Universitat Autònoma de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Performance Improvement Methodology based on Divisible Load Theory for Data Intensive Applications

Submitted by Claudia Rosas for the degree of Philosophiæ Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Barbara Sikora, done at the Computer Architecture and Operating Systems Department, PhD. in High-performance Computing

Bellaterra, May 2012

Performance Improvement Methodology based on Divisible Load Theory for Data Intensive Applications

Submitted by Claudia Rosas for the degree of Philosophiæ Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dra. Anna Barbara Sikora, done at the Computer Architecture and Operating Systems Department, Ph.D. in High-performance Computing.

Bellaterra, May 2012

Supervisor

Dr. Anna Barbara Sikora

Abstract

The recent large amount of data needing to be processed represents one of the major challenges in the computational field. This fact led to the growth of specially designed applications known as data-intensive applications. In general, to ease the parallel execution of data-intensive applications input data is divided into smaller data chunks that can be processed separately. However, in many cases, these applications show severe performance problems mainly due to load imbalance, inefficient use of available resources, and improper data partition policies. In addition, the impact of these performance problems can depend on the dynamic behavior of the application.

This work proposes a methodology to dynamically improve the performance of data-intensive applications based on: (i) adapting the size and the number of data partitions to reduce overall execution time; and (ii) adapting the number of processing nodes to achieve an efficient execution. We propose to monitor the application behavior for each iteration (query) and use gathered data to dynamically tune the performance of the application. The methodology assumes that a single execution includes multiple related queries on the same partitioned workload.

The adaptation of the workload partition factor is addressed through the definition of the initial size for the data chunks; the modification of the scheduling policy to send first data chunks with large processing times; dividing of the data chunks with the biggest associated computation times; and joining of data chunks with small computation times. The criteria for dividing or gathering chunks are based on the chunks' associated execution time (average and standard deviation) and the number of processing elements being used. Additionally, the resources utilization is addressed through the dynamic evaluation of the application performance and the estimation and modification of the number of processing nodes that can be efficiently used.

We have evaluated our strategy using a real and a synthetic data-intensive application. Analytical expressions have been analyzed through simulation. Applying our methodology, we have obtained encouraging results reducing total execution times and efficient use of resources.

Keywords: load balancing; dynamic performance analysis and tuning; Data-intensive applications; arbitrarily divisible load.

Resumen

La gran cantidad de datos que recientemente necesitan ser procesados, representa uno de los mayores retos en el campo de la computación. Esto ha conllevado al crecimiento de aplicaciones con requerimientos especiales conocidas como aplicaciones intensivas en datos. En general, para facilitar la ejecución en paralelo de aplicaciones intensivas en datos, los datos de entrada son divididos en trozos más pequeños que pueden ser procesados individualmente. Sin embargo, en muchos casos, estas aplicaciones muestran graves problemas de rendimiento debidos principalmente a desbalances de carga, uso ineficiente de los recursos de cómputo disponibles, e inapropiadas políticas de partición y distribución de los datos. Además, el impacto de dichos problemas de rendimiento puede depender del comportamiento dinámico de la aplicación.

Este trabajo propone una metodología para mejorar, dinámicamente, el rendimiento de aplicaciones intensivas en datos, en base a: (i) adaptar el tamaño y el número de las particiones de datos con el fin de reducir el tiempo de ejecución total; y (ii) adaptar el número de nodos de cómputo para conseguir una ejecución eficiente. Proponemos monitorizar el comportamiento de la aplicación para cada iteración (o consulta) y usar los datos recogidos para ajustar dinámicamente el rendimiento de la aplicación. La metodología asume que una sola ejecución incluye múltiples consultas relacionadas sobre una misma carga de trabajo particionada.

El ajuste del factor de partición de la carga de trabajo es llevado a cabo a través de la definición del tamaño inicial de los trozos de datos; la modificación de la política de planificación, para enviar primero los trozos de datos con los tiempos de procesamiento más largos; la división de dichos trozos de datos; y el agrupamiento de trozos de datos con tiempos de cómputo muy cortos. Los criterios para decidir dividir o agrupar trozos están basados en los tiempos de ejecución asociados a cada pieza (tiempo medio y desviación estándar) y en el número de elementos de cómputo que están siendo utilizados. Adicionalmente, lo inherente al uso de los recursos se abordó mediante la evaluación dinámica del rendimiento de la aplicación, junto con la estimación y consiguiente modificación del número de nodos de procesamiento que pueden ser utilizados eficientemente.

Hemos evaluado nuestra propuesta usando aplicaciones intensivas en datos reales y sintéticas. Así como también hemos analizado las expresiones analíticas propuestas a través de simulación. Luego de aplicar nuestra metodología, hemos obtenido resultados prometedores en la reducción del tiempo total de ejecución y el uso eficiente de los recursos.

Palabras clave: balanceo de carga; análisis y sintonización dinámico del rendimiento; aplicaciones intensivas en datos; carga arbitrariamente divisible.

Resum

L'augment de la quantitat de dades que necessiten ser processades actualment, representa un dels majors reptes a l'àmbit de la computació. Això ha permès el creixement d'aplicacions amb requeriments especials conegudes com aplicacions intensives en dades. En general, per afavorir l'execució en paral·lel de aquest tipus d'aplicacions, les dades d'entrada son partits en trossos més petits que poden ser processats individualment. No obstant això, en molts casos, aquestes aplicacions mostren problemes graus de rendiment, deguts principalment a desequilibris de càrrega, l'ús ineficient dels recursos de còmput disponibles, i inadequades polítiques de partició i distribució de les dades. A més, l'impacte d'aquests problemes de rendiment es pot veure acrescut pel comportament dinàmic de l'aplicació.

Aquest treball proposa una metodologia per a millorar, dinàmicament, el rendiment d'aplicacions intensives en dades, basat en: (i) l'adaptació de la grandària i nombre de les particions de dades amb la finalitat de reduir el temp d'execució total; i (ii) l'adaptació del nombre de nodes de còmput per aconseguir una execució eficient. Proposem observar el comportament de l'aplicació per cada iteració (o consulta) i utilitzar les dades recollides per a ajustar dinàmicament el seu rendiment. La metodologia assumeix que cada execució inclou múltiples consultes relacionades sobre una única càrrega de treball partida.

L'ajust del factor de partició de la càrrega de treball es fa mitjançant la definició de la grandària inicial dels trossos de dades; la modificació de la política de planificació (per a enviar primerament els trossos amb major temps d'execució); la divisió dels trossos amb major temps d'execució; i el agrupament de trossos de dades amb temps de còmput massa curts. Els criteris per a decidir si el trossos es divideixen o es agrupen estan basats en els temps d'execució associats a cada tros (com el temps mitjà i la desviació estàndard) així com també en el nombre de nodes de còmputs que s'estan utilitzant. A més a més, el referent a l'ús de recursos de còmput es va abordar mitjançant l'avaluació dinàmica del rendiment de l'aplicació, juntament amb l'estimació i modificació del nombre de nodes de processament que es puguin utilitzar eficientment.

Hem avaluat la nostra proposta usant aplicacions intensives en dades reals i sintètiques. Així com també hem analitzat les expressions analítiques propostes mitjançant simulació. Després d'aplicar la nostra metodologia, hem obtingut resultats prometedors en la reducció del temps total d'execució i l'ús eficient dels recursos.

Paraules claus: balanceig de càrrega; anàlisi i sintonització dinàmic del rendiment; aplicacions intensives en dades; càrrega arbitràriament divisible.

Acknowledgements

First and foremost to my parents and my sister,
for all the confidence and faith you have deposited on me,
you taught me to see beyond the horizon and to look one step ahead,
I'm who i am because of you. This achievement is for all of us...

To Ania, Eduardo, Josep, Toni and Tomàs,
for your guidance, advises, patience and healthy discussions,
I've learned a lot from you and
I'm grateful of being part of this research group...

to Emilio and Lola,
for receiving me in this world of science,
and for teaching me not to give up...

To Gonzalo,
for all the time, patience and care you have invested in me,
you, together with all our friends Moni and Maru, Alvaro and Zeynep,
and Andrea and Hayden, have been the best part of this experience...

To all my friends and partners at CAOS,
for everything I have learned from each one of you...
to Ronal, for encouraging me to apply for this scholarship.

Thank you all!

Contents

List of Figures

List of Equations

List of Algorithms

List of Tables

Chapter 1

Introduction

"Toutes les grandes personnes ont d'abord été des enfants."

Antoine de Saint-Exupéry. Le Petit Prince.

In an interdisciplinary world, the interaction between different fields of study has become a fact. Areas as physics, genetics, mechanics, and so on, have been positively affected by the emergence of computational science. Computational systems have provided specially developed scientific applications and high performance computing (HPC) environments to solve problems of different nature. Among some of the greatest beneficiaries of these computational systems, it is worth mentioning: accomplish weather simulations in short and specific deadlines; interpret data coming continuously from sensors; or process user-generated information. These processes, together with most scientific applications, represent some of the greatest challenges for HPC. As the applications become more complex and perform more sophisticated computations, the increasing demand for computing power highlights the need for massively parallelization of systems and applications.

Parallel systems are computer environments composed of a set of processing units that work in conjunction and simultaneously to solve a computational problem. Parallel computers have high-potential characteristics, such as processing speed, memory or disk capacity. Although such systems have their performance limits, they are more powerful than the rest of the computers and hence are more suitable for solving scientific problems demanding intensive computation. Architectures, operating systems and applications have been developed to exploit the capacities of these systems to speedup computation.

In this chapter, we present a general overview of the applications performance problem in current computational systems. In particular, our work is focused on data intensive parallel applications. This chapter introduces the motivation inspiring this work, as well as an overview of studies related to our proposal. In addition, it presents the goal and

contributions of this work and briefly describes the research method. Finally, we present the organization of this document.

1.1 Parallel Processing of Large-Scale Data

In the last few years, HPC systems have been in continuous growth to satisfy the computing power demand and to enable the parallel utilization of computational resources. Nevertheless, the size of the computational problems has growth at almost the same rate. Systems and algorithms are facing an endless data deluge coming from scientific applications such as genome sequencing, molecular dynamics simulations, weather forecasting, and worldwide banking transactions, among others [20]. For instance, a single Google search releases the power of thousands of processors looking for a query over data sets of hundreds of terabytes. This data flow has surpassed the capacities of the systems and algorithms designed a few years before, and has led to the growth of new applications known as *data intensive applications* [16], or its lest trendy name: *big-data computing* [17]. Data means "things given" in Latin –although we tend to use it as a mass noun in English, as if it denotes a substance– and ultimately, almost all useful data is given to us either by nature, as a reward for careful observation of physical processes, or by people, usually inadvertently. As a result, in the real world, data is not just a big set of random numbers; it tends to exhibit predictable characteristics. This predictable behavior will help us to estimate the performance for subsequent iterations.

Data intensive computing is concerned with addressing the technical challenges generated by the ever-growing demands for processing large-scale data sets [61]. Moreover, what makes most large-scale data *large* are repeated observations over time and/or space. For example, the Web log records millions of visits a day to a handful of pages; the retailer has thousands of stores, ten thousands of products, and millions of costumers but logs billions and billions of individual transactions a year. Scientific measurements are often made at high time resolutions and become unmanageable when they involve two or three dimensions of space as well; e.g., the whole human genome sequencing problem involves the computation of files of several gigabytes [87]. In recent years, experiments carried out with the Large Hadron Collider (LHC) at CERN, such as ATLAS [39] and CMS [38], generate data at a rate of 320 and 220 megabytes per second, respectively. Therefore, in the era of data intensive applications, computational systems are not only intended to compute but also to store and manage data. Simple tasks, such as making data available to be processed in research centers have become a huge challenge given the size of data.

In general, if a scientific application is appropriately designed to take advantage of

systems parallelism, its executions would be usually carry out in a fast and efficient way. Nevertheless, to process data efficiently is not only matter of having enough processing units, but it also depends on specific characteristics of the workload of the application. In many cases, these applications can be naturally implemented in parallel by partitioning their data sets into smaller pieces and distributing them among the processing units of the parallel system. However, each partition may have different processing times and this situation may lead to significant imbalances in the execution time of the processing units of the application.

When data intensive applications are executed for a large number of queries or iterations, we may face additional variations in overall execution time from one execution to the other. For this reason, any proposal for performance analysis strategies and load balancing techniques must be adapted to the specific characteristics of the application. In most cases, given the variability between (or within) executions, performance analysis must be carried out at run time, that is while the application is being executed. Otherwise, the proposed solution may be obsolete from one iteration to the other.

The performance of data intensive applications is closely related to the capability of overcoming performance degradation caused, in many cases, by load imbalance, inefficient use of available resources, and improper data partition and management policies. Unfortunately, as explained above, the steady growth of data sets significantly increases performance problems. The following questions arise from the analysis of this situation: how do the size of workload partitions affects the total execution time? What kinds of data-distribution strategies are best suited for data intensive applications? Are these applications able to be executed in a large number of processing units? If they are not, what should be the solution to execute them? What are the best options to achieve an efficient execution with shorter run times? Can variable behavior between executions be faced statically or dynamically? The mere posing of these questions highlights the importance of the performance analysis of data intensive applications, and the need to address this problem in current high-performance computing systems.

This work is focused on answering these questions with the aim of improving performance of parallel data intensive applications in the presence of variable behavior. To provide completeness to the scope of the work presented along this chapter, we will give now a short introduction into data management; performance issues and dynamic performance analysis and tuning.

1.1.1 Data Management

Data intensive computing represents not just an evolutionary change in computation science but also a revolutionary change in the way scientifics gather and process information, from the hardware and algorithms to the presentation of knowledge to the end user. Applications in many disciplines are driving a shift in the emphasis of data intensive computing. Applications are changing from focusing solely on large-scale data sets to the broader realm of issues dealing with the time to reach a solution when data-handling capacity is a significant factor, e.g., as in real-time processing of massive data streams.

According to Kouzes et al. [50], one of the first challenges to data intensive computation is the amount of incoming data, obtained from multiple sources and locations, with varying degrees of quality and reliability. As data size increases, the computational resources needed to process data have to grow as well. In some cases, analysis may scale linearly with data size, hence parallelization techniques are easy to implement. Nevertheless, applications may have data-depending behaviors, that require a more complex processing. These applications may scale super linearly with data size [40], [68]. Consequently, as data size increases, applications might take longer to be executed, leading to the necessity of high-performance computing systems.

With the aim of enabling efficient executions of data intensive applications over HPC systems, there are many studies that have obtained good results in time reduction and efficient use of resources; these studies range from the analysis of the effectiveness of I/O system, to the design of appropriate strategies to define and access data structures [31]. In this work, we have focused on dividing the workload of the data intensive applications into smaller chunks[1] (according to the *Divisible Load Theory*, DLT [13]) to ensure that the workload of the application can be manageable. By doing this, the size of the workload is reduced and enables parallelism. However, once the workload has been divided, other issues related to disk access or load balancing may appear. The partition of the initial data set is translated into easily tractable data. Nevertheless, complexity in parallel systems entails the design of smart data management policies to avoid performance loss.

1.1.2 Performance Issues

The performance of data intensive applications running in parallel systems is significantly affected by the dynamic effects of factors arising from characteristics of the application, the algorithm and the system. These factors typically induce load imbalance –a major

[1] A chunk refers to each independent data piece generated from dividing the initial workload. From now on, the term *data chunk* will be used to refer these pieces.

source of performance degradation in scientific applications [9]. Uneven distribution of workload among processing units in a parallel algorithm often results in some processors being idle and underutilized while others are heavily loaded. Many important problems in science and engineering have dynamically changing workload distributions. N-body simulations, adaptive meshes and discrete-event simulation are some examples. In these situations, the workload distribution is typically irregular and changes during execution.

System factors as the variability in workload processing times; processors performance at run time; and network latency, may induce load imbalance among processors. The initiation and removal of other applications that are concurrently running, and the variation in processing times due to data characteristics also increase the potential of load imbalance. In parallel systems, dynamic scheduling techniques attempt to maintain balanced loads by assigning work to idle processors at run time. These techniques are used to correct load imbalances that may result from undesirable and unexpected events, such as intrusion of operating systems call or high network latency, among others. Additionally, if there are time (or cost) restrictions when running the application, the intended goal is to execute it in the shortest possible time without wasting resources. This goal can be achieved by estimating the appropriate number of processing units according to current (or previous) application performance.

In this work, we focus on the design of load balancing strategies based on two main approaches: (i) the distribution of data chunks according to their expected processing times; and (ii) the tuning of the used processing units according to application behavior. We have chosen these approaches with the aim of providing an effective and wide-range solution to the problem of load imbalance, by taking advantage of the processing variability among the pieces (data chunks) of the workload.

1.1.3 Dynamic Performance Analysis/Tuning

Designers and developers of data intensive applications are responsible for providing the best possible behavior of these applications in parallel systems (in most cases, without considering the influence of data in the overall behavior of the application). Therefore, any application will be useless and inappropriate when its performance is below an acceptable limit. Once the application has been implemented in parallel, developers must systematically test its functionality to guarantee its correctness. Then, to provide the highest performance, optimization process may be carried out to ensure that there is no performance bottleneck during the execution of the application.

The optimization process –also known as tuning process– requires the analysis of the performance of the application and the modification of critical application parameters.

The tuning process implies then several phases. First, the performance measurements must be taken to provide information about the application. This phase is known as the monitoring phase. Here, the information related to the execution of the application is collected. Second, this information is evaluated in the analysis phase. Performance analysis finds performance degradations, deducts their causes and determines the actions to be taken to mitigate these bottlenecks. Finally, in the tuning phase, appropriate changes must be applied to the application to overcome problems and improve its performance.

The most important and complex task of the tuning process is the performance analysis [63]. The causes of performance degradations can be found at different levels, such as an error in the initial design of the communication protocol in the application; underlying hardware capabilities; or inappropriate management of the application workload, among others. As a consequence, each parameter related to the execution of the application must be evaluated. In many cases, the application performance also depends on the input data set, implying potential performance variations between executions.

Application performance analysis and tuning can be difficult and costly. In this work, given the variable behavior of data intensive applications, we consider a dynamic approach for improving performance in such applications. We did not consider the *post-mortem* analysis and tuning approach, because in applications with changing behavior, solutions proposed after one execution may be obsolete for the next one.

1.2 Motivation

Data intensive computing has begun with the analysis and translation of massive amounts of data into information. Nevertheless, the size (or complexity) of the incoming data has subtle influence in how recent scientific studies are carried out. For example, by analyzing raw data, it is possible to build (and limit the space of) models that make simulations computationally tractable, or to derive insights through computationally driven experiments (or hypothesis testing).

Currently, there are a variety of tools and strategies for the management of data intensive problems. However, major gaps in performance improvement methodologies for such applications are still a reality. The problems are exacerbated by the many data intensive applications that require the computing power available from high-performance computing systems or massively distributed clusters of commodity machines.

As data intensive applications are one of the most recent beneficiaries of the HPC systems, running efficiently such applications may be a challenge. Several solutions has been proposed in the last few years. These solutions range from the migration of data

intensive applications to *Infrastructure as a Service* (IaaS) or *Platforms as a Service* (PaaS) systems [3], to specially designed architectures to reduce system latencies, such as I/O or communication latencies [23], [85]. In most cases, proposed solutions are tightly coupled to the characteristics of the parallel system or to the application. Although previous solutions have reported outstanding performance of the applications, they have achieved it at the expense of migrating the application to a new parallel system or to a different programming paradigm.

Methods based on partitioning workload of data intensive applications into smaller (and independent) pieces often achieve far better performance results, since they enable the parallel execution of the application. Nevertheless, an inefficient data distribution may generate huge load imbalances at run time. Notwithstanding, any proposal capable of reducing load imbalances and therefore, overall execution time, will have incredibly beneficial implications for running such application in high-performance computing systems. Consequently, all factors described above summarize the motivation of this work.

1.3 Objectives

The ultimate goal of this work is to design, implement and evaluate a performance improvement methodology for parallel data intensive applications with divisible workloads and variable behavior. We address this problem by designing load balancing strategies for data intensive applications that perform related queries or iterations over data sets that can be arbitrarily partitioned into smaller pieces. We can enumerate the objectives of this work as follows:

1. Conduct a study on the general characteristics of data intensive applications, to identify those factors that influence their performance.

2. Identify the performance factors that can be tuned dynamically, taking into account results obtained in the previous point, to identify possible solutions to performance problems in data intensive applications.

3. Design and implement a dynamic performance analysis and tuning methodology capable of improving performance of data intensive applications, in terms of execution time and efficient use of available resources.

4. Conduct an analysis of the problems caused by performance factors, such as the workload partition factor and the number or used processing nodes; and evaluate the effectiveness of proposed solutions to the aforementioned problems.

5. Analyze the proposed methodology to avoid possible performance degradations and propose solutions (if necessary).

6. Evaluate the behavior of solutions proposed through the methodology by simulation-based experimental study; and real execution of data intensive applications, designed and selected to confirm the proper functioning of the proposal.

1.4 Contributions

The contributions of the work are directly related to the achievement of the objectives we have outlined in previous sections. To this end, we have designed and implemented a methodology for improving performance of data intensive applications by dynamically tuning performance factors as the number of used resources and the workload partition factor.

Our proposal has been developed making the following assumptions about data intensive applications:

- The applications process, explore and/or evaluate large-scale input data sets.
- The initial data set of the applications can be arbitrarily partitioned into independent data chunks.
- The applications perform a set of related iterations or queries on the data set, e.g., search for similarities between several related proteins in a large database, or look for similar strings on the web.
- The performance of the applications varies significantly according to the input data.
- The characteristics of the input data of the applications may be unknown.

In this work, we present the following contributions:

- The introduction of a method for reducing overall execution time of data intensive applications by partitioning the initial data set into smaller data chunks [77]. The method is based on monitoring the computation time of data chunks to determine the order in which they should be scheduled in future explorations. The proposed distribution policy avoids load imbalances by processing data chunks with large execution times first, and subsequently filling in the possible inefficiencies with data chunks with shortest associated execution times. This load

balancing method is commonly applied in a dynamic scheduling approach called *factoring* [42], [9], that delivers in each iteration, data chunks of decreasing sizes.

- A method to dynamically tuning the size of the workload data chunks [79]. We have designed a method to change the size of the data chunks with the highest and lowest associated computation time, making it possible to balance the application workload. The method includes the dynamic division and gathering of data chunks (when partitioning cost is low); and the possibility of dynamically choosing among previously generated partitions (when the partition cost is too high). In both cases, besides the computation time, the calculation of the partition factor will consider the communication cost, memory use, and the number of available computing nodes.

- The definition of a method to dynamically estimate the maximum number of processing nodes that can be used for a given data set partition factor [77], [78]. This estimation is based on monitoring performance metrics of the application, such as the overall execution time, and the rate of utilization of each processing node. The method enables to infer that the minimum execution time for an exploration is limited by the data chunk with the maximum processing time.

  Analytical expressions (described in chapter 3) are used to evaluate the collected metrics to determine the behavior of the application, and the maximum number of processing nodes that can be managed. If an execution presents a large number of idle processing nodes, the method will use gathered data to determine a new value for the processing nodes to reduce performance degradation.

  Additionally, the method is able to estimate the application execution time for an exploration using a certain number of processing nodes, to decide if this parameter should be changed. The criteria for deciding the appropriated number of workers has been defined as an index relating the estimated execution time and the efficient use of the resources. Efficiency is defined as the relation between the mean computation time for each data chunk –that is the time each node has been doing useful work– and the total time the node has been available. Consequently, the method pursues the lowest value for the number of workers that minimizes both the exploration execution time and the efficiency loss.

We have designed the proposed methodology for homogeneous clusters with the aim of being able to define a base model because they may provide steadiness in certain factors, such as processing capacity and disk and network latency, and therefore, simplifying the definition of the model.

1.5 Research Method

The research in this work is oriented to the design, implementation and evaluation of a performance improvement methodology for data intensive applications; and is framed in the academic program of applied research of the Computer Architecture and Operating System Department at the *Universitat Autònoma de Barcelona*. Throughout this work, we have followed the (iterative) hypothetico-deductive method as the methodological frame for performing the scientific analysis of data intensive applications [33]. The method is based on five major stages:

1. **Existent theories and observations**. Pose the question in the context of existing knowledge, theory and observations.

2. **Hypothesis**. Formulate a hypothesis as a tentative answer.

3. **Predictions**. Deduce consequences and make predictions.

4. **Test and new observations**. Test the hypothesis in a specific experiment/theory field.

5. **Old theory confirmed within a new context or new theory proposed.** When consistency is obtained, the hypothesis becomes a theory and provides a coherent set of propositions that define a new class of phenomena or a new theoretical concept.

The loop 2-3-4 is repeated with modifications of the hypothesis until the agreement is obtained, which leads to 5. If major discrepancies are found, the process must start from the beginning. The results of stage 5 have to be published. Theory at that stage is subject of process of natural selection among competing theories. The process can start from the beginning, but the state 1 has changed to include the new theory/improvements of old theory.

This work shares theoretical basis with the method developed and discussed in depth by Morajko [62], [63] for Dynamic Tuning of Parallel/Distributed Applications; and subsequently extended by César Galobardes [26], [25]. These proposals, together with the theory of performance analysis, constitute the first stage of the scientific research method of this work. Throughout this stage, we have conducted the first study of performance analysis for data intensive applications. Since dynamic tuning and performance improvements for applications developed under a Master-Worker approach has been widely covered in two previous theses, we are focused on the application of their theories to design,

implement and evaluate a novel and simple performance improvement methodology for data intensive applications designed under the same paradigm. In fact, steps 2 and 3 comprise the proposal of the methodology. Then, in step 4 and 5, we have evaluated and analyzed the effectiveness of the proposal. To this end, we have enhanced and extended existing performance improving strategies by including the proposed methodology. This has allowed us to develop the analytical simulator and the synthetic application used in the experimentation of our proposals.

1.6 Thesis Outline

According to the objectives and research method described above, the outline of the remaining chapters of the work is as follows.

Chapter 2: Thesis Background. Introduces some basic concepts of performance analysis and tuning of data intensive applications. In addition, related work and specific concepts about Divisible Load Theory, load balancing techniques and dynamic tuning are also introduced.

Chapter 3: Methodology Description. In this chapter, we present the proposed performance improvement methodology, specifically designed for data intensive applications with divisible load.

Chapter 4: Evaluation of the methodology. This chapter describes the test scenarios and provides the explanation of experimental results for our proposal, taking into consideration adjustments on the workload partition factor and the number of processing nodes being used.

Chapter 5: Conclusions. Concludes the work and presents the further work and open lines for the performance improvement methodology for data intensive applications with divisible load.

The list of references completes the document of this work.

Chapter 2

Thesis Background

"Ce qui embellit le désert c'est qu'il cache un puits quelque part."

Antoine de Saint-Exupéry. Le Petit Prince.

The main goal of this work, is to design and implement a methodology capable of improving performance in data intensive applications. This methodology takes advantage of the divisibility property of the workload of the applications and enables load scheduling through dynamically tuning performance parameters, such as the workload partition factor and the number of resources being used. Consequently, before introducing the developed methodology, we want to describe the theoretical basis that have framed this thesis.

A data intensive application, is an application that can explore, analyze or, in general, process large amount of data. Data may consists of results sent continuously by sensors [44]; biological databases containing a large number of biological chains of nucleic acid (DNA) or proteins [87]; or results coming from physical experiments that need to be processed to extract knowledge [39], [38]. Examples given above, have two points in common: (i) collected data does not have any sense for the researcher in its initial unprocessed state; and (ii) all these data sets need to be computed, analyzed or transformed to produce useful information, and of course, knowledge. This conversion from raw data to knowledge has been named "Information-based computing" [61].

The transformation of data into knowledge (as shown in figure 2.1) may be an endless and resource consuming task. For instance, to process data streams coming from a single experiment of the Large Hadron Collider in Geneva (tens of terabytes a day [24]) could take months. In addition, along the conception and design of new computational systems, several factors related to generated data should be considered, e.g., the storage, the distribution and the management of data. Therefore, parallel and distributed computational systems have emerged as a solution to process large-scale data. These systems

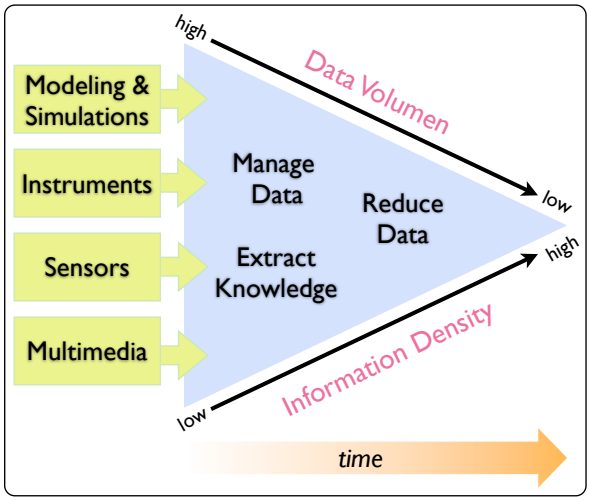


Figure 2.1: How does data becomes knowledge.

are composed of processing units working together to solve a computational problem in a concurrent manner.

To run data intensive applications in parallel and distributed computational systems, the parallel implementation of such applications is often performed by taking advantage of data parallelism (data can be distributed among available processors to be processed in parallel). Data parallelism leads to an additional challenge (besides the complex tasks of designing and implementing the application): how to distribute the workload of the application among the processing units to achieve a "good" performance? This "good" performance (depending on the application) might be minimizing the total execution time, minimizing the communication delays, and/or maximizing the resources utilization. Hence, the distribution choice becomes a *resource management problem* and should be considered as an important factor when running parallel data intensive applications.

