS et al., 2014) presented the design of a prototype system called SDS/Q that can perform scientific data analysis by processing queries directly over datasets stored in the HDF5 format, executing the queries in-memory and using the bitmap indexing to reduce the queries response time. They used a synthetic dataset

[4]http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/

and the Palomar Transient Factory (PFT) data, an automated survey of the sky for transient astronomical events, which took about of 180GB stored in PosgtreSQL. Both datasets were stored in GPFS upon using the SDS/Q. The synthetic data is used in a first evaluation by comparing PostgreSQL with the SDS/Q taking into account two access patterns, the full scan operation used to calculate the average of 12 variables, and the point lookup operation used to calculate the average of those 12 variables taking into account a specific attribute in common. A second evaluation was done by using the PFT data to compare the SDS/Q with PostgreSQL and Apache Hive, in that evaluation were considered 3 queries by applying filters and varying the time interval. The experimental environment was composed of 64 compute nodes, each node with 8 cores and 24 GB of memory, besides those nodes had no local disks.

In Palamuttam et al. (PALAMUTTAM et al., 2015) was proposed a framework built on top of Apache Spark, the SciSpark, which allows to ingest scientific data formats, such as NetCDF and HDF5 from different data sources as for example HDFS, OpenDap, local file system. However, subsequently the data have to transformed into a data type accessible in SciSpark, the Scientific Resilient Distributed Dataset (sRDD), that provides methods to work with multi-dimensional arrays. They used the MERG dataset from the Climate prediction center/NCEP/NWS for their experiments, likewise they also discusses three approaches to parallelize the initial stages of the Grab 'em, Tag 'em, Graph 'em (GTG) algorithm, a graph-theory based algorithm for identifying and tracking high precipitation events, using the Apache Spark’s API. Their experimental environment was composed of a cluster of 4 compute nodes, each node with 32 cores, 240 GB of memory and 100 GB local disk.

Table 3.1: Related work summary.

	Format	**Operation**	**DFS (*)**	**Dataset size**
Li et al., 2003	NetCDF	read/write	GPFS	64MB, 1GB
Zhao et al., 2010	NetCDF	read	HDFS / 64MB / 2	128MB, 512MB, 1.86GB, 8.15GB, 31.1GB
Buck et al., 2011	NetCDF	median	HDFS / 64MB / 3	132GB
Duffy et al., 2012	NetCDF	average	HDFS / 640MB / 3	over 300GB
Wang et al., 2012	NetCDF, HDF5, flat-files	K-means, PCA, K-nearest neighbor	PVFS	8GB, 16GB
Blanas et al., 2014	HDF5	average	GPFS	–
Palamuttam et al., 2015	NetCDF, HDF5	Grab 'em, Tag 'em, Graph 'em	HDFS, OpenDap	–

(*) DFS / block size / replication factor

Table 3.1 summarizes the related works presented which are focused mainly on the NetCDF and HDF5 scientific formats. These works perform from basic operations utilized in statistical applications to complex algorithms. Likewise these operations were run over the entire dataset or even over a subset of data by applying filters to work just

a few parameters of the entire dataset, in a time-range or an area-range which is more utilized in real applications. The solutions were implemented using MPI, Apache Hadoop or Apache Spark. And, the HDFS also was used as part of the solution when these last two frameworks were employed. In addition to this, these works evaluate the performace of their proposals in a distributed environment by varying the number of nodes and the size of the dataset used in the experiments.

4 AGRIB: PROCESSING GRIB DATA IN DISTRIBUTED ENVIRONMENTS WITH AKKA

As stated earlier the most of researches that works with scientific data are focused in the NetCDF and HDF5 formats due to their popularity, however there are many observational centers that uses the GRIB format[1,2]. In this sense, this work proposes an alternative way to process large data sets in the GRIB format in a distributed environment. Our work performs the average function over the datasets used in the experiments which were stored in the Hadoop Distributed File System (HDFS), likewise two scenarios were selected to process the GRIB files, one that uses all the parameters of the dataset and one that evaluates just 2 of 27 total parameters found in the dataset. Our proposal is referred to as aGRIB (actor GRIdded Binary), and utilizes the Manager-Worker pattern, which is implemented with the Actor model, specifically the Akka toolkit implementation. Subsequently, after analyzing the first experimental results we perform an improvement over our proposal to achieve better performance. Additionally, we compare our proposal with other mechanisms provided by the Akka toolkit as well as with one of the main frameworks currently existing for big data processing.

This chapter is divided into five sections. Section 4.1 presents the module implemented to parse and decode the GRIB messages, which is utilized in our proposal, as well as in the comparison mechanisms. Section 4.2 details our initial proposal for processing the GRIB files using the Manager-Worker pattern implemented with the Actor model. Section 4.3 describes how the comparison mechanisms used to process the GRIB files work, which use built-in router strategies provided by the Akka toolkit. Section 4.4 describes an improvement over our initial proposal, which uses metadata stored in an NoSQL database to obtain better performance in processing the GRIB messages. Section 4.5 presents another comparison mechanism to process the GRIB files using Apache Spark.

4.1 Scala client module

Initially, it was necessary to implement a module to parse and decode the GRIB messages, this module was implemented in the Scala programming language, because it is more suitable for working with the Akka toolkit.

The implementation of this module was based on the WMO documentation (WMO, 2003), (NCEP WMO GRIB2 Documentation, 2016), and the Thredds library (UNIDATA, 2017), which is mainly utilized to work with NetCDF files, nevertheless, this one has a module to parse GRIB files.

The GRIB messages on the dataset utilized in the experiments were version 2 GRIB

[1]https://www.wmo.int/
[2]http://www.cptec.inpe.br/

message, and after analyzing the dataset, it was noted that the Grid type for all the GRIB messages was the *Latitude/Longitude* type, the Product type was *Analysis or forecast at a horizontal level or in a horizontal layer at a point in time*, similarly in relation to the data packing method, only two methods were found, the *Grid Point Data - Complex Packing* and the *Grid Point Data - Complex Packing and Spatial Differencing*. Therefore, the Scala module client has been focused on the GRIB message version 2, and is just implemented for these two data packing methods.

4.2 Using the Manager-Worker pattern

The Manager-Worker pattern explained previously in Section 2.4 was employed to perform the job distribution, which means the manager partitions the work to do, and sends the tasks to be performed by the workers as requested by them, in turn, the workers request tasks from the manager and send the partial results to the manager, and at the end the manager computes the final result from the partial results obtained. The whole process is implemented using the characteristics of the Actor Model provided by the Akka toolkit.

Thus, Figure 4.1 depicts the logical representation of the components in our proposal, in this figure can be appreciated two types of machines, the Master and the Worker machines. In addition, in order to build the cluster in Akka, three types of nodes were deployed on these machines, the Seed ❶ , Master ❷ and Worker ❸ role node. The first two were contained inside the Master machine, and the third inside the Worker machines. It bears noting that each one of those Akka cluster nodes are deployed on a separated Java Virtual Machine (JVM).

– The *Seed node* is necessary to specify at least one node with this role in an Akka cluster, since this node acts as an initial contact point for new nodes joining the cluster.

– The *Master role node* is where the JobManager actor is performed.

– The *Worker role node* contains the JobWorker actors, and there can be one or more JobWorker actors per worker role node. Likewise there can be one or more worker role nodes per worker machine. This can be seen in the Subsection 5.3.1.

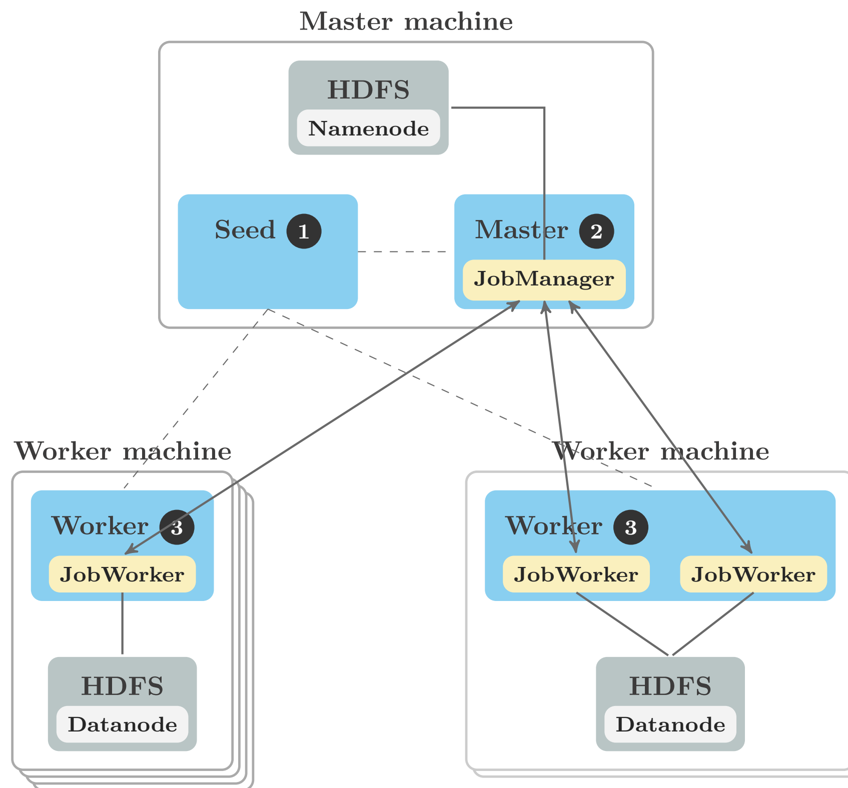
As mentioned previously, there are two actor types running inside our Akka cluster nodes, and as depicted in Figure 4.1, there is a bidirectional relationship between the JobManager actor and the JobWorker actors, however there is no communication among the JobWorker actors.

– The *JobManager* distributes the tasks among JobWorker actors, receives the partial results and computes the overall result from the partial results sent from the Job-

Worker actors. This actor works as the Manager component of the Manager-Worker model.

– The *JobWorker* is in charge of processing the tasks sent from the JobManager, sending the partially computed result and requesting tasks of the JobManager. This actor type acts as the Worker component of the Manager-Worker model.

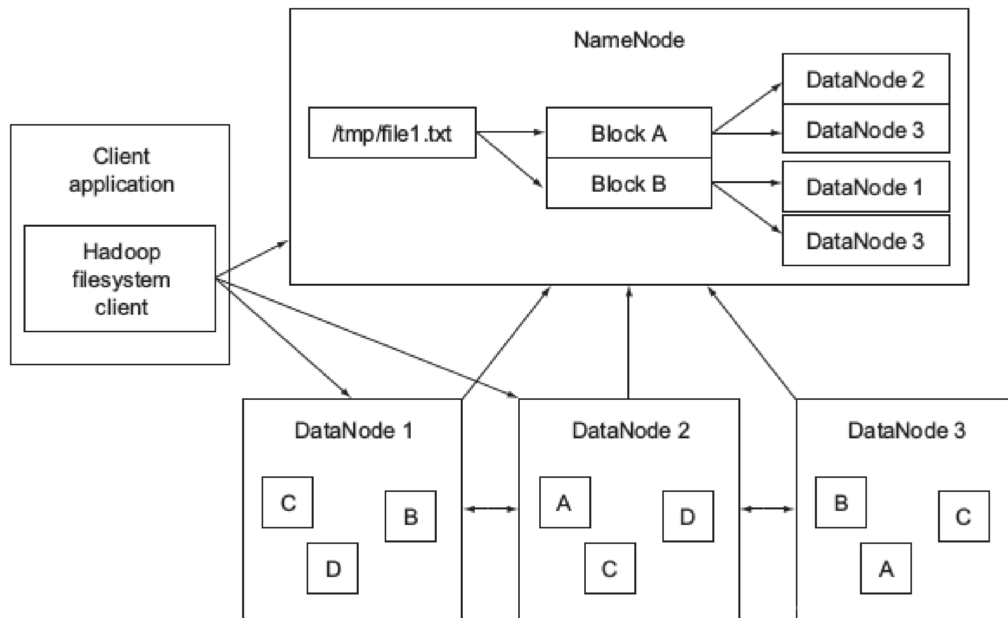
Figure 4.1: Initial architecture: Using the Manager-Worker pattern.



It is also important to mention that the dataset used in the experiments was stored in the Hadoop Distributed File System (HDFS), so that all nodes involved in the GRIB files processing may access the data, therefore, a Namenode and Datanodes were necessary where the first was deployed inside the Master machine, and the latter were deployed on the Worker machines. Figure 4.2 depicts the HDFS architecture, where the Namenode stores the metadata about the file system, which contains information about the files and the blocks location of each file distributed among the Datanodes, and the Datanode stores blocks of files, those blocks are replicated among the Datanodes, so that a client application may write and read files. Thus, in our case a client application may be a JobManager or a JobWorker actor.

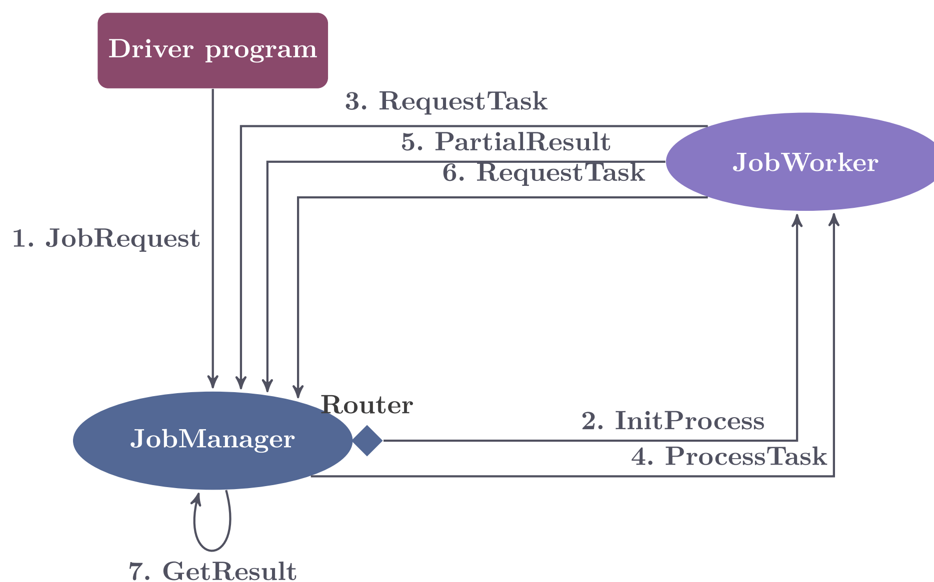
Figure 4.3 illustrates how the design of our proposal for processing GRIB files works. The actions performed by the *Driver program* are as follows:

Figure 4.2: HDFS Architecture. Figure adopted from (HOLMES, 2014).



- The necessary configurations are loaded, such as the HDFS URI, the path where the dataset is stored, router type and other parameters used in the execution of the experiments.
- A list with all the GRIB file names to be processed is obtained from the HDFS.
- Creates an ActorSystem, and a JobManager ActorRef.
- A *JobRequest* message is sent to JobManager with the list of names of GRIB files.

Figure 4.3: Design: Using the Manager-Worker pattern.



Afterwards, in the JobManager actor, a router of the BroadcastPool type is created and then, using this one router, a *InitProcess* message is sent to the JobWorker actors created in the cluster nodes. However, only this first message will be sent through the BroadcastPool router with the subsequent messages being sent through the JobManager actor itself. After sending the message, the JobManager actor changes its behavior from *default* to *processing*.

Subsequently, each JobWorker actor sends a *RequestTask* message to the JobManager asking to be registered as an actor capable of executing tasks and requests its first task. Additionally, at this point the JobWorker also changes its behavior from *default* to *processing*.

All of these messages received in the JobManager are enqueued in its mailbox and attended to in arrival order. The JobManager obtains the next task and sends a *ProcessTask* message to the JobWorker actor with information about the task to be processed.

In this step, the JobWorker actor uses our Scala client module for processing the GRIB messages stored in HDFS, and the output of each GRIB message processed is merged to calculate the partial result which will be sent to the JobManager. Thus, a *PartialResult* message is sent to the JobManager actor followed by other *RequestTask* message.

In the JobManager, the partial result is stored in memory, and one new task is sent to the JobWorker, through the *ProcessTask* message. This cycle continues until there are no more tasks to process, and when this occurs, the JobManager sends a message called *AllDone* to the JobWorker, who request an additional task. In addition to this, the JobManager actor changes its behavior from *processing* to *finish* and sends a *GetResult* message itself, in order to calculate the final result from the partial results received previously.

As will be explained in Subsection 5.3.2, the method of processing the GRIB messages of a file by each JobWorker actor may be completed in two ways, file-based and message-based, the choice of one of these two types of processing is defined in a configuration file, which is loaded at the Driver program. Hence, depending on which option was defined, the JobManager actor will select the next task to be sent to the JobWorker actor, being able to select the next file in the list (file-based) or the next group of messages of a file (message-based), and so on until to finish of processing all the files defined in the list sent in the JobRequest message. However, for this last strategy, it is necessary to parse an entire GRIB file in the JobManager to know where each GRIB message starts in order to be able to group messages.

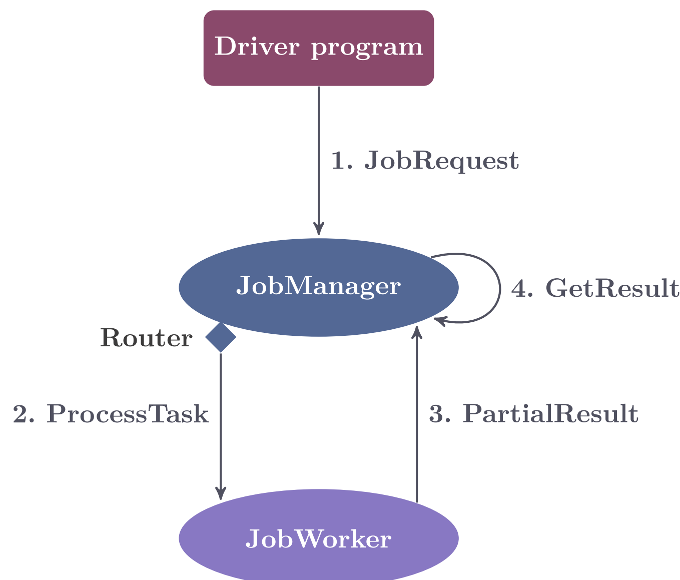
In relation to the deploy, the application code is compiled and built in the master machine and subsequently the executable JAR file is distributed manually to the worker machines, and executed from a shell script called from the master machine.

4.3 Using cluster-aware routers

With the goal of evaluating the performance of our proposal regarding the other options provided by Akka, a second approach is considered. Contrary to our proposal that uses the Manager-Worker pattern, in this second approach, the tasks are not sent on-demand. The JobManager sends the tasks to JobWorker actors from the beginning of the execution, at the *default* behavior. Thus, the built-in routers employed in this approach were the RoundRobinPool, RandomPool and AdaptiveLoadBalancingPool. The latter uses cluster metrics to distribute the tasks among the worker actors, and in this case, three different cluster metrics were utilized: CPU, Heap and the Mix metric. These metrics were collected using the Hyperic Sigar[3], which is a native package library, that provides more accurate metrics (VERNON, 2015). We chose to work with the Pool router type, since these are recommended for CPU-intensive tasks (VERNON, 2015), and also because for this type, it is not necessary to create the routees beforehand on cluster nodes, but rather they are created on-demand.

Since, we use the built-in routers from Akka, the design for this variant is simpler and the Figure 4.4 shows how this alternative approach works. In the same way as described in the Section 4.2, the *Driver program* loads the configurations from a file where these are stored, obtains the file names of all the GRIB files to be processed from HDFS, and creates the JobManager actor. Then, the list of file names is sent through *JobRequest* message to the JobManager actor.

Figure 4.4: Design: Using different cluster-aware routers.



In the JobManager actor, a router will be created and will be in charge of distributing the *ProcessTask* messages to the JobWorkers in line with the logic of the router

[3]https://github.com/hyperic/sigar

specified, and, after entrusting the processing of the first GRIB file, either as file-based or message-based, the JobManager changes its behavior from *default* to *processing*.

The *ProcessTask* messages are then enqueued in each JobWorker mailbox, and processed one by one by each JobWorker. At this point, the JobWorker actor uses our Scala client module to process the GRIB messages, and the results of each task computed are combined to calculate the partial result. Next, a *PartialResult* message containing this result is sent to the JobManager actor. This process will continue until the JobManager has entrusted all the tasks and the JobWorker actors have processed all the messages received. Once this occurs, the JobManager changes its behavior from *processing* to *finish* and sends a *GetResult* message itself to compute the final result from all of the partial results received previously.

4.4 Using Metadata

After analyzing the initial experimental results, a difference was observed between the file-based and message-based strategies. This difference refers to the time spent by the Driver program and JobManager actor. And, since the performed work for both strategies is indeed the same in the Driver program, the time difference occurs in the JobManager actor.

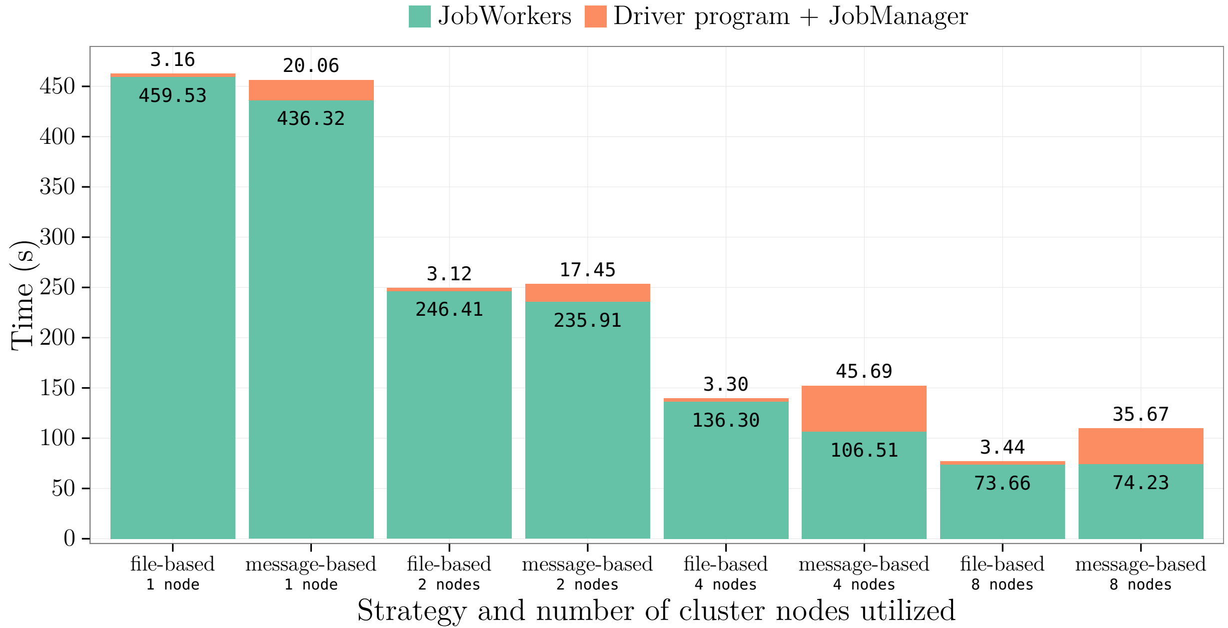
This can be seen in the Figure 4.5, which contains on each bar the maximum time spent by JobWorker actors and the time utilized by the Driver program and JobManager. The x-axis represents the strategy utilized (both file-based and message-based), as well as the number of cluster nodes used, the y-axis presents the time in seconds. Those results are taking into consideration the best performance achieved for our proposal, both for the file-based and for the message-based strategies.

As can be seen in the figure, there was a noticeable difference between file-based and message-based strategy. For the file-based, each set of the operations performed by driver program and JobManager actor took about 3 seconds, whereas this time for the message-based was longer.

This difference is caused by the message-based strategy when the JobManager selects the group of messages that will be sent to the JobWorkers, because the JobManager actor must perform a pre-processing over a GRIB file in order to identify the beginning of the first GRIB message per each group of messages. This operation is performed once per GRIB file and on-demand in the JobManager. Obviously, this was reflected in a loss of performance.

For that reason, an improvement over our initial proposal is explored, this optimization consists of storing information about GRIB messages per file, such as the byte location where each message starts in the GRIB file, a 3-tuple consisting of the discipline

Figure 4.5: Initial results for file and message-based strategy using our proposal.



code, the discipline category code, and the name code, which allow for the identification of the GRIB parameter contained in each message.

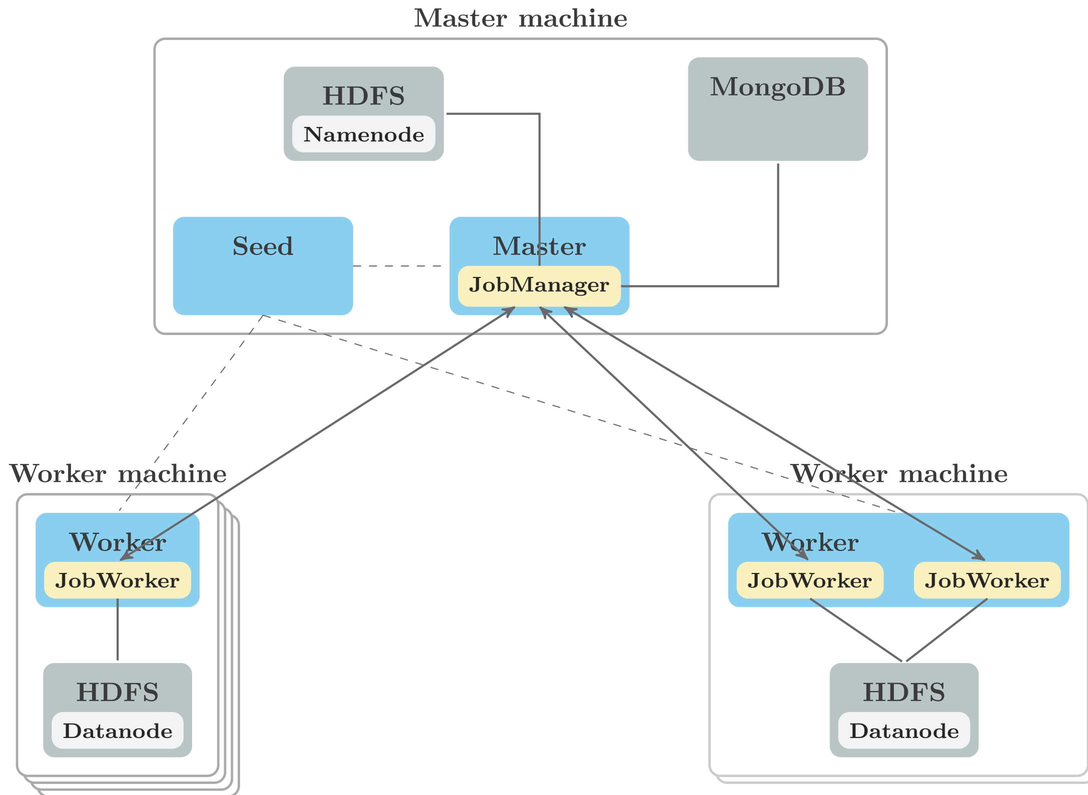
Since we only plan to store just metadata, and then perform simple queries. We choose to use MongoDB[4], one of the most popular NoSQL databases, because in Parker et al. (PARKER; POE; VRBSKY, 2013), MongoDB achieved a better performance than SQL Server – a SQL relational database – for simple queries. Likewise, Abramova et al. (ABRAMOVA; BERNARDINO, 2013) use the YCSB – Yahoo! Cloud Serving Benchmark (COOPER et al., 2010) to evaluate two NoSQL databases, both MongoDB and Cassandra. And, MongoDB obtained better results for read operations in not very large data sets, about 100.000 rows. Furthermore, MongoDB allows the storage of data as "documents" of key-value pairs.

Thus, initially a pre-processing over all the GRIB files is performed to obtain the necessary information, then, all the metadata obtained is stored in MongoDB, so it is no longer necessary to parse each GRIB file during the selection of the next set of messages to know where a message starts, since this information would be available in advance.

Figure 4.6 illustrates the complete architecture of our proposal with the improvement, thus, in this variant, the JobManager performs a query by GRIB file to MongoDB, to select only those messages that fulfil the condition set out (for example, 2 specific parameters of 27 total parameters), then divide it into groups of messages, which will be sent to the JobWorker to be processed. The result for a query that requires just the parameters Temperature (`0,0,0`) and Relative humidity (`0,1,1`) is shown in Figure 4.7, it is found in a

[4]https://www.mongodb.com/

Figure 4.6: Architecture: Using metadata stored in MongoDB.



JSON format, and contains the GRIB file name, its length in bytes, and for each message belonging to the GRIB file, the start byte location and the 3-tuple mentioned above.

Due to Scala was used in the implementation of our proposal, at this stage it was also necessary to use Casbah (CASBAH, 2016) to interact with MongoDB from Scala. Casbah was basically used to insert and query data in MongoDB.

4.5 Using Apache Spark

In order to compare our proposal with one of the main frameworks currently existing for big data processing, another approach is presented in this section. For this case, the first approach contemplated was to use our Scala client module to parse and decode the GRIB files but saving all the decoded data in the RDDs. However, in this step, we obtained JVM heap exceptions and, since the unpacked data values occupy much more memory than available, we have opted to store only the results obtained from the Scala client module in the RDDs, in the same way that it is performed in our proposal.

When Apache Spark is run in a distributed environment, it may be run mainly on Apache YARN or Apache Mesos, which allow scheduling and management the resources in the cluster. And, since Apache Hadoop was already installed to use HDFS to store the dataset, we decided to use Apache YARN, so that, the Spark application is submitted to Apache YARN in cluster mode making it necessary to execute the YARN

Figure 4.7: Sample: JSON output document from MongoDB.


```
1  {
2    "_id": ObjectId("58342d6345ef6f5c0c253271"),
3    "filename": "JULES_BRAMS05km_2015091200_2015091200.grib2",
4    "lenght": NumberLong(70801650),
5    "msgs": [
6      {
7        "start": NumberLong(2696887),
8        "params": "0,0,0"
9      },
10     {
11       "start": NumberLong(5091511),
12       "params": "0,0,0"
13     },
14     {
15       "start": NumberLong(13100719),
16       "params": "0,0,0"
17     },
18     ...
19 }
```


ResourceManager on the Master machine, and the YARN NodeManagers on the Worker machines.

The first, keeps a register of the NodeManagers and schedules and manages the available resources, named containers. The NodeManagers manage the processes running in containers, and these containers run the application-specific tasks.

This implementation with Apache Spark uses the SparkContext's binaryFiles function to read the GRIB binary data stored in HDFS, which returns a map of RDDs for each GRIB file, where the key is the path to file, and the value is a PortableDataStream object, the latter being used as input for our Scala client module. Thereafter, a new map of RDDs is defined containing the file name as key, while that the map values would be the results obtained from the Scala client module, and finally the final results are calculated from these map values.

Apache Spark is employed to perform the tasks in the cluster, however, this implementation to process the data is totally dependent upon our Scala client module to parse and decode the GRIB messages. In this case, RDDs just store the result of the Scala client module and not the entire unpacked data values, otherwise there would be JVM heap exceptions, since Apache Spark needs to have the data stored in memory before performing an operation, this works perfectly when the data fits in memory, but when this does not happen, Apache Spark have a loss of performance or even a JVM heap exception (GU; LI, 2013).

5 EXPERIMENTAL EVALUATION

In this chapter the experimental evaluation of our proposal for processing the GRIB files and the other approaches explored in this work are described. Section 5.1 describes the experimental environment in which the experiments were performed. Section 5.2 details the dataset utilized in the experiments. Section 5.3 presents the factors considered in the experimental methodology. Finally, Section 5.5 presents the experimental results and an analysis of these results.

5.1 Experimental Environment

The experiments were conducted on a cluster in Microsoft Azure, which was selected since, in Roloff et al. (ROLOFF et al., 2012), Azure gave better results in relation to cost efficiency and performance compared to Amazon and Rackspace cloud services, and also because the GPPD research group currently counts with access to an account in Microsoft Azure. The experimental cluster was composed of nine A6 instances, which has 4 virtual cores, 28 GB of memory and 285 GB local disk for instance, running a 64-bit Ubuntu 14.04 LTS operating system. An instance was dedicated for the Master machine, and the other eight instances were assigned to the Worker machines.

It must be highlighted that, the dataset used in the experiments was stored in HDFS in just four virtual machines as illustrated in Figure 5.1. HDFS was configured with a replication factor of two and with a data block size of 128 MB.

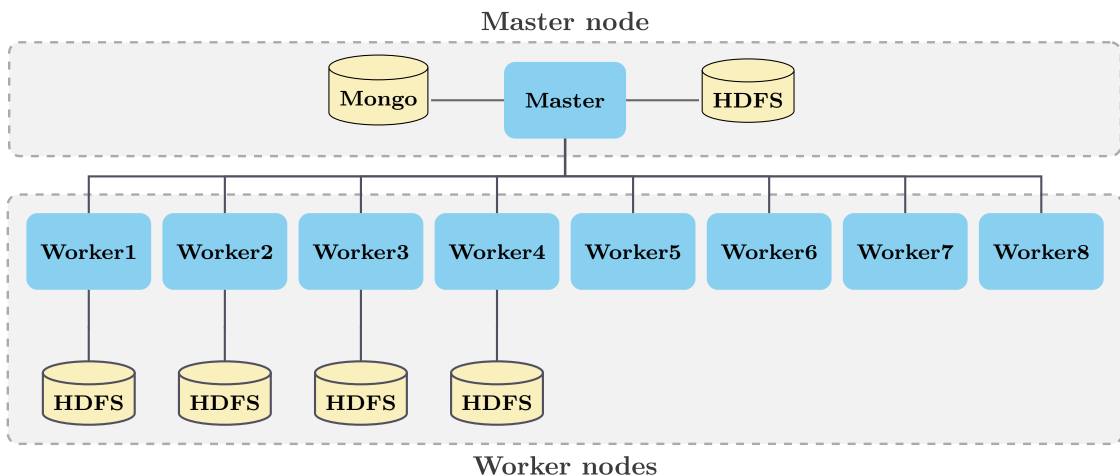


Figure 5.1: Cluster in Microsoft Azure.

As mentioned previously in Section 4.2, during the experiments using Akka, the Seed node, Master role node and NameNode were performed in the Master machine, and the Worker role node and DataNode were performed in the Worker machines. Likewise, since one of the experiments was executed using Apache Spark which utilized Apache

YARN, it was necessary to run the ResourceManager in the Master machine, and the NodeManager in the Worker machines.

The versions of the software utilized in this work are listed in Table 5.1.

Table 5.1: Overview of the software components.

Software	**Version**
Apache Maven	3.3.3
SBT	0.13.9
Java	1.8.0_60
Scala	2.11.7
Akka	2.4.4
Casbah	3.1.1
MongoDB	3.2.10
Apache Hadoop	2.6.0
Apache Spark	1.3.1

5.2 Dataset Description

The dataset used in the experiments contains 184 GRIB files, and each of these files is composed of 175 GRIB2 messages, giving a total of 32200 GRIB2 messages, all of this for just for three months. These files were obtained from the CPTEC's FTP[1], which were organized into two folders per day, one of them released at 00:00 and the other at 12:00.

The total size in a raw format is 13 GB, but if it is transformed into text-plain containing the unpacked data values, it will have a size of 378 GB, which greatly exceeds the amount of available memory in the cluster utilized. It is important to note that, this dataset size was chosen to speed up the execution of the experiments. *Latitude-longitude* is the grid type found for all the GRIB messages in the dataset, where each file is composed of 14 GRIB2 messages using the *Complex packing* and 161 GRIB2 messages with the *Complex packing and spatial differencing* data packing method. Most messages contain 1,584,353 data points or values each. Likewise, upon analyzing the entire dataset 27 different types of parameters stored in the GRIB2 messages were found. Thus, for example, *Temperature*, *Relative humidity*, *Total precipitation* had in total 3864, 3496 and 184 occurrences respectively.

5.3 Experimental Methodology

The experimental methodology considers several factors to process the GRIB files, and these are presented as follows.

[1]ftp://ftp1.cptec.inpe.br/modelos/io/tempo/regional/BRAMS05km/grib/

5.3.1 Configurations

As noted in Section 4.2, several JobWorker (JW) actors may be deployed for Worker role node (WN). Table 5.2 shows the number of JobWorker actors that there will be for each set of Virtual Machines in the experiments.

Thus, the first configuration `1WN-4JW` means that for each virtual machine 1 Worker role node will be deployed and each one will have 4 JobWorker actors, then one VM will have a total 4 JWs, two VMs will have 8 JWs, four VMs will use 16 JWs and finally eight VMs will contain 32 JWs. In the same way, the configuration `4WN-2JW` means that for each virtual machine 4 Worker role nodes will be deployed and each one will have 2 JobWorker actors, giving a total of 8 JW for 1 VM, for 2 VMs there will be 16 JWs, for 4 VMs there will be 32 JWs and for 8 VMs there will be 64 JWs in total. This configuration `2WN-8JW` means that for each virtual machine 2 Worker role nodes will be deployed and each one will have 8 JobWorker actors, thus one VM will have a total of 16 JW, two VMs will have 32 JWs, four VMs will use 64 JWs and finally eight VMs will contain 128 JWs. It is worth mentioning that each Worker role node is performed in a single JVM.

Table 5.2: Configurations: Total number of JobWorkers per set of VMs.

Configuration	1 VM	2 VMs	4 VMs	8 VMs
1WN - 4JW	4	8	16	32
2WN - 2JW	4	8	16	32
4WN - 1JW	4	8	16	32
1WN - 8JW	8	16	32	64
2WN - 4JW	8	16	32	64
4WN - 2JW	8	16	32	64
2WN - 8JW	16	32	64	128
4WN - 4JW	16	32	64	128

In short, the total number of JobWorker actors per set of virtual machines may be calculated by Equation 5.1, where wn is the number of Worker role nodes deployed, jw represents the number of JobWorker actors for each Worker role node, and vm represents the number of virtual machines running in the cluster.

$$Total\ of\ JobWorker\ actors = \#wn \times \#jw \times \#vm \tag{5.1}$$

5.3.2 Grouping messages

Furthermore, for each one of the combinations mentioned in Subsection 5.3.1, the processing of the GRIB files can be accomplished in two different ways, such as *file-based* and *message-based*. The first indicates that each JobWorker actor will process an entire file with its **175** GRIB2 messages, and in the second way the JobManager will split the file into groups of messages, where this number can vary from **22**, **44** or **88** messages, and this additional step is completed in the JobManager actor. These numbers were selected to divide each file as equitably as possible. Thus, if the number of messages to be processed by each task is 22, one GRIB file could be processed for 8 JobWorkers, seven of them will process 22 messages and one would receive 21 messages to process, and if the number of messages by each task is 44, 4 JobWorkers could process all the messages of a GRIB file, three of them will receive 44 messages to process and one would process 43 messages, and finally if the number of messages to be processed by each task is 88, one GRIB file would be processed for 2 JobWorkers, one of them would process 88 messages and the other would process 87 messages.

5.3.3 Mechanisms

In order to evaluate the performance of our proposal in relation to the other alternatives provided by Akka, all the experiments were run using 6 mechanisms: Our proposal referred to as **aGrib**, which uses the Manager-Worker pattern. The round-robin and random router strategy denoted here as **rr** and **ra** respectively, which use the RounRobinPool and RandomPool router. The other three mechanisms use the AdaptiveLoadBalancing-Pool router, the **cmc** which uses the CPU cluster metric, **cmh**, that utilizes the JVM Heap cluster metric, and the **cmm** which combines CPU, load and JVM Heap cluster metric.

Thus, for each mechanism 32 combinations in total were performed, which vary the number of JobWorker actors by Worker role node, being these: **`1WN-4JW`**, **`2WN-2JW`**, **`4WN-1JW`**, **`1WN-8JW`**, **`2WN-4JW`**, **`4WN-2JW`**, **`2WN-8JW`**, **`4WN-4JW`**, and each one of these varying the size of the data to be processed for JobWorker, 22, 44, 88 (message-based) and 175 messages (file-based).

5.3.4 Varying dataset

Since there are different applications that perform queries considering just one or a few parameters of the entire dataset, as already mentioned in Chapter 3, the experiments in this work were addressed in two scenarios. A first scenario, that calculates the average value for the **Temperature** and **Relative humidity** parameters, and a second scenario that calculates the average value for all parameters in the dataset.

5.4 Baseline

Initially, the execution time of our Scala module client serial version was measured without using the Akka Toolkit and with the dataset stored in disk, the execution time for the first scenario, which only takes into account two parameters was 1851.48 seconds and for the second scenario which takes into account all the parameters the execution time was 5258.92 seconds. In addition to this, it also measured the execution time used by GRIB API (version 1.13.0) for processing the GRIB files, thus, for the first scenario the runtime was 2139.81 seconds and for the second scenario it was 5537.92 seconds, however the GRIB API not only calculates the average but also performs other operations over the data values, such as, maximum, minimum and standard deviation. For this reason, we decided to use the serial version of the ScaGrib module as our baseline, to guarantee a fair comparison.