Under resource management problems, scheduling techniques may be applied. These techniques refer to strategies for distributing load among processors. In some cases, scheduling is carried out by applying load balancing. Load balancing techniques refer to migrating load from one processor to another [90]. In both cases, the goal is the same: *achieve a balanced execution, where all processing units are computing the same amount of work, or during the same amount of time*. Nevertheless, some authors insist on using these terms interchangeably, e.g., by calling dynamic scheduling as dynamic load balancing [81]. In this work, we consider as scheduling the distribution of workload among processing units to achieve a balanced execution. Specifically, we base our methodology (that will be discussed in the next chapter) on two scheduling strategies: (i) divisible load scheduling [12]; and (ii) scheduling using *factoring* [42].

On one hand, divisible load scheduling assumes that computation and communication loads of computational problems can be arbitrarily partitioned among the processing units [12]. This approach considers there are no precedence relations among the data, therefore load can be easily assigned to processors in a system. On the other hand, factoring derives from techniques of scheduling parallel loops. In factoring, iterations are executed using decreasing size data chunks to assure that all processors finish their computation before an estimated ideal time. From basic scheduling taxonomies defined by Casavant and Kuhl [22], divisible load scheduling and factoring techniques can be considered as dynamic scheduling techniques. Divisible load scheduling is a method that pursues an optimal value along the execution of the application (at run time), while factoring *adapts* the load distribution on the fly according to the performance of the system.

With the same aim of keeping applications balanced at run time, it is necessary to monitor and evaluate the behavior of the application. To this end, the use of performance analysis and tuning strategies is an important complement. The instant in which the analysis is carried out depends on the characteristics of the application being evaluated. If the application has a constant and predictable behavior a static performance analysis can be performed previously to the application execution. However, if the behavior of the application is random within (or between) executions –and therefore unpredictable– the analysis should be perform at run time (dynamically), otherwise any estimation will be obsolete.

In this chapter, are described the basic scheduling strategies available to balance executions (section 2.1); selected scheduling techniques for this work, are based on approaches as divisible load scheduling (described in section 2.2); and scheduling parallel iterations throughout *factoring* (described in section 2.3). To conclude with the description of the model for dynamic performance analysis and tuning used to perform adjustments and modifications in data intensive applications (described in section 2.4).

2.1 Scheduling Strategies

In parallel computing, a programmer can define multiple processes that perform one or more tasks at the same time. These tasks represent the smallest operation that can be done concurrently; and the processes are an abstract software entity that executes its assigned tasks on a processor (processing unit). In most cases, a parallel program looks for, decomposing a computational problem into tasks that can be performed by the processes. In the literature, the phases of decomposition and of distribution are often called *partitioning* [32]. The aim of partitioning is to assign equal amounts of workload

to the processing units [30]. However, the number of generated tasks and the number of processes may often be not equal. If this happens, a process can be idle or overloaded with multiple tasks [51] and hence load imbalances appear. Decisions of how to distribute the task (scheduling) and where to allocate the load (mapping) are important phases in the load management process. Consequently, the problem here is not only to decide how to split the computational problem into tasks, but also how to distribute (or schedule) and assign (or map) the generated tasks among processing units to achieve performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization [22].

The decision of how to distribute tasks in a parallel application, i.e., to define the scheduling policy, represents a resource management problem and needs to be considered an important stage when designing parallel applications. As parallel and distributed systems were gaining popularity, the performance issues related to tasks distribution were increasing too. In this work, for data intensive applications, a task is considered as each one of the obtained data chunks from partitioning the initial workload of the applications. In this case, the aim of scheduling the workload will remain the same: distribute data chunks among available processing units to ensure there are no idle processors while tasks are waiting to be processed.

Scheduling methods are typically classified into several subcategories (as shown in figure 2.2) [22]. According to Casavant and Kuhl [22], local scheduling is defined as a process performed by the operating system to fill with tasks each time fraction available in the processor. While global scheduling, refers to decide where to execute the tasks in a parallel system (set of processing units) by taking into consideration characteristics from the system or from the application. Global scheduling may be performed by a single central process, or it may be distributed among the processing units. According to the moment in which scheduling decisions are taken, global scheduling methods can be static (described in section 2.1.1) or dynamic, often called dynamic load balancing [90] (described in section 2.1.2).

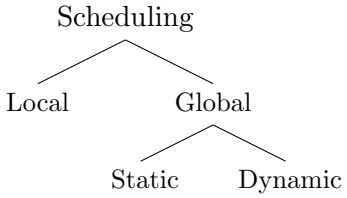


Figure 2.2: Main classification of scheduling methods [1].

[1] Adaptation from Casavant and Kuhl [22].

2.1.1 Static Scheduling

In static scheduling the goal is to minimize the overall execution time of an application while minimizing the communication delays. To this end, tasks are assigned to the processing units before launching the application; and information about execution time of tasks and available resources is assumed to be known *a priori*. Assignment of tasks is often performed without considering characteristics of the processing units, consequently a task is always executed on the processor in which it is assigned even if the processing unit presents a poor performance.

The major advantage of static scheduling algorithms is that they will not cause any run time overhead, because all the overhead of scheduling process is incurred previous to the execution. Nevertheless, for random or unpredictable execution times these algorithms will not perform well, because static scheduling algorithms rely on the estimated execution times of computing processes. For this reason, static scheduling algorithms are attractive for parallel programs that their computation times can be predicted. However, to find an optimal solution following the static approach for a large number of processors may result in an endless task [14].

The performance of a static scheduling algorithm in front of a dynamic and unpredictable program behavior can be disappointing. For example, parallel applications such as simulations of molecular dynamics have no benefits from static scheduling algorithms. Mainly because this type of applications perform a large number of calculations that require a certain degree of synchronization during the execution of the application, i.e., as atoms tends to move in the system, the computational requirements of each process may change from step to step. Consequently, the workload assigned to each processor needs to be redistributed periodically at run time and static scheduling is unable to perform this task.

The prediction of the behavior of parallel applications based on static characteristics or previous knowledge of the application, can be a long, tedious and exhausting task when the application presents a variable behavior. Therefore, when facing hard-to-predict applications a static scheduling approach may not be the most convenient to keep load balanced.

2.1.2 Dynamic Scheduling

The goal of dynamic scheduling is to redistribute tasks from heavily loaded processing units to lightly loaded processing units. In general, this process is known as *dynamic load balancing*. The aim of dynamic load balancing algorithms is to equalize the workload

among available processors while some communication costs are reduced. Nevertheless, such algorithms may generate non-negligible run time overhead, because of the analysis performed to decide the "best" processor to send the load [89].

A typical dynamic load balancing algorithm is defined by some inherent policies (adapted from a taxonomy presented by Bubendorfer and Hine [19]): (i) information policy, that specifies the amount of load information made available to task placement decision-makers; (ii) transfer policy, that determines the conditions to decide if a task should be transferred, that is, the current load of the host and the size of the task under consideration; and (iii) placement policy, that identifies the processing element to which a task should be transferred.

Dynamic load balancing algorithms may be carried out under centralized or distributed approaches. The main difference between them, is given by who is responsible of the scheduling decisions: a single processor, under a centralized approach, or all the processors, in a distributed approach. Additionally, a combination of both policies may exist, where information policies are centralized while transfer and placement policies are distributed. The decision of whether scheduling is centralized or distributed (or a combination of both) depends on the overhead introduced by the scheduling process, or on characteristics of the application, such as the communication pattern or the parallel paradigm. The advantage of dynamic load balancing (dynamic scheduling) over static scheduling is that the system do not need to be aware of the real behavior of the application before execution. Additionally, these strategies are more flexible to face unpredictable or random behaviors in the applications. Rommel [75] says that, dynamic scheduling is particularly useful when the primary performance goal is maximizing resources utilization instead of minimizing the execution time of the applications, because of the overhead introduced in the total execution time of the applications (in comparison with the static load balancing that does not introduce overhead). Nevertheless, in practice, both performance goals (minimizing total execution time and maximizing the use of computational resources) are pursued equally. The only restriction is that not always it is reasonable to pursue a global balanced state. Often, it is better relaxing the requirement of load balancing. In this way, dynamic scheduling strategies that pursue a partial balanced state between two possible extremes (completely imbalance and fully balanced) represent a certain tradeoff between the quality of the balancing and run time overhead [51].

The success of dynamic load balancing strategies depends on the likelihood of the phenomenon that an idle or lightly loaded processor and some overloaded processor coexist during the execution of an application [30].

In this work, since data intensive applications may present variable behaviors accord-

ing to data characteristics, we decide to use a dynamic scheduling scheme that can be performed at run time. In our methodology, two main approaches to keep the application balanced are considered. First, the possibility to split the workload of the applications into smaller data chunks, as proposed by Divisible Load Theory presented in subsection 2.2 and second, the capability to dynamically adapt the size of the data chunks according to the performance of the application, based on the characteristics of the scheduling parallel iterations (or loops) strategies described in subsection 2.3.

2.2 Divisible Load Scheduling

Parallel and distributed systems offer higher computing capabilities to process large computational problems. In these systems, one of the major challenges is to exploit its parallelism. In most cases, programmers are focused on improving functional parallelism. This means, to identify and to adapt the characteristics of serial programs to be properly executed in parallel. However, in data intensive applications there is another parallelism ready to be exploited namely, the *data parallelism*. Data parallelism means that large computational loads can be distributed among available processing units to be computed in parallel.

Parallel load distribution is mainly concerned to the partitioning of a single large load that originates or arrives to a processing unit. If one tries to process the original load as a whole, the resulting processing time may be unacceptable. To reduce the overall execution time, the initial load may be partitioned and distributed among the available processing nodes in the system. Nevertheless, it is important to combine knowledge from data characteristics with system-dependent constraints (such as communication delays and processor characteristics) to assure an appropriate data partitioning. The theory that studies the problem of partitioning and sharing load in parallel systems is known *Divisible Load Theory* (DLT) [12].

Divisible load theory is a mathematical model created to enable performance analysis of parallel and distributed systems by including both communication and computation issues [27]. The divisible load scheduling theory uses a system of linear equations to define the distribution of the load. Among the advantages of these models are: the ease of computation, the use of a schematic language, the equivalence to model network elements, and the facility to be applied in different fields [82].

Nevertheless, how the partitioning (or load division) can be done depends on the *divisibility property* of the load. The divisibility property refers to the characteristic that determines whether a load can be decomposed into a set of smaller pieces or not [13]. The

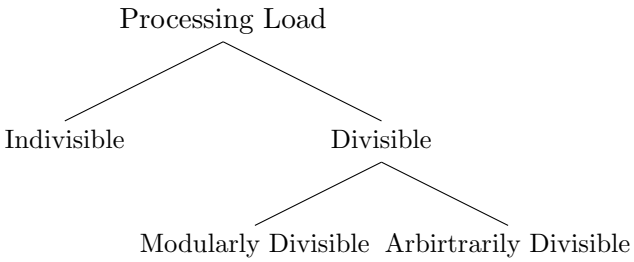


Figure 2.3: Classification of processing load [2].

divisibility property classifies load as shown in figure 2.3.

On the one hand, loads may be **indivisible**, in which case new pieces may be of different sizes, and cannot be further subdivided. Therefore, these loads do not have any precedence relations and they need to be assigned and processed entirely in a single processor. On the other hand, loads may be **modularly divisible** or **arbitrarily divisible**.

A **modularly divisible** load may be subdivided into smaller modules based on some characteristics of the load or the system. The processing of this load is completed when all its modules are processed, and the processing of these modules may be subject to precedence relations. Usually such loads are represented as tasks of interaction graphs whose vertices correspond to the modules, and whose edges represent interaction between these modules and perhaps also the precedence relationships.

A load may be **arbitrarily divisible** when all the elements in the load can be processed in the same way. These loads have the characteristic that they can be arbitrarily split into any number of load fractions (or data chunks). These load fractions may or may not have precedence relations, i.e., data can be arbitrarily partitioned but a precedence relation among the generated data chunks may exists, or if the data chunks do not have precedence relations, then each data chunk can be independently processed.

Some of the applications in which this divisibility property is satisfied, include processing of massive experimental data, image processing applications as feature extraction, edge detection, signal processing applications, and matrix computations. Additionally, since divisible load scheduling considers that both communication and computation loads can be arbitrarily partitioned among the parallel system [74], the theory is well suited for modeling a large class of data intensive computational problems. Under this scheme it is possible to model and schedule load distribution for parallel and distribution systems. These characteristics, along with the capability of arbitrarily splitting data intensive applications into smaller pieces, are the main reasons why this approach to improve

[2]Bharadwaj et al. [13].

performance in such applications is chosen.

In this work, this basic terminology of divisible load scheduling is followed:

Source: it is a processing unit (processor) that manages other processors and has all the data that should be computed. The source node divides data and distributes it among other processors (child or worker processors). At the end, this node will be responsible for collecting the results.

Job (workload): a large data file that can be arbitrarily partitioned into smaller pieces.

Chunk (piece): unit in which initial load is partitioned by the source node. The number of data chunks can be equal or greater than the number of processing units.

2.2.1 General Description of Divisible Load Scheduling

The aim of divisible load scheduling is to minimize the overall computation time while applying efficient load distribution strategies. In some cases, data partitioning algorithm is simple to implement. However, designing a strategy that efficiently utilizes the available resources in terms of computational power is not a trivial task.

The Load Distribution Model

In general, divisible load scheduling goes through the following process. The load to be processed arrives at a node, named the source or the root node. Since the architecture of the system may have processors equipped with or without front-ends, data processing may be different. For instance, in the with front-end case, in a system involving m processors, the root node partitions the load into m data chunks, starts the computation on its own chunk and simultaneously starts distributing the other data chunks to other processors one at a time in a predetermined order. The computation and communication events occur concurrently at the source node. On the contrary, in the without front-end case, the source node first distributes the data chunks to the rest of the processors and then it computes its own data chunk. The problem is then to choose the size of the load fractions in such a way that the goal of minimum processing time is met. Some times, when addressing the problem of load partitioning it is necessary to consider characteristics of both the system and the application, because not always dividing the workload into data chunks of equal size may result in a good performance.

Nomenclature

Below is described the standard notations used in the divisible load scheduling literature [27], [28], [13], [74]:

- $\alpha = (\alpha_1, \ldots, \alpha_m)$: load distribution vector (size of the load to distribute);
- α_i: load fraction allocated to processor p_i.
- $T(\alpha)$: total finish time with load distribution α.
- ω_i: ratio of the time taken by processor p_i, to compute a given load.
- T_{cp}: time taken to process a unit load (α_i) by the standard processor.
- z_i: ratio of the time taken by link l_i, to communicate a given load.
- T_{cm}: time taken to communicate a unit load on a standard link.

Then, $\alpha_i w_i T_{cp}$ is the time to process the fraction α_i of the entire load on the *ith* processor. Likewise, $\alpha_i z_i T_{cm}$ is the time to transmit the fraction α_i of the entire load over the *ith* link.

The *standard* processor or link mentioned above is any processor or link that is used as a reference. It could be any processor or link in the network or a conveniently defined fictitious processor or link.

2.2.2 Applicability of Divisible Load Scheduling

Since many parallel applications can be decomposed into smaller tasks [82], divisible load theory (or divisible load scheduling) has been easily adopted and applied into different fields, such as: managing large-scale data; image and video processing; biological and network applications; or sensor networks; among others. In addition, besides the origins of divisible load theory, its implementation has ranged from dedicated systems, such as homogeneous and heterogeneous clusters, to distributed systems as Grid and Cloud. In this section, we present an overview of some of these works.

At the very beginning, most of the studies were performed on homogeneous clusters. For instance, Drozdowski and Wolniewicz [34] employed four different testbeds (such as pattern-search, file compression, database join, and graph coloring and genetic search) to validate the effectiveness of the divisible load strategy. They concluded that divisible task model is capable of accurately describing the reality for each one of the test beds. Nevertheless, they pointed out that for some data-dependent cases, such as genetic search,

the predictions obtained trough the model were not satisfactory, because of references to disk files or memory allocation procedures introduces great amount of uncertainty and dependence on the behavior of other software using the computer. In such situations, the assumption about linear dependence of the communication time on the volume of data was not fulfilled, and communication speed decreased with data size.

The work presented by Kim [47] also considered homogeneous clusters. They proposed an improvement over the initial work presented by Cheng and Robertazzi [27]. Their proposal was based on reducing communication times by saving at each processing unit only its specific load (instead of having a duplicated record of the whole load in each processor).

Real-time scheduling algorithms and the influence of design parameters on such algorithms were studied by the authors of [53]. They proposed to combine divisible load theory with an approach that consider the earliest deadline to finish a task, to enhance the quality of service in cluster computing. As a result, they pointed out that executing partitioned sub-tasks in a homogenous cluster where processors have different available times, leads to completion times lower or equal to the estimated.

Additionally, in the work presented by Lin et al. [54] were identified three important design decisions regarding real-time divisible load scheduling. These decisions referred to workload partitioning, node assignment, and task execution order. Therefore, they proposed a scheduling framework able to configure different policies for each one of such scheduling decisions. Moreover, in the work presented by Chuprat and Baruah [29], they devised efficient algorithms to determine the smallest number of processors to complete a job according to its deadline; and determined the earliest completion time of a job on a specific number of processors.

In the work presented by Veeravalli and Ranganath [88], they used the divisible load paradigm, to schedule processing of an image onto homogeneous and heterogeneous processors. Their aim was to minimize the total processing time of the entire image that is submitted to the bus network system. They considered edge detection as an example of an image processing application, which qualifies to use a divide-and-process-like strategy, where initial load can be partitioned into smaller independent data pieces and hence is supported by divisible load theory model.

Later, Drozdowski and Wolniewicz [35][36], considered scheduling divisible loads on a distributed computing system with physical restrictions, such as limited available memory. They took into account communication delays and heterogeneity of the system. The problem they studied consists of finding a distribution of the load in which the communication and computation time are the lowest possible. Their method was tested on

systems connected by star network, and networks in which binomial trees can be embedded (meshes, hypercubes, multistage interconnections), demonstrating that in many cases memory limitations does not restrict efficiency of parallel processing as much as computation and communication speeds does.

Brest and Žumer [15] studied the improvement in total execution time, for applications developed under a Master-Worker pattern using divisible load scheduling. Their method decomposes the program workload into computationally homogeneous sub-tasks, which may be of different size according to the current load of each machine in the heterogeneous computing system. They evaluated their proposal using the problem of continuous speech recognition and the asymmetric traveling salesman problem with promising results on improving total execution time of the applications. Their study reported the influence of the initial size of data partitions in the overall execution time of the application.

An additional approach to schedule divisible workloads on heterogeneous systems was proposed by authors of [10] and [11]. In their work, they proposed a multi-round method for resources selection based on the speed of the processors and on communication links, specifically they tackled situations where the communication-to-computation ratio was not too high. Similarly, in the work presented by [91], they studied the distribution of divisible loads in heterogeneous distributed systems using parallel applications designed under a Master-Worker pattern. They used multi-round scheduling to sent load to each worker, sending several data chunks rather than a single one. To solve this particular divisible load scheduling problem, in their work they defined the subset of workers that should be used, the sequence of communication to these workers, and the sizes of each chunk of load.

The introduction of an equal-size allocation scheme for divisible load is introduced first in the work presented by [49]. They have considered equal allocation scheduling as a reasonable policy when prior knowledge about processor and link capacities is missed. At the same time, they tried to identify how much the finish time is affected in comparison to using optimal scheduling policies.

Recent studies, are focused on translating divisible scheduling policy to distributed environments, such as Grid. In [69], a load distribution model was designed to achieve an estimated performance level for large jobs –common in grid applications. They defined an adaptive model that took into consideration both computation time and communication time to estimate the optimal distribution. This work was followed by [70], where an enhancement for the previous adaptive model was proposed. The aim of the new model was to distribute loads over all grid sites to achieve an optimal *makespan* for large scale jobs.

Moreover, many data grid applications may be decomposed into multiple independent tasks for parallel execution and analysis. This property has been successfully exploited for scheduling divisible load on large scale data grids by using genetic algorithms. In [1] an adaptive genetic algorithm was proposed to improve the representation of the data partitions and the initial population, and hence reducing total execution time. In addition, authors of [2] designed a load distribution model that considers both the communication time and the computation time to minimize the total processing time. This time reduction was achieved by an optimal estimation of the completion time and the optimal distribution of the tasks among the available processors in the Grid.

Finally, the work presented by [92] introduced algorithms of divisible load scheduling for data intensive applications. In their work, they identify the polynomial time algorithms to partition the input data and generate optimal mappings to collection of autonomous and heterogeneous computational systems.

In this work, since considered data sets may be arbitrarily divided, we propose to partition large-scale data sets into smaller independent pieces. This division is performed to enable parallelism in the applications and to facilitate the reduction of their total execution time.

2.3 Factoring in Master-Worker Applications

One of the most popular application patterns is the Master-Worker. That consists of one master process and several worker processes. The master sends data to workers, which process these data in parallel and send their results back to the master. According to the terminology defined by Massingill et al. [56], an application may be implemented under the following paradigms: embarrassingly parallel, separable dependencies, or geometric decomposition, among others. The difference between such paradigms depends on the nature of the problem. In *separable dependencies* or *geometric decomposition* paradigms, the master process may have to wait for an answer from all workers before sending them new tasks (because there are data dependencies among tasks). Otherwise, in the *embarrassingly parallel* paradigm the master just has to wait for the answer from each individual worker before sending a new task to that worker.

When there are data dependencies among tasks, the performance of the applications mainly depends on two factors: (i) the load balancing among available workers; and (ii) the number of workers being used. However, when tasks are mutually independent, the performance of the application only depends on the number of workers, because load balancing is achieved at run time (if there are enough tasks to be distributed). In both

cases, the proper number of workers depends on tasks granularity, resource cost, and the relation between computation and communication time [26].

In general, the purpose of any load balancing technique is to improve the efficiency of the application. For example, the technique has to be able to minimize the overall execution time, and to permit that all the processors involved in the computation complete their tasks at the same time. Our main source of inspiration to undertake the load imbalance problem has been the adaptation of *factoring* [42], [7] performed by Moreno et al. [65]. They analyzed the separable dependencies type of Master-Worker pattern; pointing out that such applications present performance problems that cannot be avoided statically because the actual behavior depends on the dynamic conditions at run time (computing power of the processors, communication features of the system, additional load on the system, and so on). Their adaptation of the original factoring equations for the Master-Worker pattern are described in subsection 2.3.1. This modification led to performance improvements in Master-Worker applications when there is a high variability of the computational load associated to the processes.

2.3.1 General Description of Factoring

In dynamic scheduling strategies load is balanced by assigning work to idle processors at run time. As a result, dynamic strategies consider variations on the behavior of the application. To overcome variable behaviors at run time, mainly caused by workload variable processing times or by the performance of processing nodes performance, the scheduling strategy has to be able to dynamically re-assign tasks. Consequently, dynamic scheduling algorithms are a powerful tool towards performance improvement of parallel applications via load balancing [7].

For dynamically load balancing parallel applications, algorithms that derive from theoretical advances in research on scheduling parallel loop iterations with variable running times, has been widely studied [52], [71], [42]. In the last decade, some dynamic loop scheduling algorithms based on *factoring* [42] were proposed.

Factoring-based studies derive from scheduling parallel loops with variable running times, and accommodate load imbalances caused by predictable phenomena, such as irregular data, as well as unpredictable phenomena, such as data-access latency and operating system interference. In these algorithms, loops can be executed in decreasing size data chunks. In this way, earlier larger data chunks cause relatively little overhead, and the unevenness produced can be smoothed by the smaller data chunks.

The factoring implementation for parallel-loops uses a decreasing chunk size and is inspired by a probabilistic model where processor run times are modeled as identical in-

dependent random variables. The main principle behind deciding the size of the chunks is that every piece must be processed before the ideal execution time of the whole parallel loop. Therefore, the chunk sizes are dynamically calculated during execution of the application.

The work presented by Moreno et al. [65] is focused on the parallel loop iteration distribution problem. They proposed a strategy to achieve load balancing based on pieces of variable sizes. In this approach based on parallel loops, the minimum work to be assigned to a worker is the parallel loop iteration, and usually the application must wait the end of the execution of a parallel loop iteration to continue its execution.

For an embarrassingly parallel pattern as Master-Worker, an *iteration* is considered the total workload that should be processed by the workers. Iterations may have some dependencies, and therefore to start the execution of a new iteration they need that the previous one has already finished. Additionally, they defined a *task* as the minimum amount of work that can be assigned to a worker. The set of tasks that a worker receives from the master is named *chunk*, and the sum of P (the number of processors) data chunks with the same number of tasks is named *batch*. Finally, the processing of the set of continuous *batches* with no mutual dependence form an *iteration*.

In the Master-Worker pattern, Moreno et al. [65] considered that the execution time of every task cannot be measured. In this sense, they proposed to measure the execution time of a chunk and then, to infer the task processing time rate by dividing the data chunk execution time by the number of tasks within the chunk. To this end, they defined a random variable as the task processing time rate; based on the number of tasks to be processed and represented as statistical parameters, such as mean and standard deviation. Moreover, they explain that if the use of a homogeneous system is assumed, all processors will have the same mean and standard deviation of task processing time rate.

2.3.2 Load Balancing Methods for Iterative Loops

In load balancing methods for iterative loops, the sizes of the data chunks are defined with the aim of minimizing the overall loop execution time. According to Cariño and Banicescu [21] these load balancing methods can be classified as *non-adaptive*, when the chunk sizes are predictable from information that is available or assumed before loop run time, or *adaptive*, when the chunk sizes depend on information available only during loop execution.

Non-Adaptive Methods

Non-adaptive load balancing methods for loops generate equal size data chunks or predictable decreasing size data chunks.

A *Static Scheduling* load balancing strategy consists of distributing all parallel loop iterations among the processors in a single batch and then wait the results. This type of strategy is inefficient because faster processors may be idle while the last processor finishes its assigned parallel loop iterations.

An opposite approach is the *Self Scheduling* strategy. In this strategy, a minimum chunk is distributed to each available processor following a multi round approach. Consequently, the best theoretical load balancing is expected because the imbalances in the execution times are compensated by faster processors. Nevertheless, this type of strategies may introduce a high overhead in scheduling and communications.

Fixed Size Chunking [52] is a generalization of non-adaptive methods that uses a constant chunk size. This technique searches for the optimal chunk size that minimizes the total execution time. The chunk size is constant throughout the process, and it is limited at one end by distributing only one parallel loop iteration at a time (Self Scheduling), and at the other end, by distributing all the parallel loop iterations in a single batch (Static Scheduling).

More sophisticated strategies are based on using decreasing data chunks sizes. In methods that generate predictable decreasing chunk sizes, the underlying idea is to initially allocate large data chunks and later use the smaller data chunks to smooth the unevenness of the execution times of the initial larger data chunks. It was first introduced by *Guided Self Scheduling* [71], which uses a greater chunk size for the first batches and then decreases according to the amount of remaining tasks to be distributed. The imbalances are caused mainly by workers that waste too much time to process last chunks. If the last data chunks have fewer tasks, the imbalance probability will be minimized. The chunk size is determined by the relation between the total remaining tasks and the total number of processors.

Factoring [42], schedules iterations in batches, where the size of a batch is a fixed ratio of the unscheduled iterations and the batch division into P data chunks. In general, the ratio depends on the mean and standard deviation of the iteration execution times (when these statistics are not known, a fixed ratio is used). The next batch size is calculated after all the data chunks in the current batch are scheduled or are calculated.

A combination of factoring and *tiling*, a technique for organizing data to maintain locality, is known as *fractiling* [5]. Finally, *Weighted factoring* [43] is the first strategy that incorporates information about processor speeds when determining chunk sizes. Thus, the

faster processors get bigger data chunks than slower processors. The relative processor speeds are assumed to be fixed throughout the execution of the loop, so the size of data chunks executed by a given processor monotonically decreases in size.

Adaptive Methods

In time stepping applications containing parallel loops that have to be executed in every step, the amount of computation in a loop may evolve as the application progresses. Additionally, the loads of the processors running the application may also be changed by the operating system. Under these evolving conditions, assigning chunk sizes to processors based on the above non-adaptive load balancing methods for loops may not yield the best possible performance. *Adaptive Weighted Factoring* [6], [8], attempts to incorporate both loop characteristics and processor capacities when determining the data chunks sizes. Within a time step, the iteration of a parallel loop is assigned to processors as in *Weighted factoring*; however, the processor weights are adjusted at the end of a step using information collected during the current and previous steps.

Adaptive factoring [7] relaxes the assumptions in the original *factoring* method that the mean and standard deviation of the iteration execution times are known before launching the application, and that these are the same on all processors. Adaptive factoring dynamically estimates these statistics at run time. Initial estimates are obtained from the execution times of data chunks from an arbitrary sized initial batch. Then, the sizes of succeeding data chunks are calculated according to obtained mean and standard deviation of previous measurements. Therefore, new estimations are refined by using more information from recently executed data chunks.

In this work, we incorporate the adaptation of the size of data chunks (based on the concept of factoring) to avoid load imbalances caused by larger pieces. To this end, generated data chunks may be adapted dynamically at run time to reduce long execution times and increase resources utilization.