5.5 Experimental Results

All the experiment results in this section show the average over 5 runs. The results related to the first scenario were run in a cluster located in West Europe Azure region, whereas the results related to the second scenario were run in a cluster located in North Europe Azure region. Besides, the initial experiments were considering the following configurations: `2WN-2JW`, `4WN-1JW`, `2WN-4JW`, `4WN-2JW`, `2WN-8JW` and `4WN-4JW`, which were performed in the same cluster, whereas the configurations: `1WN-4JW` and `1WN-8JW` were performed on different clusters.

In addition, for each task processed for a JobWorker actor, we logged the number of received tasks, received messages, processed messages, the time spent reading the GRIB file from HDFS, the time spent processing the GRIB messages with our module client, the worker machine where was deployed the actor. Likewise, we use the sysstat[2], a suite of monitoring tools for the Linux operating system, which was used to collect information about of the CPU and memory usage in an interval of 5 seconds. All this was done with the aim to understand better the why of some results.

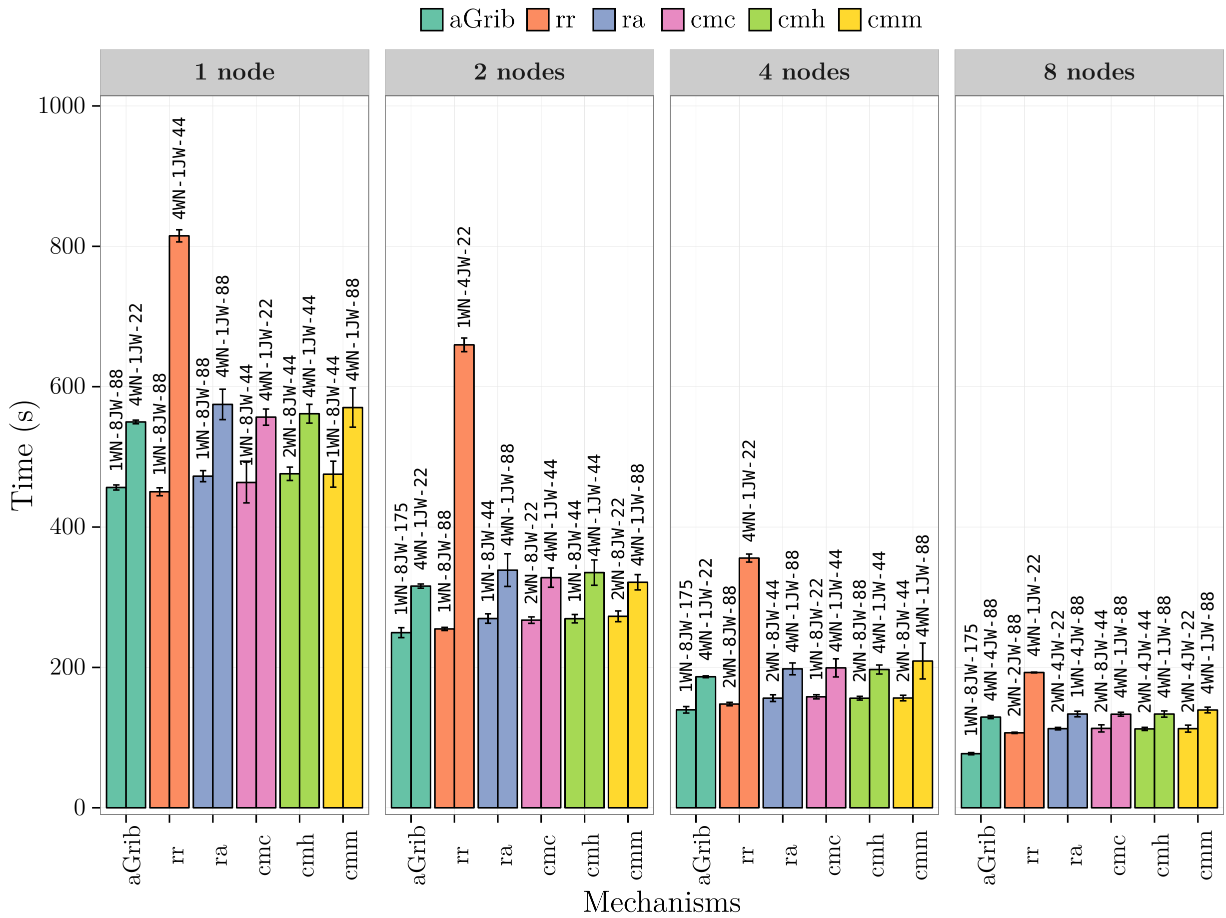
It is worth noting that the performed experiments with the mechanisms: rr, ra, cmc, cmh and cmm using the file-based strategy did not complete their execution correctly. Likewise, experiments using the `1WN-16JW` configuration failed during their execution, due to the JVM run out of memory, so it was not possible to have 16 JobWorker actors per Akka worker node.

[2]https://github.com/sysstat/sysstat

5.5.1 First scenario: Using two parameters

This subsection covers the results of processing just two parameters of the dataset. The total execution times considering the best and worst cases for each mechanism using 1, 2, 4 and 8 worker virtual machines are shown in Figure 5.2, where the error bars represent the standard deviation. Nonetheless, all the execution times for this scenario are presented in the Figures B.1, B.2, B.3, and B.4, which are found in Appendix B.

Figure 5.2: First scenario: Best and worst configuration for each mechanism.



Thus, taking into account only when **1 worker virtual machine** was used. The **rr** obtained the best performance from all mechanisms using the `1WN-8JW-88` configuration with a time spent of 450.28 seconds. The same had a gain of 3.08% and 80.98% in relation to its second best and worst cases `2WN-4JW-88` and `4WN-1JWW-44` respectively. This latter configuration also had the worst time from all mechanisms.

The second best time from all mechanisms resulted from using **aGrib**, which spent a time of 456.38 seconds. The `1WN-8JW-88` configuration obtained the best performance, which had a gain of 1.38% and 20.46% with regard to the second best and worst cases `2WN-4JW-175` and `4WN-1JW-22` respectively.

In addition to this, for all mechanisms, the difference between the best and second best cases ranged from 0.33% to 3.08%. Likewise, the best performance achieved for aGrib had a loss of 1.34% in relation to the rr mechanism, and a gain of 3.54%, 1.56%, 4.28% and 4.15% in relation to the best cases reached for the ra, cmc, cmh and cmm mechanisms respectively.

Nevertheless, when we examined the total execution times and the time spent by the JobWorker actors, a difference in time between message-based aGrib and the other approaches including file-based aGrib was found. As mentioned in Section 4.4, we refer to this time difference as the time spent by the Driver program and JobManager actor. Thus, this time ranged from 11 to 29 seconds for the message-based aGrib, being the configurations `1WN-4JW`, `2WN-2JW`, `4WN-1JW`, `1WN-8JW`, `2WN-4JW` and `4WN-2JW` which took a longer time, whereas it was about 3 seconds for the file-based aGrib, and it ranged from 3 to 8 seconds for the other mechanisms, being the configuration `4WN-4JW-88` which took more time. Taking into consideration only the best cases, this time was 20.06 seconds for the message-based aGrib. And, it was between 3 and 4 seconds for the best cases of the other mechanisms.

The reason for this time increment in the message-based aGrib was because the JobManager actor needs to parse a GRIB file with the aim to know the beginning of the first message of each group of messages that will be sent in response for each JobWorker request, all of this producing a bottleneck in the JobManager. This was not the case for the file-based aGrib, where the list of GRIB files to be processed are known beforehand. When the message-based strategy for the other five approaches is used, the job carried out by the JobManager is performed from the beginning of the processing in parallel to the JobWorker operations, as described in Section 4.3.

As can be seen in the figure, the rr mechanism had a poor performance with the `4WN-1JW-44` configuration. This was because the messages, that correspond to Temperature and Relative humidity variables have a fixed position in all the GRIB files, so there are actors that receive a number of messages but do not necessarily decode all the GRIB messages, it may even be the case that the actor does not decode any of the received messages. Similar behavior was seen with the configurations: `1WN-4JW`, `2WN-2JW`, `4WN-1JW`, `1WN-8JW`, `2WN-4JW`, `4WN-2JW` when processing groups of 22 messages and the configurations: `1WN-4JW`, `2WN-2JW`, `4WN-1JW` for tasks grouping 44 messages.

For example, both Table 5.3 and Table 5.4 detail the number of received and processed messages using the rr and aGrib with the `4WN-1JW-22` configuration. In both tables, the first iteration for rr and aGrib is taken in consideration. Thus, for the rr mechanism (see Table 5.3), 3 actors received 8096 tasks and 1 actor received 7912, the first three processed 2576, 1840 and 2944 messages respectively while the latter processed 0 messages. The actor named as `c4` did not decode any messages, but utilized a time of 90.36 seconds to parse the assigned GRIB messages to know whether it corresponded with either of the two parameters (Temperature or Relative humidity) to be evaluated, while

the actor characterized as c3 used a time of 715.78 seconds to parse and decode the GRIB messages.

Table 5.3: Number of received and processed messages using round-robin for the 4WN-1JW-22 configuration.

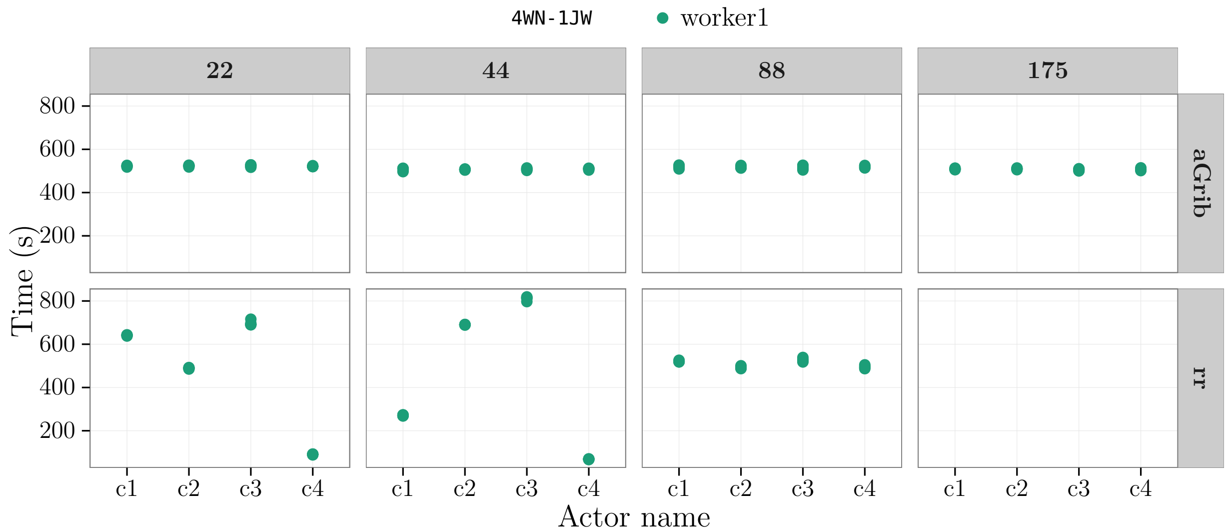
Actor	Received	Processed
c1	8096	2576
c2	8096	1840
c3	8096	2944
c4	7912	0

Table 5.4: Number of received and processed messages using aGrib for the 4WN-1JW-22 configuration.

Actor	Received	Processed
c1	8504	1805
c2	8309	1803
c3	7879	1830
c4	7508	1922

On the other hand, for aGrib (see Table 5.4), all the actors processed a similar number of messages and had an uniform time for parsing and decoding the GRIB messages, which indicates that there was a better balancing of the tasks using aGrib. This can be clearly seen in Figure 5.3, which depicts the time spent by the JobWorker actors, for both rr and aGrib mechanisms using the 4WN-1JW configuration.

Figure 5.3: Time spent by the JobWorkers using the 4WN-1JW configuration with rr and aGrib.



When **2 worker virtual machines** were used. The best performance from all mechanisms was obtained using the 1WN-8JW-175 configuration of **aGrib**, where the time spent for this configuration was 249.53 seconds. Likewise, the gain with regard to the second best and worst cases of this mechanism was 1.53% and 26.59%, 2WN-4JW-175 and 4WN-1JW-22 respectively.

The second best mechanism resulted from using the **rr** with the 1WN-8JW-88 configuration, which had a gain of 0.98% and 158.83% with respect to its second best and

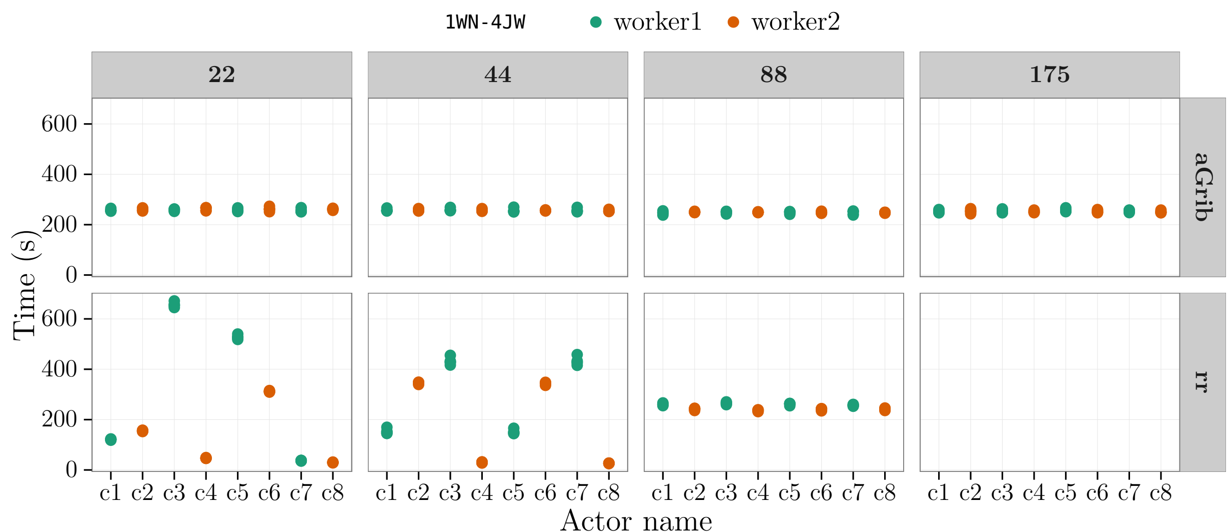
worst cases, `2WN-4JW-88` and `1WN-4JW-22`, respectively. This latter configuration had the worst performance from all mechanisms.

The difference between the best and second best cases ranged from 0.05% to 1.98% for all mechanisms. In addition, the best performance of aGrib reached a gain of 2.13%, 8.06%, 7.15%, 7.97% and 9.34% in relation to the best cases achieved for the rr, ra, cmc, cmh and cmm mechanism respectively.

As in the previous case, there was a time difference between the message-based aGrib and the file-based aGrib for the work performed by the JobManager. This time ranged from 11 to 29 seconds for the message-based aGrib, using the configurations `1WN-4JW`, `2WN-2JW`, `4WN-1JW`, `1WN-8JW`, `2WN-4JW` and `4WN-2JW` which took a longer time, whilst the file-based aGrib was about 3 seconds, and it was between 3 and 15 seconds for the other mechanisms, using the `4WN-4JW-88` configuration which took more time. Considering only the best cases, this time was 17.45 seconds for the message-based aGrib, and between 3 and 6 seconds for the best cases of the other mechanisms.

Once again, in this specific case, the rr mechanism had a poor performance when used the configurations: `1WN-4JW`, `2WN-2JW`, `4WN-1JW` and the tasks processed group of messages of 22 and 44. This was because there were actors that processed a significantly smaller number of messages (even none at all) than other actors. This can be seen in Figure 5.4, which shows the number of messages processed per task upon using the `1WN-4JW` configuration and the time spent by the JobWorker actors for both aGrib and rr.

Figure 5.4: Time spent by the JobWorkers of the rr and aGrib mechanism using the `1WN-4JW` configuration.



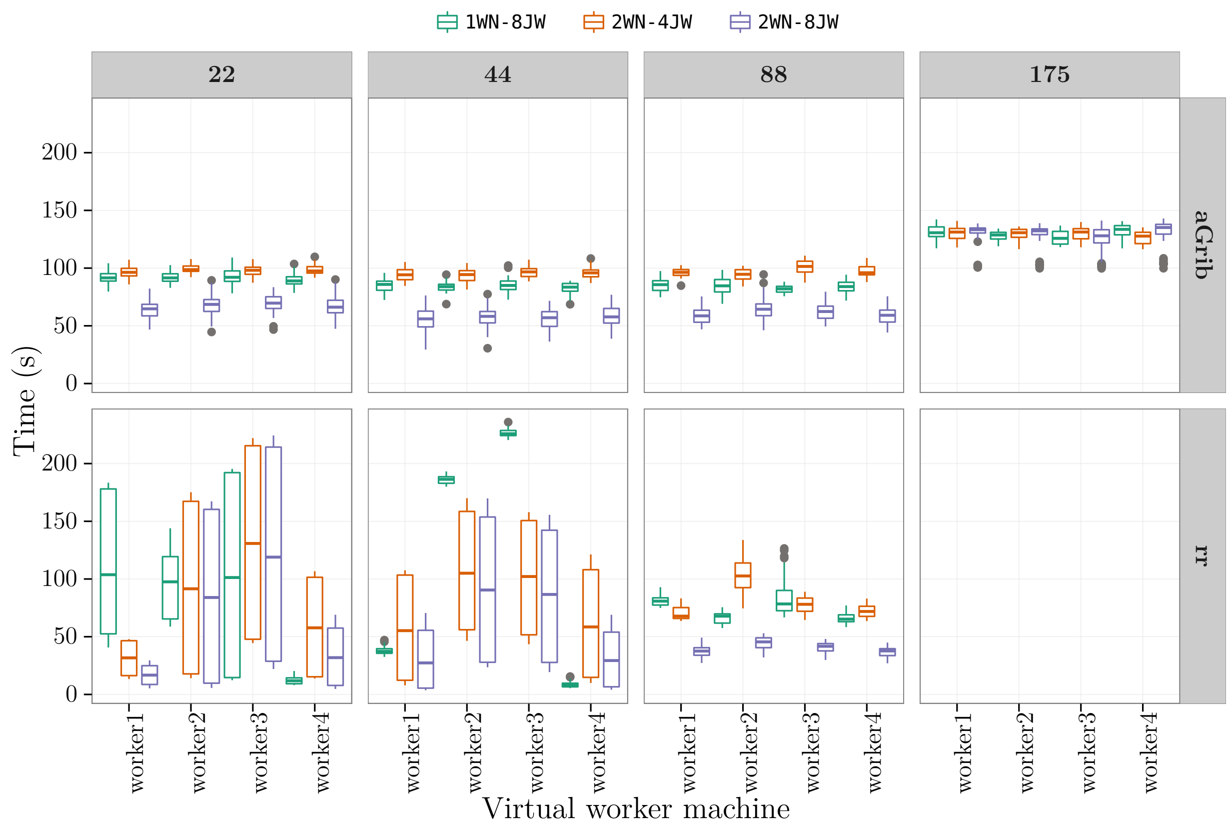
When **4 worker virtual machines were utilized**. The **aGrib** obtained the best performance among all mechanisms using the `1WN-8JW-175` configuration and an execution

time of 139.60 seconds, which had a gain of 1.25% and 33.74% regarding its second best and worst cases, `2WN-4JW-175` and `4WN-1JW-22` respectively.

The second best mechanism was the **rr**, which achieved its best time using the `2WN-8JW-88` configuration, while its second best time was achieved using the `2WN-4JW-88` configuration with a time increment of 0.19% and its worst performance was using the `4WN-1JW-22` configuration with an increase in time of 140.74%. The worst case of this mechanism had, once again, the worst case among all mechanisms.

In general for this case, for all mechanisms, except aGrib, the gain between the best and second best cases was less than 1%. The best time of the aGrib reached a gain of 5.88%, 11.91%, 13.28%, 11.84% and 12.03% in relation to the best cases obtained for the rr, ra, cmc, cmh and cmm mechanisms respectively.

Figure 5.5: Time spent by the JobWorkers of the rr and aGrib mechanism using `1WN-8JW`, `2WN-4JW`, `2WN-8JW` in 4 VMs.



When four worker machines were used, the time difference between the total execution time and the maximum time utilized by the JobWorker actors had a substantial increase for all mechanisms. Thus, this time was between 20 and 81 seconds for the message-based aGrib, using the configurations `1WN-8JW`, `2WN-4JW`, `4WN-2JW`, `2WN-8JW` and `4WN-4JW` which took a longer time, whereas it was about 3 seconds in the case of file-based aGrib and ranged from 3 to 96 seconds for the other mechanisms, using the same configuration

as the message-based aGrib which had the increase in time, most notably when using the `2WN-8JW` configuration.

Concerning only the best cases, this time was 45.69 seconds for the message-based aGrib. The best case for rr used a time of 96.40 seconds by the JobManager operations, however, this was not the case for the other rr configurations, such as `2WN-4JW-88` and `4WN-1JW-22` where the operations used 33.32 (second best case) and 4.77 (worst case) seconds respectively. And, it ranged from 37 to 56 seconds for the other mechanisms.

Recall that the `1WN-8JW-175` configuration using the file-based aGrib achieved the best performance, whereas the `2WN-8JW-88` configuration achieved the best time upon using the rr mechanism, and the `2WN-4JW-88` configuration obtained the best performance for the message-based aGrib. Figure 5.5 shows the time spent by the JobWorker actors for those 3 configurations using four worker machines. As can be seen in this figure, the `2WN-8JW-88` configuration of rr had a better performance among the other configurations, however, the operations carried out in the Driver program and JobManager actor have a large influence in the total runtime.

We chose some of the results as similar as possible to compare the time utilized by the JobWorker actors for both aGrib and rr mechanisms upon using the `2WN-8JW-88` configuration, which has 64 JobWorker actors in total. The information shown in Table 5.5 details the number of received and processed messages for each actor, the file reading time from HDFS and the time spent for processing GRIB messages using our client module per each JobWorker. As can be seen in the table, the time taken with aGrib was considerably greater than those with the rr for a similar number of received and processed messages. This could explain why the JobWorkers had better performance upon using the rr mechanism with the `2WN-8JW-88` configuration. However, a further analysis is necessary to determine the reason of this result which may be due to the fact that although they have similar tasks, the number of data points may not be the same.

Table 5.5: Time spent for the processing the GRIB files using the `2WN-8JW-88` configuration.

Mechanism	**Machine**	**Actor**	**Received**	**Processed**	**HDFS Time**	**GRIB Time**
rr	worker1	c50	435	95	262	30224
rr	worker1	c58	435	95	265	28541
aGrib	worker3	c43	435	95	436	65736
rr	worker2	c13	528	126	567	41267
rr	worker3	c9	528	126	1105	41316
aGrib	worker3	c49	525	120	462	56099
aGrib	worker4	c14	527	124	3079	59183
rr	worker4	c48	522	114	290	34763
aGrib	worker4	c32	522	114	469	63248

When **8 worker virtual machines were used**. The **aGrib** reached the best performance from all mechanisms using the `1WN-8JW-175` configuration with a time spent

of 77.11 seconds. This configuration had a gain of 2.46% and 67.63% with respect to its second best and worst cases, `2WN-4JW-175` and `4WN-4JW-88` respectively.

The second best mechanism was once again the **rr**, where its best performance resulted from using the `2WN-2JW-88` configuration. Nonetheless, it also had the worst performance in general when each task processed groups of 22 messages.

Thus, the best performance for this specific scenario was reached by the file-based aGrib, which obtained a gain of 38.54%, 46.24%, 46.73%, 45.78% and 46.25% in relation to the best cases achieved by the rr, ra, cmc, cmh and cmm mechanisms respectively. In addition to this, the time difference between the best and second best cases for the rr, ra, cmc, cmh and cmm mechanisms was 0.25%, 0.71%, 1.67%, 0.77% and 0.17%, respectively.

When eight worker machines were used, there was also a time difference between the total runtime and the maximum time utilized by the JobWorker actors, thus, the time employed by the JobManager actor was between 31 and 81 seconds for the message-based aGrib, whereas it was between 3 and 4 seconds for the file-based aGrib, and it ranged from 4 to 85 seconds for the other mechanisms, giving the configurations `1WN-8JW`, `2WN-8JW` and `4WN-4JW` which took more time. And, taking into consideration only the best cases, this time was 35.67 seconds when the message-based aGrib was used, while it was 3.44 seconds when the file-based aGrib was used. On the other hand, the rr spent a total time of 30.57 seconds in the operations performed for the Driver program and the JobManager, whilst this time was between 40 and 64 seconds when the alternative approaches were used.

Figure 5.6 and Figure 5.7 display the average CPU usage as a percentage considering the use of 8 virtual machines for aGrib and round-robin, respectively. The CPU usage plotted is only of the worker machines and not of the master machine. The two figures contain all the configurations utilized, as well as the group of messages processed per task. As can be appreciated in both figures, the aGrib with the file-based strategy made better use of the resources, especially with the `1WN-8JW-175` configuration, regarding the other approaches experimented in this work. As can also be seen in the figures, configurations which required less worker role nodes, also used less resources at the deploying phase, as there were less JVMs to deploy, this also applied at the beginning of the processing for the message-based strategy. In addition to this, the first few seconds (3-4) were time employed by the Driver program to obtain the information of the GRIB files from HDFS, and subsequently the creation of JobWorker actors from the JobManager actor.

Figure 5.6: First scenario: CPU utilization for the aGrib using 8 worker machines.

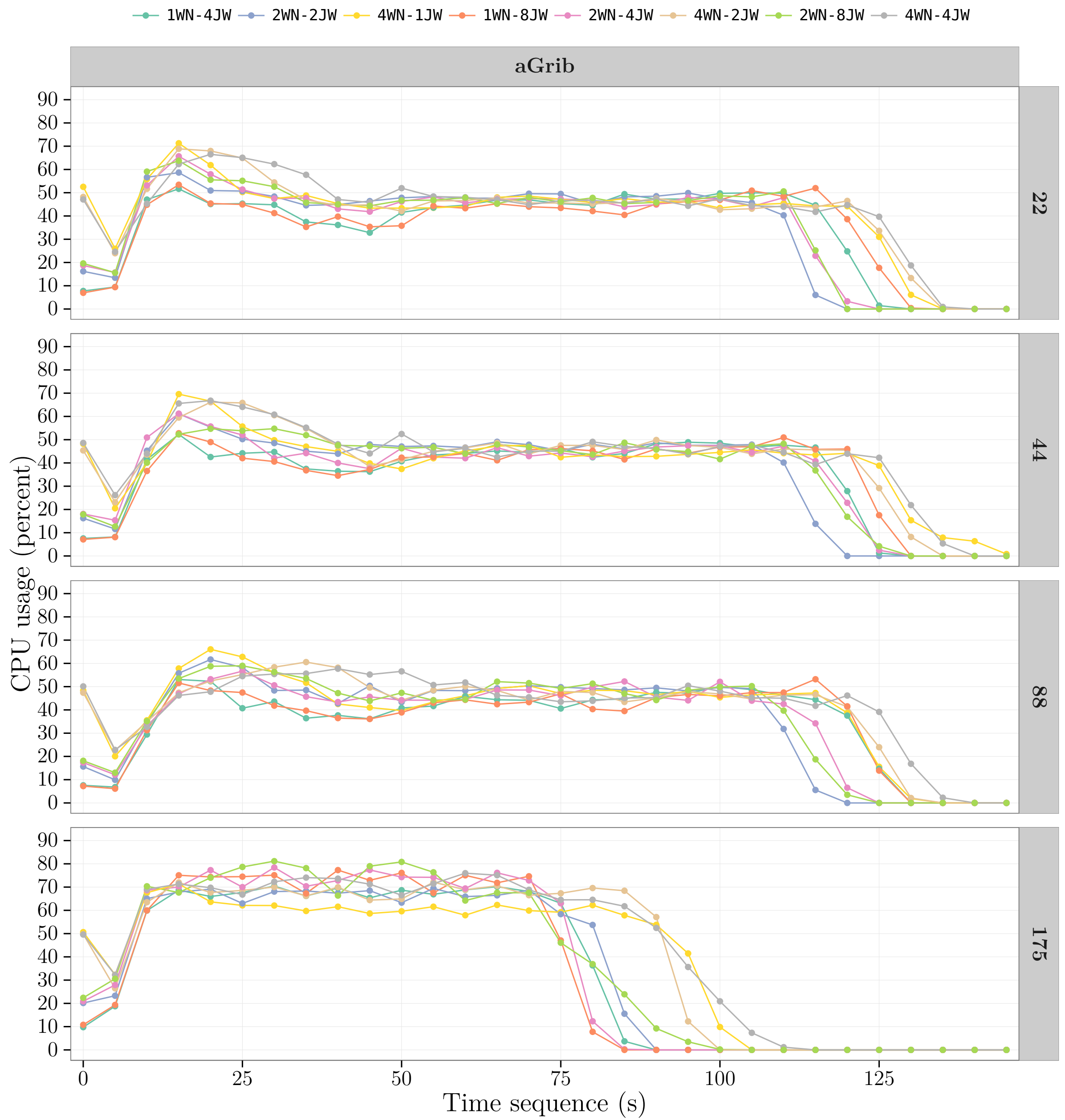


Figure 5.7: First scenario: CPU utilization for the round-robin using 8 worker machines.

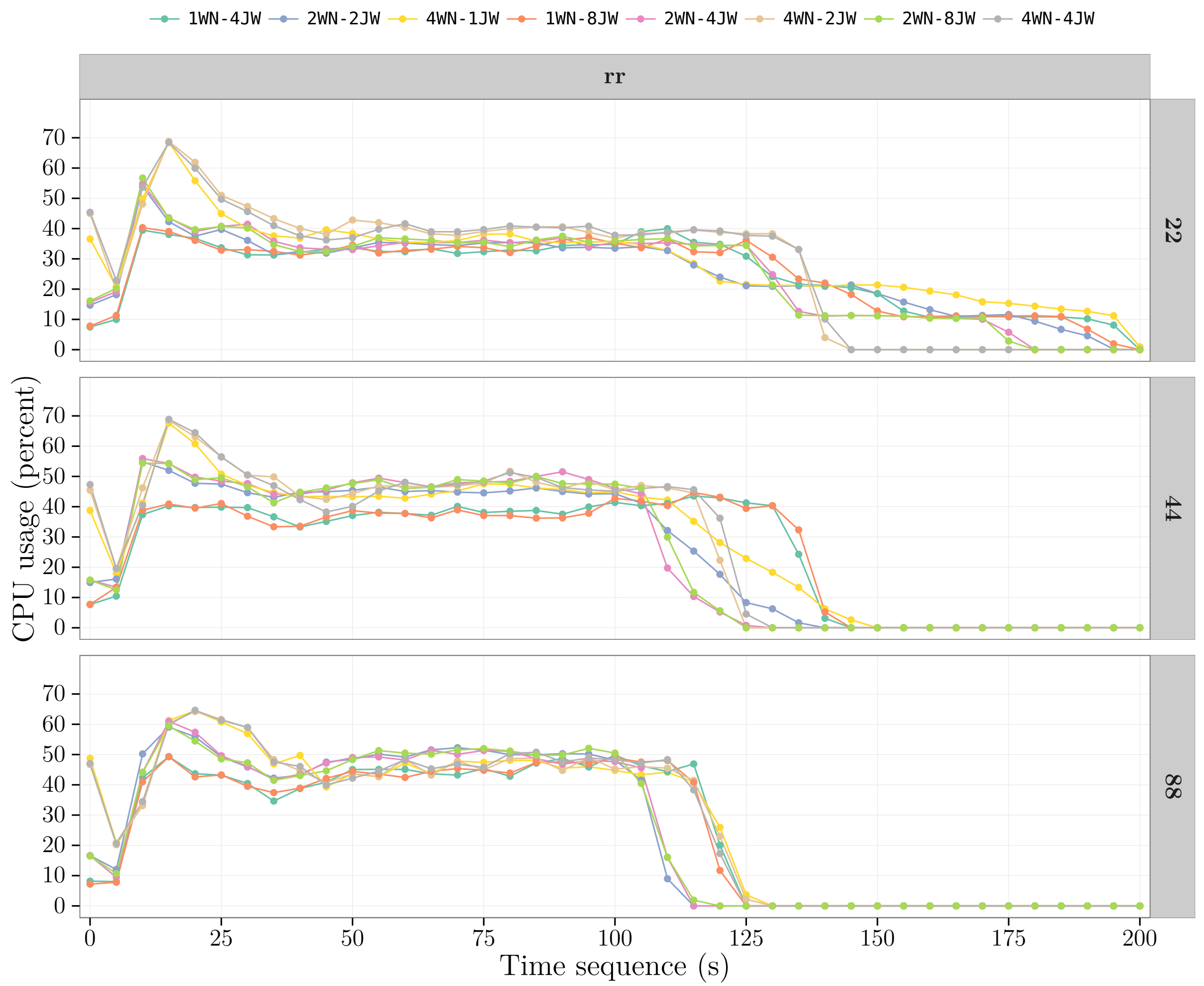


Figure 5.8 depicts the average memory usage as a percentage over time by employing the aGrib. In the figure, it can be appreciated that the less worker role nodes we have, the less memory will be utilized, as indeed less memory will need to be allocated by a JVM than by two JVMs. Thus, configurations `1WN-4JW` and `1WN-8JW` maintained an approximate memory usage of 25%, whereas the configurations: `2WN-2JW`, `2WN-4JW`, `2WN-8JW` maintained an approximate peak memory usage of 45%, and the configurations: `4WN-1JW`, `4WN-2JW`, `4WN-4JW` reached a peak approximate memory usage of 60%. The same behavior was observed in the other mechanisms.

Figure 5.8: First scenario: Memory utilization for the aGrib using 8 worker machines.

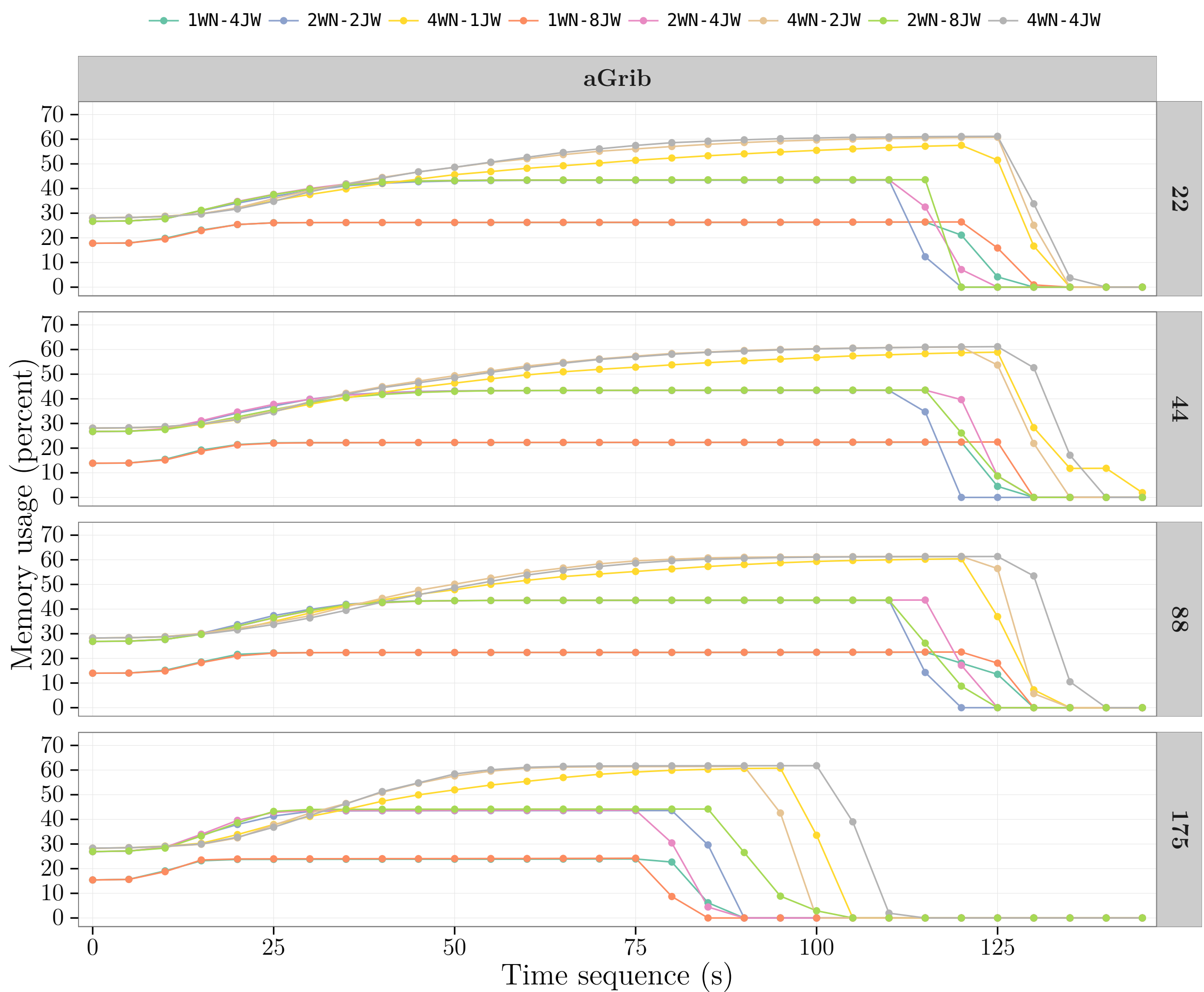
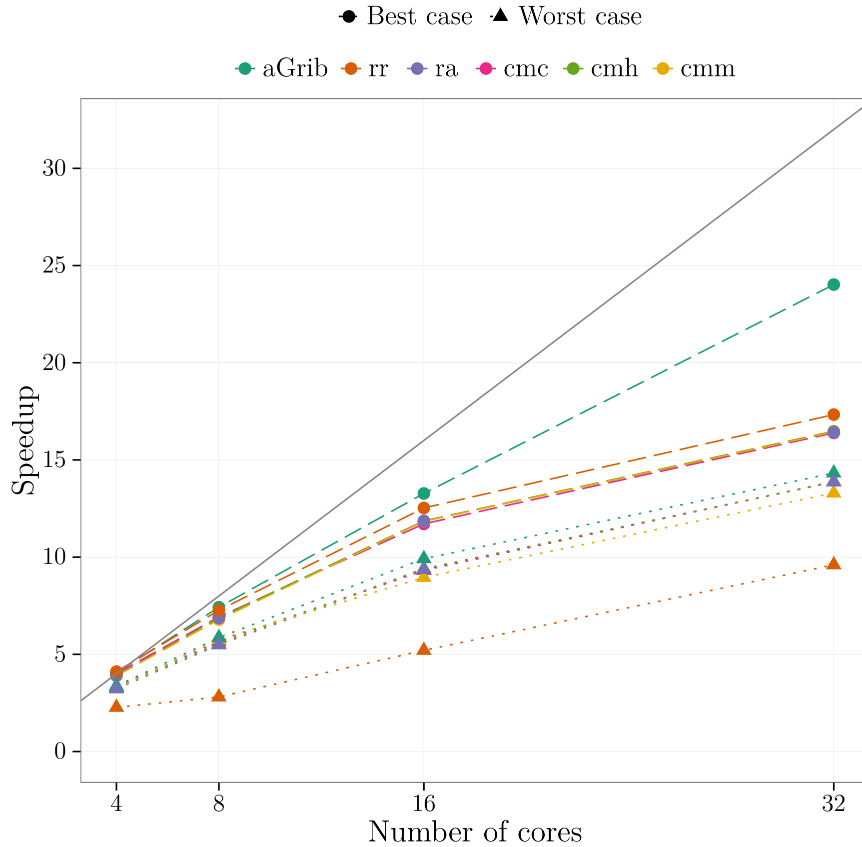


Figure 5.9 shows the speedup achieved in the experiments for all mechanisms, taking into account the best and worst cases. The baseline utilized was our Scala client module, which is used internally for aGrib and the other mechanisms utilized in the experiments. Thus, considering the best cases, results show that increasing the number of cluster nodes causes all of the mechanisms to obtain good scalability, especially our

proposal. However the speed gain diminishes slightly after 4 VMs for aGrib, and this gain diminishes considerably for the other mechanisms.

Figure 5.9: First scenario: Best and worst speedup for each mechanism.



Summary: The processing of the GRIB files with aGrib obtained the best performance using the `1WN-8JW` configuration. Thus, when 1 VM was used, it was better at processing groups of 88 GRIB messages per task, whereas the file-based strategy had better results upon using 2, 4 and 8 VMS. This means that for our proposal, it was better to have 8 JobWorker actors running in just one worker role node. Nonetheless, our proposal did not achieved the best performance upon using 1 VM, this was due to the fact that aGrib with the `1WN-8JW-88` configuration utilized a longer time in the JobManager actor compared with the best case obtained using round-robin.

In relation to the message-based strategy, the aGrib and rr mechanisms achieved better results by processing tasks with groups of 88 GRIB messages, whereas the cmc, cmh and cmm mechanisms had better results by processing groups of 22 and 44 GRIB messages per task, this indicates that those which used the adaptive load balancing presented better performance with smaller tasks.