2.4 Performance Analysis and Tuning of Parallel Applications

The aim of the methodology presented in this work is to improve performance of data intensive applications by solving load imbalances when partitioning and distributing data. Along this chapter, the presence of performance issues such as load imbalance in data intensive applications has been discussed. Load imbalances are often solved throughout

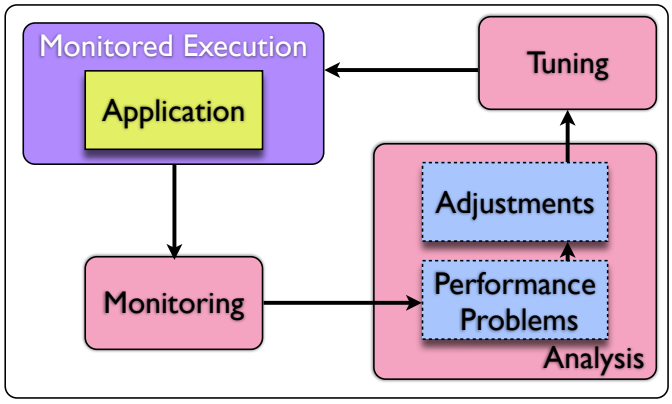


Figure 2.4: Performance Analysis/Tuning cycle.

efficient scheduling strategies, such as divisible load theory and factoring.

Nevertheless, to tackle inefficient situations may not be possible without carrying out an evaluation process. Consequently, one of the most important stages when improving performance of parallel applications is the *performance analysis*. Through this evaluation process, users may identify execution inefficiencies (such as the load imbalances mentioned in previous sections) and correct them to achieve a better performance (e.g., balanced executions). In literature, a performance analysis task may be generally described as three major steps, shown in figure 2.4 [45]: monitoring, data collection; analysis, data evaluation; and tuning, modification of the application.

In the **monitoring** step, knowledge about the behavior of the application is obtained. In this step, applications are often instrumented and executed to collect performance data, i.e., measurements of application performance. Gathered data is often translated into a graphical representation (traces) to facilitate understanding to the users. If the application processes a large amount of data or has long processing times, the size of these traces data may grow significantly.

Next, in the **analysis** step, user or tool evaluates data collected in the previous step to identify possible performance bottlenecks in the application. This process can be performed during the execution of the application or once the execution has finalized (dynamic or post-mortem). The decision of performing the analysis after or during run time mainly depends on the applications characteristics and the level of expertise of the analyst.

Finally, once the performance bottlenecks has been identified, in the **tuning** step, the appropriate modifications to the application are introduced.

As in the previous step, this process can be performed automatically or not. The behavior of the application is a key factor to choose whether performing the tuning process

statically or dynamically. A static tuning may be easy to perform, but if the application presents a highly variable behavior, most of the introduced modifications may not be appropriated for subsequent explorations. Nevertheless, besides of the flexibility of dynamic tuning, this process may introduce greater overhead in the overall execution time of the application.

2.4.1 Performance Analysis and Tuning Classification

Performance analysis can be classified into the two major groups shown in figure 2.5 [62], [26], [46]. This classification is based on who performs the analysis (the user or a tool), and when the performance analysis is carried out, that is if the analysis is performed along the execution of the application or after the application has finished (post-mortem).

When a performance analysis is carried out manually, the analysis and subsequent modifications are performed by the user. In most cases, after introducing the modifications, the application must be re-compiled, re-linked and restarted. While under an automatic approach, performance analysis and modifications may be carried out by a tool (dynamically or post-mortem), in some cases without needing to stop the application. However, the application performance analysis requires performance data gathered from the application executions or previous knowledge about the application. Therefore, the application must be monitored or modeled to get such data.

For the performance analysis, a "measure and modify" approach may be followed. Under this approach, the analysis involves a monitoring and a visualization step. Monitoring must be performed to achieve a global view of the behavior of the application. Through monitoring, performance data (measurements) are collected from the application execution. Visualization tools are used to present graphically gathered data. Under a manual approach, users must analyze these graphics to identify problematic regions and adapt the application source code, while under an automatic approach, the tool may perform (or suggest) such modifications. This process can be carried out until a certain performance is achieved.

On the contrary, a performance analysis based on a predictive approach looks for the construction of a performance model able to describe the behavior of the applications. This performance model eases the understanding of the performance issues and the prediction of future executions. Nevertheless, users have to be able to process the performance information to improve the application.

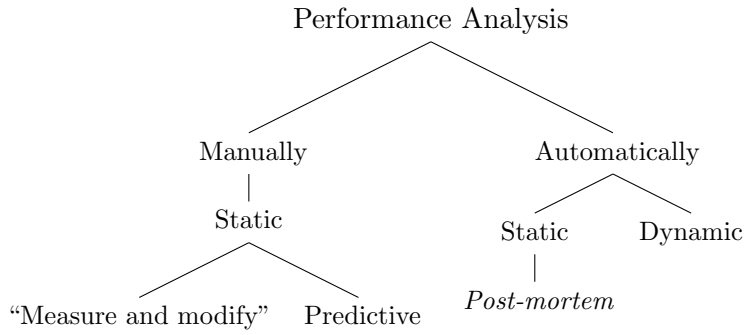


Figure 2.5: General classification of performance analysis.

Performance Monitoring

The performance monitoring process consists of two main phases: instrumentation and measurement collection [62]. The application is executed under a monitoring tool that helps to measure and collect performance data (this data is used to understand the application behavior). Applications are instrumented (statically or dynamically) at every point that must be evaluated. Once the application is instrumented, is compiled and linked with the appropriate monitoring libraries. The instrumentation is used to measure and collect performance data during the execution of the application.

The most used monitoring techniques, such as timing, profiling and tracing, are described next.

- **Timing**: this technique is based on measuring execution times. To this end, specific calls to timing libraries are introduced in the source code of the application. Such calls collect the execution time of the whole application execution or just from certain parts of the application, such as: functions, loops, basic blocks, among others. This technique is considered a simple and fast method to get an overview of the requested execution time for certain parts of the application.

- **Profiling**: this technique is based on getting accumulated values of a specific part of the code. Profiling facilitates the collection of a reduced set of performance data by measuring the number of times a part of the application (such as a loop, function, etc.) is called during the execution. The report provided by profiling is an overview of the execution of the application. It does not shows exactly where a performance problem is, but indicates dominating functions in the execution.

  Profiling may be carried out in two ways [62]:

  - *Counting*: records the number of times an event has occurred.

  - *Sampling*: takes a "snapshot" of a system state in a period. The program is sampled at fixed time intervals and collected data is saved.

- **Tracing**: this technique saves a sequence of some significant activities in the application execution. Traces may contain information about actions that have occurred: what was invoked (a function, etc.), which process call it (machine name, process identifier, etc.), and where in the code it is (in which line is the function, etc.). A generated global trace is representative of application behavior. Nevertheless, if it is no controlled, it can be one of the most invasive techniques because of the overhead introduced, and will generate a large amount of data.

After the application is instrumented and performance data is collected, generated trace files (or log files) are transformed into graphical representation by **visualization** tools. By doing this, users may observe illustrated information about the performance of the application (usually presented in Gantt charts, bar charts or pie charts) and detect performance problems.

Predictive Analysis

Predictive analysis of the performance of applications is based on the description of their behavior through analytical models. The aim of such models is to predict how future executions of the application will behave. The advantages of this type of modeling is that it eases the comprehension of the performance of applications for different input parameters (data or system parameters). Nevertheless, the construction of such performance models requires a high level of expertise from the programmer and its definition may result a complex process. Given this difficulty, some performance analysis tool facilitate this task by abstracting the application behavior model in high level expressions that are useful for the programmers [62].

Classical Performance Analysis and Tuning

In classical performance analysis, applications are instrumented by user/programmer to collect performance measurements from their executions. This data collection, as shown in figure 2.6 is performed by a monitoring tool and is often displayed to the user through visualization tools to facilitate the analysis process (using Gantt charts, or bars charts, among others). Understanding collected data is not always an easy task and requires a high level of expertise from the programmers. Next step requires programmers to look among the data log generated for the monitoring tool to identify possible performance

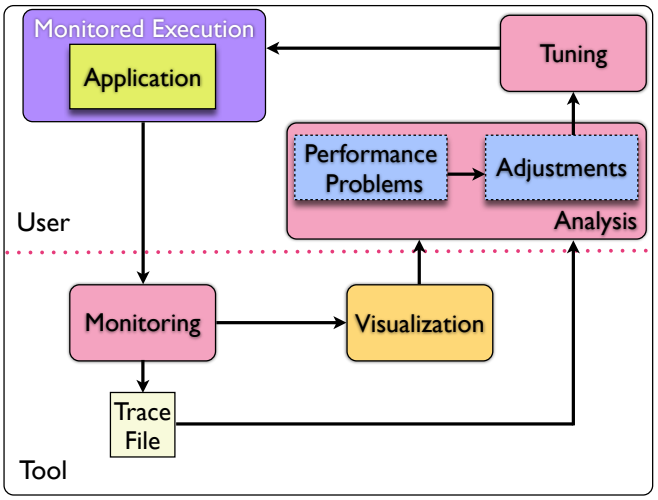


Figure 2.6: Classical Performance Analysis (Adapted from Morajko [62]).

issues and their causes. Then, a programmer has to manually relate detected performance issues to the source code. Finally, the application can be tuned (problems found in the application are fixed by changing the source code). After this step, the program has to be re-compiled, re-linked and restarted [62]. Therefore, the analysis of complex applications may become a difficult task to perform.

Analysis and tuning steps are performed statically (i.e., *off-line*) in the classical performance analysis. They may be performed previous to the execution of the application (in some cases even previous the implementation of the application) or after executing the application. A *predictive performance analysis* uses analytical models of the behavior of the application to provide an early insight of the performance of the application. While in the *trace-based* approach, the programmer analyzes trace files obtained from the monitoring step to discover performance problems. In both cases (in the construction of the model or in the analysis of large trace files), performing analysis and tuning tasks may be too hard and time consuming for the programmer/user.

In this sense, although the classical performance analysis has been widely used for many years it has some deficiencies:

- In most cases, the programmer needs a high level of expertise (and plenty of time) to find and modify the performance issues in the source.
- Proposed solutions based on a single execution might not work well in other executions when the behavior of the application is variable.
- If the application has long execution times, collected performance data (trace files) may not be easy to handle and understand.

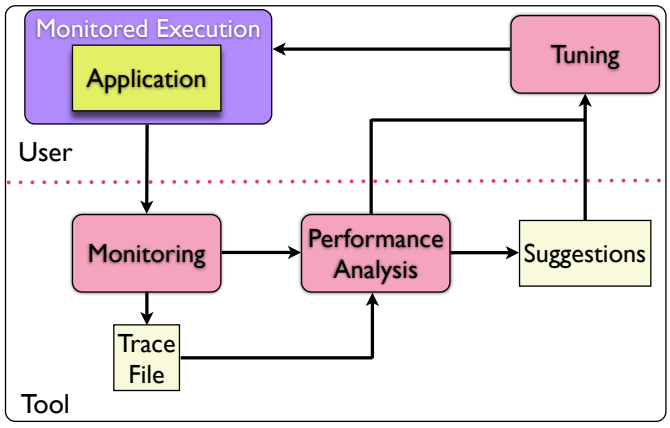


Figure 2.7: Automatic Performance Analysis (Adapted from Morajko [62]).

Automatic Performance Analysis and Tuning

As analysis and tuning steps may result too complicated to the programmer to perform, advances in the performance analysis process have propose to relieve the programmer of such tasks. In this sense, an automatic analysis approach is defined. Under this approach, parallel applications are instrumented previous to the execution and collected performance data is processed by the analysis environment (as shown in figure 2.7). The automatic performance analysis environment uses performance models (specifically design to each type of application) to evaluate the trace files or profiling obtained from collected measurements. This analysis looks for possible performance issues (*bottlenecks*) in the execution of the application and proposes some possible modifications to each bottleneck found.

This type of performance analysis process is commonly named *post-mortem static analysis*, and the visualization of performance data is replaced by an automatic analysis. Among all the advantages of using this type of performance analysis, it is worth mentioning: the significant reduction in the time invested in the identification of the performance problems; collected performance data tends to be more precise; and obtained observations may be useful to understand applications behavior.

Nevertheless, automatic performance analysis is also affected by some of the main problems described for the classical performance analysis [26]:

- The generation and storage of large trace files may result difficult to handle.
- Since the analysis is carried out after executing the application, if the application presents variable behavior the proposed modifications may be obsolete for subsequent executions.

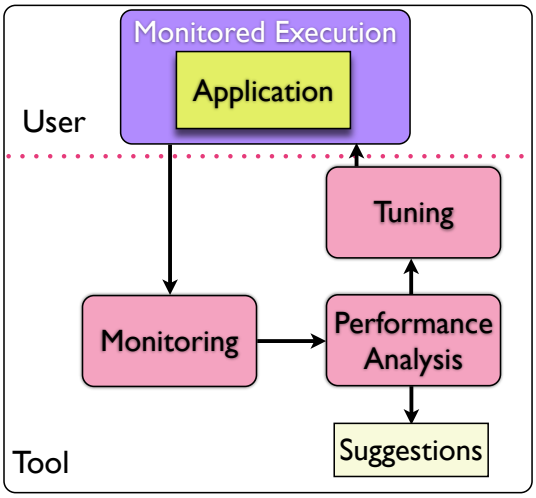


Figure 2.8: Dynamic/Automatic Performance Analysis (Adapted from Morajko [62]).

- The overhead introduced by instrumentation (needed to gather the performance data) is hard to predict and control.
- The modifications should be introduced manually, and the application must be re-compiled, re-linked and restarted.

Dynamic Performance Analysis and Tuning

The automatic performance analysis can be advantageous in comparison to the classical approach. Nevertheless, in both approaches the tuning step has to be performed manually by the programmer. This means modifying the source code and restart the program manually. To this end, programmers must have a high level of expertise and knowledge about the performance of the analyzed application. Besides, most of the commented approach need a trace file either to visualize a program execution or to make an automatic analysis. Not mentioning that if the applications have variable behavior according to the input data, the results of a *post-mortem* analysis may be useless or obsolete for subsequent executions of the same application. Consequently, static analysis are not suitable when there are dynamic conditions, such as variable behavior depending on the input data and/or variable behavior throughout the application execution.

In the dynamic performance analysis/tuning all the optimization steps are done during the execution of the application, as shown in figure 2.8. The search for performance problems is not based on trace files, because measurements needed to analyze the performance of the application are collected and introduced dynamically to the analysis step. Additionally, tuning step does not require manual modification of the application because

it can be supported at run time.

Dynamic performance analysis/tuning is an approach flexible enough to enable the modification of certain performance parameters of the application along its execution. This method bases performance modifications in previous and current behavior of the application. To this end, dynamic performance analysis/tuning should handle the following situations: (i) dynamic instrumentation and monitoring of the execution of the application; (ii) automatic performance analysis at run time; and (iii) automatic tuning of the application at run time. Consequently, this type of performance analysis relieves the programmer to modify the execution of the application manually, because the whole process is carried out automatically by the analysis tool.

In this work, we choose the dynamic performance analysis and tuning approach to evaluate the performance of data intensive applications. This decision is based on the significant variability in the execution times of the application between (and within) executions. Additionally, this approach facilitates the incorporation of our performance improvement methodology to the execution of the applications.

2.4.2 Performance Model

To improve performance in any parallel application, a detailed analysis process must be carried out. The analysis requires information of the application behavior to determine the existence of performance issues during the execution. In some cases, monitoring, analysis and tuning tools base their decisions in the observed application performance. One of the main differences between existent tools is the way how performance tuning decisions are made. For example, environments such as ActiveHarmony [86] determine good values for tunable parameters by searching the parameter value space using heuristic algorithms (in this case the Simplex Method). Although this heuristic approach may report good estimations for the tunable parameters, they treat the application as a black box, user does not know what is happening in the application.

In this work, an important factor is the identification of performance parameters and tuning points in data intensive applications that facilitates the analysis and estimation of adequate values for the tunable parameters. In this sense, a work developed in the Computer Architecture and Operating System Department at the Universitat Autònoma de Barcelona is the Monitoring, Analysis and Tuning Environment (MATE) [63]. This dynamic tool enables the automatic and dynamic analysis and tuning of parallel applications. The tool is based on analytical performance models provided by the user/programmer that describe the functioning of the application. Such performance models enable the estimation of minimal execution times for the applications and helps to predict the

performance of the application at run time[62].

The performance model may contain formulas and/or conditions that facilitate the estimation of the appropriate behavior for the application, by calculating the expected value of some of the performance parameters. The formulas use as an input data some previously defined information coming from the application –measurements to establish corresponding modifications. Once the parameters are modified, the performance of the application is improved.

In this work, the design of the performance analysis model for data intensive applications is based on the definition of performance models proposed by Morajko [62]. Therefore, a performance model for parallel applications should consider the following parts:

- The **measure points** or applications parameters, that describe specific values that should be monitored during the execution of the application. These values are used to evaluate the performance expressions and/or strategies.
- The **performance expressions and/or strategies**, represent the essence of the performance model. Designed expressions are used to identify possible performance issues and to determine the corresponding modifications to the face those situations. Therefore, selected expressions have to be able to describe the behavior of the application.
- The **tuning points and actions**, represent the modifications (changes) that should be introduced to the execution of the application to improve its performance. These "modifiers" consider safety conditions [26] to avoid possible inconsistencies in the execution of the application.

Based on the theoretical background presented in this chapter, the proposed methodology is based on dividing the application workload into smaller data chunks and scheduling those data chunks efficiently to achieve a balanced execution. Specifically, the methodology pursues the adjustment of: (i) the size of the data chunks; and (ii) the number of processing nodes used to assure an efficient execution.

Additionally, in accordance to the dynamic performance analysis and tuning strategy, the methodology follows four main steps: (i) the identification of the performance issues that can be solved at run time; (ii) the definition of the performance parameters that should be monitored; (iii) the definition of the parameters that had to be changed to overcome the performance problems; and (iv) the designing of the performance expressions or strategies that determine the appropriate tuning actions. The development and description of these steps are presented in the following chapter.

Chapter 3

Methodology for performance improvement of data intensive applications.

"Droit devant soi on ne peut pas aller bien loin."

Antoine de Saint-Exupéry. Le Petit Prince.

The recent data deluge needing to be processed represents one of the major challenges in the computational field. This fact led to the growth of specially designed applications known as data intensive applications. In general, to facilitate the parallel execution of data intensive applications input data is divided into smaller data chunks that can be processed separately. However, in many cases, these applications show severe performance problems mainly due to load imbalances, inefficient use of available resources, and improper data partition policies. In addition, the impact of these performance problems can depend on the dynamic behavior of the application.

In data intensive applications, load imbalances may be avoided by taking into consideration the ability to determine and support an adequate distribution of data and computational workloads. For example, to overcome load imbalances, it is important to find expressions and/or strategies that can be broadly applied for an efficient distribution of the workloads in the system [59]. In data intensive applications, this type of performance improvement can be achieved by adjusting specific factors, such as: the scheduling policy, the workload partition factor or some specific characteristics of both the system and the application, such as the number of processing units (computational resources), to name just a few.

The main goal of this work is to propose a methodology that enables performance improvement for parallel data intensive applications. These performance improvements

are closely related to: (i) the reduction of the execution time of the applications, i.e., to provide a fast processing of large-scale data without modifying the algorithm of the applications; and (ii) the reduction of long periods of idleness in the processing nodes, i.e., to efficiently execute the application. Consequently, the proposed methodology tunes dynamically the execution of data intensive applications with the aim of processing large-scale data as fast and efficiently as possible.

The methodology is based on: (i) adapting the size and the number of data partitions to reduce overall execution time; and (ii) adapting the number of processing nodes to achieve an efficient execution. These modifications are carried out under a dynamic performance analysis and tuning approach. That means, the proposed tuning solutions are based on the evaluation of the execution time of the applications during monitoring process. From this evaluation are obtained the *performance metrics* that analytically describe the overall behavior of the application, e.g., average execution time, or standard deviation, among others. Once the application is monitored and evaluated, tuning process can be carried out. Nevertheless, in this work, the tuning of the workload partition factor and the number of processing units should be performed at run time because of the long and variable execution time of applications.

The rest of the chapter is organized as follows. The overall description of the methodology and the definition of the initial assumptions are explained in section 3.1. Then, the specific data management techniques designed in this work for data intensive applications are defined in section 3.2. The strategy to adjust the number of processing nodes based on the performance of the application is exposed in section 3.3. Finally, the performance improvement methodology is summarized and discussed in section 3.4.

Before going further, it is necessary to introduce an overview of the methodology in which all the management techniques are framed, as well as the initial assumptions about data intensive applications and some basic notation.

3.1 Overview and Initial Assumptions

In this section, the main characteristics of the proposed performance improvement methodology for data intensive applications are described. Execution time of data intensive applications varies depending on the characteristics of input data. Since input data characteristics are difficult to predict and data intensive applications have long run times, the methodology has to be able to monitor, analyze and tune the execution of the application dynamically. Otherwise, solutions proposed to overcome specific performance issues in one point of the execution might be obsolete at the moment they are applied.

This variability and duration of executions demands efficient and dynamic adaptations of performance factors.

Although it is difficult to predict the execution time of the application based on the characteristics of its input data, the behavior of the application in different iterations or queries over a specific data set may be considered analogous [18]. This assumption is based on the inherent similarity between the iterations. For example, when looking for a topic on the web, users may use as many correlative words as possible. In this way, each search performed in a period can be handled as a query, and there will be resemblance between them. When processing images of astronomic data, pictures taken from a certain galaxy, planet, black hole, etc. are processed and filtered following a similar pattern [20], [59]. Genome sequencing processes vast amounts of data closely related (from similar organisms or species) [80], [87], [72]. Iterative algorithms [41], [37] of indexing, data mining and machine learning that are applied in different fields of science, such as understanding the brain behavior based on data coming from magnetic resonance imaging (MRI) [73], present similar processing schemes that may be used to predict performance of subsequent explorations.

Moreover, to facilitate managing the input data, in most cases (if data may be divided), input data can be split into smaller and independent data chunks. These data chunks may be of equal size or not. For the sake of clarity, initial partition of the application workload is done in a fixed number of data chunks with the same size (as will be discussed in section 3.2.1).

After the workload is partitioned, data chunks should be scheduled to balance the load among available processing nodes. The methodology is based on the execution of applications in homogeneous clusters of workstations, where the computation capacity may be considered as constant and, in most cases, the disk and network latency are stable. Moreover, to make easier the initial design we used a *shared nothing* [60] processing approach. Under this approach, each node (consisting of processor, local memory, and disk resources) shares nothing with other nodes in the cluster.

In this work, we considered applications developed under a Master-Worker pattern, because of the following reasons: (i) it is one of the most popular patterns in parallel applications; (ii) it facilitates the analysis of data distribution among the processing nodes; (iii) the variation in the number of workers is a very influential performance parameter; and (iv) it has a simple communication pattern if workers do not communicate between them.

Therefore, the proposed methodology is focused on two performance parameters in data intensive applications: the workload partition factor and the number of process-

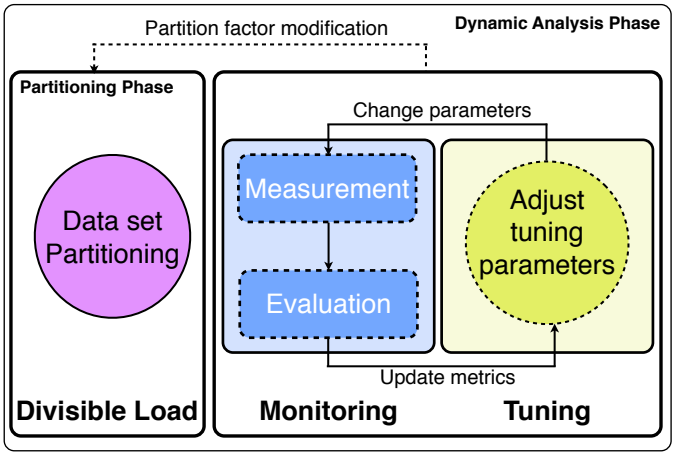


Figure 3.1: General description of the load balancing methodology.

ing units (computational resources) being used. In the design of the methodology, the following assumptions about data intensive applications are considered:

1. The initial data set of the application can be arbitrary partitioned into independent data chunks.

2. The application performs a set of related iterations or queries on the data set, e.g., the application searches similarities for several related proteins on a large database, or looks for similar strings on the web.

3. The performance of the application varies significantly (according to the input data), justifying the use of a dynamic performance tuning tool.

4. The characteristics of the input data of the application may be unknown.

The methodology, represented on figure 3.1, includes a partitioning phase and a dynamic analysis phase. In the partitioning phase, the initial workload is divided into smaller pieces named data chunks, and multiple alternative partitions are generated whether the cost of generating new partitions during the application execution is too high. The aim of this approach is to take advantage of the adaptation made in [64] to the *factoring* [42][7] load balancing policy. The modification proposed by Moreno et al. [64] is based on distributing the workload in chunks of decreasing size to keep execution balanced.

Along these lines, if the cost of dynamically generating new partitions is acceptable, the generation of data chunks will be performed at run time. However, the cost of partitioning the workload can be high for data intensive applications. If this happens, multiple partitions before executing the application are generated and then, during the execution the most appropriated one is chosen (as will be discussed in sections 3.2.3 and 3.2.4).

The dynamic analysis phase of the methodology is summarized in the algorithm 1, and notation used along this work (when referring the methodology) is described in Table 3.1. In this algorithm are shown the main processes that are executed in this phase: (i) the measurement phase; and (ii) the evaluation of the model, and (iii) the tuning phase.

Algorithm 1 Dynamic phase of the Load Balancing Methodology.

Require: N_q, N_w
Ensure: $tq_i(n)$
1: $n \leftarrow N_w$
2: $N_f \leftarrow max(N_f)$
3: $SP \leftarrow FCFS$
4: $i \leftarrow 1$
5: $j \leftarrow 1$
6: **while** $i \leq N_q$ **do**
7: **if** $i \neq 1$ **then**
8: $SP \leftarrow HFF$ {N_f distributed by processing time in decreasing order}
9: **end if**
10: **while** $j \leq N_f$ **do**
11: *master* sends data chunk j to the *worker*
12: *worker* process query i in data chunk j
13: *master* receives computation time C_{ij} from *worker*
14: $Ts_i \leftarrow Ts_i + C_{ij}$
15: $j \leftarrow j + 1$
16: **end while**
17: **update** μC_i
18: **update** $Tmax_{ij}$
19: **update** tq_i
20: **sort** data chunks {by processing time C_{ij} in decreasing order}
21: $Nw_{max} \leftarrow Ts_i/Tmax_{ij}$ {determines maximum number of workers}
22: $n \leftarrow min(tq_i(n), \rho)$ {updates number of workers to be used}
23: $N_f \leftarrow min(tq_i(n), Ef_n)$ {updates partition factor to be used}
24: $i \leftarrow i + 1$
25: **end while**

In the dynamic analysis phase, performance monitoring and tuning are carried out. Metrics such as data chunks processing time are collected, and the performance model described along this chapter is evaluated.

Both tasks are carried out dynamically to determine which tuning parameters must be adjusted for the next iteration. Specifically, to improve performance of the application at run time the following three performance parameters are taken into consideration:

1. The workload partition factor: this factor is determined based on characteristics of both the system and the application. The number of generated data chunks

Notation	Description
N_f	number of data chunks.
N_q	number of explorations (*queries*).
N_w	maximum number of processing nodes available.
j	data chunk identifier ($0 < j < N_f$).
i	exploration identifier ($0 < i < N_q$).
n	number of active processing nodes ($0 < n \leq N_w$).
$size$	data chunk size (in *MByte*).
λ	communication cost by MByte (BW^{-1}).
C_{ij}	computation cost (in secs) for the ith exploration and the jth data chunk.
C_i	total computation time (in secs) for the ith exploration. $\left(C_i = \sum_{j=1}^{N_f} C_{ij}\right)$
μ_i	average computation time (in seconds) for the ith exploration. $\left[\mu_i = (\sum_{j=1}^{N_f} C_{ij})/(N_f)\right]$
σ_i	standard deviation of computation time (in seconds).
ρ_n	performance index for n number of workers.
Ts_i	total sequential computation time for the ith exploration. $\left(\forall i \in N_q : Ts_i = \sum_{j=1}^{N_f} C_{ij}\right)$
T_{max_i}	maximum computation time for ith exploration.
T_{ideal}	ideal computation time for a parallel execution.
y	number of divisions for data chunks with $(C_{ij} > T_{ideal})$
$T_{group_{id}}$	computation time for the grouped data chunks.
Nw_{max}	maximum number of workers $\left(Nw_{max} = \frac{Ts_i}{Tmax_{ij}}\right)$
SP	scheduling policy.

Table 3.1: Summary of notation

should be large enough to guarantee that all processing nodes (workers) receive data to process.

2. The scheduling policy: to assure a load balance (all workers processing data the same amount of time), data chunks must be delivered dynamically in accordance to the behavior of the application.

3. The computational resources (processing nodes) to be used: to avoid long periods of idleness for the processing nodes (and therefore, inefficient executions), the tuning of this value is based on the total execution time of the application for a certain input data.

The execution of the application starts with a set of default values for both the partition factor and the number of processing nodes. The selection of these values is based on the criteria described in following sections. These values can be modified at run time and hence, they may improve the overall performance of the application during the execution.

The *measurement* phase is used to collect data chunks associated computation times. In the first exploration, a *First Come First Serve* (FCFS) scheduling policy is used because there is no previous information about data chunks computation time. Starting from the second exploration, once the computation times has been collected, the scheduling policy is updated to a *Heaviest Fragments First* (HFF) approach, where data chunks are sent according to their processing times in decreasing order, as described in section 3.2.2.

After this point, gathered data is evaluated in the *model evaluation* phase; and the corresponding modifications in the execution of the application are introduced in the *tuning* phase (if necessary). Through this process, the workload partition factor and the number of processing nodes can be adjusted. The tuning of such performance parameters is carried out to minimize the total execution time while keeping an efficient use of resources. The workload of the application might have been partitioned prior to execution, but the tuning of these parameters will be done dynamically and continuously at run time.

3.2 Data Management

In this section, we present the adaptation of the workload partition factor. Tuning process is performed trough the definition of the initial size of the data chunks; the modification of the scheduling policy to send first data chunks with large processing times; the division of the data chunks with the biggest associated computation times; and the join of data chunks with small computation times. The criteria for dividing or gathering chunks are

based on the chunks associated execution time (average and standard deviation) and in how many processing nodes (workers) are being used.

Conceptually, in the adjustment and management of the workload of the application (the modification of the size of the data chunks and the use of different scheduling policies) it is important to characterize data that constitutes the workload [58]. Data characterization provides to the user a broader vision of what to expect when processing it and facilitates management decisions. One main characteristic of data intensive applications analyzed in this work is the *divisibility property* of their workload. Our methodology takes advantage of this characteristic by arbitrarily dividing the workload into smaller independent data chunks. Then, the scheduling of generated data chunks among the available processing nodes is performed while the application is being executed. This process is based on the data chunks associated computation time and follows the *Heaviest Fragments First* approach that will be described later on this section. In this way, the application behavior is monitored and data distribution is carried out based on current observations.

The rest of the section is organized as follows. The description of the selection of the initial partition factor for the workload based on characteristics of both the system and the application is explained in section 3.2.1. Then, the scheduling policies used to deliver data chunks to the processing nodes (*First Come First Serve* and *Heaviest Fragments First*) are defined in section 3.2.2. The strategy to partition data chunks dynamically to reduce the total execution time of the application is explained in section 3.2.3. Finally, the grouping technique to reduce the amount of data chunks with short execution times that have to be sent to the processing nodes, is exposed in section 3.2.4.

3.2.1 Selection of the initial partition factor

In general, data intensive applications explore, analyze or process large data sets. This computation, if done serially, may be too time consuming or, in some cases, given the amount of data it may be even impossible to perform. To facilitate running this type of applications in current computational systems, some parallelization techniques have to be applied. In most cases, programmers take advantage of functional parallelisms to implement parallel applications and achieve efficient executions. Nevertheless, if the application does not allows for data parallelism, when processing large-scale data, the total execution time will remain excessive. In this sense, if the workload of the data intensive applications enables partitioning, input data can be split into smaller data chunks. By doing this, the parallelism of the systems can be exploited and the size of the input data can be manageable.

Notwithstanding, once the divisibility condition is fulfilled, questions about the num-

ber and size of data chunks may arise. Selecting how many data chunks should be generated, i.e., the workload partition factor, is a non-trivial task. On one hand, if considering unlimited processing resources, as more data chunks are generated shorter application execution times will be obtained. The main questions are: how small a data chunk should be? or how good is the cost/benefit relation between data chunk size and total execution time of the application? On the other hand, for limited number of resources, the following questions arrives: how many data chunks should we have to guarantee the shorter total execution time using efficiently all the available resources? Therefore, the difficulty resides in how to choose a trade-off between a well balanced execution and low execution times. If applications are executed using a large number of data chunks (i.e., a high partition factor), it may be easier to avoid load imbalances. However, the replication of the serial fraction of processing each chunk may introduce some overhead in the total execution time.

To decide a specific number of data chunks (or an initial common partition factor for all data chunks), we took into consideration some initial constraints, based on: (i) hardware parameters, such as the available physical memory, the network bandwidth, and the number of available resources (nodes); and (ii) application parameters, such as the total size of the workload, the cost of partitioning the workload (time), and the number of iterations or queries. Most of these parameters can be measured previously to the execution of the application and will not change at run time. Here, the main parameters of the system taken into consideration in the methodology are described:

- Available physical memory: the workload of a data intensive application can easily surpass the memory capacities of the processing nodes. During the execution, the processors share the physical memory of the node among application processes and operating system processes. Since data intensive applications have long runtimes, the selected size for the data chunks has to be small enough to avoid memory overflow, otherwise application performance could be degraded by paging or virtual memory accesses.

- Network bandwidth: since data chunks are moved through the network to reach each worker, the size of the data chunks should be small enough to avoid network saturation, but large enough to guarantee maximum utilization of the link. The identification of this value has to be done empirically by evaluating the behavior of the network when sending data chunks of variable sizes.

- Available processing nodes: the number of data chunks is directly affected by how many processors are available. A simple scheme is to set the number of

data chunks to be equal to the number of worker nodes available. However, this scheme may cause load imbalance as the processing time of each data chunk may vary significantly. Therefore, the number of data chunks should be large enough to assure all processing nodes to execute them in parallel.

In this work, the initial partition factor is chosen as an intermediate value that takes into consideration the physical characteristics of the available cluster. Extensive studies about data chunks size definition based on the physical characteristics of the cluster may be carried out, but they are out of the scope of this work.

Additionally, the methodology has been designed based on the following applications parameters:

- Total size of the workload: as the workload of data intensive applications has a continuous growth, the number of generated data chunks should be proportional to its total size. Thus, for a total workload size of around hundreds of Gigabytes, the size of generated data chunks should not be of Kilobytes (that means having a large number of small data chunks) because there will be too many pieces to distribute, and therefore, a greater management overhead may be introduced. At the same time, if the size of the workload is of a few tens of Gigabytes, the number of partitions should be (at least) similar to the number of processing nodes to assure all the processors will have work to compute.

- Partitioning cost: we consider partitioning cost as the amount of time invested in generating the number of data chunks. This factor is important because there are data intensive applications in which their workload must be preprocessed or formatted before being analyzed. Moreover, the frequency for updating input data has also subtle influence in the partitioning process. In such cases, partitioning the workload can be a long and tedious process and in consequence, data sets with costly partitioning process (in comparison with the total execution time of the application) that are updated too often, e.g., every day, represent workloads that may not be take advantage of our methodology.

- Number of iterations or queries: the total execution time of data intensive applications greatly depends on how many iterations or queries the application must perform. The number of times the application should be executed to complete all its processing makes easier the decision of how partitioned the workload must be. For instance, as more iterations the application performs, more partitioned the initial workload should be, because distribution and allocation time is lower, and therefore, (if possible) different iterations may be overlapped.

In this way, once the number of data chunks is empirically estimated, workload partitioning can be performed at any time. Nevertheless, as commented above, depending on the type of data intensive application, the partitioning cost determines whether all partitions or only the initial ones, are going to be generated before the execution of the application.

If the partitioning cost (time of creating data chunks from input data) is too high, workload partitioning must be carried out off-line. In this case, the workload may be replicated under different partition factors. Then, the proposed methodology will dynamically choose the best partition factor among those partition factors available. On the contrary, (if the partitioning process is fast enough to be carried out at run time), the methodology will generate the best partition dynamically.

Once data chunks are generated using the defined size, the application can be executed. After executing the application, and given the characteristics of data intensive applications, it can be seen that even when all data chunks are of the same size, the average computation time by data chunk may vary. We use this variation to determine future modifications in the size of the data chunks.

We propose to start the computation using a relatively high partition factor (determined by system and application characteristics described above) because there is no initial information about the cost of processing each data chunk. In this way, the methodology initially tries to meet the load balancing goal by distributing smaller data chunks.

3.2.2 Changing Scheduling Policy

In some applications, the distribution of data chunks to the processing nodes (or scheduling) is an influential parameter in the overall performance of the application. Scheduling looks for balanced executions where all workers compute the same amount of time. Scheduling decisions may be based on the speed of the processing nodes (which processors are faster), the cost of processing the computational load, or a combination of both. In this part of the work, given the special characteristics of some data intensive applications, we decided to fix the computational resources. Working with dedicated homogeneous clusters may provide, in most of the cases, constant computation and communication times. Therefore, this work is focused on the variability in computation times introduced by the combination algorithm-input data.

The computation time of the data chunks is not known beforehand and this makes the methodology impossible to establish the distribution order without executing the application. For this reason, some knowledge from the behavior of the application is

collected at the beginning of the execution. Since queries (or iterations [1]) may present some degree of similarity between them, the knowledge obtained from previous iterations can be used for subsequent ones. In this case, the initial iteration serves to label data chunks according to their associated computation times.

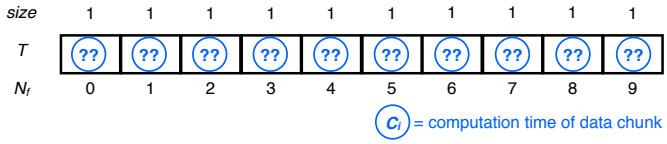
The methodology contemplates dynamic scheduling schemes, where, data chunks are distributed while the application is running. The reason of doing this is to avoid workers waiting too long for data chunks, e.g., as could happen with static scheduling schemes. Initially, data chunks are distributed following two on-demand approaches. First, under an approach named *First Come First Serve* (FCFS) that delivers data chunks of the workload to the requesting worker without following any rule; and second, under an approach named *Heaviest Fragments First* (HFF) [77] that delivers data chunks according to their associated computation time. Specifically, data chunks with highest execution times are distributed first. This way, possible load imbalances caused by the variability in data chunks computation times are smoothed at the end sending faster data chunks. Main characteristics and advantages of both policies are described next.

First Come First Serve (FCFS)

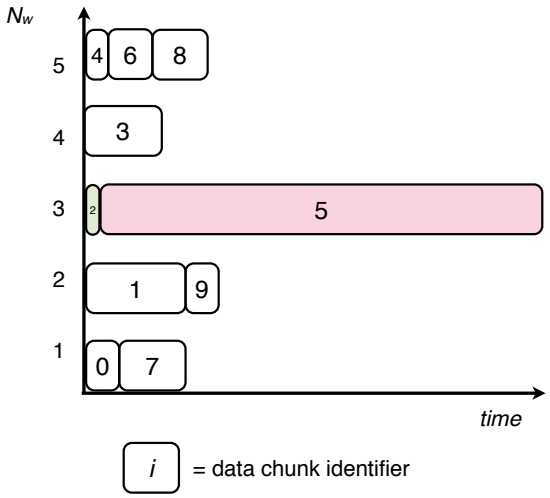
On-demand schemes are useful in applications with variable execution times because the load is distributed when is required by the processing nodes. By following an on-demand approach, the workers waiting time can be reduced. In this work, for initial executions there is no knowledge about the behavior of the application hence, based on the characteristics of the system and workload described in section 3.2.1, the application is launched using default values for the tunable parameters: a high number of workers (limited by the number of nodes available in the cluster) and a high workload partition factor. Data distribution is carried out following a *First Come First Serve* approach that delivers one data chunk at a time to every idle worker.

As previously mentioned, our methodology assumes that the iterations may be related to each other and they are processed sequentially. Once an iteration is processed, the execution time for each data chunk is stored and historical statistics updated. When distributing data under the FCFS scheduling policy, there is no guarantee for a balanced execution. For example, in figure 3.2, the workload is partitioned in ten data chunks of equal size and the computation time of the chunks is unknown (as shown in figure 3.2(a)). Data chunks are distributed to the available workers under the FCFS approach (figure 3.2(b)). Each chunk may have the associated computation times shown in figure 3.2(c).

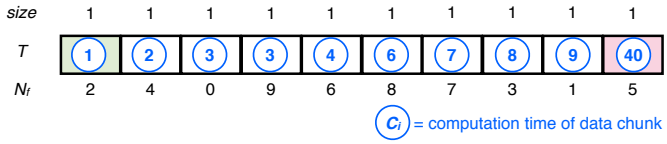
[1] In this work, the terms query, exploration and iteration are used interchangeably



(a) Initial Workload



(b) First Come First Serve Policy



(c) Measured Computation Time

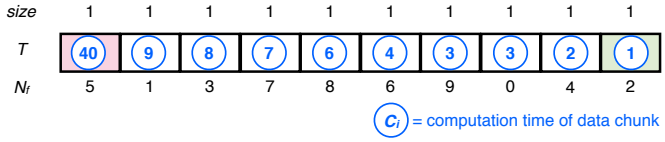
Figure 3.2: Initial Execution using First Come First Serve Scheduling Policy

In many cases, data chunks with highest execution times may be the last to be delivered. When this happens, if there are no more data chunks to distribute, workers without data to process will be idle until the last worker finishes processing. This situation may be improved by sending data chunks in accordance to their associated processing times, more specifically distributing first (in following iterations) those with the highest execution times.

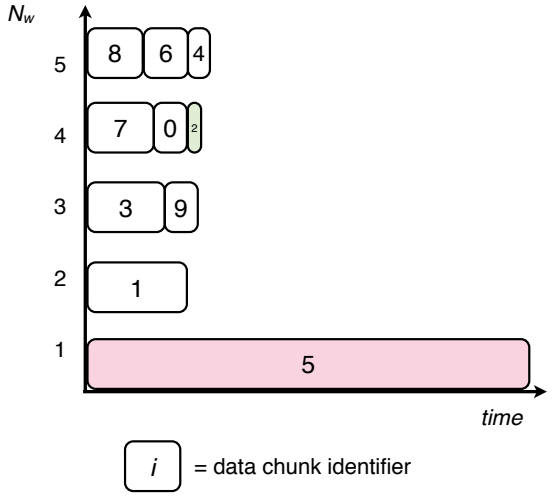
Heaviest Fragments First (HFF)

In subsequent explorations, once the data chunks are labeled and after considering a certain degree of similarity between the executions, data chunks can be delivered according to their processing time. Based on the collected metrics, data chunks are sorted in decreasing order before being sent to the workers. By doing this, it is possible to smooth

possible load imbalances caused by the variability in the execution time of each data chunk. Nevertheless, these imbalances may persist if there are large differences between the computation time of the data chunks. For example, in figure 3.3(a), collected metrics from the previous example are sorted in decreasing order. Then, these data chunks are distributed among a fixed number of processing nodes. Unfortunately, load imbalances still persist. This situation may be solved by adapting both the workload partition factor or the number of processing nodes. Both strategies will be discussed later in this chapter.



(a) Workload sorted in decreasing order



(b) Heaviest Fragments First Policy

Figure 3.3: Subsequent Executions using Heaviest Fragments First Scheduling Policy

When collecting data chunks associated computation times using the initial workload partition factor, two kinds of data chunks may be identified: (i) those whose computation time C_{ij} is above the average computation time of the exploration μ_i; and (ii) those data chunks whose computation time C_{ij} is below the average computation time of the exploration μ_i. For example, here $\mu_i = 8.00$, therefore maximum and minimum values are represented by data chunks number 5 ($C_{i5} = 40.00$) and 2 ($C_{i2} = 1.00$). The data chunk with the maximum computation time is labeled as T_{max_i}. In parallel executions using a fixed number of processing nodes, any node that finishes before the worker processing the data chunk labeled as T_{max_i} will be idle until that worker finishes. This waiting time results in a processing node being idle, and therefore, in an inefficient execution. For

example, in figure 3.3(b) it can be seen that total execution time may not be lower than the time imposed by data chunk number 5.

Given this time restriction imposed by data chunks with highest execution times, the proposed methodology also adapts dynamically the partition factor with the aim of balancing the load among workers . Therefore, our previous scheduling technique based on distributing data chunks according to their processing times in decreasing order (HFF scheduling policy) is complemented with the modification of the size of data chunks at run time to reduce total execution time. We named this strategy, **HFF + factor**. This strategy considers: (i) to divide data chunk(s) with highest associated computation time into smaller new pieces named *sub-data chunks*; and (ii) to gather data chunks with low associated computation times into bigger pieces named *parent-data chunks*.

The main criterion to decide when to partition, or when to group, is given by estimating the best possible computation time. In this particular case, *ideal* time (T_{ideal}, shown in expression (3.1)), represents the relation between the serial computation time of the entire workload, Ts_i, and the total number of available processing nodes N_w.

$$T_{ideal} = \frac{T_{s_i}}{N_w} = \frac{(\sum_{j=1}^{Nf} C_{ij})}{N_w} \tag{3.1}$$

Consequently, monitoring the execution time of the data chunks C_{ij} allows for calculating the average computation time μ_i and standard deviation σ_i, which are used for deciding the chunks that should be partitioned and the chunks that should be grouped. In next two sections, we describe the main characteristics of the techniques to modify data chunks sizes. First, *partitioning* is exposed in section 3.2.3, under this strategy, data chunks with highest execution times are divided into smaller pieces. Second, data chunks with short execution times are joined together under the *grouping* scheme explained in section 3.2.4. This approach is used to reduce communication and scheduling overheads.

3.2.3 Partitioning

As seen in previous sections, load imbalances may persist when data chunks are too big. In this case, if there is a time limit imposed for data chunks with highest execution times, adding more computing resources may not reduce the application overall execution time. On the contrary, if more workers are added and there is a big difference in execution time between the first and the last worker finishing computing (for instance, several tens of minutes), the efficiency of the execution drops almost completely.

The number of partitions impacts on the performance of the application[55]. In gen-

eral, the number of data chunks should be large enough to assure all workers to process them in parallel. A simple scheme is to set the number of data chunks to be equal to the number of worker nodes available. However, this scheme may cause load imbalances as the processing time of each data chunk may vary significantly. In this sense, when executing a data intensive application in parallel, the total execution time C_i is given by the time that last worker takes to finish processing. As seen in examples shown in previous sections, this delay may be caused by processing data chunks with long execution times. In accordance to the available computational resources, the methodology considers to divide this (or these) data chunk(s) into smaller pieces, and reallocate the generated pieces among the nodes. By doing this, it may be possible to reduce total execution time and improve load balancing.

Nevertheless, data chunks cannot have unlimited partitions, there is always a limit in the size of the data chunk that defines if a new chunk division is possible or not. In this work, we chose a conservative approach that prevents unnecessary divisions of data chunks with short execution times. The methodology has a threshold, defined by expression (3.2) that is used to determine whether a data chunk should be partitioned or not. Expression (3.2) defines that every data chunk with an associated computation time C_{ij} greater than expected ideal time, should be partitioned to met this constraint, i.e., to execute the application in T_{ideal}.

We have observed that in some cases, after dividing a data chunk, its computation time does not scale linearly; i.e., if the data chunk has a computation time T, and it is divided into 2 new pieces, the computation time associated to the pieces does not necessarily is going to be $T/2$. This behavior has been deduced through experimentation, and this non-linearity characteristic depends on both the algorithm and the data.

$$C_{ij} > T_{ideal} \tag{3.2}$$

Additionally, if the workload of the application enables an arbitrary number of partitions and the partitioning cost is not too high, new divisions can be generated during the execution of the application. Otherwise, the number of data chunks that must be generated from a specific data chunk should be chosen from pre-partitioned sets of the workload. For instance, in a workload of the application whose partitioning cost is too high, a single data chunk may be divided into predefined "sub"-data chunks (as shown in figure 3.4). In this case, each level corresponds to a specific partition factor. Therefore, if a data chunk has to be partitioned in 6 or 7 new data chunks, the methodology will select and distribute the set of data chunks closer to the estimated value, e.g., the partition

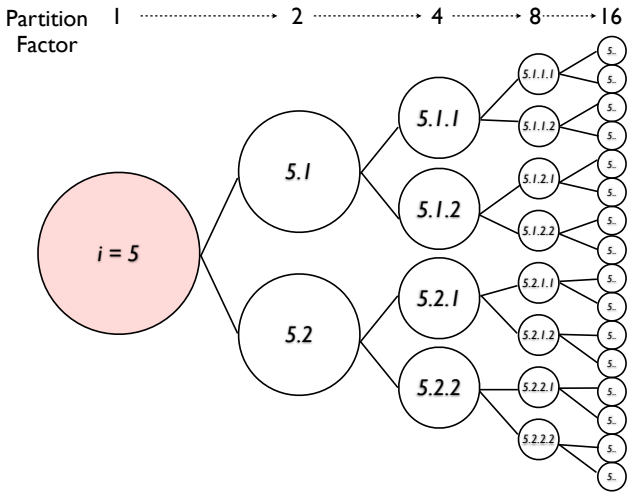


Figure 3.4: Data chunks partitioning in workloads with high partitioning costs.

factor in which this data chunk has been partitioned in 8 smaller pieces.

When partitioning data chunks, to close the gap between the expected execution time of an iteration (T_{ideal}) and the time of the data chunks with the highest execution time, there is a need to estimate the computation time for partitioned data chunks. This estimation facilitates the decisions about the order in which data chunks should be distributed. With this aim, a statistic of order N_w (E_{N_w} shown in expression (3.3)) estimates the upper bound for computation time of the new data chunks. This estimation is based on the average computation time, the standard deviation of the data chunks computation times, and the number of processing nodes used. By using these values in expression (3.5), it has been possible to establish in how many pieces the data chunks with highest execution times has to be partitioned.

$$E_{N_w} = \mu_i + \sigma_i * \sqrt{\frac{N_w}{2}} \tag{3.3}$$

For statistical purposes and with the aim of enabling the estimation of the number of new partitions, we have introduced to the statistic the corresponding modifications to keep consistent expression (3.3), resulting on the expression (3.4). These modifications are based on the characteristics of sampling distribution, where a certain probability is assigned to each variable to estimate its possible values. Centrality and variability qualities, such as mean and standard deviation are used to define those statistics.

$$E_{N_w} = \left(\frac{C_{ij}}{y}\right) + \left(\frac{\sigma_i}{y}\right) * \sqrt{\frac{N_w}{2}} \tag{3.4}$$

In this expression, the average computation time of data chunks that meets the time restriction (3.2) has been given by the relation between their associated computation time C_{ij} and the number of new data chunks generated y. Similarly, standard deviation of new data chunks will be represented as the relation between the standard deviation of processing the whole workload and the number of newly generated pieces. These assumptions are sound because the mean value of the sampling distribution of means is the same as the population mean; and the variance of the sampling distribution of variances equals the population variance divided by the sample size.

$$\left(\frac{C_{ij}}{y}\right) + \left(\frac{\sigma_i}{y}\right) * \sqrt{\frac{N_w}{2}} \leq T_{ideal} \tag{3.5}$$

Finally, to define the number of new data chunks to be generated, y must be calculated from expression (3.5), leading to expression (3.6). This expression is called into the methodology at the analysis phase for collected performance measurements (as shown in algorithm 2). Obtained result for the expected number of new data chunks is approximated to its closer integer value. In this way, if the expression indicates that a data chunk should be partitioned in *3.2* pieces, the methodology will approximate to 3 data chunks (if partitioning can be performed dynamically) or to 4 data chunks (if the closest size of data chunks is selected from previous partitions, as shown in figure 3.4).

$$y = \frac{N_w * \left(C_{ij} + \sigma_i * \sqrt{\frac{N_w}{2}}\right)}{(\mu_i * N_f)} \tag{3.6}$$

Algorithm 2 Partitioning data chunks with the highest execution times.

Require: N_f, T_{ideal}
Ensure: y
1: $j \leftarrow 1$
2: **for** $j = 1 \rightarrow N_f$ **do**
3: **if** $C_i > T_{ideal}$ **then**
4: calculate y
5: divide i into y pieces
6: **end if**
7: **end for**
8: **sort** data chunks in decreasing order
9: **continue** the execution of the application

For example, if an iteration is executed with the number of processing nodes $N_w = 5$ and partition factor $N_f = 10$, the resulting scheduling using the HFF policy may look as on figure 3.5. In this case, there are data chunks with different associated computation times, and the data chunk with the largest computation time (about 40 *time units*) can be easily identified. In this example the corresponding values for the average computation time, the standard deviation, and the expected ideal time are $\mu_i = 8.00$, $\sigma_i = 11.49$, and $T_{ideal} = 16.00$.

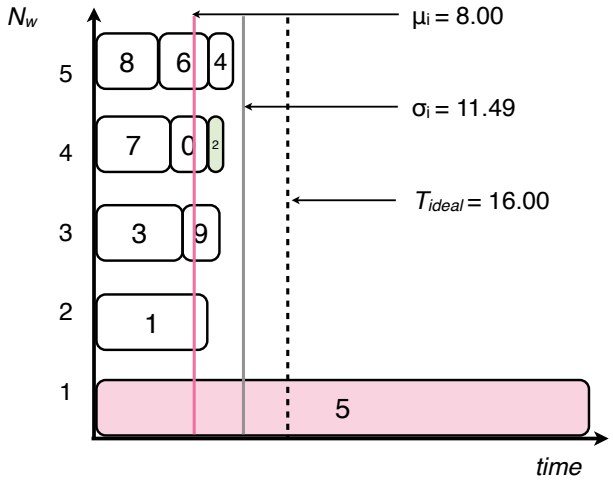


Figure 3.5: Initial execution under HFF scheduling policy.

After evaluating the restriction given by expression (3.2), and solving y from expression (3.6), the resulting value (rounded) for the number of pieces in which data chunk $j = 5$ should be partitioned is $y = 4$ (as indicated in figure 3.6). In this example, the workload can be arbitrarily partitioned at run time and new four data chunks (of equal size) are obtained from the data chunk number five.

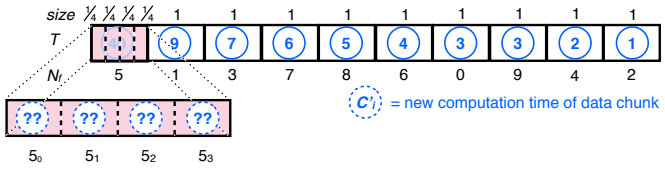
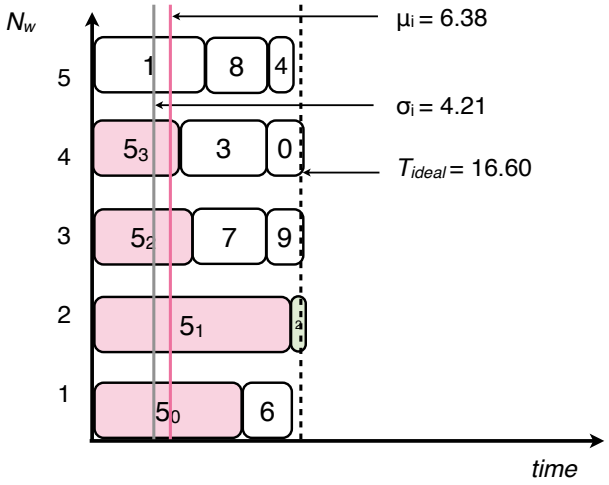


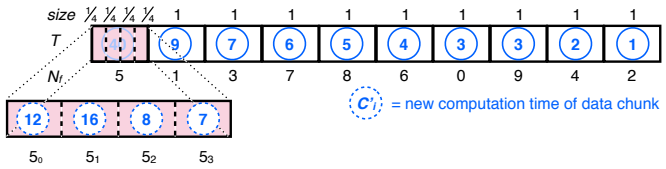
Figure 3.6: Partitioning data chunk with the highest execution times.

However, the computation times for the new data chunks are unknown. As proposed for the initial iterations, data chunks have to be sent to collect their associated computation times. Then, a subsequent exploration is used for labeling new data chunks with their associated computation times. In addition, to avoid possible load imbalances, new

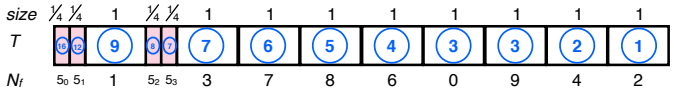
data chunks are scheduled when their original data chunk was expected to be sent (figure 3.7(a)). The new chunks can be labeled (figure 3.7(b)) and rearranged in decreasing order of computation time, for the next exploration, as shown in figure 3.7(c).



(a) Measuring



(b) Computation Times



(c) Sorting

Figure 3.7: Collecting the execution time of data chunks that have been obtained through *partitioning* process.

By using the partitioning scheme, it is possible to break time restrictions imposed by data chunks with highest execution time. Therefore, when having more data chunks to distribute it is easier to smooth the load balancing among available processing nodes. Consequently, the overall execution time of the application can be greatly reduced.

3.2.4 Grouping

In data intensive applications, partitioning the workload into small data chunks may improve load balancing and hence reduce total execution time. Nevertheless, having too

many data chunks can also produce time overheads caused by scheduling, communication or computing, as well as it may introduce greater variability in computation times between all the pieces. To avoid these overheads, the performance of the application is evaluated at run time and decisions of grouping or distributing bigger data chunks are taken dynamically.

In this sense, every data chunk that its associated computation time is lower than the expected ideal time, can be considered to be grouped. In *grouping*, according to the characteristics of the workload, data chunks can be joined in bigger pieces at run time or they can be chosen from previously partitioned data chunks. If the cost of gathering together data chunks is too high (equal or greater than the total execution time of the application) or the workload needs certain pre-processing, such as being formatted before being used, the resulting data chunk is selected from sets of data chunks generated before executing the application. In this case, to keep the grouping process simple, only consecutive pieces are considered to be joined (as shown in figure 3.8). Otherwise, all data chunks that meet the time restriction (of having associated computation times below the ideal time T_{ideal}) can be evaluated and grouped dynamically (at run time) in a bigger data chunk.

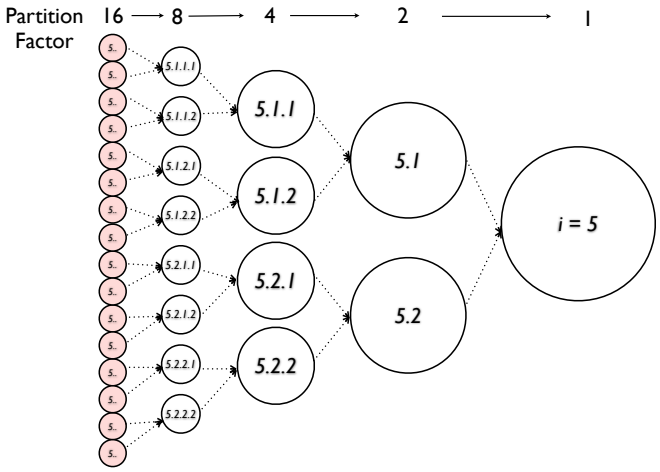


Figure 3.8: Data chunks grouping in workloads with high partitioning costs.