When 1 and 2 worker virtual machines were used, the time spent for the rr, ra, cmc, cmh and cmm mechanisms in the Driver program and JobManager actor was minimum

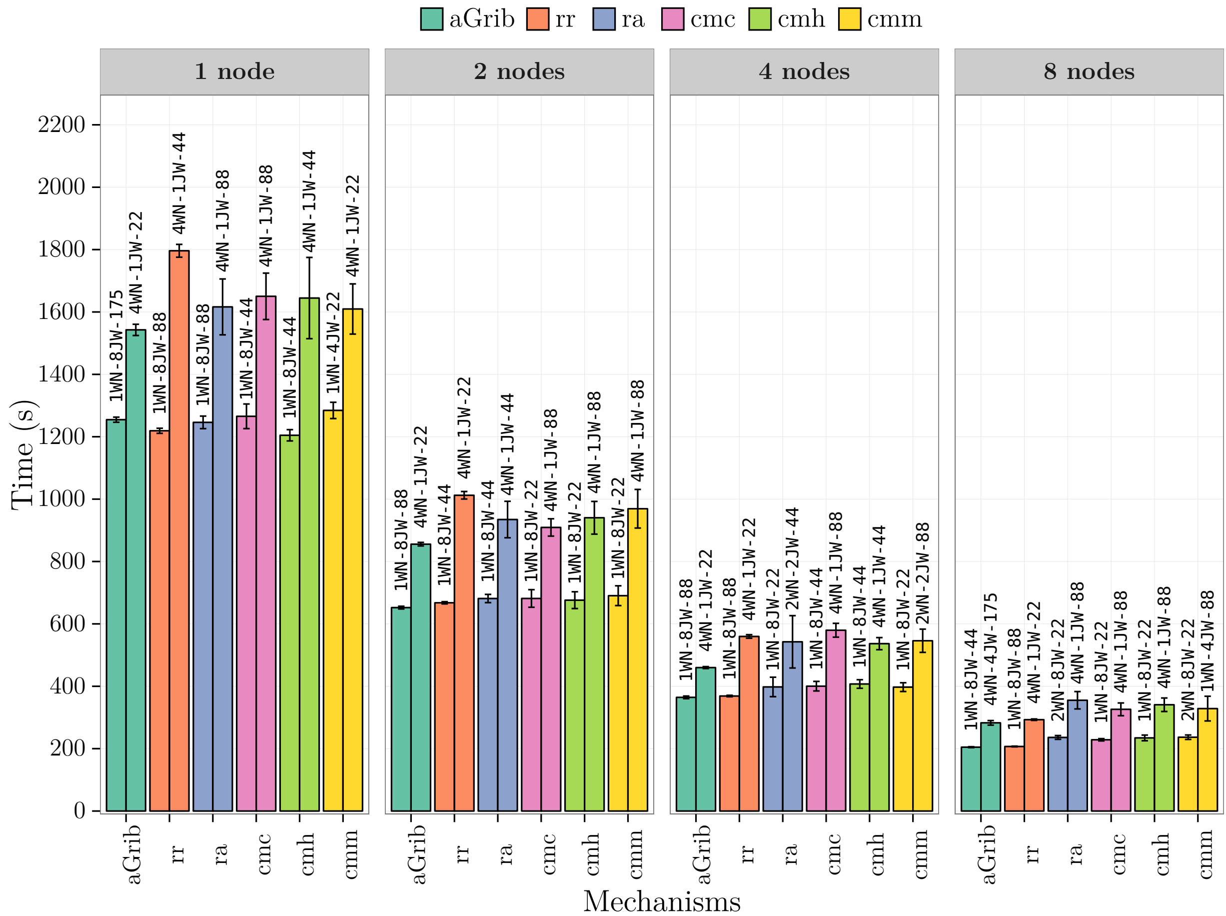
in relation to the utilized for the message-based aGrib. Likewise, the `1WN-8JW` configuration achieved better results for these cases.

On the other hand, when 4 and 8 worker virtual machines were used, the time spent by the JobManager actor experienced a significant increase, especially for the configurations: `1WN-8JW`, `2WN-8JW` and `4WN-4JW`. This happened because there were a greater number of JobWorker actors running in the cluster, so the tasks performed by the JobWorker actors were accomplished faster than the task distribution by the JobManager actor itself. Likewise, by using 2 Akka worker nodes presented better times and we observed that if only the time spent by the JobWorker actors is taken into account, the `2WN-8JW` configuration obtained better performed the work in a shorter time, however the total execution time was influenced by the time used by the distribution of tasks in the JobManager actor.

5.5.2 Second scenario: Using all parameters

In this scenario, all parameters into the dataset are evaluated. The total run times considering the best and worst cases for each mechanism upon using 1, 2, 4 and 8 worker virtual machines are shown in Figure 5.10, besides the error bars represent the standard deviation. Nonetheless, all the run times for this scenario are detailed in the Figures B.5, B.6, B.7, and B.8, which are found in Appendix B.

Figure 5.10: Second scenario: Best and worst configuration for each mechanism.



Thus, when **1 worker virtual machine** was used. The best performance from all mechanisms was reached by the **cmh** using the `1WN-8JW-44` configuration with a time spent of 1204.70 seconds. This configuration had a gain of 2.20% and 36.52% in relation to its second best and worst cases, `1WN-8JW-22` and `4WN-1JW-44`, respectively.

The second best mechanism resulted from using the **rr**, its best time was obtained using the `1WN-8JW-88` configuration, with a gain of 0.47% and 47.34% with respect to the second best and worst cases, the `1WN-8JW-44` and `4WN-1JW-44` configuration, respectively, the latter one was also the worst case among all mechanisms.

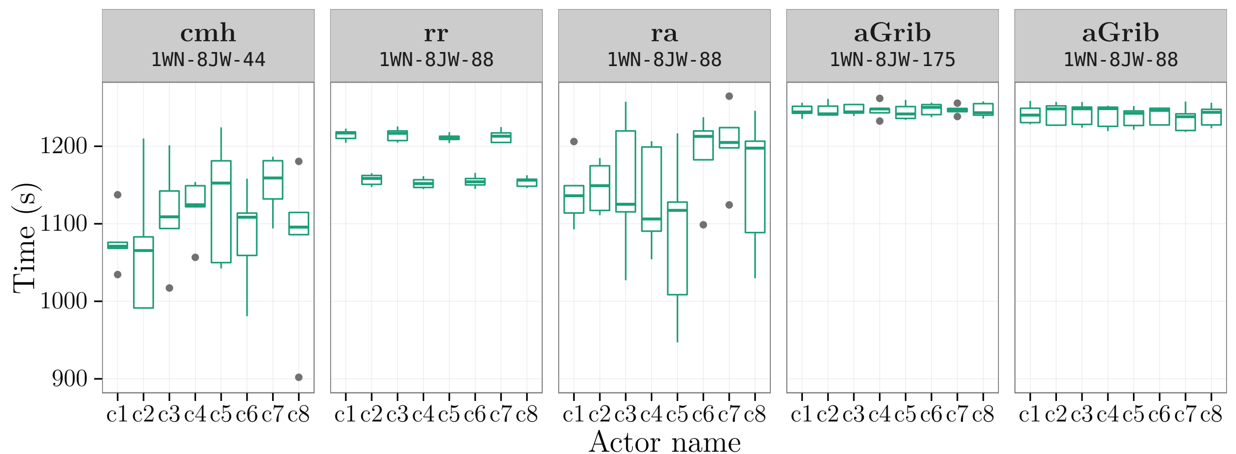
Regarding the **ra**, the best performance was obtained through the `1WN-8JW-88` configuration, followed by the `1WN-8JW-44` while the `4WN-1JW-88` configuration had the worst case. The best performance of this mechanism had a gain of 1.40% in relation to its second best case and 29.71% with regard to its worst case.

The **aGrib** reached its best performance using the `1WN-8JW-175` configuration with a time spent of 1254.68 seconds, which had a gain of 0.52% and 22.95% regarding the second best and worst cases, `1WN-8JW-88` and `4WN-1JW-22` respectively. The best performance according to aGrib had a loss of 3.98%, 2.84% and 0.69% in relation to the best cases for the cmh, rr and ra mechanisms respectively, and a gain of 0.86% and 2.37% with respect to the best cases reached by the cmc and cmm mechanisms.

Taking into consideration the best cases for these experimental results, the time used by the Driver program and JobManager actor in relation to the message-based aGrib was 17.64 seconds, whereas this time ranged from 3 to 4 seconds for the message-based rr, ra, cmc, cmh and cmm, and file-based aGrib.

Figure 5.11 illustrates the times involved by all JobWorker actors taking into account the best mechanisms upon using 1 virtual machine as worker. The JobManager sent 736, 368 and 184 tasks when using the `1WN-8JW-44`, `1WN-8JW-88` and `1WN-8JW-175`, respectively.

Figure 5.11: Second scenario: Best cases using 1 worker virtual machine.



As shown in Figure 5.11, the task distribution for aGrib was performed more uniformly than the other mechanisms. Thus, considering just the first iteration of the experiments for this scenario, each JobWorker upon using the `1WN-8JW-175` configuration processed 4025 GRIB messages, whereas with the `1WN-8JW-88` configuration processed between 4023 and 4027 GRIB messages in the case of aGrib. On the other hand, the cmh mechanism with its `1WN-8JW-44` configuration processed a number of GRIB messages ranging from 3541 to 4770, while the best case of the rr mechanism processed between 4002

and 4048 GRIB messages. However, aGrib did not achieved the best case among all mechanisms, so that a further analysis is required to determine the reason why the performed tasks using aGrib took a longer time for this scenario.

When just **2 worker virtual machines** were used. The best performance among all the mechanisms was obtained using **aGrib** with the `1WN-8JW-88` configuration, the time spent for this configuration was 652.19 seconds. This configuration had a gain of 1.19% and 31.18% with regard to its second best and worst performances, `1WN-8JW-44` and `4WN-1JW-22` respectively.

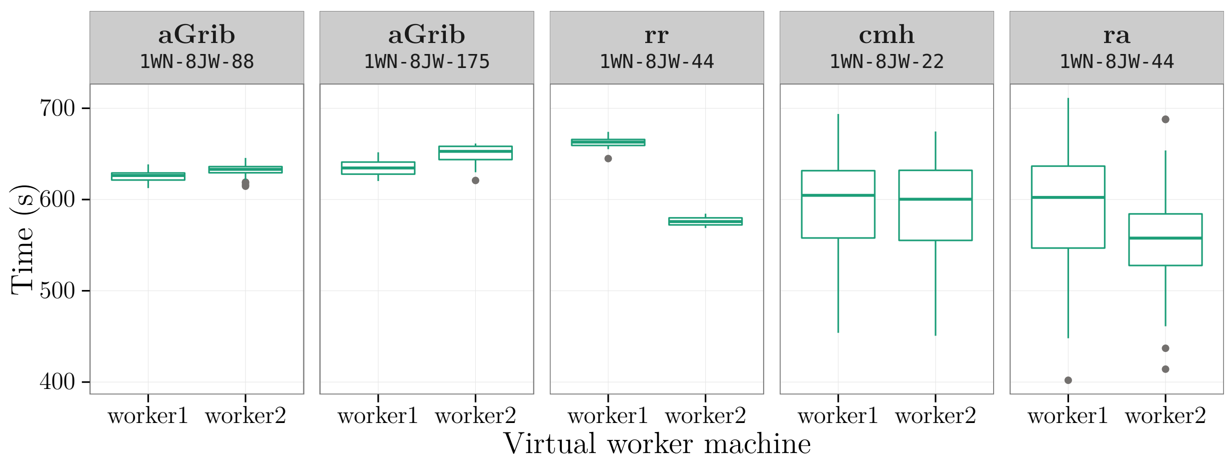
The second best mechanism resulted from using the rr with the `1WN-8JW-44` configuration, followed by the `1WN-8JW-88` configuration. The `4WN-1JW-22` configuration had the worst performance with an increase in time of 51.64% regarding the best time. It was the worst case among all the mechanisms using two virtual machines.

The best performance of aGrib reached a gain of 2.37%, 4.47%, 4.5%, 3.66% and 5.86% in relation to the best cases obtained for the rr, ra, cmc, cmh and cmm mechanism respectively. Additionally, the time difference between the best and second best case ranged from 0.37% to 2.69% considering all mechanisms.

For these results, it was also pointed out that there was a difference in the time taken by the JobManager actor between the message-based aGrib and the other approaches. This time utilized in the message-based aGrib was 14.34 seconds, whilst for the other mechanisms it was between 3 and 5 seconds.

Figure 5.12 depicts the times spent by the JobWorkers in the best cases when 2 worker virtual machines were used. This figure presents the file and message-based strategy for aGrib, as well as rr, cmh and ra mechanisms. As can be seen in the figure, regarding the time spent by the actor groups in worker1 and worker2, there is not a wide variation for aGrib, whilst for rr this variation is noticeable, and it is even higher for the other mechanisms.

Figure 5.12: Second scenario: Best cases using 2 worker virtual machines.



When **four virtual machines** were used. The **aGrib** achieved the best performance among all mechanisms using the `1WN-8JW-88` configuration with a time spent of 364.41 seconds, which had a gain of 0.84% and 26.20% in relation to its second best and worst cases, `1WN-8JW-44` and `4WN-1JW-22` respectively.

The second best mechanism was using the **rr** with its `1WN-8JW-88` configuration, which had a gain of 1.36% and 51.78% with respect to the second best and worst case of the rr mechanism, `1WN-4JW-88` and `4WN-1JW-22` respectively.

The best time of the aGrib achieved a gain of 1.19%, 9.18%, 9.85%, 11.76% and 9% regarding the best cases reached by the rr, ra, cmc, cmh and cmm mechanisms respectively. Likewise, the difference in time between the best and second best cases for all mechanisms was between 1.35% and 3.33%. Just as when 4 worker virtual machines were used in the first scenario, the second scenario also presented an increase in the time spent by the JobManager actor. Thus, the JobManager spent a time between 11 and 29 seconds when using the message-based aGrib, and it was between 3 and 20 seconds when the built-in routers from Akka were utilized. Additionally, if only the best cases are taken into consideration, the message-based aGrib spent a time of 14.52 seconds, whilst it ranged from 3 to 9 seconds for the other message-based mechanisms.

When **8 virtual worker machines** were used to process the entire dataset of GRIB files. The best performance from all mechanisms resulted from using **aGrib** with its configuration `1WN-8JW-44`, taking a time of 204.77 seconds. This configuration achieved a gain of 0.29% and 26.61% regarding the second best and worst cases of the aGrib mechanism, which were `1WN-8JW-22` and `4WN-1JW-22` respectively.

The second best mechanism was once again the **rr**, the `1WN-8JW-88` configuration obtained the best time for this mechanism. This configuration had a gain of 2.44% and 41.55% in relation to its second best and worst cases, `1WN-4JW-88` and `4WN-1JW-22` respectively.

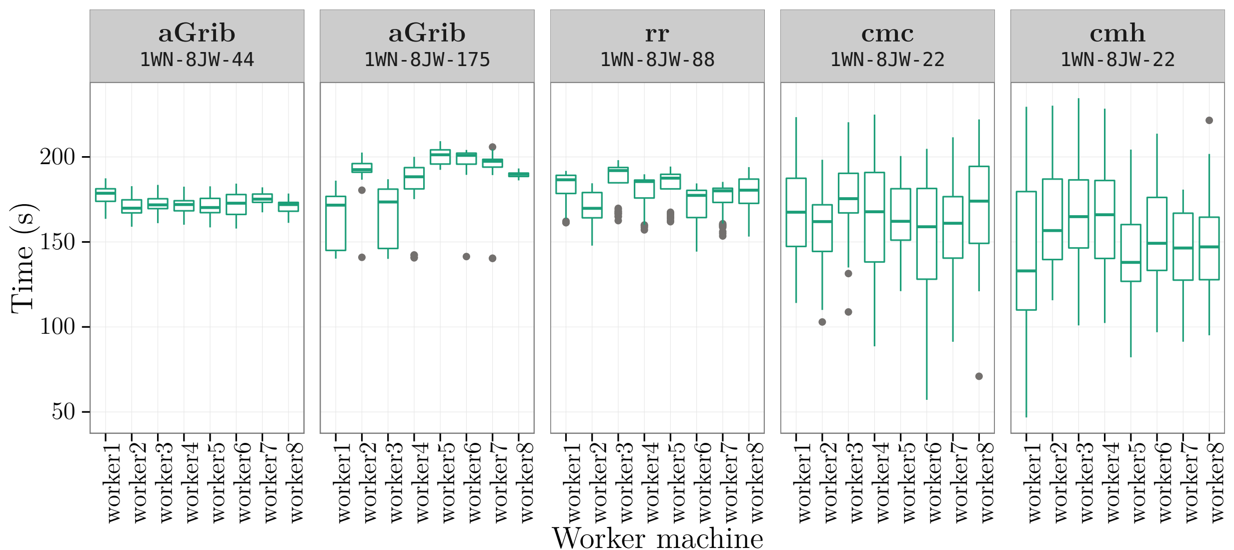
Thus, the best performance for this scenario was reached by the message-based aGrib strategy, which obtained a gain of 1.04%, 15.22%, 11.56%, 14.49% and 15.58% in relation to the best cases obtained for the rr, ra, cmc, cmh and cmm mechanisms respectively.

For this case, a time increment for the operations performed by the JobManager upon using the message-based strategy was also observed. Considering only the best cases, the time spent by the JobManager using the message-based aGrib was 20.84 seconds, as opposed to 3.40 seconds for the file-based aGrib. On the other hand, it took a time of 11.05 seconds when using rr, and this time ranged from 6 to 14 seconds when using the other mechanisms. Nonetheless, the `2WN-8JW-44` and `2WN-8JW-88` configurations of the aGrib mechanism spent 74.08 and 48.34 seconds, respectively, while the `2WN-8JW-88` configuration of the rr mechanism took 39.26 seconds.

The time used by the JobWorkers considering the best cases of the aGrib, rr, cmc and cmh mechanisms are shown in Figure 5.13. As this figure implies, the message-based

aGrib presents a more uniform distribution than the other approaches. And, considering the first iteration of the best cases obtained upon using 8 virtual machines, each of the 64 JobWorkers for aGrib with `1WN-8JW-88` configuration received between 10 and 13 tasks and processed between 438 and 569 GRIB messages, while that the JobWorkers for aGrib with `1WN-8JW-175` configuration received between 2 and 3 tasks and processed between 350 and 525 GRIB messages. On the other hand, the JobWorkers in the best case of the rr mechanism received between 5 and 6 tasks and processed between 435 and 528 GRIB messages. And, regarding the cmc and cmh mechanisms, their JobWorker actors processed 218-743 and 284-702 GRIB messages, respectively.

Figure 5.13: Second scenario: Best cases using 8 worker virtual machines.



As similarly presented for the first scenario, Figure 5.14 and Figure 5.15 display the average CPU usage as a percentage considering the use of 8 virtual machines for aGrib and round-robin, respectively. We can observe in those figures that the resource utilization had a more sustained use in comparison with the graphics shown for the first scenario (Figure 5.6 and Figure 5.7), except in the case of round-robin when tasks contained groups of 22 and 44 messages. Additionally, for this scenario, the average memory usage showed a behavior similar to that observed in the first scenario illustrated in Figure 5.8.

Figure 5.14: Second scenario: CPU utilization for the aGrib using 8 worker machines.

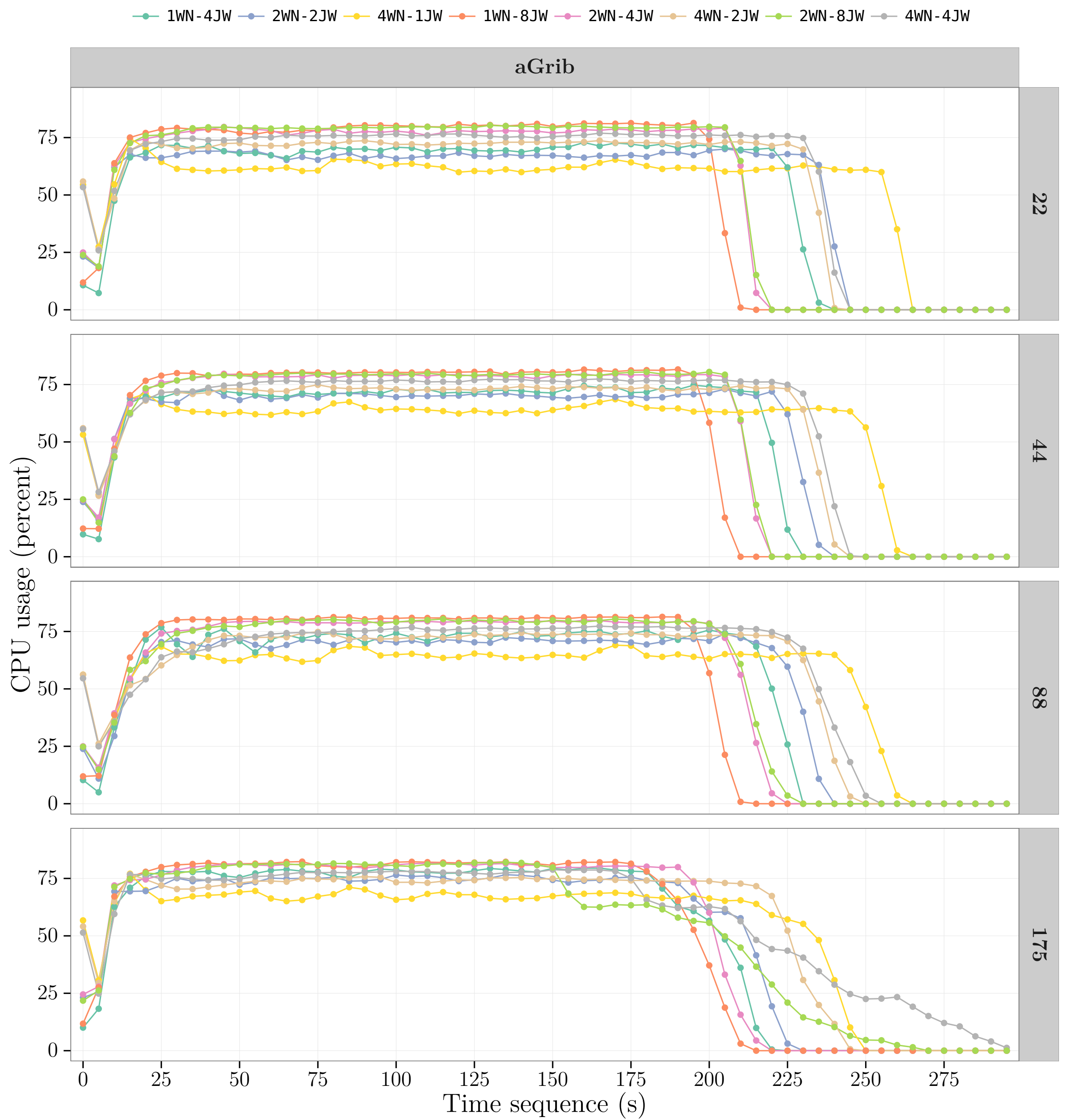
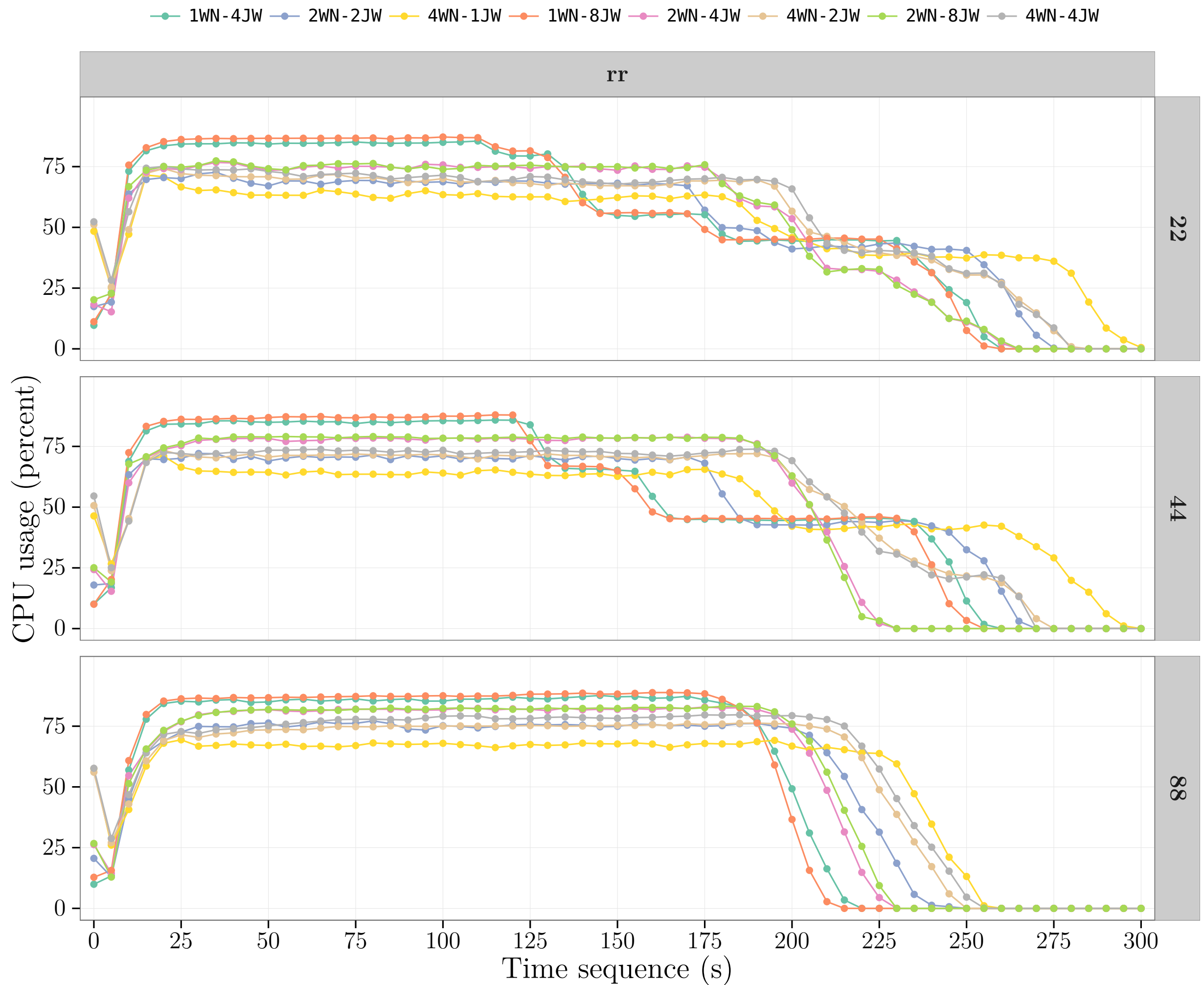


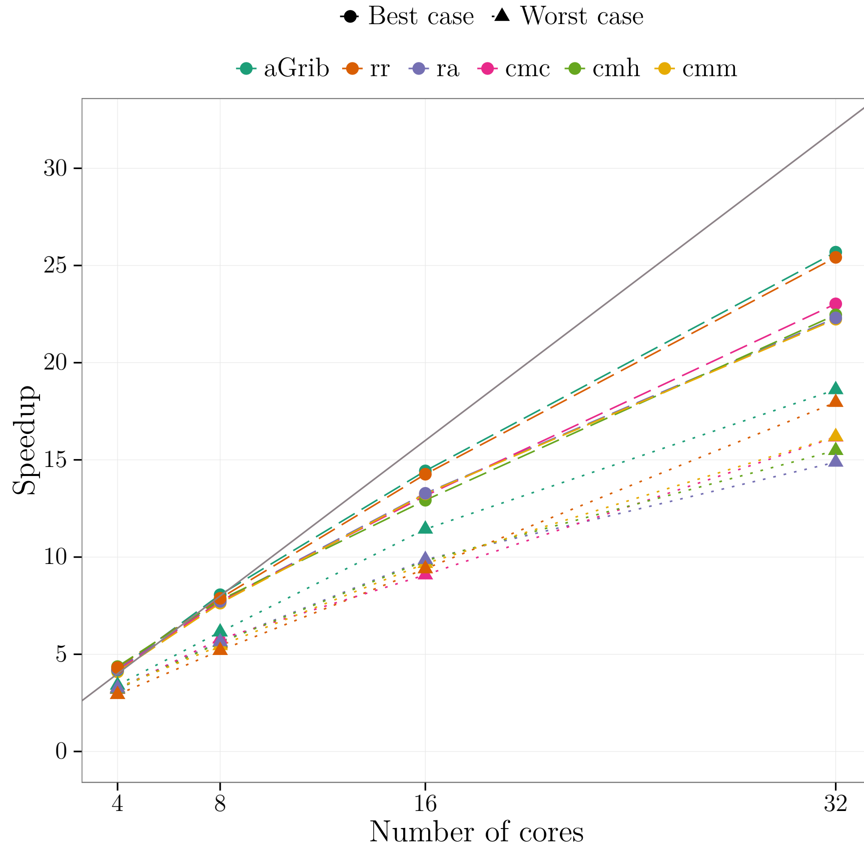
Figure 5.15: Second scenario: CPU utilization for the round-robin using 8 worker machines.



The speedup achieved taking into account the best and worst cases for this scenario is shown in Figure 5.16, the baseline utilized was also our Scala client module. As might be expected, processing the GRIB files considering the full set of parameters presented better results than the previous scenario for all mechanisms. A better scalability was reached by processing all the variables for all the mechanisms utilized. This is notable because there were no GRIB messages parsed needlessly for applying a search criteria on each message and there were also no unnecessary reads from HDFS.

Summary: For our proposal, the aGrib, the best cases were obtained using the `1WN-8JW` configuration, however when just 1 VM was used, the `1WN-8JW-175` was not the best case among all of the performed mechanisms. Nonetheless, aGrib achieved the best performance among all mechanisms upon using the `1WN-8JW-88`, `1WN-8JW-88` and `1WN-8JW-44` for 2, 4 and 8 VMs, respectively. A key point to note is that the aGrib and the round-robin

Figure 5.16: Second scenario: Best and worst speedup for each mechanism.



had a similar number to performed by the JobWorker actors. As in the first scenario, regarding the message-based strategy, the cmc, cmh and cmm mechanisms had better results by processing groups of 22 and 44 GRIB messages per task, obtaining a better performance with smaller tasks. Whereas the aGrib presented better times by processing groups of 88 GRIB messages per task when 1, 2 and 4 worker machines were used, and groups of 44 messages per task upon using 8 worker machines.

Furthermore, it was noted that in most cases, the best times for each mechanism were achieved by varying the size of the message groups processed by a task using the same configuration. So for example, when using 8 worker virtual machines and the message-based aGrib, the best performance was achieved by using the `1WN-8JW-44`, the second best case was `1WN-8JW-22`, and third best case was the `1WN-8JW-88` configuration.

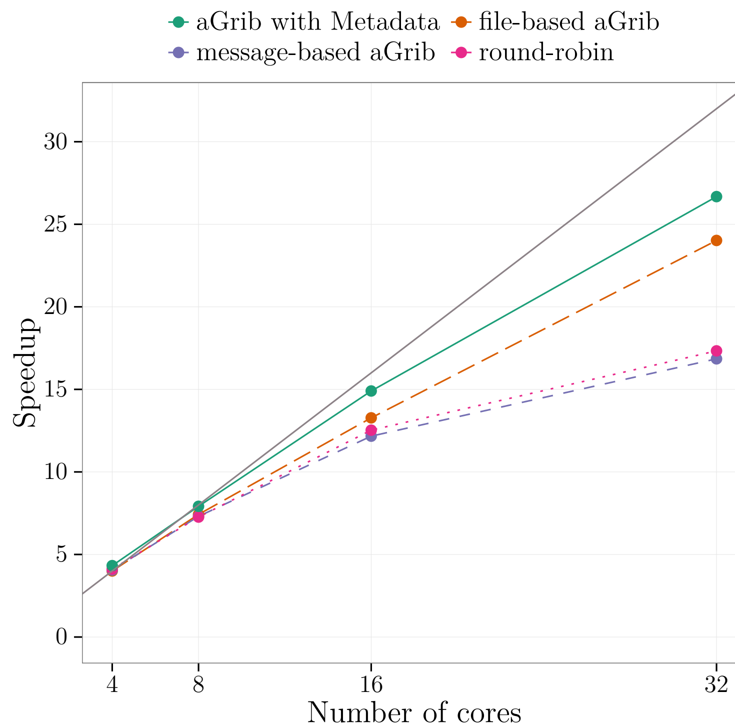
Additionally, when using 1, 2, 4, and 8 virtual machines as workers, there was a pattern in the time spent for processing the GRIB files. When there was the same number of JobWorker actors (4, 8 ,16) by worker virtual machine, it was better to have less worker role nodes by worker machine. Thus, `1WN-4JW` obtained a better performance than `2WN-2JW`, and this latter one was better than `4WN-1JW`. In the same way, `1WN-8JW` had a better performance than `2WN-4JW`, and this last one had was better than `4WN-2JW`. Consequently, `2WN-8JW` resulted in being better than `4WN-4JW`.

5.5.3 Using Metadata

The pre-processing of the GRIB files and the insertion of the metadata in MongoDB took a total time of 179.47 seconds, however it must be taken into account that this pre-processing is performed only once and can be used multiple times to process GRIB files.

Figure 5.17 shows the speedup achieved for aGrib without the optimization (file and message-based strategies), round-robin and aGrib with the optimization. The x-axis shows the number of cores utilized while that of the y-axis shows the respective speedup. As can be appreciated in the figure, while the speedup keeps growing for the message-based aGrib using metadata and the file-based strategy, this is not the case for the message-based aGrib without using metadata and round-robin, where the speedup did not hold that constant from 16 cores (4 VMs).

Figure 5.17: First scenario: Speedup achieved for aGrib, round-robin and aGrib using metadata.



Since, the first scenario only evaluates 2 – Temperature and Relative humidity – of 27 total parameters. And, there are 21 and 19 GRIB messages that corresponds with the Temperature and Relative humidity parameters, respectively, per GRIB file. The fact of processing a GRIB file in groups of 88 GRIB messages per task does not provide any benefit, since upon using the metadata, the maximum of messages that can be processed by a GRIB file would be 40 GRIB messages. So, the best times were obtained with the configurations: `1WN-4JW-44`, `1WN-8JW-44`, `1WN-8JW-22` and `1WN-8JW-22` when using 1, 2, 4 and 8

worker virtual machines, respectively. In addition, for this case, the time spent by the Driver program and the JobManager actor ranged from 4 to 5 seconds.

The **aGrib using metadata** obtained a better performance in contrast to the other three cases, thus, when 1 VM was used with regard to file-based aGrib and message-based strategy the gain was 8.06% and 6.58% respectively. With 2 VMs regarding the file-based strategy, the optimization achieved a gain of 6.68%, and when 4 and 8 VMs were used, the gain was slightly better, giving values of 12.35% and 11.07%, respectively. Nonetheless, in relation to aGrib and round-robin message-based when using 8 worker virtual machines, the gains became even greater, giving values of 58.3% and 53.88%, respectively. Although there is no a great gain between the best cases of the initial proposal and the optimization, there does exist a considerable gain compared to the results obtained using the message-based strategy, especially when we refer to the experiments run in a cluster of 4 and 8 virtual machines, which was the motivation for this improvement.

The Table 5.6 presents the execution time and the standard deviation obtained considering the aGrib using metatada, as well as the aGrib without the optimization for both file and message-based strategies and the round-robin which obtained the best performance when just 1 VM was used.

Table 5.6: First scenario: Summary of the results for aGrib, round-robin and aGrib using metadata

Mechanism	#VMs	Configuration	Exec. Time	(StDev)
aGrib with Metadata	1	`1WN-4JW-44`	428.19	(± 3.035)
file-based aGrib	1	`2WN-4JW-175`	462.69	(± 3.929)
message-based aGrib	1	`1WN-8JW-88`	456.38	(± 3.639)
round-robin	1	`1WN-8JW-88`	450.28	(± 5.657)
aGrib with Metadata	2	`1WN-8JW-44`	233.90	(± 3.395)
file-based aGrib	2	`1WN-8JW-175`	249.53	(± 7.061)
message-based aGrib	2	`1WN-8JW-88`	253.36	(± 3.852)
round-robin	2	`1WN-8JW-88`	254.86	(± 2.202)
aGrib with Metadata	4	`1WN-8JW-22`	124.25	(± 0.971)
file-based aGrib	4	`1WN-8JW-175`	139.60	(± 4.617)
message-based aGrib	4	`2WN-4JW-88`	152.19	(± 1.663)
round-robin	4	`2WN-8JW-88`	147.81	(± 2.410)
aGrib with Metadata	8	`1WN-8JW-22`	69.42	(± 1.122)
file-based aGrib	8	`1WN-8JW-175`	77.11	(± 1.545)
message-based aGrib	8	`2WN-2JW-88`	109.90	(± 1.513)
round-robin	8	`2WN-2JW-88`	106.82	(± 0.905)

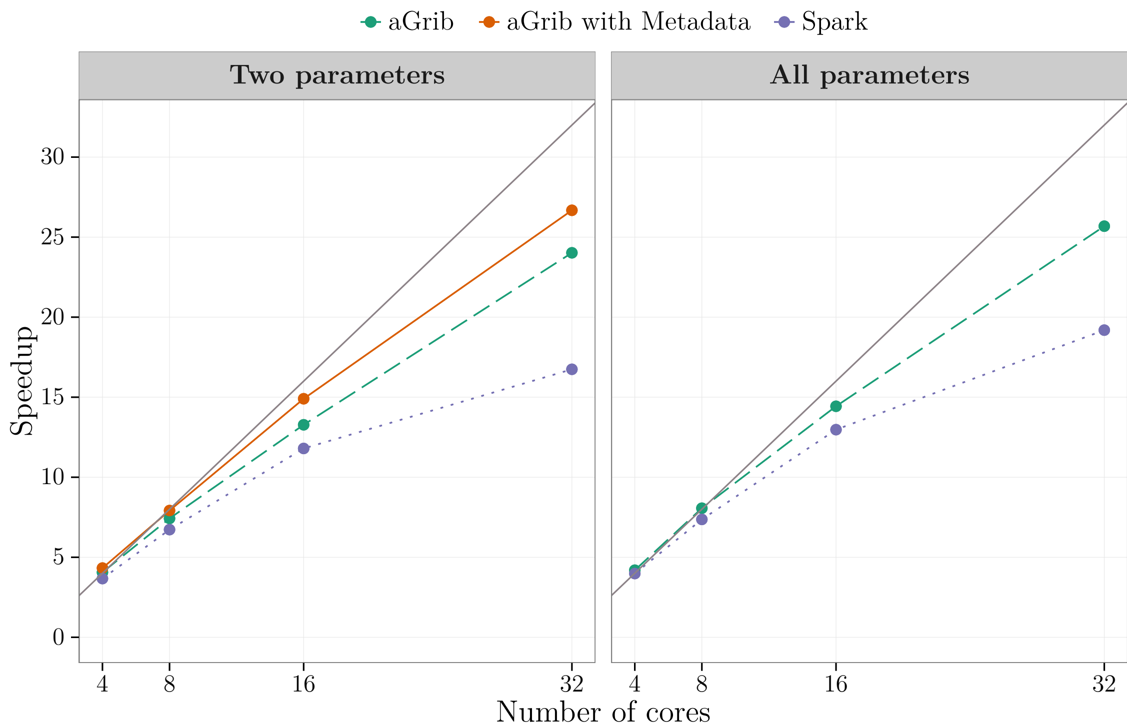
5.5.4 Using Apache Spark

At this point, we present a comparison between our proposal and our version that processes the GRIB files using Apache Spark. According to the experiments conducted with Spark, we use the default configurations. The cluster had 1 virtual machine as master, and 8 virtual machines as workers. The experiments took into consideration 1 *executor* per worker virtual machine and 4 *cores* per *executor*. In addition to this, each *executor* had 24GB of memory allocated.

Figure 5.18 displays the speedup reached for the first scenario on the left side, as well as for the second scenario on the right side, both considering the best cases of our proposal and the results obtained with Apache Spark. Since aGrib with the optimization just was applied over the first scenario, it does not appear in the figure for the second scenario. The x-axis shows the number of cores utilized and the y-axis shows the respective speedup.

With regard to the first scenario the **aGrib without the optimization** reached a gain of 10.65%, 10.33%, 12.44% and 43.48% when using 1, 2, 4 and 8 VMs, respectively. Obviously, the **aGrib with the optimization** reached a better performance with a gain of 17.94%, 17.71%, 26.32 and 59.36% when 1, 2, 4 and 8 VMs were utilized, respectively. And for the second scenario, **aGrib** achieved a gain of 5.35%, 9.56%, 11.26% and 33.84% upon using 1, 2, 4 and 8 worker virtual machines, respectively.

Figure 5.18: Speedup achieved for aGrib and Apache Spark in the two scenarios.



One explanation for the fact that there is a great difference between the results using 4 and 8 VMs, is that the GRIB files were initially stored in the HDFS using just 4 nodes, so that for 8 nodes a greater data transmission was necessary causing a loss of performance in Apache Spark, which in its default behavior needs to have all the data in memory before performing any operation. This would indicate that Apache Spark performs poorly when it does not have direct access to data.

Table 5.7 and Table 5.8 summarize the results obtained in the two scenarios for each set of virtual machines, both tables present the best configurations using the aGrib and the number of *executors* and *cores* per *executor* utilized for the Spark's execution, the time spent and standard deviation considering aGrib, as well as Apache Spark.

Table 5.7: First scenario: Summary of the results for aGrib and Apache Spark.