For every data chunk that can be gathered with other data chunks, the grouping strategy will stop when the sum of the associated computation times of the data chunks exceeds T_{ideal}, therefore generating too many data chunks with similar computation times. In this work, as a first strategy, we are assuming that the execution time of bigger data chunks may be at least equal to the sum of the smaller pieces that are gathered to generated them (their times may be greater because of the replicated serial fractions of each data chunk). Then, there should be enough data chunks with short execution times to fill the *gaps* left by imbalances along the exploration. In this sense, the methodology includes

a more precise approach to estimate the total time for the grouped data chunks. Under this approach, every data chunks that could be added to a bigger data chunk is evaluated using the threshold defined in expression (3.7), where the sum of the computation time of each data chunk in the group should not exceed the ideal time. Thus, computation time of new data chunks is kept way below ideal time enabling us to dispose of data chunks small enough to fill gaps on execution time and facilitate a balanced execution.

$$T_{group} \leq T_{ideal} \tag{3.7}$$

When applying the grouping strategy (described in algorithm 3), the execution of the application may look as on figure 3.9. In this case, data chunks with short execution times may be grouped with "larger" data chunks to create a lower number of pieces with similar computation times. For the example shown in this section, that data chunks are grouped without any additional restriction. In this case, the process of gathering data chunks may involve any of them, as long as the total execution time of the new data chunks is kept below the ideal time.

For the sake of clarity, in the example shown in this section, data chunks are left in the same order as in previous execution (figure 3.9). Then, bigger data chunks are created by grouping faster data chunks (those data chunks with the lowest execution times) while the restriction defined in expression (3.7) is met. The computation time when grouping T_{group} is evaluated by adding the computation time of the selected data chunks. If the sum is lower than the threshold, we continue grouping the chunks. For this particular example, the defined threshold is $T_{ideal} = 16.00$ and data chunks 5_1 and 6 ($T_{(5_1+6)} = 16.00$), 1 and 8 ($T_{(1+8)} = 14.00$), 5_2 and 7 ($T_{(5_2+7)} = 14.00$), and 3 and 5_3 ($T_{(3+5_3)} = 14.00$) can be grouped (as shown in figure 3.10).

As in the *partitioning* strategy, associated computation time for new data chunks is unknown. To avoid load imbalances, the associated computation time for new data chunks is considered the sum of the time of each piece that has been grouped. Therefore, new data chunks are distributed and executed when the data chunk with the highest execution time was expected to be distributed, i.e., grouped chunks will be delivered when chunk 5_1, 1, 5_2 and 3 were expected to be delivered (as represented in figure 3.11(a)). After this iteration, it is possible to label the data chunks with their new associated computation times. Collected measurements are used to sort the whole data set in decreasing order for the following iterations.

The advantages of using grouping when the workload is partitioned under fine-grained divisions are mainly related to one factor: the reduction of the number of data chunks.

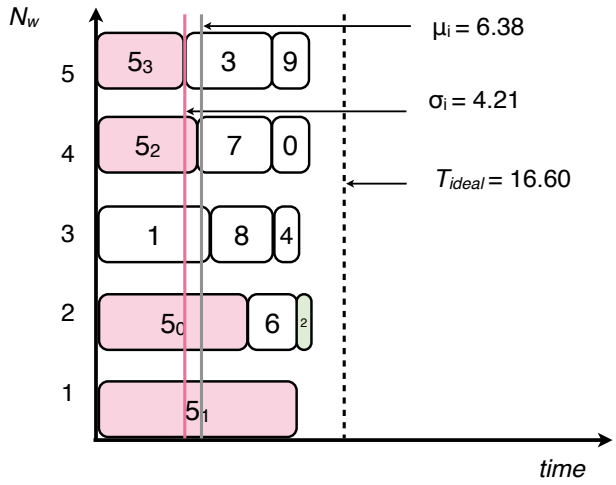


Figure 3.9: Execution before grouping using HFF scheduling policy.

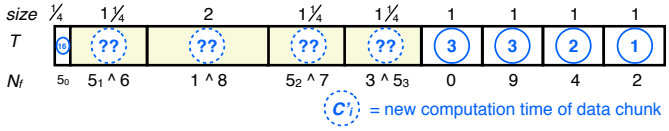
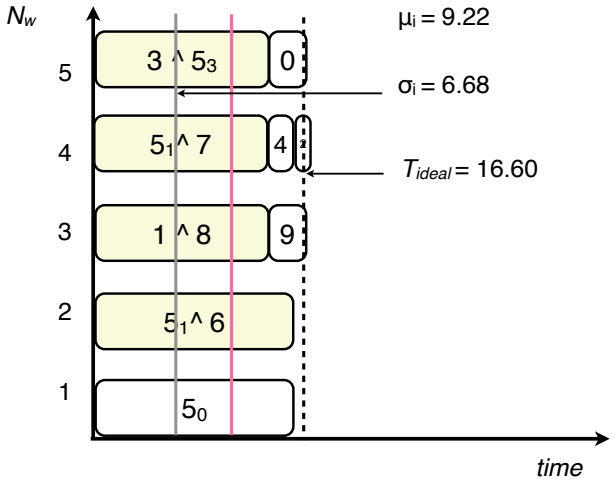
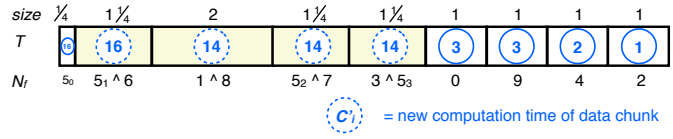


Figure 3.10: Data chunks to be group for subsequent exploration.



(a) Measuring



(b) Computation Times

Figure 3.11: Collecting the execution time of data chunks that have been obtained through *grouping* process.

Algorithm 3 Grouping data chunks with the lowest execution times.

Require: N_f, T_{ideal}
Ensure: grouped data chunks

```
1: j ← 1
2: a ← 0
3: b ← 1
4: create auxiliar vector[N_f]
5: for j = 1 → N_f do
6:    if C_i < T_ideal then
7:       saves i into vector[a]
8:       a ← a + 1
9:    end if
10: end for
11: while !vector[] do
12:    aux_vec[b] ← vector[a+1]
13:    T_group ← T_group+ vector[a]
14:    while T_group < T_ideal do
15:       T_group ← T_group+aux_vec[b]
16:       if T_group < T_ideal then
17:          vector[a] and aux_vec[b] are grouped
18:       else
19:          b ← b + 1
20:       end if
21:    end while
22:    a ← a + 1
23: end while
```

This reduction saves communication overheads that may be caused by sending too many data chunks along the network, as well as scheduling overheads that may be given by an inefficient distribution of data from the source (master) node. By sending less data chunks (which times are below the ideal), the variability among the workload in terms of computation times may be greatly reduced. In this way, data chunks with variable sizes are executed during the same amount of time.

3.3 Resource Management

As commented at the beginning of this chapter, to improve performance of data intensive applications both the workload partition factor, described in previous sections, and the number of workers to be used can be adapted. In this section, the description of the strategy for tuning the number of processing nodes to be used is exposed. In general, in data intensive applications (or almost every parallel application) the number of ma-

chines in which they are executed represents one of the most influential parameters in the performance of the application.

Applications initially designed to be executed under a parallel approach are the most benefited when running in large-scale parallel systems. This benefit is obtained after evaluating the application to enable its parallelism degree at almost every level (functionality, data, structures, etc). Unfortunately, not all the applications are designed to consider such parallelism, most of the applications have a serial version that after few modifications can be executed in parallel. Nevertheless, their performance is quickly degraded when adding more processing nodes. This performance degradation may be caused by inefficient distributions of the load to the processing nodes or by time constraints established by data.

Generally, data intensive applications have long execution times given by the amount of data to process or by the behavior of the algorithm with different input data. The reduction of such times may be achieved by adding more computational resources. However, this reduction can be conditioned by data characteristics. In this work, improvements in overall execution time of data intensive applications are restricted by some specific data chunks. Data chunks average computation time (and highest computation times) serve as indicator of the maximum possible improvement when executing the application. For this reason, after enabling a balanced execution by modifying the workload partition factor, the methodology assesses the evaluation of the number of processing nodes that should be active to achieve efficient executions, i.e., an execution where all processing nodes finish at the same time.

3.3.1 Adjusting Number of Workers

The presented methodology focuses on data intensive applications implemented under a Master-Worker pattern. First, the load is balanced by adjusting the size of the data chunks so that the difference between their associated computation times gets reduced. Then, if the workload partition factor is adapted, the number of processing nodes being used has to be modified to reduce possible inefficiencies (workers with longer periods of idleness).

The importance of these modifications is based on the idea of efficiently executing data intensive applications in the lowest execution time possible using the available resources. In an ideal scenario where there is no communication overhead or time constraints given by data, adding more resources will scale well. However, this situation is not common in this type of applications. In this work, these performance issues are assessed by adapting the number of computational resources used to provide an efficient execution.

In this section, the performance model used in the methodology to adjust the number of workers for data intensive applications is described. In this description, some initial constraints for the selection of the number of workers are explained first. Then, the expressions to estimate the behavior of the application after modifying the performance parameter are exposed. Finally, the definition of certain performance indexes to decide whether it is necessary to adapt the number of workers or not is presented.

Initial restrictions

When tuning data intensive applications, some of their performance factors cannot be adjusted without considering the influence that other factors have on them. In this sense, after tuning the workload partition factor (described in the previous section), and once the load is as balanced as possible, the number of computational resources (processing units) that are going to be used can be assessed. As a first approximation, the processing time for each data chunk is measured and collected to determine the maximum number of processing nodes n that may be used. This data is evaluated using expressions (3.8) and (3.9).

$$n \leq \frac{Ts_i}{T_{max_{ij}}} \tag{3.8}$$

$$n \leq \frac{\sum_{j=1}^{N_f} C_{ij}}{\lambda * size * N_f} \tag{3.9}$$

Since the minimum execution time for an exploration is limited by data chunks with the highest computation time, it is possible to infer from expression (3.8) that the best execution time may be achieved when using the number of workers defined by this expression. In this case, this restriction is used to calculate the maximum number of workers that would process all the workload efficiently (without long periods of idleness caused by workers waiting for other workers to finish their tasks).

An important characteristic when evaluating performance in parallel applications implemented under Master-Worker patterns is that the Master process may become a bottleneck. Therefore, it is necessary to estimate the maximum number of workers that can be managed by this Master process. If this limit is exceeded, processing nodes may become idle (long periods of waiting time shown in figure 3.12).

This estimation of the maximum number of workers is based on the cost of sending data chunks (communication time) to the workers and the time invested by the workers

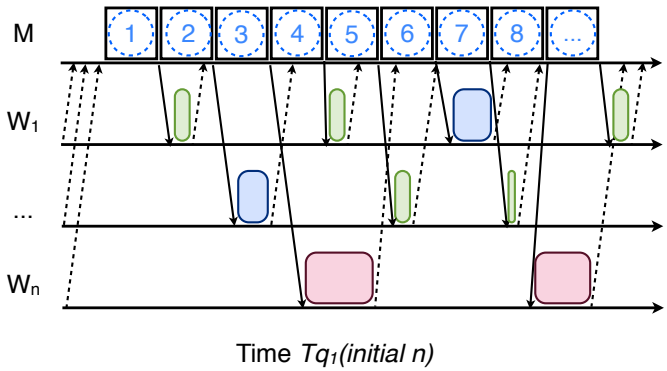


Figure 3.12: Idleness caused by the saturation of the Master process.

to compute the received workload (computation time). Thus, expression (3.9) describes this relation and estimates how many workers the Master process can handle. That is, a number for each the Master is capable of managing all the queries without becoming a bottleneck. Otherwise, if this restriction is not met, the Master process will be saturated and the undesirable situation of workers becoming idle while waiting for data may appear.

Based on the similarity principle between iterations, the methodology evaluates the expressions (3.8) and (3.9), and the minimum of these values is proposed as the number of processing units (workers) that should be active for subsequent explorations. Once this limit is found, it is possible to estimate the behavior of subsequent iterations or queries for this new value of processing units, based on previous collected measurements.

Total execution time estimation

The estimation of certain parameters values plays an important role in dynamic performance analysis. The estimation of future behavior of the application (or the possibility to establish a range for the expected total execution time) is relevant when proposing solutions to the performance problems. These values might be of great help to decide whether some modifications are adequate or not. For instance, when predicting how long will the application execution take in a certain parallel machine and estimating how many resources should be available to perform an efficient execution. Estimation of such values is closely related to specific parameters of the application. In this work, the estimation of the total execution time for subsequent explorations depends (directly) on the number of workers that are being used.

The estimation of the expected total execution time for an iteration is done through the expression (3.10). Total execution time calculation considers the time needed for sending the first data chunk to all workers ($\lambda * size * (n-1)$). Thus, the cost of making available

to each worker the initial data to process. In this work, sending step is done serially and the worker starts processing after it has received all the data chunk (blocking and synchronous sends). This type of communication facilitates the design of the methodology. The consideration of different communication schemes has been left as an open line.

Furthermore, to estimate the total execution time for a specific number of workers, the expression (3.10) relates two additional factors: (i) the computation time of each worker $((\lambda * size) + \mu_i)$; and (ii) the number of chunks it expects to receive $\left(\frac{N_f}{n}\right)$.

$$tq_i(n) = [\lambda * size * (n-1)] + \left(\frac{N_f}{n}\right) * [(\lambda * size) + \mu_i] \tag{3.10}$$

This expression does not consider the presence of data chunks with long execution times or the heterogeneity in the size of the data chunks. This absence is given by the difficulty to generalize the expression if such scenarios were considered. Moreover, by applying the restrictions discussed above (and the performance indexes described below), it might be improved the estimation of how many workers should be used.

Efficient use of resources

Data-intensive applications may present additional inefficiencies when adding more computational resources. In some cases, for this specific type of parallel applications it is possible to reach performance barriers that cannot be avoided. For example, the restriction imposed by data chunks with highest execution times represents one of the major limitations for such applications. When adding more resources, there is a point where a limit in total execution time may appear. In previous sections of this work, these performance issues where tackled by adapting the size of some "problematic" data chunks (those with the highest execution times). Nevertheless, inefficiency problems can be faced from another side: tuning the number of processing nodes being used.

The methodology evaluates constraints related to the computational capacity of the workers and the amount of data that needs to be processed. These values, together with the total execution time estimation, provide an overview of the expected application behavior when tuning the number of processing units (worker nodes).

Although values obtained from previous performance expressions are useful, it is necessary to measure the efficiency degree achieved in the execution of the applications. The criterion for deciding the appropriated number of workers has been defined as an index that relates the estimated execution time ($tq_i(n)$) to achieve efficient executions. Efficiency is defined by expression (3.11), and is described as the relation between the mean

computation time for each chunk (μ_i), which is the time each node has been doing useful work, and the total time the node has been available ($tq_i(n)$).

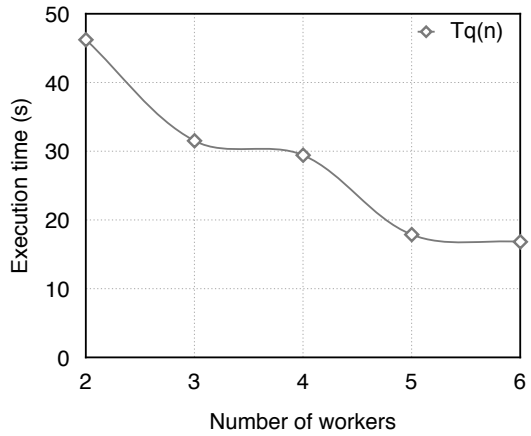
$$Ef_n = \frac{\mu_i * N_f}{n * tq_i(n)} \tag{3.11}$$

In the efficiency index Ef_n, obtained results describe the utilization of the processing nodes for a specific iteration. The greater the efficiency, the less workers idle, and therefore the greater the value of the index Ef_n. By applying expression (3.11), it is possible to evaluate if the tuning proposed for the number of workers does not affect the performance of the application, i.e., if this value does not cause future performance problems. For example, when observing a specific scenario (expected total execution time for an estimated partition factor and processing nodes) where Ef_n is evaluated, it may be possible to estimate how idle will the workers be in comparison with the overall execution of the application.

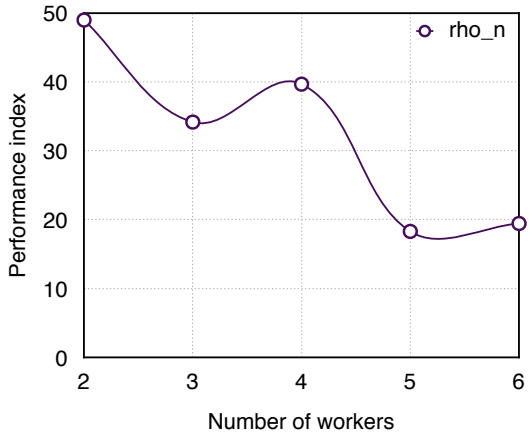
Under similar circumstances, the performance index shown in expression (3.12) enables the selection of active workers for the current execution. This expression reflects the maximum value of processing nodes that can be active without loosing efficiency in the overall execution of the application. Result obtained from ρ_n indicates that beyond that value, the total execution time may not be reduced and the worker nodes will become idle at some point (as shown in figure 3.13), i.e., the relation between reducing total execution time and having workers idle starts rising, and therefore no performance improvement is obtained. In this work, this behavior is interpreted as an inefficient execution, and it is one of the main situations our methodology tries to avoid.

$$\rho_n = \frac{tq_i(n)}{Ef(n)} = \frac{n * tq_i^2(n)}{\mu_i * N_f} \tag{3.12}$$

The methodology modifies the number of processing nodes that are being used based on the minimum value between the performance index ρ_n and the constraints for the number of workers described by expressions (3.8) and (3.9). To dynamically tune the number of workers, the constraints, the performance index and the efficiency of the application are calculated. In figure 3.13, are shown the expected execution time $tq_i(n)$ and performance index ρ_n for the example presented in section 3.2. In this example, after partitioning and grouping the data chunks the partition factor obtained is $N_f = 9$. Figures 3.13(a) and 3.13(b) presents the estimated execution time and the performance index, for workers ranging from 2 to 6. Here, it can be seen how the total execution time decreases



(a) Total execution time using $N_f = 9$ (example from previous section) and variable number of workers



(b) Performance index using $N_f = 9$ (example from previous section) and variable number of workers

Figure 3.13: Example of total execution time vs. performance Index.

when adding more processing units (workers). Nevertheless, the increasing in the number of workers, if there are load imbalances, may cause efficiency loss, and hence a variable behavior in the performance index. The analysis phase looks for the lowest value for both $tq_i(n)$ and ρ_n. In this example this minimum is around 5, i.e., the number of workers active should be $N_w = 5$ to provide an efficient execution.

During the analysis phase, to update λ and $size$ values, the following parameters have to be monitored: network parameters, such as bandwidth and setup overhead; communication parameters, such as message size; and computation parameters, such as processors (CPU) utilization, as well as the average execution time estimated by expression (3.10). Reported values from the performance analysis determine the corresponding modifications to the number of workers for subsequent iterations. After doing this, every step in the methodology is covered and a next iteration can begin.

Throughout this chapter, we have described the characteristics of our performance improvement methodology for data intensive applications with divisible load. The proposed methodology takes advantage of the divisibility properties of the workload of certain applications. First, to reduce overall execution time of data intensive applications, their workload is divided into smaller pieces named data chunks. Data chunks are distributed on demand to available workers to collect their associated computation times. Once all measurements have been gathered (and for subsequent iterations), the methodology sorts and distributes data chunks according to their associated processing times in decreasing order. Therefore, initial load imbalances caused by data chunks with the highest executions time are smoothed with faster data chunks. Second, our methodology considers the

adaptation of the size of the data chunks at run time to reduce their associated computation time and to reduce communication and distribution overheads. This process has been performed by dynamically partitioning data chunks with the highest computation times (or selecting the appropriate size from pre-partitioned sets in the case of high partitioning costs); and by joining fast data chunks into bigger pieces, respectively. Finally, the methodology assesses the estimation of how many processing nodes should be active for efficiently processing an iteration.

3.4 Summary

Although the methodology covers different aspects for load balancing and could be compared with any other method, we limit the comparison only to the most representative features of method. For comparison purpose, our proposal is based on three major fields of research: Divisible Load Theory (DLT) [13], Factoring adaptations [42], [6], [64] and Dynamic Performance Analysis and Tuning [63], [25].

Divisible Load Theory and Factoring rely on strategies to adapt the size of the workload to enable applications parallelism and reduce load imbalances. Additionally, both *scheduling* strategies (DLT and factoring) require previous information about processors capacity to define the partition factor for the workload. On the contrary, the proposed methodology uses application knowledge (recent and historical behavior of the application) to improve the performance of the application, by dynamically deciding whether to partition or not the workload. The size of the data chunks is chosen at run time, and corresponding adaptations are based on partitioning or joining data chunks according to their associated computation times.

On the side of dynamic performance analysis, our proposal uses gathered information to adjust the size of the workload partitions for the next iteration (as in methods based on DLT and factoring). In many cases, there is a root process (in our context a master process) that controls the amount of data that has to be distributed to the worker nodes. Root process takes scheduling decisions and evaluates the performance for subsequent explorations. In addition, once the load is balanced, the proposed methodology evaluates the efficient use of available processing nodes to avoid long periods of idleness in the workers.

The design of the methodology has followed the characteristics of performance monitoring, analyzing and tuning techniques described by Morajko et al. [63]. The purpose of our design it to facilitate the implementation of the method into dynamic performance analysis and tuning tools.

Additionally, it has been observed that, in data intensive applications, the total execution time may directly depend on the number of data chunks (and its sizes) and the number of processing nodes that are available. Therefore, the methodology considers the division of the initial workload into smaller pieces. These pieces are distributed on demand to the available workers nodes, and their associated execution time is collected for decision making. Decisions are taken in the Master process, and they are based on the following assumptions: workload partitioning may show a linear behavior, and there is certain similarity between iterations. Therefore, the methodology evaluates and adapts for next iterations both the workload partition factor and the number of processing nodes to be used. These analysis and tuning decisions are taken dynamically, i.e., while the application is running.

The proposed methodology requires the collection of information about the behavior of the application to make tuning decisions. This situation demands the execution of an initial iteration to "label" generated data chunks. Once data chunks are labeled, it is possible to determine the order in which they should be distributed to the available worker nodes for subsequent iterations. At the same time, labeling process makes easier the identification of data chunks with highest and lowest execution times. These data chunks are the candidates to be modified to improve the performance of the application, i.e., reduce total execution time or reduce overheads caused by distributing too much smaller pieces. In next iterations, the number of workers will also be tuned based on the current number of data chunks and their corresponding behavior.

To conclude, proposed performance improvement methodology allows the execution of data intensive applications with arbitrarily divisible loads in parallel systems. This suggest the use of our methodology as a feasible solution to the load imbalances problems in current parallel data intensive applications (as shown in next chapter). In addition, the structure of the methodology and parallel applications considered, may facilitate the implementation of a similar technique in platforms such as Cloud, which have been gaining popularity in the last few years.

Chapter 4

Evaluation of the methodology

"-Il est bien plus difficile de se juger soi-même que de juger autrui. Si tu réussis à bien te juger, c'est que tu es un véritable sage."

Antoine de Saint-Exupéry. Le Petit Prince.

In this chapter, we present the evaluation of the data management techniques and the resource management method explained in previous chapter. This evaluation is carried out to show how the proposed methodology improves the performance of data intensive applications. Performance improvement is achieved by dynamically adjusting the workload partition factor and tuning the number of processing nodes. It is worth noting that the proposed methodology considers data intensive applications, which workload can be arbitrarily partitioned into smaller pieces (*data chunks*), and that perform several related queries or iterations in this data. In addition, the methodology may report good results in presence of slightly differences between the iterations (as it can be seen later in section 4.1.3).

The purpose of the evaluation process is to confirm the benefits of using the proposed methodology. To this end, we have defined a set of metrics to observe both functional and performance features of the proposal. In this work, the most representative metrics are the total execution *time* and *efficiency*. Since our methodology is developed considering applications implemented under a Master-Worker paradigm, total execution time is determined by last worker finishing its processing. Additionally, we consider efficiency as the relation between average execution time and total execution time for one iteration. These metrics have been chosen with the aim of measuring several aspects related to the behavior of the applications in the presence of data chunks with variable execution time. In addition, some significant scenarios of data intensive applications, in which our methodology can be applied, have been included in the evaluation of the proposal.

In section 4.1, the selected *test beds* for the evaluation of the methodology are presented: a real data intensive application BLAST is explained in section 4.1.1; a distributed merge sort is described in section 4.1.2; and section 4.1.3 presents the analytical simulator. Then, we present the evaluation of the method and results of each strategy separately. The evaluation method used for selecting and adapting the workload partition factor are explained within section 4.2. Finally, evaluation of the method and results for tuning the number of processing nodes are described in section 4.3.

4.1 Selected Test beds of Data Intensive Applications to Evaluate the Methodology

The methodology is tested on homogeneous and dedicated clusters, where one single process is executed on a single core. Additionally, analyzed data intensive applications have been developed under a Master-Worker paradigm, where each worker is assigned to a different processing node and they do not communicate between them. Under this approach, a homogeneous cluster consists of a head node, connected via a switch to N processing nodes. We assume that all processing nodes have the same computational power and that all links from the switch to the processing nodes have the same bandwidth. The methodology assumes a typical cluster environment in which the head node does not participate in computation. The role of the head node is to divide the workload and distribute data chunks to processing nodes.

The methodology is evaluated using: (i) a real data intensive application, BLAST (subsection 4.1.1); (ii) a synthetic application based on the merge sort algorithm (subsection 4.1.2); and (iii) an analytical simulator (subsection 4.1.3). Each test bed has been selected to analyze different stages of the methodology. First, BLAST is used to check the overall behavior of real applications when applying our proposal. Then, the synthetic application facilitated the analysis of the workload partition factor adaption at run time (because the partitioning process is faster than for BLAST). Finally, the analytical simulator enabled the evaluation of the methodology for a wide range of scenarios of the selected data intensive applications.

4.1.1 Basic Local Alignment Search Tool - BLAST

As a real application, we choose NCBI BLAST (Basic Local Alignment Sequence Tool) [4] for assessing our proposal because it is one of the most widely used bioinformatics tools, and it satisfies the assumptions presented in section 3.1 about data intensive applications:

1. The initial data set of the application can be arbitrary partitioned into independent data chunks;

2. The application performs a set of related iterations or queries on the data set;

3. The performance of the application varies significantly (according to the input data);

4. The characteristics of the input data of the application may be unknown.

BLAST searches for *regions of similarity* in biological queries (nucleotides or proteins). It calculates the statistical significance of matches comparing the entrance query with large databases of sequences, such as GenBank [1] or Swiss Prot [2]. Based on heuristics, BLAST algorithm improves up to 10 times the exact match Smith-Waterman Algorithm [83].

BLAST is both a CPU and a data intensive application that can be executed in parallel and distributed systems by: (i) partitioning the query (input sequences) so that each of worker looks for similarities of a subset of queries (figure 4.1(a)), or (ii) partitioning not only the input sequences but also the database (figure 4.1(b)). The second approach allows the database to grow and still fit on the local disk or local memory of worker nodes. In some cases, workers communicate among themselves to generate the final combined comparison result. This application presents irregular processing times due to data characteristics. It processes biological databases of hundreds of GB [3] –and growing– that can be arbitrarily divided into non-dependent data chunks (in BLAST literature these chunks are called fragments).

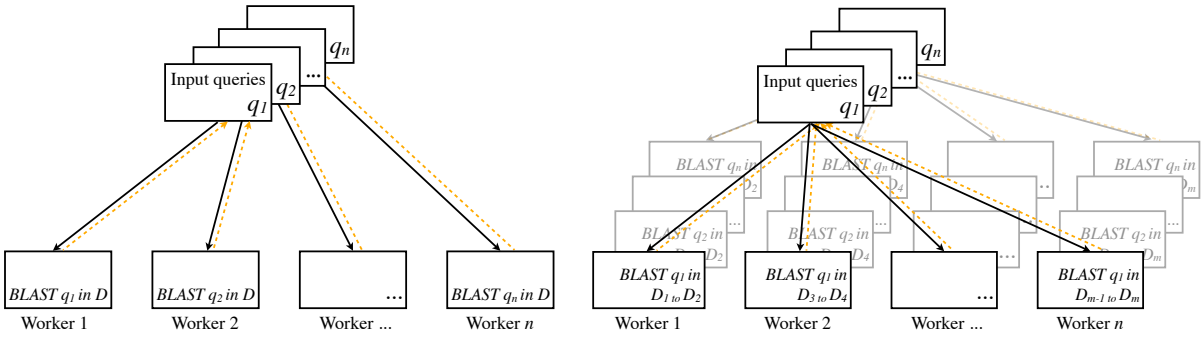
The first approach (i.e. partitioning only the input sequences) can be easily implemented using a scripting language to identify the input sequences, execute BLAST, and concatenate the result. However, according to Matsunaga et al. [57], in practice users still face the following challenges: (i) finding the ideal number of workers to use; (ii) identifying in how many partitions should the database be split; (iii) providing balanced executions; and (iv) recovering from the potential failure of some workers to avoid obtaining only partial results.

Most parallel BLAST versions, such as mpiBLAST [31], ScalaBLAST [68], CloudBLAST [57], CloudBurst [80] and AzureBlast [55], have been developed for homogeneous and heterogeneous clusters [31], Grid [68] and Cloud ([57], [80], [55]) platforms using the Master-Worker paradigm and taking advantage of the parallelism of the systems using

[1] http://www.ncbi.nlm.nih.gov/genbank/

[2] http://web.expasy.org/groups/swissprot/

[3] The *wgs* database available at the FTP server of the National Center for Biotechnology Information –NCBI– (ftp://ftp.ncbi.nlm.nih.gov/blast/db/) is of approximately 360 GB.



(a) Partitioning the query.

(b) Partitioning the query and the database.

Figure 4.1: Approaches to implement BLAST in parallel.

database or query partitioning. In general, input data is partitioned and the generated database fragments are distributed between all available workers. Next, each worker searches for similarities between the input sequence and the database fragment. Finally, it returns the obtained results to the master, which collects all the results and concatenates them into one output file.

In the case of BLAST, the application may present load imbalances given by variations in the computation time associated to the complexity of the query. In this application, these variations are caused by the content of the data chunks because the more similarities BLAST encounters in a data chunk, the more time is required for processing a fragment. Moreover, BLAST represents a good candidate to benefit from our proposal.

In this work, BLAST has been used to evaluate the effectiveness of our methodology to improve performance in real data intensive applications when dynamically adapting: (i) the scheduling policy; (ii) the size of the data chunks; and (iii) the number of processing nodes used.

Selected Workloads for BLAST

Along this chapter, all experiments performed with BLAST have been using the ncbi-blast-2.2.23 [66] version of the application (available in the web page of the National Center for Biotechnology Information [4]). Since the performance of BLAST (total execution time) is highly influenced by the characteristics of the biological queries and databases, we choose one of the biggest database of nucleotides named *nt* (of approximately 50 GB) and the scenarios described below to observe variability.

BLAST has been executed using three different workloads: a heavy workload (tagged

[4]Latest versions of BLAST are available in ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/

Slow), using queries with many similarities with the database; a medium workload (tagged *Common*), decreasing the number of similarities; and a light workload (tagged *Fast*), reducing even more the number of similarities. All queries contain biological sequences of the same size (1MB each):

- *Slow*: a 1,036,416 chars long sequence, literally chopped from the last part of the *nt* database. This piece was selected due to its long associated execution time (a couple of hours in our computing platform).
- *Common*: a 1,076,380 chars long sequence, created from randomly selected lines from the *nt* database. This sequence has an associated execution time about minutes.
- *Fast*: a sequence of 1,015,156 chars, taken from another database of nucleotides: the *yeast* DNA database. This sequence has fast associated execution times of only a few seconds.

The database has been previously partitioned and formatted into the following partition factors: $N_f = \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$. These workloads are available in the local disk of each worker node at run time.

4.1.2 Distributed Sorting Algorithm

Sorting algorithms have been widely used since the beginning of computer science. Knuth [48] said: *"Computers have historically spent more time sorting than doing anything else. A quarter of all mainframe cycles were spent sorting data."* Although scientific applications are not commonly executed in mainframes, sorting algorithms are often used to evaluate performance. The special behavior of such algorithms and the differences in computation time when processing unsorted and sorted files have become sorting algorithms as one of the most interesting in the computational field. The steadiness of certain sorting algorithms makes easier the estimation of total execution time of such applications.

Additionally, many sorting algorithms, such as *merge sort* have included modifications to facilitate sorting large input files in short time. Merge sort parallelizes well due to its divide-and-conquer method. The variability in computation time and the capability of processing large input files in parallel, together with the possibility of arbitrarily dividing the file to be sorted into smaller pieces, have led us to develop a distributed version of the algorithm.

This application was developed as a Master-Worker and its input data files (of unsorted items) were generated using the *gensort* program [67]. These files contain items

represented as lines of 100 ASCII characters. This application can sort input files of medium size (e.g. several tens of GB); the files can be split into smaller data chunks; and the chunks are processed by worker nodes under a round-robin approach. The size of generated files is of up to 32 GB, and each file was divided into 128 data chunks. This number of pieces was selected because is large enough to guarantee that all workers of the parallel system described at the end of this section will have data chunks to process.

The distributed version of the merge sort replicates the sorting engine at each worker process. Workers receive their data chunks following the scheduling policies described in section 3.2.2. Moreover, to introduce variability in the computation time associated to some of the data chunks in the workload, unsorted and sorted data chunks have been randomly combined in the workload. The workload has a variable percentage of sorted data chunks (e.g. a 25%, 50% and 75% of the total workload) to generate the time variation expected in the application.

As the stable version of merge sort algorithm has a computational complexity of $(n * log(n))$, once the workload is partitioned the total execution time is greatly reduced. Nevertheless, for comparison purpose obtained results might not be comparable with results from a serial version. Therefore, to keep integrity in the total execution time, after each worker performs a distributed merge sort in the received data chunks, and once all its data chunks have been sorted, the workers merge their processed data chunks into a bigger file. In this way, the distributed sorting time and the final merge time can be collected.

In this work, we have used this application to analyze the effect of dynamically modifying: (i) the scheduling policy; (ii) the size of the data chunks; and (iii) the number of processing nodes used. Consequently, distributed merge sort facilitates the analysis of the results of applying the proposed methodology, because it enables the modification of the size of the data chunks at run time without introducing additional processing overhead. Dynamic *partitioning* and *grouping* strategies (defined in previous chapter) have been easily applied to this application.

4.1.3 Analytical Simulator

In the performance analysis process, one of the most useful tools is simulation. Simulation enables the evaluation of scenarios that are difficult to observe and model. In many cases, simulation techniques validate the results obtained from empirical analysis and modeling. Nevertheless, implementing, debugging and executing a whole analysis environment to evaluate its functioning may result a big challenge. In this work, some analytical expressions have beens defined to: (i) describe the behavior of data intensive applications;

and (ii) choose the appropriate values for performance parameters, such as the workload partition factor and the number of processing nodes used. Therefore, their functionality have been evaluated through simulation.

In this sense, an analytical simulator has been implemented to evaluate the proposed load balancing methodology described in chapter 3 in a wide range of scenarios for the selected applications. The developed tool facilitates the observation and analysis of the performance parameters influence in the execution of the application. For instance, the simulator allows for the visualization of certain variations in data chunks processing time, and the evaluation of how changes in the partition factor affect the performance of the application. Thus, the simulator is able to reproduce such situations for a wide range of situations.

As a first step, the simulator has been fed with different scenarios of input data. Initial input data included the following parameters: the execution time of every data chunk (in seconds); the size of the data chunks (in megabytes); the communication time per megabyte and the number of processing nodes. The result from the simulation process is the total execution time of the application (expressed in seconds) for each scenario.

The simulator has been designed to reproduce a Master-Worker paradigm. Additionally, it presents a synchronous communication pattern between the master and workers; i.e., a data chunk cannot be sent until the sending of the previous data chunk has finished. In this way, communication time has been modeled as the product of the size of the data chunk by the communication time per megabyte.

Additionally, the expression (3.10) (described in section 3.3.1) has been used to model the total execution time of the application in the simulator. Since the response time in most of the scenarios is almost negligible, this time (the time of sending the results once the worker has finished processing its data chunks) has not been considered in the total execution time. Moreover, the model assumes that the total execution time is determined by the last worker to finish its computation.

Summarizing, the simulator has been developed to: (i) quickly assess the proposed methodology using a greater number of processing nodes when varying scheduling techniques, as well as introducing errors in the similarity between iterations; (ii) reproduce the general behavior of data intensive applications with divisible load; and (iii) to observe and evaluate the performance improvement capability of the methodology in such applications. These aspects will be fully evaluated in the rest of this chapter.

Experimental Environment

The experiments explained in the following sections were carried out in a homogeneous cluster with 32 processing nodes, and 12 GB of memory per node. Each node consists of 2 dual core Intel Xeon 5160 at 3 Gh with 4 MB of L2 Cache and memory at 667 Mhz FSB.

4.2 Data Management

In this section, we describe the experiments performed with the aim of evaluating the performance improvement methodology proposed in this work. Experiments shown in the following sections are based on the evaluation of each phase of the methodology using the applications (BLAST and distributed merge sort) and the evaluation environment (analytical simulator) described in section 4.1. First, in section 4.2.1 the improvements obtained when changing the scheduling policy are exposed. Then, in section 4.2.2 the method and results of tuning the size of data chunks according to the performance of each application are described.

4.2.1 Evaluation of the scheduling policy

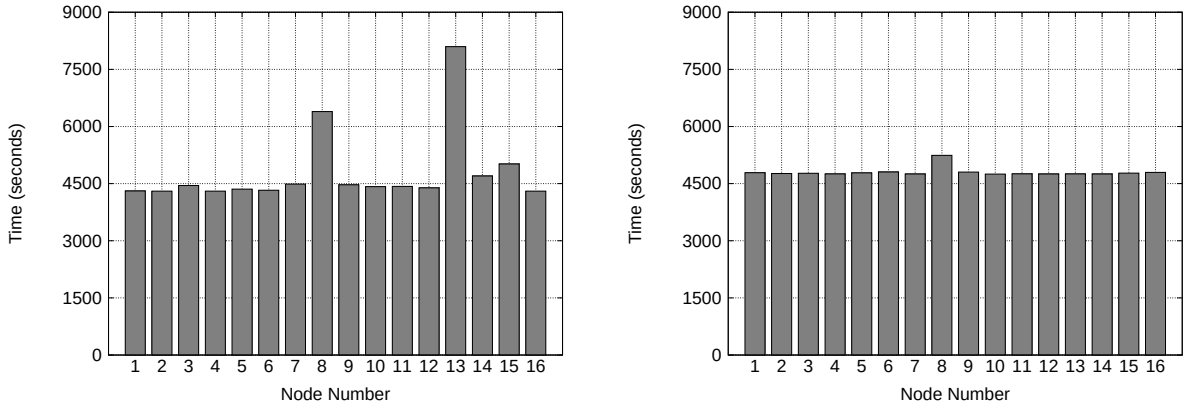
As a first step, the behavior of the application when changing the scheduling policy is analyzed. The applications are executed using the *First Come First Serve* (FCFS) and the *Heaviest Fragments First* (HFF) scheduling policies (that have been described in previous chapter in section 3.2.2). According to our methodology, the time spent on processing each fragment should be recorded to be used for scheduling data chunks for next iterations, explorations or queries. The experiments shown in this section are carried out using a partition factor of $N_f = 128$, because: (i) it gives enough information about data chunks processing times; (ii) it allows a higher load balancing; and (ii) it is not difficult to graphically see the time associated to each fragment.

Here, we present experiments performed using the applications described in section 4.1. First, the method and results obtained for the evaluation using BLAST are exposed. Then, the corresponding results for the merge sort application are shown. Finally, the experiments performed using the analytical simulator are explained and discussed.

Evaluation of the scheduling policy through BLAST

Results shown in this section (in figure 4.2(a)) are obtained by using 16 workers, $N_w = 16$, the partition factor $N_f = 128$, the *Slow* scenario for BLAST and applying the *First Come*

First Serve (FCFS) scheduling policy. Here, load imbalances can be clearly observed. The cause of such situation is given by data chunks with highest computation times that are scheduled at the end of the execution. This situation can be initially solved by changing the scheduling policy from *FCFS* to *Heaviest Fragments First* (HFF). Results obtained when applying this latter scheduling policy are presented at the right side of the graphic (figure 4.2(b)). The advantage of reordering data chunks according to their associated computation times is clear because *HFF* softens possible load imbalances by sending the heaviest fragments first (and filling the gaps with faster data chunks). A reduction of up to 40% in total execution time can be observed for this specific scenario (in comparison to not having a predefined distribution order).



(a) Processing time for *blastn*, *Slow* query, $N_w = 16$, $N_f = 128$ and *FCFS*.

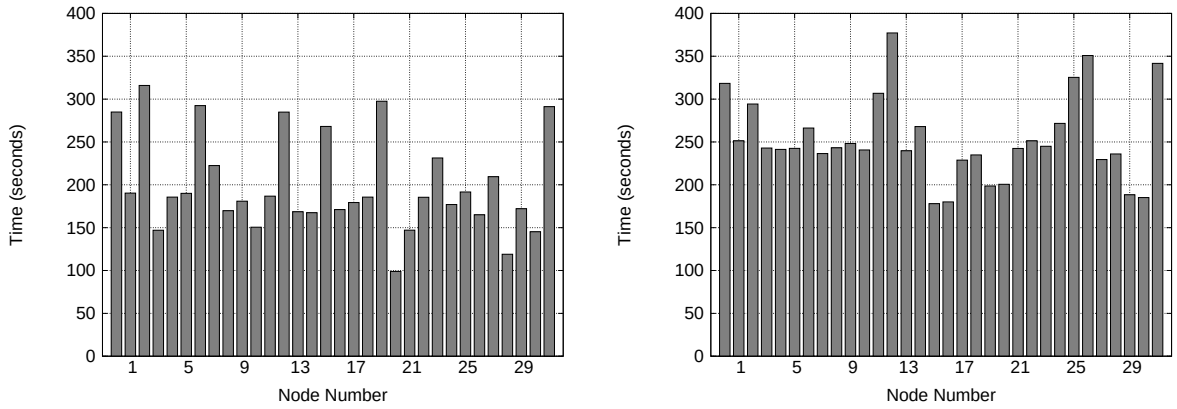
(b) Processing time for *blastn*, *Slow* query, $N_w = 16$, $N_f = 128$ and *FCFS & HFF*.

Figure 4.2: BLAST: Comparison of application performance using $N_w = 16$, $N_f = 128$, *Slow* scenario and, First Come First Serve (FCFS) and Heaviest Fragment First (HFF) scheduling policies.

Evaluation of the scheduling policy through Distributed Merge Sort

For the distributed merge sort scenario, data chunks are distributed using the *First Come First Serve* (FCFS) and the *Heaviest Fragments First* (HFF) approaches. Experiments were performed using 32 workers, $N_w = 32$. Obtained results in terms of total execution times are shown in figures 4.3(a) and 4.3(b). In figure 4.3(a) data chunks are distributed without any pre-defined order; while in figure 4.3(b), data chunks with highest execution times are delivered first.

From the reported results, it can be concluded that changing only the distribution



(a) Execution with distributed merge sort, for $N_f = 128$ and $N_w = 32$ using *FCFS*.

(b) Execution with distributed merge sort, for $N_f = 128$ and $N_w = 32$ using *HFF*.

Figure 4.3: Distributed Merge Sort: Load imbalances using First Come First Serve (FCFS) and Heaviest Fragment First (HFF) scheduling policies.

policy may not solve the load imbalance problem for this application. This situation persists because there is a final merge time that is not considered when distributing data chunks. The final merge is performed by each worker with the received data chunks and it is influenced by the total number of pieces this worker has received. If a worker has processed too many data chunks with low computation times, this performance improvement may disappear when merging the final file causing unexpected and undesirable load imbalances.

Evaluation of the scheduling policy when introducing error in data chunks processing time predictions using the Analytical Simulator

The aim of this experiment is to evaluate the effectiveness of the proposed methodology for the distribution policy, when introducing different degrees of error in the predictions of the data chunks processing times. The proposed load balancing methodology is based on sending first those data chunks with higher processing times. To accomplish this, the history of the execution time measured for each data chunk is gathered and then this information is used to decide the scheduling order. However, predictions are likely to fail in some degree, and expected results might differ because the execution time of the same data chunk may vary from iteration to iteration.

Here, we evaluate how the total execution time is being affected when a certain degree of error is introduced to the prediction of the processing time associated to each data

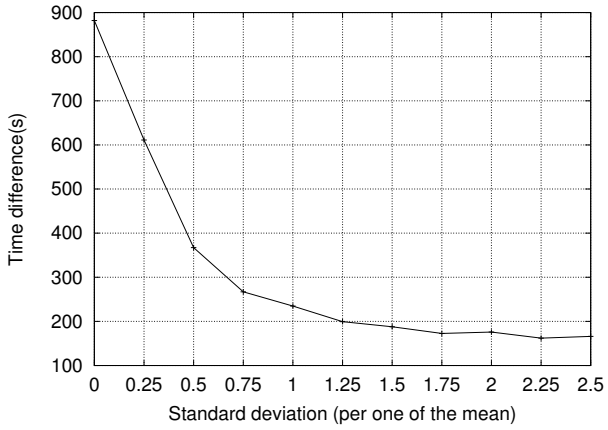


Figure 4.4: Analytical Simulator: Mean time differences when introducing errors in data chunk prediction times using FCFS and HFF scheduling policies.

chunk. To this end, the simulation environment is set to consider a partition factor equal to 128, $N_f = 128$, and 64 workers, $N_w = 64$.

The input data of the simulator is generated following a normal distribution from an initial data set obtained from real measurements. Then, a certain degree of variation is introduced in the computation times of each data chunk (the degree of error is ranged from no variation –0%– up to 250% of variation of the measured time). The greater this degree of variation, the greater the variability in the time associated to the data chunk.

Obtained data sets are evaluated using both scheduling policies: (i) *Fist Come First Serve* (FCFS): the data chunks are distributed with not previous order; and (ii) *Heaviest Fragments First* (HFF): the data chunks are distributed by processing time in decreasing order. Reported results are the differences between the total execution time when applying both scheduling policies.

From results shown in Figure 4.4, it is observed that introducing variability in the processing times of each data chunk tends to degrade the performance of both scenarios. Increasing the variability in the execution time of the data chunks leads to less accurate predictions for the HFF scheduling policy. Consequently, as expected, it can be observed how the differences between the total execution time of each scheduling policy is reduced. However, it also can be seen that even for a 100% of variability, HFF policy is leading to significant improvements over FCFS.

In some data intensive applications, performance improvement is achieved by reducing total execution time. To this end, it is necessary to tackle time restrictions imposed by data chunks with highest computation time. To achieve this goal, the methodology considers the partitioning of such data chunks, and the grouping of data chunks with short execution times. These strategies are evaluated in the following section.

4.2.2 Adapting the size of data chunks

The objective of these experiments is to evaluate the performance of different data intensive applications when tuning the workload partition factor. Results shown in this section are represented by the total execution time of the applications when applying the strategies described in sections 3.2.3 and 3.2.4. Both *partitioning* and *grouping* strategies are applied at each experiment. Data chunks with highest execution times are partitioned into smaller pieces, and bigger data chunks are created by joining pieces with short execution times.

First, results obtained by adapting the workload partition factor in BLAST are shown. Then, our method is applied to the distributed merge sort. To finalize, with the evaluation of the adaptation in the size of the data chunks in larger scenarios through simulation.

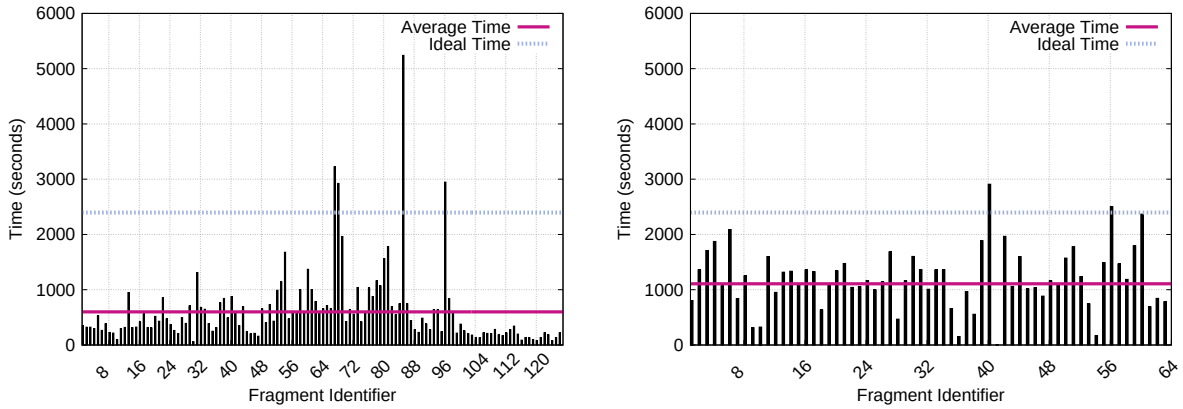
Evaluation of adjusting the workload partition factor in BLAST

In BLAST executions, the modification of the partition factor is done as explained in section 3.2.3 when the cost of partitioning at run time is too high. In this case, database is previously partitioned into different number of data chunks (different partition factors) and our methodology chooses the appropriate size of data chunks from these sets of data chunks. From initial partitions, a partition factor equal to 128, $N_f = 128$ is selected. The execution is carried out in $N_w = 32$ workers using the *Slow* type of query.

From this configuration the following results are obtained: a total execution time of $C_i = 5,263.51$ seconds; an average execution time of $\mu_i = 599.08$ seconds; a standard deviation of $\sigma_i = 670.86$; and an expected ideal time equal to $2,396.33$ seconds. These results and the computation time of each data chunk are shown in figure 4.5(a). Under the same scenario, figure 4.5(b) presents the results of a new partitioned workload after gathering and dividing the data chunks. In this case, N_f is reduced from 128 to 64 and the expected total execution time (time of the slower data chunk) is now equal to $2,910.03$ seconds (a reduction in the total execution time of up to 55%).

This time reduction is obtained by selecting from the previously partitioned data sets the most adequate size of data chunks. For this specific experiment, we can observe 2 data chunks with large execution times, each of them are divided in 4 (smaller) data chunks coming from a partition factor equal to 512. Data chunks with short execution times are grouped with their contiguous data chunks to create bigger pieces [5]. This scheme of partitioning and grouping is used to preserve the relations between different data chunks in the workload.

[5]For a partition factor of $N_f = 256$ are obtained 2 data chunks with large execution. Each one of these data chunks are divided into 2 smaller pieces that substitute their "parents" in next iterations.



(a) Initial processing time by data chunk

(b) Processing time by tuned data chunks

Figure 4.5: BLAST: Variations in the execution time of the data chunks when partitioning and grouping the workload.

In addition, the behavior of BLAST when using the HFF scheduling policy with and without adapting the size of the data chunks (workload partition factor modification) is compared. The evaluation is performed ranging the number of workers, N_w, from 4 to 32. Obtained total execution times are contrasted with the expected ideal time for each case; results are showed in figure 4.6. The difference between tuning or not the size of the data chunks is clearly shown in this figure. Here, HFF presents a constant total execution time from 16 workers because of data chunks with the highest execution time (these data chunks impose a time restriction and applications total execution time cannot be lower than this time). On the contrary, when applying partitioning and grouping strategies, the total execution time of the application is greatly reduced. This time reduction is achieved by breaking data chunks that impose time restrictions and by distributing them first.

Evaluation of adjusting the workload partition factor in Distributed Merge Sort

In this experiment, the performance of the distributed merge sort when adapting the size of the data chunks is evaluated. This functionality is introduced with the aim of: (i) reducing the execution time of data chunks with high processing times; or (ii) reducing the number of data chunks with low computation times by sending less pieces of greater size. The experiment is performed using the scenario described in the section 4.1.2. This experimentation is carried out to provide the comparisons between the results presented in figure 4.3 (in section 4.2.1) and figure 4.7. Data chunks are partitioned and grouped at

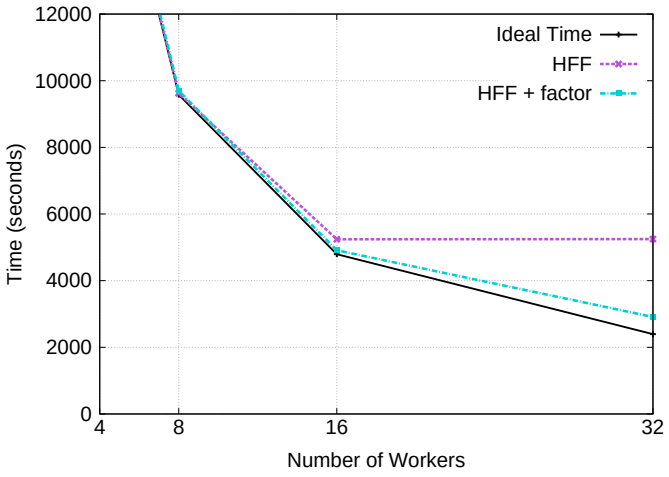


Figure 4.6: BLAST: Performance improvement in total processing time for $N_f = 128$ and the *Slow* scenario for BLAST, using *HFF* scheduling policy + workload partition factor adaptation (*HFF + factor*).

run time. The methodology evaluates collected execution times and establishes the data chunks that should be partitioned and grouped for the subsequent exploration.

Results presented in figure 4.7 show a more balanced execution in comparison with non-adapting the workload partition factor (as shown in figure 4.3(b)). This improvement is obtained after grouping and generating data chunks of almost the same total execution time for each worker. By doing this, the variability introduced by the final merge becomes constant and the processing time of each one of the workers is almost the same.

Tuning the workload partition factor enabled a well balanced execution among available workers nodes (finishing time of all workers is about 200 seconds, in comparison with previous FCFS and HFF distributions presented in section 4.2.1), as well as a reduction of the total execution time of approximately a 32%, (from comparing 371.20 seconds of HFF scheduling strategy to 279.63 seconds using the HFF + factor scheme).

Evaluation of adjusting the workload partition factor using the Analytical Simulator

The simulator is used to evaluate the analytical expressions presented in chapter 3 to show the performance improvement when the size of the data chunks is changed. Simulations are performed under the following conditions:

- An initial partition factor $N_f = 128$.

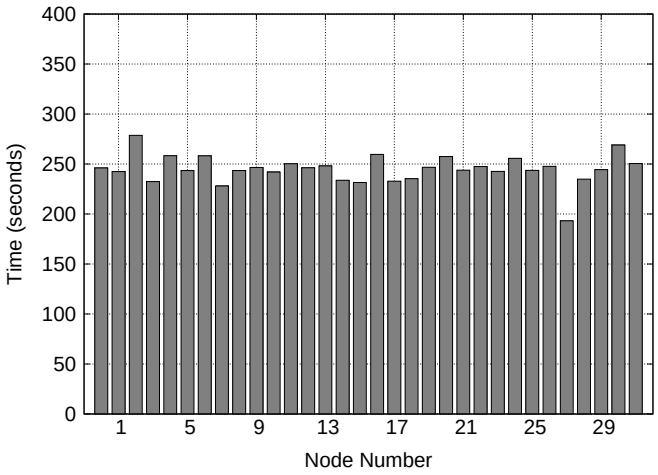


Figure 4.7: Distributed Merge Sort: Total execution time by worker for $N_f = 128$ and $N_w = 32$ using *HFF + factor*.

- N_w values ranging from 10 to 80.
- Two different scenarios: *Heaviest Fragments First* with and without workload partition factor modification (*HFF + factor* and *HFF*, respectively).
- Execution time of data chunks is generated following a normal distribution based on measurements obtained from real executions of BLAST.

From the results shown in figure 4.8, it can be observed the difference between the maximum execution time $T_{max_{ij}}$ obtained with and without applying the tuning strategy for the workload partition factor. Here, the execution time limitation imposed by data chunks with large computation times can be softened or diminished through dividing those data chunks into smaller pieces. It is worth noticing that once this barrier is removed, the total execution time of the application may be reduced by adding more workers.

As for all the experiments, the history of the computation time measured for each data chunk is used to adapt the workload partition factor, when measurements vary from one exploration to another predictions are likely to fail in some degree.

As previously done in section 4.2.1, the variation in total execution time is analyzed. This variation is obtained when a certain degree of error is introduced in the data set, e.g., changing the size of data chunks. The simulation environment is set to use $N_f = 128$ and $N_w = 64$, and a certain degree of variation in the time of each element of the set is introduced. The greater this percentage of variation introduced in the computation

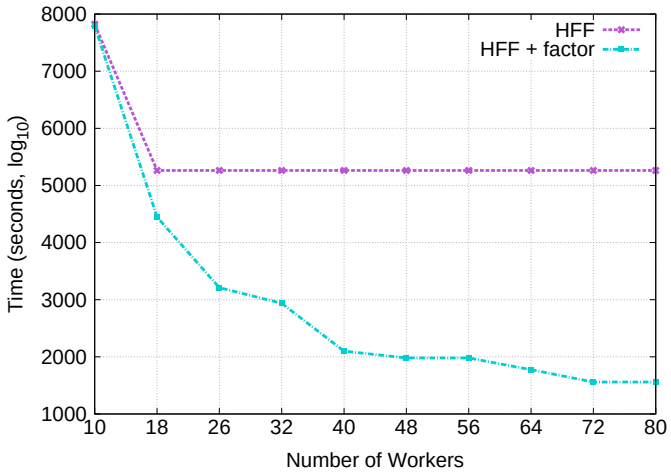


Figure 4.8: Analytical Simulator: Performance improvement using HFF + factor scheduling strategy for variable numbers of workers.

time of each data chunk, the greater the variability obtained for each new data chunk. The generated data set is evaluated for the HFF scheduling policy with and without adapting the workload partition factor. The simulation process is repeated 500 times and the results are the average values of computation time for both cases.

Introducing variability in the computation time of data chunks tends to degrade the performance of the HFF strategy. In figure 4.9, it can observed that time degradation is significantly reduced when adapting the size of the data chunks, i.e. using our *HFF + factor* strategy.

In general, when data chunks with large computation times are divided into smaller data chunks, their associated computation time is greatly reduced. Consequently, it is possible to reduce the total execution time of the application by adding more computational resources. In addition, the methodology proved to be efficient enough to dynamically improve the performance of data intensive applications. Obtained results are encouraging in terms of applying our method to a wider range of data intensive applications with divisible load. These improvements are achieved even in the presence of failures in the estimation of the associated computation times of the data chunks, i.e., when subsequent queries are not closely related.

Once the load balancing using the HFF scheduling policy and the workload partition factor modification is achieved, the focus can be shifted to assess the resource utilization.