Mechanism	**#VMs**	**Configuration**	**Exec. Time**	**(StDev)**
aGrib	1	`1WN-8JW-88`	456.38	(± 3.639)
Spark	1	`1 executor - 4 cores`	505.00	(± 10.131)
aGrib	2	`1WN-8JW-175`	249.53	(± 7.061)
Spark	2	`2 executors - 8 cores`	275.32	(± 4.601)
aGrib	4	`1WN-8JW-175`	139.60	(± 4.617)
Spark	4	`4 executors - 16 cores`	156.96	(± 1.106)
aGrib	8	`1WN-8JW-175`	77.11	(± 1.545)
Spark	8	`8 executors - 32 cores`	110.63	(± 2.292)

Table 5.8: Second scenario: Summary of the results for aGrib and Apache Spark.

Mechanism	**#VMs**	**Configuration**	**Exec. Time**	**(StDev)**
aGrib	1	`1WN-8JW-175`	1254.68	(± 8.254)
Spark	1	`1 executor - 4 cores`	1321.76	(± 28.398)
aGrib	2	`1WN-8JW-88`	652.19	(± 4.428)
Spark	2	`2 executors - 8 cores`	714.51	(± 15.343)
aGrib	4	`1WN-8JW-88`	364.41	(± 4.269)
Spark	4	`4 executors - 16 cores`	405.44	(± 1.586)
aGrib	8	`1WN-8JW-44`	204.77	(± 1.449)
Spark	8	`8 executors - 32 cores`	274.06	(± 1.60)

6 CONCLUSION

Due to the overwhelming growth of scientific data in the last few years, data-intensive analysis on this vast amount of scientific data is very important to extract valuable scientific information. GRIB is a widely adopted format within the meteorological community and is used to store historical meteorological data and weather forecast simulation results. However, current options for processing the GRIB files do not perform the computation in a distributed environment. This situation limits the analytical capabilities of scientists who need to perform analyses on large data sets in the shortest time possible.

In this sense, we propose an alternative way to process large datasets in the GRIB scientific data format. Our proposal uses the implementation of the Actor model provided by the Akka toolkit to implement the well-know Manager-Worker pattern. In this pattern, the manager component partitions and distributes the work to do by the worker components and computes the final result from the partial results obtained from the workers, and the worker component is charged of processing the work sent from the manager, sending the results of each executed work to the manager, as well as requesting tasks of the manager. Furthermore, we compare our proposal with built-in router strategies provided by the Akka toolkit used to distribute the tasks among the workers, the round-robin, random and an adaptive load balancing were the strategies chosen, as well as we compare our proposal with one of the main frameworks currently existing for big data processing, the Apache Spark.

In addition to this, our proposal could be performed on an individual computer, a cluster of machines or in the cloud without having to modify the code or recompile it, in contrast with the current options to process the GRIB format.

The results presented in Chapter 5 show that our proposal scales well with increasing number of worker nodes. We have considered two different scenarios for processing the GRIB files. Thus, our proposal reached a better performance by using 8 worker virtual machines upon being compared with other approaches evaluated in this work, especially when is applied a filter to select only 2 of 27 total parameters found in the datasets, for this case our proposal reached a gain of 38.54%, 46.24%, 46.73%, 45.78%, 46.25% and 43.28% in relation to the best cases obtained for the rr, ra, cmc, cmh, cmm mechanisms and the Apache Spark, respectively. On the second scenario that evaluates all the parameters of the dataset, our proposal obtained a gain of 1.04%, 15.22%, 11.56%, 14.49%, 15.58% and 33.84% with respect to the best cases obtained for the rr, ra, cmc, cmh, cmm mechanisms and the Apache Spark, respectively. Likewise, after analyzing the first experimental results we perform an improvement over our proposal to achieve better performance, that improvement uses metadata stored in a NoSQL database to avoid the overhead caused for the time utilized in the manager component to distribute the tasks among their workers, thus, for this case our proposal by using 8 worker virtual machines

achieved a gain of 53.88%, 62.42%, 62.97%, 61.92%, 62.44% and 59.36% in relation to the best cases obtained for the rr, ra, cmc, cmh, cmm mechanisms and the Apache Spark, respectively.

Finally, we conclude that a tool that is able to monitor the execution of the actors during their life cycle in the different Java threads would be very useful for a better understanding, and subsequently a better exploitation of the Actor model implemented in the Akka toolkit.

As future work, we plan to extend the Manager-Worker patter to have a hierarchy of actors which will be responsible for the task distributions among the workers, as well as to generalize our implementation, so that it can be applied to other scientific data formats. Thus, initially we plan to extend the prototype to support the NetCDF data format using the Java NetCDF library from Unidata.

Furthermore, it would also be advantageous to improve the current deploy method, probably by using containers, such as the Docker containers.

REFERENCES

ABRAMOVA, V.; BERNARDINO, J. NoSQL Databases: MongoDB vs Cassandra. In: **Proceedings of the International C* Conference on Computer Science and Software Engineering**. New York, NY, USA: ACM, 2013. (C3S2E '13), p. 14–22.

AGHA, G. **Actors: A Model of Concurrent Computation in Distributed Systems**. Cambridge, MA, USA: MIT Press, 1986.

ALEXANDROV, A. et al. The Stratosphere Platform for Big Data Analytics. **The VLDB Journal**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 23, n. 6, p. 939–964, 2014.

ARMSTRONG, J. **Making Reliable Distributed Systems in the Presence of Software Errors**. Thesis (PhD) — The Royal Institute of Technology, Stockholm, Sweden, December 2003.

BALAJI, P. **Programming Models for Parallel Computing**. [S.l.]: The MIT Press, 2015.

BLANAS, S. et al. Parallel Data Analysis Directly on Scientific File Formats. In: **Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2014. (SIGMOD '14), p. 385–396.

BUCK, J. **Extending MapReduce for scientific data**. Thesis (PhD) — University of California, Santa Cruz, 2014.

BUCK, J. B. et al. SciHadoop: Array-based Query Processing in Hadoop. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2011. (SC '11), p. 66:1–66:11.

BUSCHMANN, F. et al. **Pattern-Oriented Software Architecture - Volume 1: A System of Patterns**. [S.l.]: Wiley Publishing, 1996.

CANDANEDO, J. A.; PARADIS, E.; STYLIANOU, M. Building Simulation Weather Forecast Files for Predictive Control Strategies. In: **Proceedings of the Symposium on Simulation for Architecture & Urban Design**. San Diego, CA, USA: Society for Computer Simulation International, 2013. (SimAUD '13), p. 4:1–4:6.

CARREÑO, E. D. **Migration and Evaluation of a Numerical Weather Prediction Application in a Cloud Computing Infrastructure**. Dissertation (Master) — PPGC - Federal University of Rio Grande do Sul, 2015.

CASBAH. 2016. Accessed Jul. 31, 2017. Available from Internet: <https://mongodb.github.io/casbah/>.

CHANDY, K.; TAYLOR, S. **An introduction to Parallel Programming**. [S.l.]: Jones and Bartlett, 1992.

CHEN, C. P.; ZHANG, C.-Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data . **Information Sciences**, v. 275, p. 314 – 347, 2014.

CLINGER, W. D. **Foundations of actor semantics**. Thesis (PhD), 1981. PHD.

COOPER, B. F. et al. Benchmarking Cloud Serving Systems with YCSB. In: **Proceedings of the 1st ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2010. (SoCC '10), p. 143–154.

DECANDIA, G. et al. Dynamo: Amazon's Highly Available Key-value Store. In: **Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2007. (SOSP '07), p. 205–220.

DOBRE, C.; XHAFA, F. Parallel Programming Paradigms and Frameworks in Big Data Era. **International Journal of Parallel Programming**, v. 42, n. 5, p. 710–738, 2014.

DUFFY, D. Q. et al. **Preliminary Evaluation of MapReduce for High-Performance Climate Data Analysis**. [S.l.], 2012. Accessed Jul. 31, 2017. Available from Internet: <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120009187.pdf>.

FORTNER, B. **The Data Handbook: A Guide to Understanding the Organization and Visualization of Technical Data**. [S.l.]: Springer-Verlag, 1995.

GU, L.; LI, H. Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark. In: **High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on**. [S.l.: s.n.], 2013. p. 721–727.

HAYASHIBARA, N. et al. The Phi accrual failure detector. In: **Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.** [S.l.: s.n.], 2004. (SRDS'04), p. 66–78.

HEWITT, C. Viewing control structures as patterns of passing messages. **Artificial Intelligence**, v. 8, n. 3, p. 323 – 364, 1977.

HEWITT, C. Actor Model for Discretionary, Adaptive Concurrency. **CoRR**, abs/1008.1459, 2015. Accessed Jul. 31, 2017. Available from Internet: <https://arxiv.org/abs/1008.1459v38>.

HEWITT, C.; BAKER, H. G. Laws for Communicating Parallel Processes. In: **IFIP Congress**. [S.l.: s.n.], 1977. p. 987–992.

HEWITT, C.; BISHOP, P.; STEIGER, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. (IJCAI'73), p. 235–245.

HIBBARD, B. et al. Visualization in Earth System Science. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 36, n. 4, p. 5–9, 2002. ISSN 0097-8930.

HOLMES, A. **Hadoop in Practice**. 2nd. ed. Greenwich, CT, USA: Manning Publications Co., 2014.

JANSSEN, C.; NIELSEN, I. **Parallel Computing in Quantum Chemistry**. [S.l.]: CRC Press, 2008.

KAMBATLA, K. et al. Trends in big data analytics. **Journal of Parallel and Distributed Computing** , v. 74, n. 7, p. 2561 – 2573, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

KARAU, H. et al. **Learning Spark: Lightning-Fast Big Data Analytics**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015.

KOSTER, J. D.; CUTSEM, T. V.; MEUTER, W. D. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In: **Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control**. New York, NY, USA: ACM, 2016. (AGERE 2016), p. 31–40. ISBN 978-1-4503-4639-9.

LI, J. et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In: **Proceedings of the 2003 ACM/IEEE Conference on Supercomputing**. New York, NY, USA: ACM, 2003. (SC '03), p. 39–.

Lightbend Inc. **Akka Scala Documentation**. [S.l.], 2016. Accessed Jul. 31, 2017. Available from Internet: <http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf>.

MARKIEWICZ, Ł. et al. Operational Enhancement of Numerical Weather Prediction with Data from Real-time Satellite Images. In: WEINTRIT, A. (Ed.). **Marine Navigation and Safety of Sea Transportation: Navigational Problems**. [S.l.]: CRC Press, 2013. chp. 5, p. 153–160.

MARQUES, A. C. **Implementação de dados obtidos com imagens do sensor TM do LANDSAT 5 e da missão SRTM no modelo atmosférico BRAMS**. Dissertation (Master) — Curso de Sensoriamento Remoto, Centro Estadual de Pesquisas em Sensoriamento Remoto e Meteorologia - Universidade Federal do Rio Grande do Sul, 2009.

NCEP WMO GRIB2 Documentation. 2016. Accessed Jul. 31, 2017. Available from Internet: <http://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc.shtml>.

ORTEGA-ARJONA, J. L. The Manager-Workers Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming. In: **Proceedings of the 9th European Conference on Pattern Languages of Programms**. Irsee, Germany: UVK - Universitaetsverlag Konstanz, 2004. (EuroPLoP '2004), p. 53–64.

PALAMUTTAM, R. et al. SciSpark: Applying In-memory Distributed Computing to Weather Event Detection and Tracking. In: **Proceedings of the 2015 IEEE International Conference on Big Data**. Washington, DC, USA: IEEE Computer Society, 2015. (BIG DATA '15), p. 2020–2026.

PARKER, Z.; POE, S.; VRBSKY, S. V. Comparing NoSQL MongoDB to an SQL DB. In: **Proceedings of the 51st ACM Southeast Conference**. New York, NY, USA: ACM, 2013. (ACMSE '13), p. 5:1–5:6.

REW, R.; DAVIS, G. NetCDF: an interface for scientific data access. **IEEE Computer Graphics and Applications**, v. 10, n. 4, p. 76–82, July 1990.

ROESTENBURG, R.; BAKKER, R.; WILLIAMS, R. **Akka in Action**. 1st. ed. Greenwich, CT, USA: Manning Publications Co., 2016.

ROLOFF, E. et al. Evaluating High Performance Computing on the Windows Azure Platform. In: **2012 IEEE Fifth International Conference on Cloud Computing**. [S.l.: s.n.], 2012. p. 803–810.

ROSÀ, A.; CHEN, L. Y.; BINDER, W. Profiling Actor Utilization and Communication in Akka. In: **Proceedings of the 15th International Workshop on Erlang**. New York, NY, USA: ACM, 2016. (Erlang 2016), p. 24–32.

ROSE, C. A. F. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre RS, Brasil: Editora Sagra Luzzatto, 2003.

Spark Documentation. 2016. Accessed Jul. 31, 2017. Available from Internet: <http://spark.apache.org/docs/latest/>.

TASHAROFI, S.; DINGES, P.; JOHNSON, R. E. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In: **Proceedings of the 27th European Conference on Object-Oriented Programming**. Berlin, Heidelberg: Springer-Verlag, 2013. (ECOOP'13), p. 302–326.

The HFD5 Format. **Hierarchical Data Format, version 5**. 2002. Accessed Jul. 31, 2017. Available from Internet: <https://www.hdfgroup.org/>.

UNIDATA. 2017. Accessed Jul. 31, 2017. Available from Internet: <https://github.com/Unidata/thredds>.

VERNON, V. **Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2015.

WANG, Y.; JIANG, W.; AGRAWAL, G. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In: **Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)**. Washington, DC, USA: IEEE Computer Society, 2012. (CCGRID '12), p. 443–450.

WMO. **Guide to the WMO Table Driven Code Form Used for the Representation and Exchange of Regularly Spaced Data In Binary Form: FM 92 GRIB Edition 2**. [S.l.], 2003. Accessed Jul. 31, 2017. Available from Internet: <http://www.wmo.int/pages/prog/www/WMOCodes/Guides/GRIB/GRIB2_062006.pdf>.

ZAHARIA, M. et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: **Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 15–28.

ZAHARIA, M. et al. Spark: Cluster Computing with Working Sets. In: **Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing**. Berkeley, CA, USA: USENIX Association, 2010. (HotCloud'10).

ZASADZINSKI, M.; MUNTÉS-MULERO, V.; SIMO, M. S. Actor Based Root Cause Analysis in a Distributed Environment. In: **2017 IEEE/ACM 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)**. [S.l.: s.n.], 2017. p. 14–17.

ZHAO, H. et al. Parallel Accessing Massive NetCDF Data Based on Mapreduce. In: **Proceedings of the 2010 International Conference on Web Information Systems and Mining**. Berlin, Heidelberg: Springer-Verlag, 2010. (WISM'10), p. 425–431.

Appendix A — GRIB SAMPLE

This Appendix presents a sample of the GRIB2 format. The Data section presents only some values, due to it would be extensive to place all values.

Section Octet	Message Octet	Value	Meaning
			Indicator Section
1-4	1-4	GRIB	"GRIB"
5-6	5-6	0	Reserved
7	7	0	This GRIB2 message contains Meteorological products (the product discipline)
8	8	2	The GRIB Edition Number is 2
9-16	9-16	427018	The total length of this GRIB message is 427018 bytes
			Identification Section
1-4	17-20	21	This Section is 21 bytes long
5	21	1	This is Section 1
6-7	22-23	255	Missing value
8-9	24-25	0	There is no originating/generating sub-centre
10	26	1	GRIB Master Tables Version implemented on 7 November 2001
11	27	0	Local tables not used. Only table entries and templates from the current Master table are valid.
12	28	0	Analysis
13-14	29-30	2015	Year: 2015
15	31	7	Month: 7
16	32	5	Day: 5
17	33	0	Hour: 0
18	34	0	Minute: 0
19	35	0	Second: 0
20	36	0	This GRIB2 message contains Operational products
21	37	0	This GRIB2 message contains Analysis products
			Local Use Section
1-4	38-41	0	This Section is 0 bytes long
			Grid Definition Section
1-4	38-41	72	This Section is 72 bytes long
5	42	3	This is Section 3
6	43	0	Grid Specified in Code table 3.1
7-10	44-47	1584353	There are 1584353 data points in this grid
11	48	0	There is no optional list of numbers defining number of points
12	49	0	There is no appended list

Table A.1 – *A GRIB message sample*

Section Octet	Message Octet	Value	Meaning
13-14	50-51	0	The Grid Definition Template Number is 3.0: Latitude/longitude projection
15	52	6	Earth assumed spherical with radius of 6,371,229.0 m
16	53	All 1's	There is no scale factor for the radius of the spherical earth
17-20	54-57	All 1's	There is no scaled value of the radius of the spherical earth
21	58	All 1's	There is no scale factor of major axis of oblate spheroid earth
22-25	59-62	All 1's	There is no scaled value of major axis of oblate spheroid earth
26	63	All 1's	There is no scale factor of minor axis of oblate spheroid earth
27-30	64-67	All 1's	The scaled value of minor axis of oblate spheroid earth
31-34	68-71	1159	There are 1159 points along a parallel (Ni)
35-38	72-75	1367	There are 1367 points along a meridian (Nj)
39-42	76-79	0	The basic angle for all latitudes and longitudes is 1 degree
43-46	80-83	All 1's	The unit of the subdivisions of basic angle for all latitudes and longitudes is 10^{-6} degrees
47-50	84-87	-4.7E7	The latitude of the first grid point (La1) is -47.0 degrees
51-54	88-91	2.77867008E8	The longitude of the first grid point (Lo1) is 277.867 degrees
55	92	00110000	x (i) direction increments given, y (j) direction increments given. Resolved u and v components of vector quantities relative to easterly and northerly directions
56-59	93-96	1.10196E7	The latitude of the last grid point is 11.0196 degrees
60-63	97-100	3.330968E8	The longitude of the last grid point is 333.0968 degrees
64-67	101-104	47694.0	The i direction increment is 0.047694 degrees
68-71	105-108	42474.0	The j direction increment is 0.042474 degrees
72	109	01000000	Points in the first row or column scan in the $+y$ ($+j$) direction
			Product Definition Section
1-4	110-113	34	This Section is 34 bytes long
5	114	4	This is Section 4
6-7	115-116	0	There are no coordinate values after the Product Definition Template

Table A.1 – *A GRIB message sample*

Section Octet	Message Octet	Value	Meaning
8-9	117-118	0	The Product Definition Template Number is 4.0: Analysis or forecast at a horizontal level or in a horizontal layer at a point in time
10	119	3	The parameter category is 3: Mass
11	120	5	The parameter number is 5: Geopotential height (in gpm)
12	121	All 1's	No information on the type of generating process is provided
13	122	All 1's	There is no background generating process identifier
14	123	All 1's	There is no analysis or forecast generating process identifier
15-16	124-125	3	The observational data cut-off was 3 hours after the reference time
17	126	All 1's	There is no minutes of observational data cut-off after reference time
18	127	1	The time is given in hours
19-22	128-131	12	The forecast time is 12 hours after the reference time
23	132	1	The first fixed surface is Ground or water surface
24	133	All 1's	There is no scale factor of first fixed surface
25-28	134-137	All 1's	There is no a scaled value of first fixed surface
29	138	All 1's	There is no second fixed surface
30	139	All 1's	There is no scale factor of second fixed surface
31-34	140-143	All 1's	There is no a scaled value of second fixed surface
			Data Representation Section
1-4	144-147	49	This Section is 49 bytes long
5	148	5	This is Section 5
6-9	149-152	1584353	There are 1584353 data points for which values are specified in Section
10-11	153-154	3	The Data Representation Template Number is 5.3: Grid point data - complex packing and spatial differencing
12-15	155-158	-1.0000001E-20	The reference value (R) is -1.0000001E-20 (IEEE 32-bit floating-point value)
16-17	159-160	1	The binary scale factor (E) is 1
18-19	161-162	0	The decimal scale factor (D) is 0
20	163	9	9 bits are used for each packed value in the Data Section
21	164	0	The original field values were floating point numbers
22	165	1	The method General Group Splitting is used

Table A.1 – *A GRIB message sample*

Section Octet	**Message Octet**	**Value**	**Meaning**
23	166	0	No explicit missing values included within the data values
24-27	167-170	0	Primary missing value substitute
28-31	171-174	0	Secondary missing value substitute
32-35	175-178	23230	23230 groups of data values into which field is split
36	179	0	Reference for group widths
37	180	4	4 bits used for the group widths
38-41	181-184	1	Reference for group lengths
42	185	1	Length increment for the group lengths
43-46	186-189	705	True length of last group
47	190	10	Number of bits used for the scaled group lengths
48	191	2	Second-Order Spatial Differencing is used
49	192	2	Number of bytes required in the data section to specify extra descriptors needed for spatial differencing
			Bitmap Section
1-4	193-196	6	This Section is 6 bytes long
5	197	6	This is Section 6
6	198	255	A bit map does not apply to this product
			Data Section
1-4	199-202	426816	This Section is 426816 bytes long
5	203	7	This is Section 7
		...	
		324.0	
		344.0	
		366.0	
		386.0	
		404.0	
		416.0	
		422.0	
		416.0	
		396.0	
		360.0	
		...	
			End Section
1-4	427015-427018	7777	Indicates the end of the message

Appendix B — PERFORMANCE EVALUTION

This Appendix presents the total execution times obtained from the experiments. Figure B.1, B.2, B.3, B.4 depict the total execution times for the first scenario (processing just 2 parameters) upon using 1, 2, 4 and 8 worker virtual machines, respectively. Whereas Figure B.5, B.6, B.7, B.4 depict the total execution times for the second scenario (processing all the parameters) upon using 1, 2, 4 and 8 worker virtual machines, respectively. Besides, the total execution times using metadata on 1, 2, 4 and 8 worker virtual machines are shown in Figure B.9.