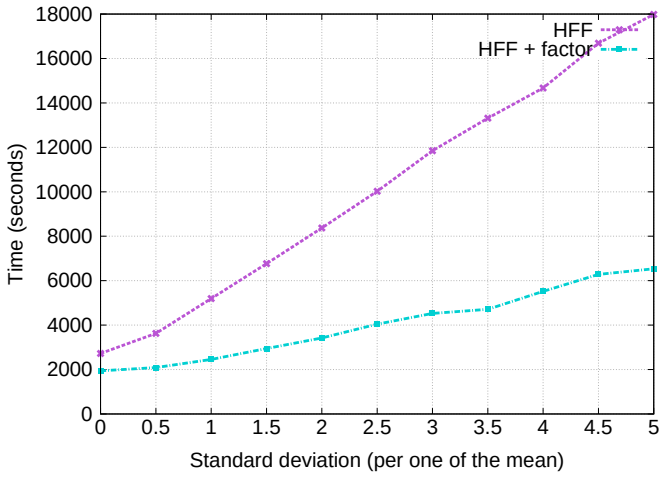


Figure 4.9: Analytical Simulator: Performance improvement when introducing errors in data chunk prediction times using FCFS and HFF scheduling policies.

4.3 Resource Management

In this section, the resource management scheme proposed in our methodology to improve the performance of data intensive applications is evaluated. Experiments are carried out using the applications and the simulator described in section 4.1. The evaluation of adjusting the number of processing nodes being used is exposed in section 4.3.1.

4.3.1 Adjusting Number of Workers

By adapting the size of the workload, an important reduction of the execution time is achieved. Nevertheless, in most cases, there is an instant in which there is no more possibility to reduce total execution time. When this happens, the goal is shifted to efficiently adapt the number of processing nodes. This adaptation is performed with the aim of avoiding worker being idle during the execution of the application. It is worth mentioning that workers may be idle for both load imbalances and communication dependencies. Nevertheless, in this work, the Master-Worker model observed and implemented does not consider the aforementioned dependencies. To attain an efficient execution using a certain number of processing nodes, and since total execution time is given by the last worker that finishes its processing, collected metrics are evaluated to tune the number of workers being used in parallel executions under a Master-Worker paradigm.

In this section, the proposed strategy for tuning the number workers nodes using the previously defined data intensive applications is evaluated. First, the results obtained from

BLAST are presented, followed by the evaluation of the distributed sorting algorithm. Finally, for a wide range of scenarios for the applications, the method is analyzed and evaluated through simulation.

Estimating the number of resources for BLAST

To show the advantages of tuning the number of workers used by the application, BLAST is executed using five different partition factors $N_f = \{128, 256, 512, 1024, 2048\}$; and five different numbers of workers $N_w = \{2, 4, 8, 16, 32\}$. The value of λ is measured experimentally as the inverse of the network bandwidth ($\approx 112.5MB/s$, which is the expected best-case data bandwidth measured between two nodes of the cluster for a Gigabit Ethernet network). The average sizes of the data chunks for the selected partition factors are shown in Table 4.1.

N_f	128	256	512	1024	2048
size [MB]	55.8	28.2	13.9	7.0	3.5

Table 4.1: BLAST: Data chunks and chunks sizes

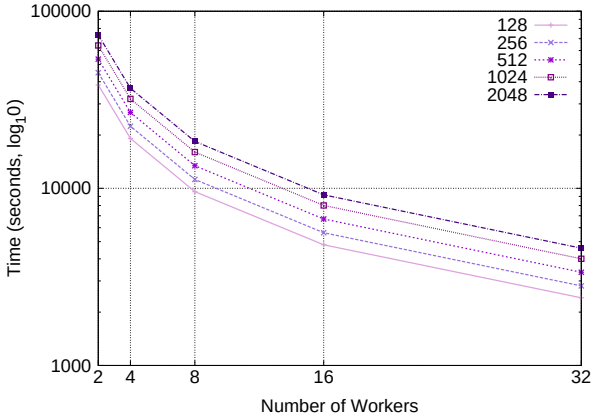
Figure 4.10(a) shows the evaluation of expression tq_i (3.10) for each value of N_w using the *Slow* scenario, figure 4.10(b) shows the real execution time of BLAST, and figure 4.10(c) shows the real execution time after tuning the partition factor. It can be observed that when the restrictions defined by expressions (3.8) and (3.9) are met [6], there is a small difference (of up to a 8% for most cases, as shown in Table 4.2) between estimated total execution time and real (measured) execution time for most of the partition factors and numbers of workers.

N_w	Expected Time [sec]	Real Time [sec]	**Error (%)**
2	38,375.81	38,410.30	**0.09**
4	19,136.69	19,261.70	**0.65**
8	9,584.92	9,620.99	**0.37**
16	4,798.99	5,242.65	**8.46**
32	2,409.03	5,264.33	**54.23**

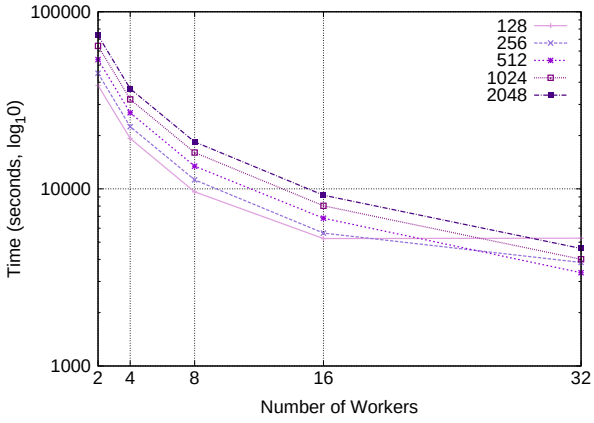
Table 4.2: BLAST: Error calculation for $N_f = 128$ between expected and real execution time when varying the number of workers.

Therefore, as expected, expression (3.10) facilitates the estimation of the total execution time of the application for subsequent iterations. Results shown in these figures

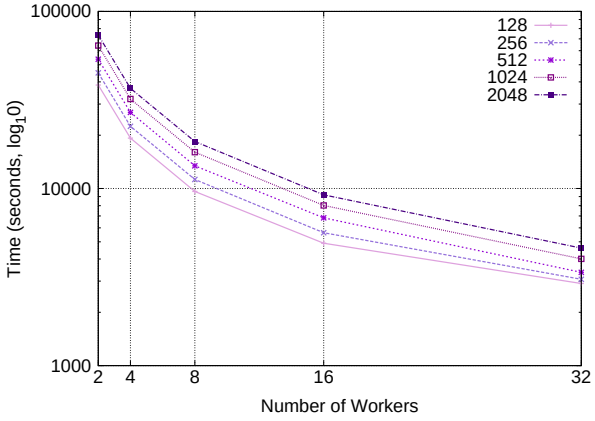
[6]Expression (3.8) represents the maximum number of workers that will finish the processing at the same time. Expression (3.9) estimates the maximum number of workers that the master may handle without causing idle time, i.e. before the master becomes a bottleneck.



(a) Expected (via analytical expressions) processing time for *Slow* type query, N_f variation and *FCFS & HFF* scheduling policies.



(b) Real processing time for *Slow* type query, N_f variation and *FCFS & HFF* scheduling policies.



(c) Real processing time with *Slow* type query, N_f variation and *FCFS & HFF + factor* scheduling policies.

Figure 4.10: BLAST: Comparison between expected and real execution times.

present a major improvement when tuning the number of workers. For almost all the cases, when varying from 2 to 32 workers the reduction of the total execution time reaches a 93.7% (as shown in Table 4.3). These results will be compared later with obtained results from the evaluation using the analytical simulator (where it is possible to simulate more than 32 workers).

Expected Performance (HFF)			
N_f	$tq_i(2)$ [sec]	$tq_i(32)$ [sec]	**Improvement (%)**
128	38,375.81	2,409.03	**93.72**
256	44,966.65	2,814.81	**93.74**
512	53,686.05	3,359.47	**93.74**
1024	64,083.39	4,004.90	**93.75**
2048	73,537.14	4,595.66	**93.75**
Real Performance (HFF)			
N_f	$tq_i(2)$ [sec]	$tq_i(32)$ [sec]	**Improvement (%)**
128	38,410.30	5,264.33	**86.29**
256	44,994.70	3,843.83	**91.46**
512	53,722.10	3,362.27	**93.74**
1024	64,152.50	4,009.22	**93.91**
2048	73,643.00	4,609.62	**93.74**

Table 4.3: BLAST: Expected and real performance improvement

The cases where the estimated execution time greatly differs from the real execution time (cases with $N_w = 32$ workers using the partition factors $N_f = 128$ and $N_f = 265$) can be explained through the constraints indicated by expressions (3.8) and (3.9) (minimum total execution time to finalize the execution and maximum capacity of the master, respectively). In these cases, when using 128 data chunks, the associated computation time of the chunk with the highest processing time is of about 5,245.61 seconds ($Tmax_{ij}$, in Table 4.4). Therefore, when the number of workers is changed from 16 to 32, there is no improvement in total execution time because of it. Consequently, for similar cases, increasing the number or processing nodes may not reduce the total execution time.

In this sense, for the same partition factor $N_f = 128$ and without adapting the size of the data chunks, obtained result from evaluating the expression (3.8) is equal to: $\frac{Ts_i}{Tmax_{ij}} = \frac{76{,}598.19}{5{,}245.61} = 15$. This value (the maximum number of workers that should be active) indicates that above 15 workers the total execution time will remain the same (there is no improvement and the probability of having idle workers increases). Likewise, for a partition factor equal to 256 data chunks, the maximum number of workers that can be used is 23, as shown in Table 4.4. Figure 4.11 illustrates this discussion by showing for $N_f = 128$ and $N_f = 256$ the best possible execution time (calculated by $tq_i(n)$); and the

real (measured) execution time (given by the maximum execution time $Tmax_i$). Since adding more workers does not guarantee a reduction in total execution time, the best decision is to adjust the number of workers being used (reducing the number of workers that are active) to increase the efficient use of processing resources.

N_f	μC_i [sec]	Ts_i [sec]	$Tmax_i$ [sec]	Nw_{max}
128	598.42	76,598.19	5,245.61	**15**
256	350.57	89,747.16	3,842.51	**23**
512	209.64	107,317.37	2,648.39	**41**
1024	125.06	128,061.95	1,417.55	**90**
2048	71.79	147,018.66	860.31	**171**

Table 4.4: BLAST: Maximum number of workers (Nw_{max}) for Slow queries.

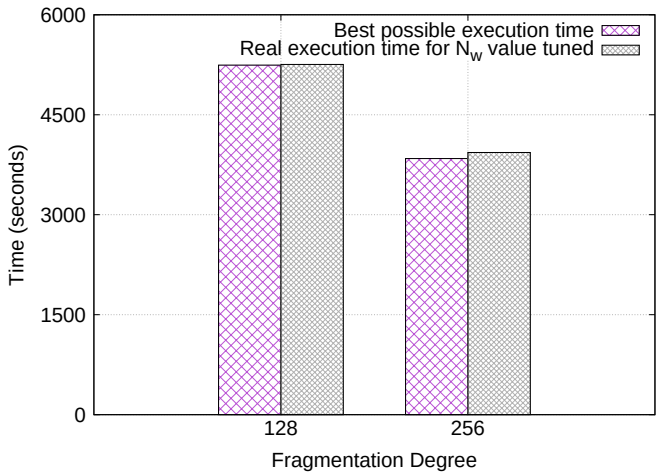


Figure 4.11: BLAST: comparison of the best possible execution time and real execution time when adapting the number of workers, for *Slow* scenario using $N_w = 15$ for $N_f = 128$, and $N_w = 23$ for $N_f = 256$.

In addition, the *Slow* scenario and a partition factor $N_f = 128$ are used to illustrate the use of the performance index described in expression (3.12) when tuning the number of workers. Performance index indicates the efficient use of resources during the execution of the application. This index relates total execution time of the application and average execution time to estimate the total time workers are idle, i.e. without processing data. Figure 4.12 shows the results reported for the performance index ρ_n when using 2, 4, 8, 16 and 32 workers.

It can be seen that there is no efficiency loss (the curve does not start to climb) for the selected application, and therefore, more workers may be added. However, if the number of workers is increased, there will be no gain in total execution time (because of

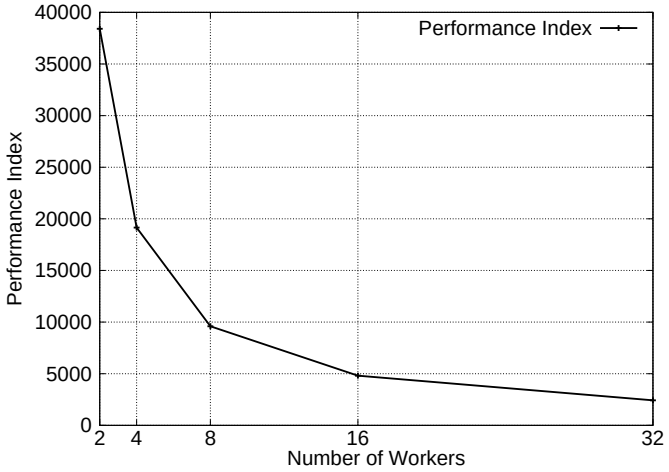


Figure 4.12: BLAST: performance index for *Slow* query and $N_f = 128$.

the data chunk with the highest computation time) and more workers will remain idle. This idleness is translated as an efficiency loss and it must be avoided.

To conclude this section, an additional evaluation of the influence of adapting both the number of processing nodes and the size of the data chunks in BLAST is presented. Although partition factors can be modified, performance improvement for BLAST may remain constant (as shown in figure 4.10). This stability may be improved by dynamically adapting the size of the data chunks. In the following lines, the total execution time of the application with and without changing the size of the data chunks for initial partition factors equal to $N_f = 128$ and $N_f = 256$ is evaluated. Figure 4.13 shows the total execution time for each partition factor without adapting data chunks sizes (left column labeled *HFF*) and when the tuning is applied (right column labeled *HFF + factor*).

Once the workload partition factor is adjusted, time limitations imposed by data chunks with highest computation times are softened. Consequently, a larger number of processing nodes can be used without losing efficiency. For 32 workers and a partition factor $N_f = 128$, obtained results have reported up to 55% of reduction in the overall execution time when applying the proposed methodology (as shown in figure 4.13). Reported results show the advantage of adapting the size of the data chunks dynamically. Nevertheless, there is an implicit overhead caused by the serial fraction of each data chunks. This fraction penalized the total execution time when adding data chunks times. For example, using a partition factor $N_f = 256$ it can be seen a slight increment in the total execution time when adapting the size of the data chunks, because the serial fraction is replicated when the computation times of the data chunks are summed. Therefore, as more data chunks we have, the more overhead is introduced.

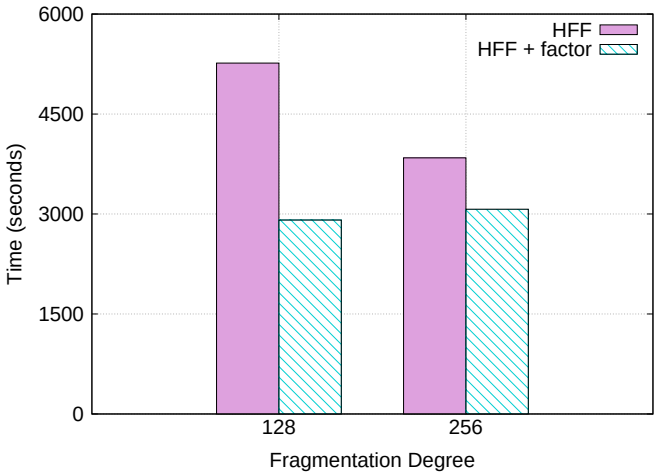


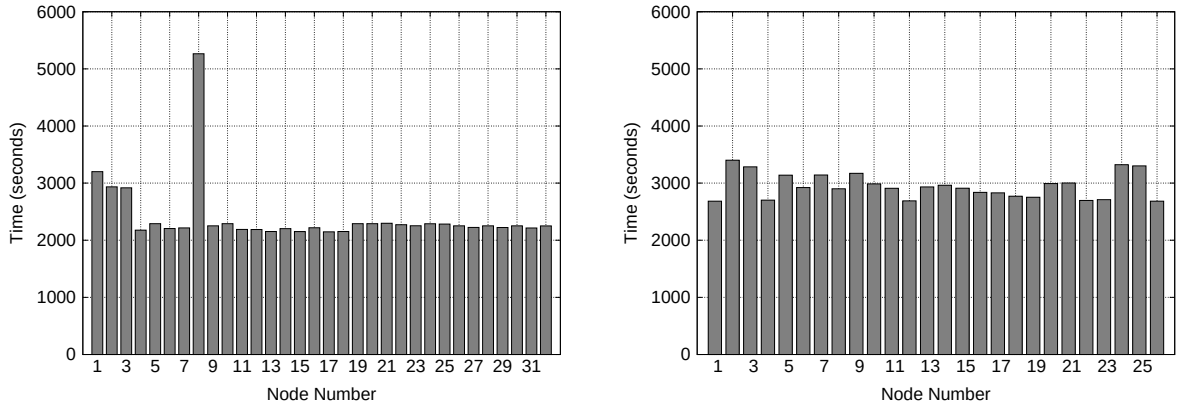
Figure 4.13: Execution with *Slow* query, for $N_w = 32$ using *HFF* and *HFF + factor*.

Consequently, figure 4.14 shows the reported results when tuning both the partition factor and the number of workers. Here, once the workload partition factor is adapted to reduce total execution time, expressions (3.8) and (3.9) defined that the number of workers must be reduced to $N_w = 26$ for next iterations to improve the efficient use of resources (figure 4.14(b)). From this experiment, focusing on the partition factor $N_f = 128$, it can be observed a time reduction between the first and last worker finishing processing of up to 76.99% (from 3116.10 seconds to 717.06 seconds). Additionally, processing nodes are used a 14.52% more ($Ef_{32} = 1.8$ and $Ef_{26} = 2.1$) and there is an overall improvement in total execution time and efficient use of resources of approximately 45% ($\rho_{32} = 2886.59$ and $\rho_{26} = 1593.97$). Therefore, by applying our methodology it is possible to reduce total execution time of BLAST application and increase the efficient use of available resources.

Estimating the number of resources for Distributed Merge Sort

The intent of this experiment is to analyze the behavior of the distributed sorting algorithm when adding more processing nodes. This experiment is carried out using the initial workload described in section 4.1.2, a unsorted file of 32 GB [7] divided into 128 data chunks. To introduce variability in the computation time associated to some of the data chunks in the workload, unsorted and sorted data chunks are randomly combined in the workload. Therefore, 25% of data chunks are previously sorted to generate the expected time variation when executing the application. The total execution time when

[7]Input files have been generated using the *gensort* program [67].



(a) Initial BLAST execution

(b) Tuned BLAST execution

Figure 4.14: BLAST: Efficient use of resources when tuning the partition factor and the number of workers (HFF + factor). Initial partition factor $N_f = 128$, initial number of workers $N_w = 32$; tuned values $N_f = 64$ and $N_w = 26$.

changing the number of workers is the observed metric. For this experiment, the value of N_w used is ranged from 16 to 56 workers (each workers is executed on one processor of the node). The reason of using such limited values is to facilitate data interpretation and results presentation. Figure 4.15 shows the reported results when applying the following scheduling strategies: (i) *First Come First Serve* (FCFS); (ii) *Heaviest Fragments First* (HFF); and (iii) *Heaviest Fragments First* adapting the partition factor (HFF + factor).

Here we can observe a steady reduction in the total execution time for the HFF scheduling policy with modification of the data chunks sizes (HFF + factor) when adding more processing resources. Consequently, it can be ascertain a constant reduction in total execution times when applying all the stages of the proposed methodology: (i) changing distribution policy, (ii) adapting the size of the data chunks, and (iii) adding more processing resources. These results are encouraging for improving the performance of data intensive applications in terms of total execution time reduction.

Estimating the number of resources using the Analytical Simulator

In this section, the analytical simulator is used to evaluate the performance of data intensive applications under the aforementioned scheduling policies when: (i) adding a large number of workers; and (ii) introducing variability in the execution time of the data chunks. First, the scheduling techniques when increasing the number of workers are compared. In figures 4.16 and 4.17 are shown the results of the simulations performed under the following conditions:

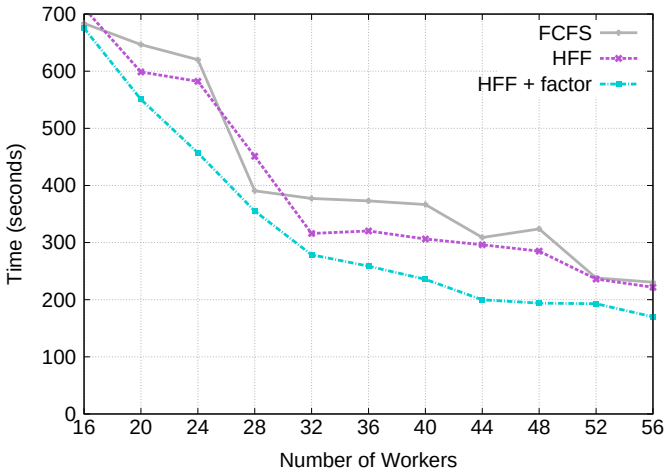


Figure 4.15: Distributed Merge Sort: Comparison between FCFS, HFF and HFF + factor, when varying the number of processing nodes for $N_f = 128$.

- An initial partition factor $N_f = 128$.
- N_w values ranging from 10 to 64.
- Three different scenarios: *First Come First Serve* (FCFS) scheduling policy, and using *Heaviest Fragments First* with and without workload partition factor modification (HFF + factor and HFF, respectively).
- Execution time of data chunks were generated following a normal distribution based on measurements obtained from real executions of BLAST.

In figure 4.16, it can be seen a reduction in total execution time when applying all the scheduling policies. When comparing reported results from all the strategies, FCFS reports the worst performance results in terms of total execution time. This strategy presents the inherent overhead of the application without applying any scheduling policy. In addition, HFF scheduling policy, as observed in previous sections, reaches a steady point where the total execution time cannot be reduced by adding more workers. As commented before, this barrier depends on data chunks with highest execution times. Consequently, when data chunks sizes are modified dynamically and the number of processing nodes is increased, HFF + factor reports the greater reduction in total execution time, and therefore, the best results for performance improvement.

Consequently, the HFF scheduling policy with data chunks sizes modifications is more flexible. Thus, this policy is more scalable than the other scheduling techniques presented

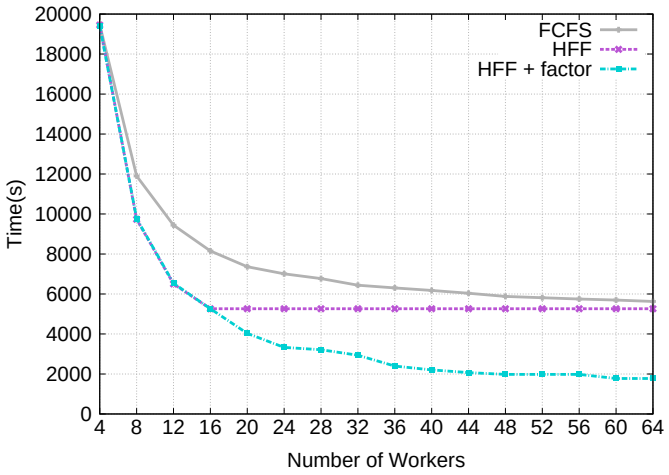


Figure 4.16: Analytical Simulator: Total execution time when varying the number of workers for the FCFS, HFF and HFF + factor scheduling policies.

in this work. The reason of this behavior is that data chunks with large computation times are not a major restriction for the HFF + factor strategy, because their sizes are tuned in accordance to the application behavior through partitioning or grouping.

Results reported from the evaluation of the performance index for the scheduling policies (shown in figure 4.17) corroborate this conclusion. It can be seen how the FCFS strategy reaches first the performance inflection point represented by the performance index. For instance, if using FCFS or HFF (without modifying the size of the data chunks) the performance may be quickly degraded by adding more workers, i.e., both FCFS and HFF curves start to climb. This behavior indicates that the execution of the application reached its best possible configuration of processing nodes, and from this point it may become inefficient. From this experiment, an execution under the FCFS scheduling policy gets its minimum execution time (and maximum efficiency) using between 12 and 16 workers. The same happens when the HFF scheduling policy is used, the "best" performance for this configuration may be achieved using only 16 workers. These values correspond with similar points in figure 4.16, and in consequence, after 16 workers there is no significant improvement in the execution of the application.

Additionally, reported results for HFF + factor show that under this scheduling technique the maximum number of workers to achieve the minimum execution time (without loosing efficiency) has not been found yet. Therefore, as more workers are added lower execution times may be obtained.

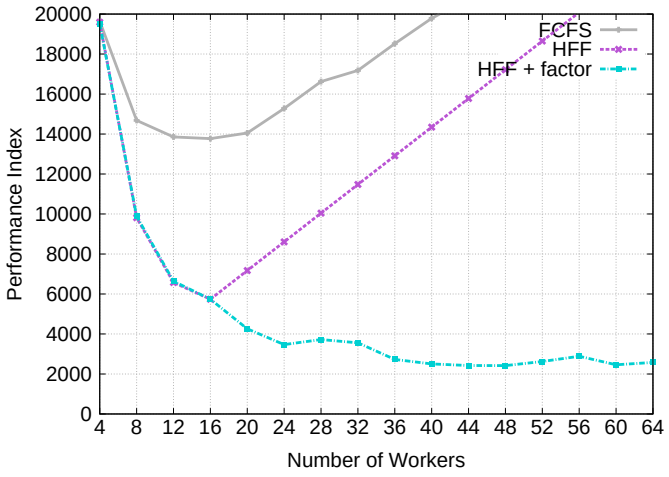


Figure 4.17: Analytical Simulator: Performance index when varying the number of workers for the FCFS, HFF and HFF + factor scheduling policies.

4.4 Summary

Throughout this chapter, we have shown how certain data intensive applications have been benefited by the proposed performance improvement methodology. Considered applications process large-scale workloads that may be arbitrarily divided into smaller pieces, and perform several related iterations or queries in this data. Some of these applications may be iterative algorithms, such as indexing, machine learning, data mining; web search applications; astronomical image processing; or genome sequencing, among others. In general, any data intensive application that match with the context described in previous chapter (that has a large-scale workload that can be arbitrarily divided into smaller pieces, and that performs several related queries or iterations in this data) may easily take advantage of our proposal.

This work has been focused on certain use cases of data intensive applications, such as the life science application Basic Local Alignment Sequence Tool – BLAST and a stable merge sort algorithm. In particular, BLAST has been selected because of its relevance as computational/data intensive application and merge sort because of its versatility. In addition, we have implemented an analytical simulator to facilitate the observation and analysis of the behavior of our methodology. The simulator has enabled us to evaluate the methodology in a larger number of processing nodes, as well as to analyze situations that otherwise would be too time consuming, e.g., executions with variable performance between iterations.

The selected applications have been used to evaluate each one of the main characteristics of the proposed methodology. In general, experimental results have shown an encouraging performance improvement in total execution time reduction and efficient use of resources in such applications. For example, the execution of BLAST, using our performance improvement proposal, has shown great reductions in total execution time and certain improvements in the use of available resources. This results in faster executions with lower idleness periods for the workers. Distributed merge sort (a simple parallel implementation of merge sort algorithm) has facilitated the analysis of performance improvements when dynamically tuning (during the execution of the application) the workload partition factor, showing load imbalances reduction when the size of the data chunks is modified.

Additionally, using simulation it has been possible to observe the performance of selected data intensive applications when: (i) a larger number of processing nodes is used; or (ii) iterations or queries performed are loosely related. In the latter case, although there is a certain performance degradation when having variable behaviors between iterations, the proposed methodology reports acceptable performance improvements for the executions of data intensive applications.

In general, by applying a simple strategy for performance analysis and tuning, it has been possible to show the viability and effectiveness of our proposal. Reported results have shown that the performance improvement methodology described in this work, has been able to achieve efficient and faster executions in data intensive applications with divisible load. Additionally, simulation results have indicated the viability of applying our methodology in real and larger applications.

Chapter 5

Conclusions

"On risque de pleurer un peu si l'on s'est laissé apprivoiser..."

Antoine de Saint-Exupéry. Le Petit Prince.

In previous chapters, we have presented the motivation, conception, design, implementation and evaluation of the contribution of this work: a methodology to improve performance in parallel data intensive applications. The aim of the methodology is to reduce the total execution time of data intensive applications, as well as augmenting the efficient use of computational resources.

In the design of the methodology, we have considered that: (i) the workload of the application can be arbitrarily partitioned into smaller pieces called data chunks; and (ii) applications perform several related queries or iterations in the data. Each execution has been monitored to collect the associated computation time by data chunk. Then, a set of carefully selected performance metrics, such as the average and maximum execution times, has been analyzed through our defined performance model. Based on obtained results, the workload partition factor and the number of processing nodes used (tuning points), have been tuned during the execution of the application.

The main contributions of this work are:

- Enabling parallel execution of data intensive applications by partitioning the initial data set into smaller data chunks.

  Data intensive applications process or analyze large-scale data. We have take advantage of divisibility properties of their workloads to split them into smaller pieces and distribute the generated data chunks among all available worker nodes. In this way, applications may be easily executed in parallel, and total execution times may be greatly reduced. In addition, we have faced existent variability

among the associated processing time of the data chunks, by proposing a scheduling technique that delivers first data chunks with highest execution times. Consequently, possible load imbalances are smoothed by filling with faster data chunks resulting gaps in workers processing times. This scheduling policy has been named *Heaviest Fragments First* and has been described in section 3.2.2.

- Reducing total execution time of the application by dynamically tuning the workload partition factor, i.e., changing the size of the data chunks with the highest and lowest associated computation times.