Each figures presents all mechanisms utilized (aGrib and the other five approaches), the strategies utilized (file-based and message-based), and each one of sub-graphics presents the configurations in the x-axis and the time spent in the y-axis.

Figure B.1: First scenario: Execution time using 1 worker machine.

Figure B.2: First scenario: Execution time using 2 worker machines.

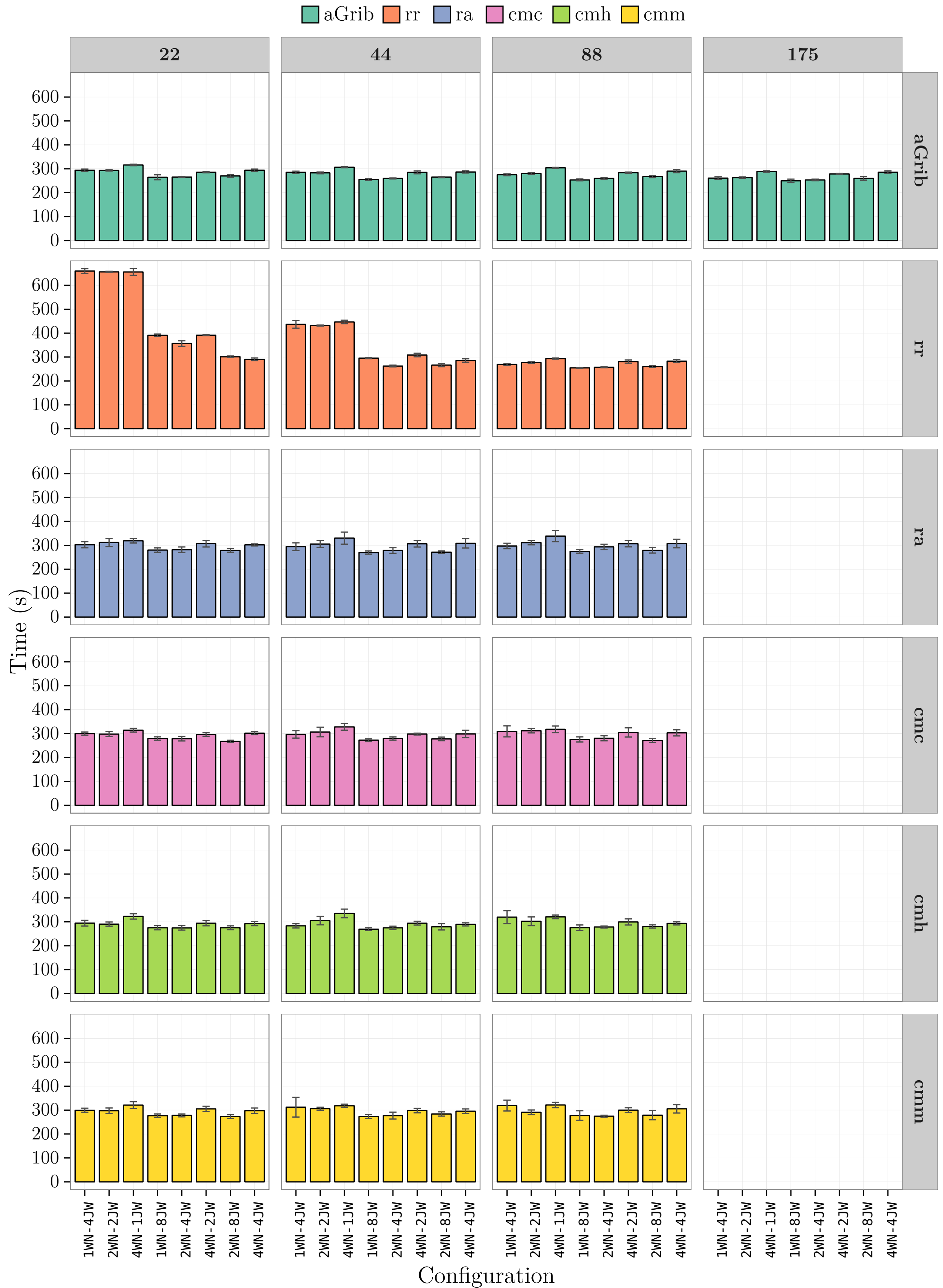


Figure B.3: First scenario: Execution time using 4 worker machines.

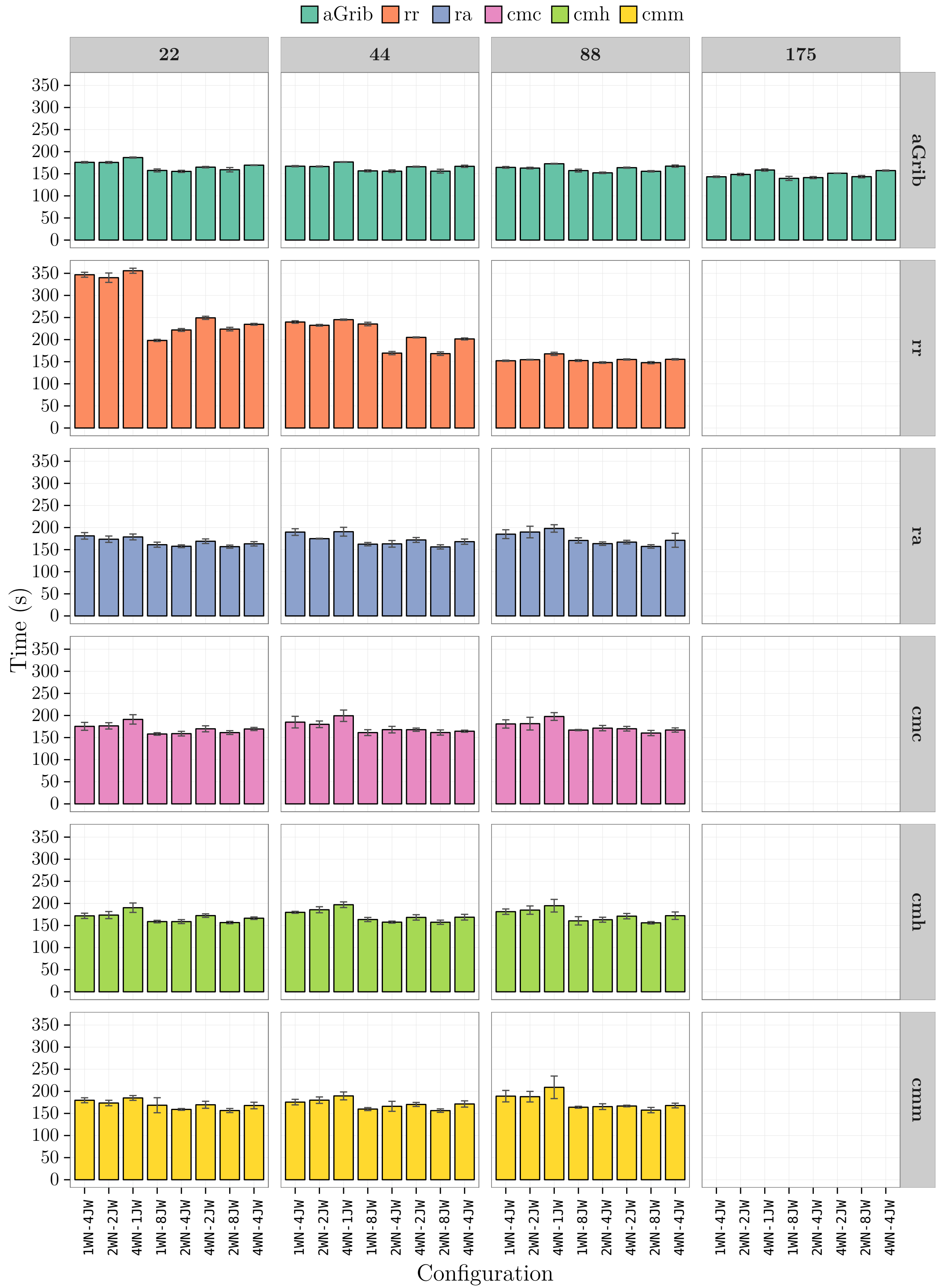


Figure B.4: First scenario: Execution time using 8 worker machines.

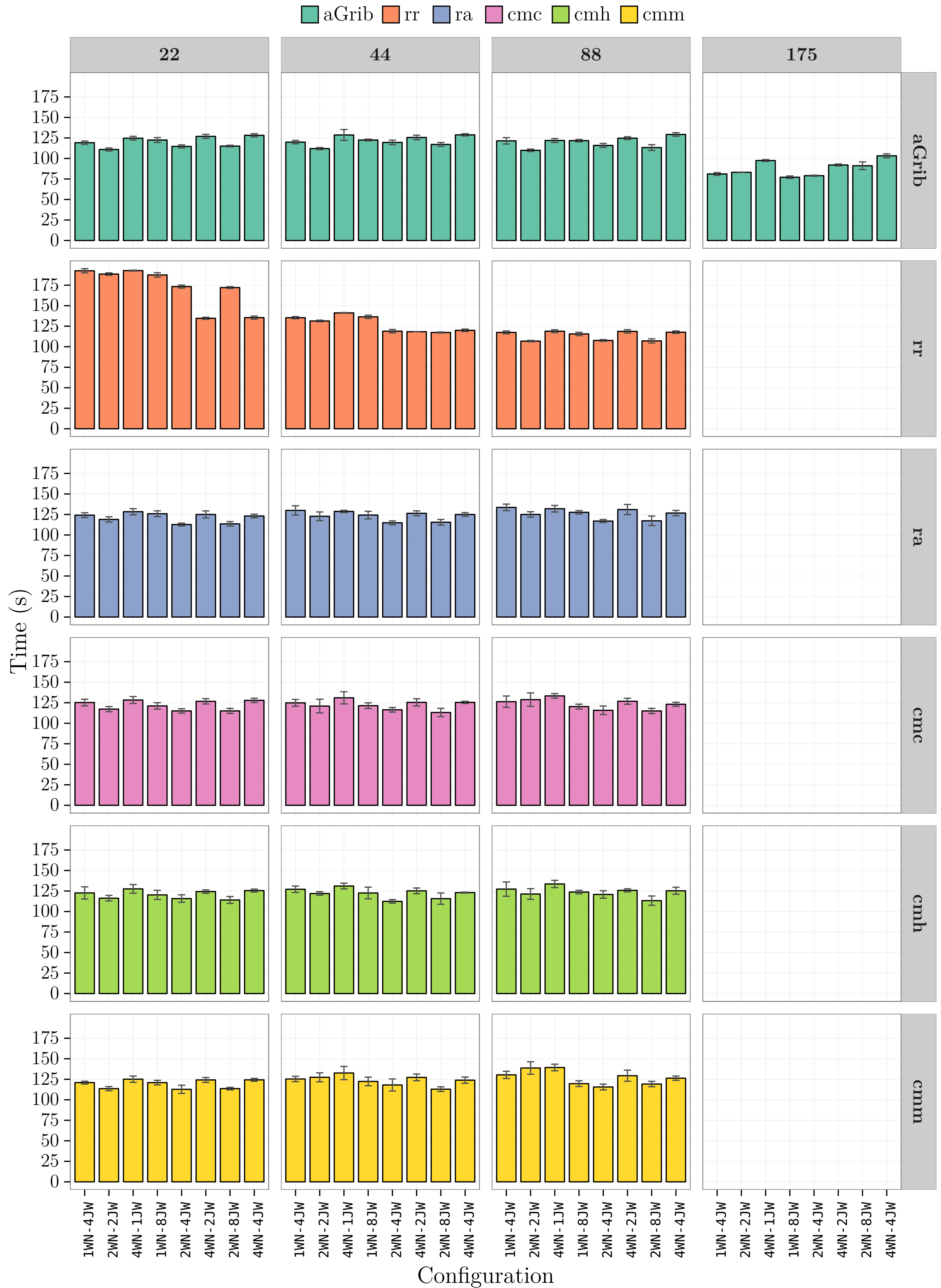


Figure B.5: Second scenario: Execution time using 1 worker machine.

Figure B.6: Second scenario: Execution time using 2 worker machines.

Figure B.7: Second scenario: Execution time using 4 worker machines.

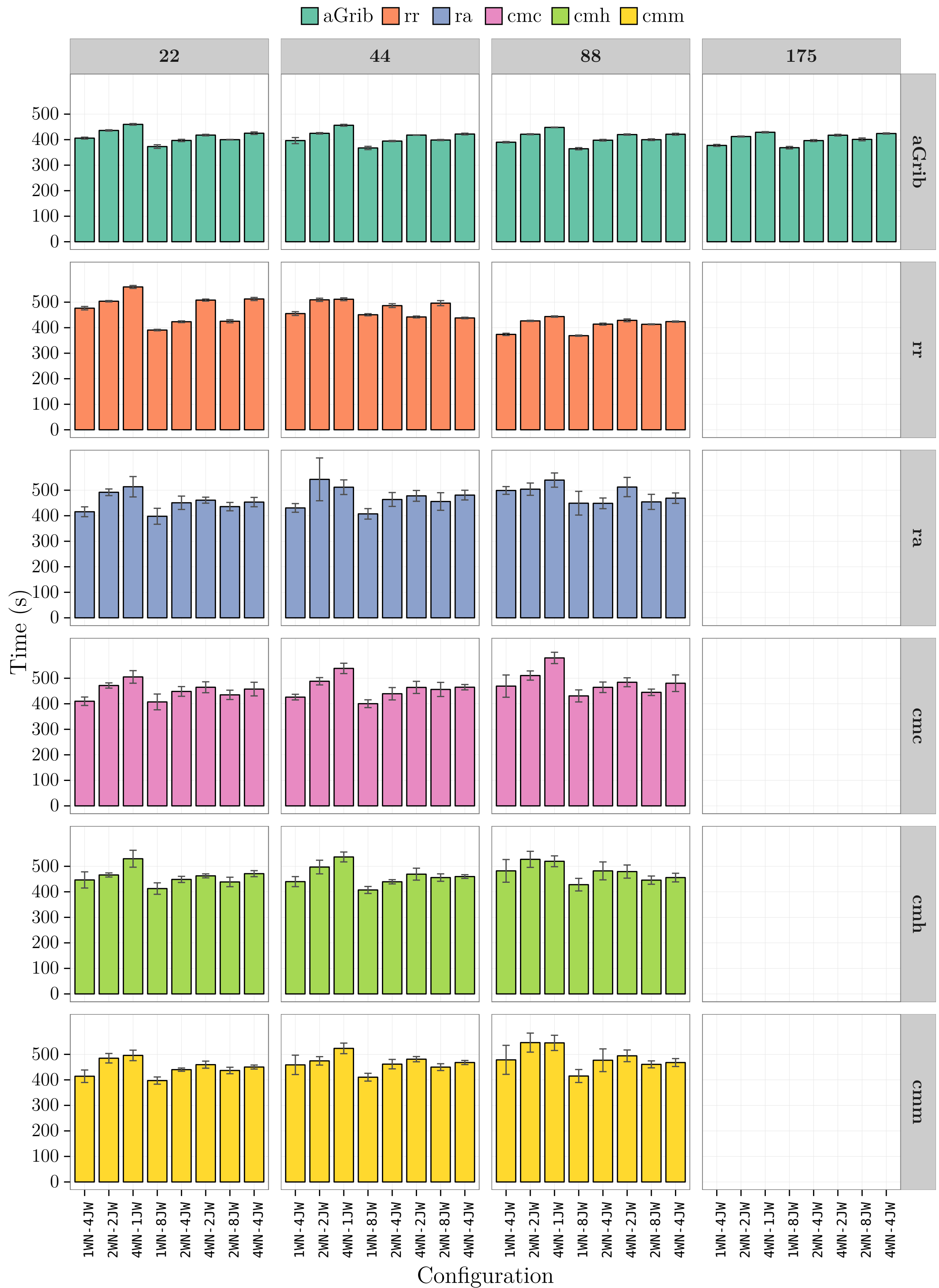


Figure B.8: Second scenario: Execution time using 8 worker machines.

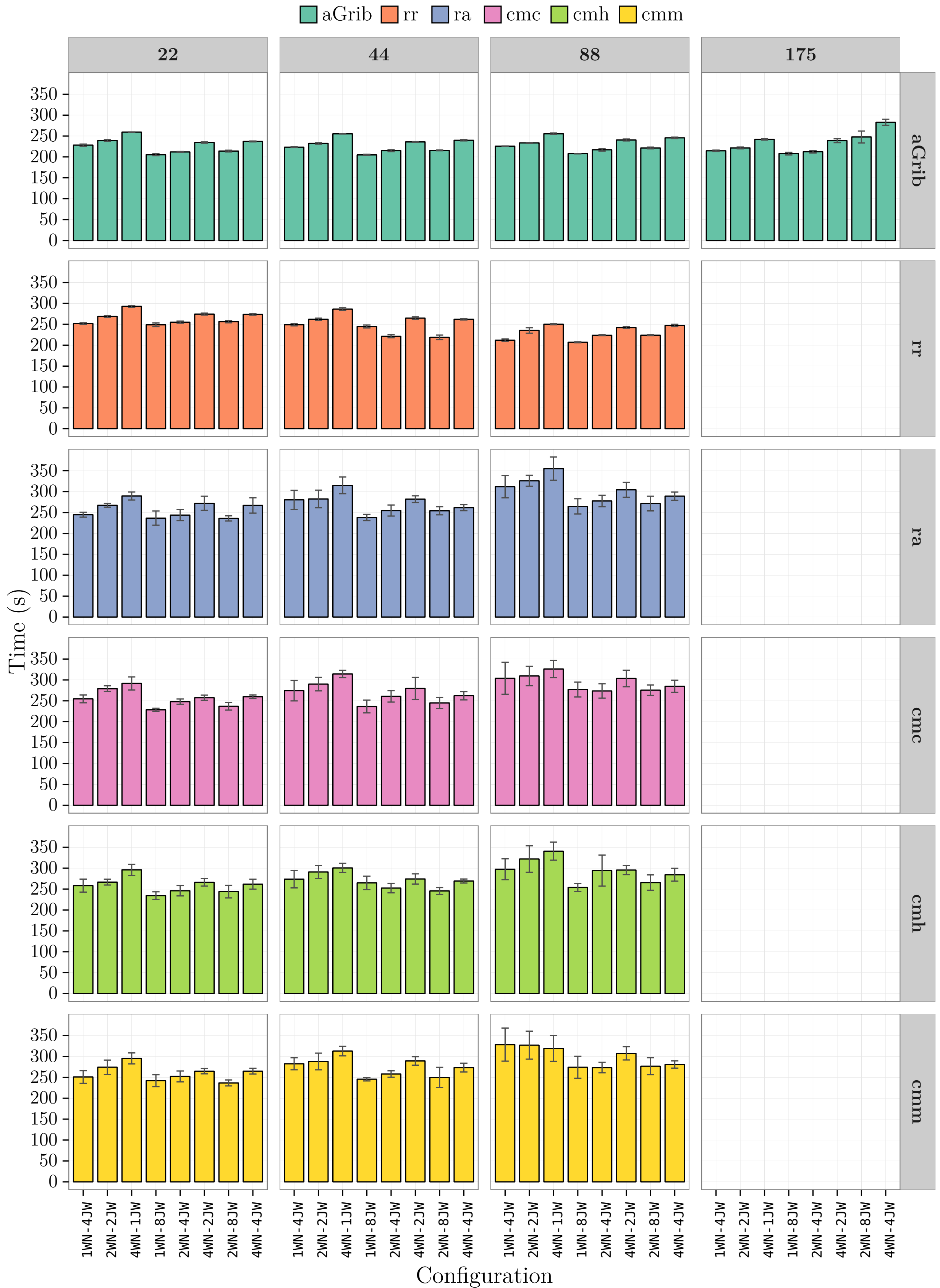


Figure B.9: First scenario: Total execution times of our proposal by using metadata.