  The associated execution time of data chunks has been greatly reduced by adapting their sizes at run time. We have proposed to dynamically modify their sizes by: (i) dividing into smaller pieces those data chunks with highest execution times; and (ii) gathering fastest data chunks into bigger pieces to reduce distribution overheads. These techniques are triggered in accordance to the processing time of the data chunks, and have been named *partitioning* (presented in section 3.2.3) and *grouping* (introduced in section 3.2.4), respectively.

- Increasing the efficient use of computational resources (without periods of idleness) by estimating the maximum number of processing nodes that can be used for a given data set partition factor.

  The total execution time of applications developed under a Master-Worker paradigm is given by the last worker finishing computation. To avoid such situation, we have applied our methodology to dynamically determine how many processing nodes should be active. By doing this, the number of computational resources has been adapted to provide an efficient execution. We have considered an execution as efficient when all the workers are finishing their computation at the same time and without being idle for a long period. This tuning is performed at run time, and has been described in section 3.3.1.

Throughout this work, we have followed the steps of the scientific research method; ranging from the planning and discussion of objectives and methods, up to the testing and validation of proposals as described next. The definition and introduction of our objectives, motivation and challenges in chapter 1, has been the first step in the development of this work. Then, theoretical background (exposed in chapter 2) has served us to complement the motivation of our work, and to justify decisions taken while designing our performance improvement methodology. Taking such concepts and definitions as starting point, we have been capable of performing the analysis of influent factors in the

development of our proposal, putting special emphasis on the adaptation of the workload partition factor and tuning the number of processing nodes. From this approach, we have been able to provide solutions to the problem of load imbalances in chapter 3. Our proposal has been implemented and widely evaluated in different scenarios for data intensive applications: from real to synthetic applications, as well as analytical simulation, that have been described in chapter 4. Having completed all the stages that comprise our research, we are now able to present the final conclusions and further work of this study.

5.1 Final Conclusions

The continuous growth of data coming from sensors, biological and physical experiments, and information generated by users, needing to be processed, has led to the design of new methods to satisfy its processing requirements. Concepts as *data intensive* or *big-data* computing have risen in the last few years and, along with these terms, approaches like dividing the workload of the applications into smaller pieces (data chunks), have become more common. With all this in mind, the number of performance problems related to load balancing has also increased.

In our work, one of the main contributions is that we have provided a novel and satisfactory solution to the problem of load imbalances in data intensive applications. In some data intensive applications, partitioning its workloads into data chunks of equal size does not guarantee similar execution times. From experimental observations, the computation time associated to each data chunk is closely related to the characteristics of data and the behavior of the algorithm as well. To this end, we have designed and implemented a performance improvement methodology to be applied on a subset of data intensive applications. In particular, we have considered applications that perform multiple related explorations (queries) on the same workload.

The structure of the methodology has been influenced by the characteristics of dynamic performance analysis and tuning process. First, this process presents a *monitoring* phase where performance metrics are collected along the execution of the application. Then, in the *analysis* phase, gathered data is used to determine the corresponding modifications in certain tuning points. In this work, tuning parameters included are: (i) the workload partition factor; (ii) the distribution of generated data chunks among the application processes; and (iii) the number of computational resources (processing nodes) to be used by the application. Finally, the methodology has a *tuning* phase, in which these parameters are changed at run time.

First, the methodology considers the dynamic adaptation of the partition factor or,

for the case of high partitioning costs, the generation of multiple division of the workload using different partition factors before executing the application; and then, the dynamic selection of the most adequate one according to the behavior of the application. Then, tuning the workload partition factor has been described as the modification of the size of data chunks during the execution of the application. This modification has been achieved by dividing or gathering specific data chunks according to application performance. Additionally, data chunks have been distributed according to their associated computation time. In this case, data chunks with highest processing times have been delivered first. Once the execution of the application is as balanced as possible, it has been carried out the adaptation of the processing nodes that should be used to assure efficient executions, i.e. all workers finishing at the same time without large periods of idleness.

The methodology has been tested using the following applications: (i) the widely known bioinformatics tool BLAST, that handles a broad number of queries over large-scale biological databases; (ii) a distributed merge sort algorithm that process medium-scale text files. In addition, the main aspects of the proposal have been analyzed and evaluated through an analytical simulator. Using simulation, it has been possible to extrapolate observed results from the real and synthetic applications to analyze the behavior of the methodology on a wide range of scenarios. The results obtained have shown the capability of the methodology to improve the performance of data intensive applications with divisible load.

In most cases, sending data chunks with highest computation times first (this factor has been evaluated in section 4.2.1) and adapting the size of the data chunks at run time (evaluated and discussed in section 4.2.2), have greatly reduced total execution time of the applications (in comparison with obtained results when our methodology is not applied). Additionally, the adjustments in the active number of processing nodes (described in section 4.3.1) have permitted the maximum utilization of available computational resources, i.e., worker nodes in a parallel application developed under a Master-Worker paradigm.

Summarizing, the performance improvement achieved in data intensive applications when using our proposal, have been determined by three main reasons:

- Changing the scheduling policy, from a *First Come First Serve* (FCFS) to a *Heaviest Fragments First approach* (HFF).
- Adapting the partition factor of the workload at run time to reduce the time constraint imposed by data chunks with highest computation times.
- Tuning the number of workers doing computation to avoid inefficient executions, i.e. workers idle for long time.

Obtained results are promising in terms of reducing total execution time and efficient use of processing resources for different scenarios of data intensive applications. Furthermore, these results have shown that the proposed methodology can be widely used to improve performance in real data intensive applications.

5.2 Further Work and Open Lines

This work, as any work covering several aspects within a certain field of science, is intended to be complete and entirely closed. However, this research also gives rise to a wide range of affordable open lines and further work. These open lines are described below.

One of the points that can be further improved in the proposed methodology, in terms of data management, is the use of knowledge-based techniques for deciding the initial workload partition factor. To this end, it would be necessary to conduct a complete study of the relation between several parameters of the scenario, including: (i) systems parameters, such as network latency, memory and disk capacities of the processing nodes, and communication patterns, among others; and (ii) application parameters, such as data structures, complexity of the algorithm, dependance among data chunks, etc. In addition, the results of the study to estimate the appropriate initial workload partition factor could be used as inputs to improve our current performance model for partitioning and grouping data chunks at run time. Therefore, the partitioning process will not only assume linear divisions but also partitions suited to specific characteristics of the system or of the application.

A second point that should be considered and treated in the future is to provide estimations of data chunks executions times without having to execute the application. The relevance of this issue is given by applications where the number of iterations is too low to use the initial execution for labeling. In this case, estimations should be based on the characterization of the relation between the features of both data chunks and algorithms to approximate an expected execution time. Expected result might not necessarily be the optimum, but it should be close enough to real execution time to decide when the data chunk should be delivered in following iterations.

Finally, it would be interesting to incorporate to the methodology the capability to reuse data chunks that have been already sent to the workers. To this end, the distribution process would be not only based on the associated processing time of each data chunk, but also in *who* has processed it before. Therefore, overheads caused by not having the data chunk loaded in virtual memory would be reduced.

Among others, the most important topics are:

- Incorporate the methodology on the dynamic performance analysis and tuning tool MATE [63], to enable the execution of the application without participation of the user.

  MATE is a performance improvement environment for parallel applications that is based on monitoring, analyzing and tuning the execution of the applications at run time. Its tuning decisions are programmed into tunlets that the user introduces to the environment. An interesting research line might be the adaptation of the performance model of our methodology as a tunlet for MATE. In this way, the environment would be able to analyze the behavior of data intensive applications and tune them at run time.

  Recent scalable versions of MATE (that are developed in our research group), require the capability of taking local and global decisions about the performance of the application. Consequently, one of the major inconveniences that might appear is that our methodology has not considered other parallel paradigms than Master-Worker. Under this paradigm, decisions are taken in the master, and therefore, there is only a global view of the application.

- Improve the design of the performance model to enable considering heterogeneous processing capacities, as well as non-stable communication times or non CPU-bound applications. The relevance of this point is given by the characteristics of high performance computing systems available at this time. Most of them are mainly heterogeneous at both communication and computation levels.

  This work is focused in homogeneous clusters and CPU-bound applications with the aim of being able to define a base model. Nevertheless, our intention is to extend it adding new considerations. On a heterogeneous cluster more work has to be assigned to the faster nodes, otherwise they would be idle while the slower nodes are still executing their tasks. Therefore, different processing speeds for each worker, and different bandwidths of Master-Worker communication links should be considered to enhance our proposal.

  An initial and naive approach might be sending the heaviest data chunks (those with highest execution times) to faster processors and links. Nevertheless, if there is any fail in the distribution, it might result in unexpected load imbalances or overloading fast elements (processor and link). Therefore, when data and computations are not evenly distributed to processors, minimizing communication overhead becomes a challenging task.

On the other hand, if an application is not CPU-bound, we have considered that it could be IO-bound or Communication-bound. In the first case, to avoid having bottlenecks accessing to data, data chunks might be previously distributed to the processing nodes (to its local disks). In this way, disk latency of a worker might not affect the performance of other workers. In the second case, in which applications may be communication bound, we have considered that the simplest strategy would be to generate data chunks as small as possible to avoid possible communication delays (or interference between the workers).

- Design scheduling and tuning techniques that consider a wider range of data intensive applications. This point would supposed the inclusion of some algorithm fed with the main characteristics of the applications and its workload, such as:

  - *Complexity of the algorithm* (logarithmic; quadratic; linear; etc) that requires a different strategy to group or divide data chunks based on the resulting computation time.

    Divisible Load Theory have proved to be appropriate for workloads that have linear computational complexities. Nevertheless, an interesting open line for this research, is to define the partition sizes of the workload in accordance to the computational complexity of the application. Many algorithms used in real-time modeling and simulation of complex systems (e.g. signal and image processing, cryptography, and genetic algorithms) require processing load of nonlinear complexity, i.e., the computational time of the given data/load is a nonlinear function of the load size. In most of the algorithms, which require nonlinear computational complexity, it is possible to divide the loads arbitrarily and process them independently such that the total processing time is less than the processing time on a single processor [84].

    We have considered that for evaluating the partition factor that suits better for the application workload (based on the complexity of the algorithm), a first portion of an iteration of the application can be analyzed using different sizes of the data chunks. By doing this, it might be possible to improve the total processing time or to avoid data chunks sizes that might generate unexpected overheads. Nevertheless, one of the main issues we might face is that sampling phase (the initial phase to determine which partition factor suits better for a determine data intensive application) might become a bottleneck or might result too time consuming. If this happens, computational

complexity of the algorithm should be provided by the user, as well as the information of how the workload should be partitioned.

- *Non-arbitrarily divisible data* that must be distributed and processed following a certain predefined order to assure a correct functioning of the applications, e.g. image processing data that must be computed under pipeline paradigms. Data dependences, in addition to communication costs and control overhead, may lead to slow the whole load balancing process down to the pace of the slowest processors.

  Additionally, in the case of initial workloads that should be divided into one or more dimensions, we have analyzed these cases and thought that while the workload of the application enables division into smaller pieces, our methodology might be applied. In this sense, for divisions in more than one dimension, a record of the previous position of the data chunks (its coordinates) might help us to identify the pieces and determine both their distribution to the processing nodes, and the final order for generated data.

  Note that there is no reason *a priori* to restrict to a uni-dimensional partitioning of the data, more general data partitioning, such as two-dimensional, recursive, or even arbitrary slicing into rectangles, could be considered. But uni-dimensional partitioning is very natural for most applications, and, in some cases, finding the appropriate partition is already a hard task. Therefore, this situation together with the possibility of having data dependencies between data chunks have been included as an interesting open line for this work. Implementation details such as how to manage the different types of data sets, depend on the application and should be externally provided.

- Design and implement a variant of the methodology that could be used in virtualized environments (or non-dedicated clusters) specifically those as Cloud, in which the MapReduce programming paradigm has similar features with the Master-Worker scheme used in this work. In addition, the implementation of cost functions would facilitate the estimation of the processing nodes that should be contracted to met a certain budget.

  Virtualized environments offer highly heterogeneous systems, as well as certain economic restrictions that should be taken into consideration when launching data intensive applications. On the one hand, platforms such as Hadoop and Microsoft's Windows Azure have shown an efficient and powerful system to create highly scalable applications with many aspects of parallel computing auto-

matically provided. In this sense, our method to avoid load imbalances and inefficient use of resources could take advantage of the mechanisms for managing node failures, data and jobs allocation provided by Cloud platforms.

Besides studying the problem of load balancing for data intensive applications on virtualized platforms, an interesting open line might be to provide fast and efficient executions (considering a fixed budget) in computational systems with processing nodes of different capabilities and costs. Here, we might find heterogeneity and scalability issues that have not been considered in this work.

5.3 List of Publications

The work presented in this thesis has reported the following publications:

1. **C. Rosas, A. Morajko, J. Jorba, A. Espinosa, T. Margalef. *Performance Modeling for a Bioinformatics Parallel Application*, In Proceedings of the XXI Spanish Conference on Parallelism, pp. 459–465, Valencia, Spain, September 2010.** [76]

   This work focuses on the performance analysis of bioinformatics applications, such as the Basic Local Alignment Tool, BLAST [4]. In this paper, the main performance problems related to data intensive applications were identified: the influence of the scheduling policy, and the variability in processing times caused by data.

2. **C. Rosas, A. Morajko, J. Jorba, E. César. *Workload Balancing Methodology for Data-Intensive Applications with Divisible Load*, In Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 48–55, Vitória, Brazil, October 2011**. [77]

   This work proposes a methodology for improving performance of data intensive applications based on performing multiple data partitions prior to the execution, and ordering the data chunks according to their processing times during the application execution. Executions were monitored and the processing time of each exploration was used to dynamically tune the performance of the application. Performance improvement was achieved by adapting the scheduling policy and the number of processing nodes to be used by the application.

This paper has been selected as **Best Paper in the Applications Track** of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).

3. **C. Rosas, A. Morajko, J. Jorba, A. Moreno, E. César. *Improving Performance on Data-Intensive Applications Using a Load Balancing Methodology based on Divisible Load Theory*, In International Journal of Parallel Programming, *In Press*, Sent: December 2011, Accepted: May 2012**. [78]

   This work describes a methodology for dynamically improving the performance of data intensive applications based on its behavior. In this work the size and number of data partitions, as well as the number of processing nodes were adapted to provided faster and efficient executions. In this work, application execution was monitored and gathered data was used to dynamically tune the performance of the application. The tuning parameters included in the methodology were the partition factor of the data set, the distribution of the generated data chunks, and the number of processing nodes used.

4. **C. Rosas, A. Sikora, J. Jorba, A. Moreno, E. César. *Dynamic Tuning of the Workload Partition Factor in Data-Intensive Applications*, In Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC), *In Press*, Liverpool, UK, June 2012.** [79]

   This work proposes a strategy for dynamically tuning the partition factor of generated the data chunks. With the aim of decreasing the load imbalance and therefore the overall execution time, this strategy divided the data chunks with the biggest computation times and gathered contiguous chunks with the smallest computation times. The criteria to divide or join chunks was based on the associated execution time of each data chunk (average and standard deviation), as well as on the number of processing nodes to be used.

5. **C. Rosas, A. Sikora, J. Jorba, A. Moreno, E. César. *Dynamic Tuning of the Workload Partition Factor and the Resource Utilization in Data-intensive Applications*, In Future Generation Computer Systems, *Unpublished*, Sent: April 2012.**

   This work focuses on the evaluation of the methodology to dynamically improve the performance of data intensive applications based on: (i) adapting the size

and the number of data partitions to reduce overall execution time; and (ii) adapting the number of processing nodes to achieve an efficient execution. Evaluation was performed using real and synthetic data intensive applications, as well as analytical simulation. Reported results showed the viability of applying the performance improvement methodology in a wide range of data intensive applications.

Bibliography

[1] M. Abdullah, M. Othman, H. Ibrahim, and S. Subramaniam. An integrated approach for scheduling divisible load on large scale data grids. In *Computational Science and Its Applications – ICCSA 2007*, volume 4705 of *Lecture Notes in Computer Science*, pages 748–757. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74468-9. doi:10.1007/978-3-540-74472-6_61.

[2] M. Abdullah, M. Othman, H. Ibrahim, and S. Subramaniam. Optimal workload allocation model for scheduling divisible data grid applications. *Future Generation Computer Systems*, 26(7):971 – 978, 2010. ISSN 0167-739X. doi:10.1016/j.future.2010.04.003.

[3] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke. Software as a service for data scientists. *Communications of the ACM*, 55(2):81–88, Feb. 2012. ISSN 0001-0782. doi:10.1145/2076450.2076468.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410(8), October 1990.

[5] I. Banicescu and S. Flynn Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM. ISBN 0-89791-816-9. doi:10.1145/224170.224306.

[6] I. Banicescu and V. Velusamy. Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, IPDPS '01, pages 791–801, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-0990-8. doi:10.1109/IPDPS.2001.925034.

[7] I. Banicescu and V. Velusamy. Load Balancing Highly Irregular Computations with the Adaptive Factoring. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 195, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1573-8. doi:10.1109/IPDPS.2002.1015661.

[8] I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing*, 6: 215–226, 2003. ISSN 1386-7857. doi:10.1023/A:1023588520138.

[9] I. Banicescu, R. L. Carino, J. P. Pabico, and M. Balasubramaniam. Overhead Analysis of a Dynamic Load Balancing Library for Cluster Computing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, volume 2 of *IPDPS '05*, page 122b, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi:10.1109/IPDPS.2005.320.

[10] O. Beaumont, A. Legrand, and Y. Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Computing - Parallel matrix algorithms and applications*, 29(9):1121–1152, Sep. 2003. ISSN 0167-8191. doi:10.1016/S0167-8191(03)00095-4.

[11] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: results and open problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):207–218, Mar. 2005. doi:10.1109/TPDS.2005.35.

[12] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818675217.

[13] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Computing*, 6:7–17, 2003. ISSN 1386-7857. doi:10.1023/A:1020958815308.

[14] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3): 207–214, Mar. 1981. ISSN 0018-9340. doi:10.1109/TC.1981.1675756.

[15] J. Brest and V. Žumer. A Simple Method for Dynamic Scheduling in a Heterogeneous Computing System. *Journal of Computing and Information Technology*, 10(2):1330-1136, Jun. 2002. doi:10.2498/cit.2002.02.02.

[16] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical report, Carnegie Mellon Univ., May 2007.

[17] R. E. Bryant, R. H. Katz, and E. D. Lazowska. Big-Data Computing. White paper, Computing Research Association, 2008.

[18] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21:169–190, 2012. ISSN 1066-8888. doi:10.1007/s00778-012-0269-7.

[19] K. Bubendorfer and J. H. Hine. A Compositional Classification for Load Balancing Algorithms. Technical Report CS-TR-99-9, School of Mathematics and Computer Science. Victoria University of Wellington, 1998.

[20] M. Cannataro, D. Talia, and P. K. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Computing - Parallel data-intensive algorithms and application*, 28(5):673–704, May 2002. ISSN 0167-8191. doi:10.1016/S0167-8191(02)00091-1.

[21] R. Cariño and I. Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing*, 44:41–63, 2008. ISSN 0920-8542. doi:10.1007/s11227-007-0148-y.

[22] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2): 141–154, Feb. 1988. ISSN 0098-5589. doi:10.1109/32.4634.

[23] A. Caulfield, L. Grupp, and S. Swanson. Gordon: An improved architecture for data-intensive applications. *IEEE Micro*, 30(1):121 –130, jan.-feb. 2010. ISSN 0272-1732. doi:10.1109/MM.2010.18.

[24] CERN. LHC Computing. Online, Apr. 2012. URL http://public.web.cern.ch/public/en/lhc/Computing-en.html.

[25] E. César, A. Moreno, J. Sorribes, and E. Luque. Modeling Master/Worker applications for automatic performance tuning. *Parallel Computing - Algorithmic skeletons*, 32(7-8):568–589, 2006. ISSN 0167-8191. doi:10.1016/j.parco.2006.06.005.

[26] E. César Galobardes. *Definition of framework-based performance models for dynamic performance analysis*. PhD thesis, Universitat Autònoma de Barcelona, 2006. URL http://hdl.handle.net/10803/5760.

[27] Y.-C. Cheng and T. G. Robertazzi. Distributed computation with communication delay [distributed intelligent sensor networks]. *IEEE Transactions on Aerospace and Electronic Systems*, 24(6):700 –712, Nov. 1988. ISSN 0018-9251. doi:10.1109/7.18637.

[28] Y.-C. Cheng and T. G. Robertazzi. Distributed computation for a tree network with communication delays. *IEEE Transactions on Aerospace and Electronic Systems*, 26: 511–516, May. 1990. doi:10.1109/7.106129.

[29] S. Chuprat and S. Baruah. Scheduling Divisible Real-Time Loads on Clusters with Varying Processor Start Times. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 15–24, 2008. ISBN 978-0-7695-3349-0. doi:10.1109/RTCSA.2008.23.

[30] Z. Cvetanovic. The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems. *IEEE Transactions on Computers*, 36(4):421–432, Apr. 1987. ISSN 0018-9340. doi:10.1109/TC.1987.1676924.

[31] A. E. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo*, 2003.

[32] K. D. Devine, E. G. Boman, and G. Karypis. Partitioning and Load Balancing for Emerging Parallel Applications and Architectures. In *Parallel Processing for Scientific Computing*, chapter 6. Society for Industrial and Applied Mathematics, Philadephia, PA, 2006. doi:10.1137/1.9780898718133.ch6.

[33] G. Dodig-Crnkovic. Scientific Methods in Computer Science. In *Proceeding of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, 2002. URL http://www.mrtc.mdh.se/~gdc/work/cs_method.pdf.

[34] M. Drozdowski and P. Wolniewicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 311–319, London, UK, 2000. Springer-Verlag. ISBN 3-540-67956-1. doi:10.1007/3-540-44520-X_40.

[35] M. Drozdowski and P. Wolniewicz. Divisible Load Scheduling in Systems with Limited Memory. *Cluster Computing*, 6:19–29, 2003. ISSN 1386-7857. doi:10.1023/A:1020910932147.

[36] M. Drozdowski and P. Wolniewicz. Optimum divisible load scheduling on heterogeneous stars with limited memory. *European Journal of Operational Research*, 172(2): 545 – 559, 2006. ISSN 0377-2217. doi:10.1016/j.ejor.2004.10.022.

[37] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi:10.1145/1851476.1851593.

[38] European Laboratory for Particle Physics. CMS Experiment, Mar. 2012. URL http://cms.web.cern.ch/.

[39] European Organization for Nuclear Research. ATLAS Experiment, Mar. 2012. URL http://www.atlas.ch/computing.html.

[40] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41:30–32, 2008. ISSN 0018-9162. doi:10.1109/MC.2008.122.

[41] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm Data-Intensive Scientific Discovery*. Microsoft, Redmon, WA., 2009. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/.

[42] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35:90–101, Aug. 1992. ISSN 0001-0782. doi:10.1145/135226.135232.

[43] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 318–328, New York, NY, USA, 1996. ACM. ISBN 0-89791-809-6. doi:10.1145/237502.237576.

[44] Z. Ivezic, J. Tyson, R. Allsman, J. Andrew, R. Angel, et al. LSST: from Science Drivers to Reference Design and Anticipated Data Products. Technical report, Large Synoptic Survey Telescope, 2008. URL arXiv:0805.2366v2. Version 2.0.9.

[45] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Wiley professional computing. Wiley, 1991. ISBN 978-0-471-50336-1.

[46] J. Jorba Esteve. *Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes*. PhD thesis, Universitat Autònoma de Barcelona, 2006.

[47] H.-J. Kim. A Novel Optimal Load Distribution Algorithm for Divisible Loads. *Cluster Computing*, 6(1):41–46, Jan. 2003. ISSN 1386-7857. doi:10.1023/A:1020915000287.

[48] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2nd edition, 1998. ISBN 0-201-89685-0.

[49] K. Ko and T. G. Robertazzi. Equal allocation scheduling for data intensive applications. *IEEE Transactions on Aerospace and Electronic Systems*, 40(2):695 – 705, Apr. 2004. ISSN 0018-9251. doi:10.1109/TAES.2004.1310014.

[50] R. Kouzes, G. Anderson, S. Elbert, I. Gorton, and D. Gracio. The Changing Paradigm of Data-Intensive Computing. *Computer*, 42(1):26–34, 2009. doi:10.1109/MC.2009.2.

[51] P. Krueger and M. Livny. Load Balancing, Load Sharing and Performance in Distributed Systems. Technical report, Computer Science Department. University of Wisconsin, 1987. URL http://ftp2.cs.wisc.edu/pub/techreports/1987/TR700.pdf.

[52] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, Oct. 1985. ISSN 0098-5589. doi:10.1109/TSE.1985.231547.

[53] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling for cluster computing. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '07, pages 303–314, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2800-7. doi:10.1109/RTAS.2007.29.

[54] X. Lin, A. Mamat, Y. Lu, J. Deogun, and S. Goddard. Real-time scheduling of divisible loads in cluster computing environments. *Journal of Parallel and Distributed Computing*, 70(3):296 – 308, 2010. ISSN 0743-7315. doi:10.1016/j.jpdc.2009.11.009.

[55] W. Lu, J. Jackson, and R. Barga. AzureBlast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi:10.1145/1851476.1851537.

[56] B. Massingill, T. Mattson, and B. Sanders. A pattern language for parallel application programs. In *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 678–681. Springer Berlin / Heidelberg, 2000. ISBN 978-3-540-67956-1. doi:10.1007/3-540-44520-X_93.

[57] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3535-7. doi:10.1109/eScience.2008.62.

[58] R. Mian, P. Martin, A. Brown, and M. Zhang. Managing data-intensive workloads in a cloud. In *Grid and Cloud Database Management*, pages 235–258. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20045-8. doi:10.1007/978-3-642-20045-8_12.

[59] C. Miceli, M. Miceli, B. Rodriguez-Milla, and S. Jha. Understanding performance of distributed data-intensive applications. *Philosophical Transactions of The Royal Society*, 368(1926):4089–4102, Sep. 2010. doi:10.1098/rsta.2010.0168.

[60] A. Middleton. HPCC Systems: Introduction to HPCC (High-Performance Computing Cluster). White paper, LexisNexis Risk Solutions, 2011. URL http://hpccsystems.com/community/white-papers/hpcc-intro.

[61] R. Moore, T. A. Prince, and M. Ellisman. Data-intensive computing and digital libraries. *Communications of the ACM*, 41(11):56–62, Nov. 1998. ISSN 0001-0782. doi:10.1145/287831.287840.

[62] A. Morajko. *Dynamic tuning of parallel/distributed applications.* PhD thesis, Universitat Autònoma de Barcelona, 2003. URL http://www.tdx.cbuc.es/TDX-1124104-171625/.

[63] A. Morajko, T. Margalef, and E. Luque. Design and implementation of a dynamic tuning environment. *Journal of Parallel and Distributed Computing*, 67(4):474–490, Apr. 2007. ISSN 0743-7315. doi:10.1016/j.jpdc.2007.01.001.

[64] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef, and E. Luque. Dynamic Pipeline Mapping (DPM). In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 295–304, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85450-0. doi:10.1007/978-3-540-85451-7_32.

[65] A. Moreno, E. César, J. Sorribes, T. Margalef, and E. Luque. Task distribution using factoring load balancing in Master–Worker applications. *Information Processing Letters*, 109:902–906, 2009. ISSN 0020-0190. doi:10.1016/j.ipl.2009.04.014.

[66] NCBI. Blast homepage. http://blast.ncbi.nlm.nih.gov/, Jul. 2010. URL http://blast.ncbi.nlm.nih.gov/.

[67] C. Nyberg. Sort Benchmark Data Generator and Output Validator, 2011. URL http://www.ordinal.com/gensort.html.

[68] C. Oehmen and J. Nieplocha. ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17:740–749, Aug. 2006. ISSN 1045-9219. doi:10.1109/TPDS.2006.112.

[69] M. Othman, M. Abdullah, H. Ibrahim, and S. Subramaniam. Adaptive divisible load model for scheduling data-intensive grid applications. In *Proceedings of the 7th international conference on Computational Science, Part I*, ICCS '07, pages 446–453, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72583-1. doi:10.1007/978-3-540-69384-0_30.

[70] M. Othman, M. Abdullah, H. Ibrahim, and S. Subramaniam. A2DLT: Divisible Load Balancing Model for Scheduling Communication-Intensive Grid Applications. In *Proceedings of the 8th international conference on Computational Science, Part I*, ICCS '08, pages 246–253, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-69384-0_30.

[71] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12): 1425–1439, Dec. 1987. ISSN 0018-9340. doi:10.1109/TC.1987.5009495.

[72] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S.-H. Bae, Y. Ruan, S. Ekanayake, S. Wu, S. Beason, G. C. Fox, M. Rho, and H. Tang. Data intensive computing for bioinformatics. Technical report, Indiana University, Bloomington, IN, 12/29/2009 2009. URL http://grids.ucs.indiana.edu/ptliupages/publications/DataIntensiveComputing_BookChapter.pdf.

[73] J. Qiu, J. Ekanayake, T. Gunarathne, J. Choi, S.-H. Bae, H. Li, B. Zhang, T.-L. Wu, Y. Ruan, S. Ekanayake, A. Hughes, and G. Fox. Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics*, 11(Suppl 12):S3, 2010. ISSN 1471-2105. doi:10.1186/1471-2105-11-S12-S3.

[74] T. G. Robertazzi. *Networks and Grids: Technology and Theory*. Springer Publishing Company, Inc., 1st edition, 2007. ISBN 038768235X, 9780387682358.

[75] C. G. Rommel. The probability of load balancing success in a homogeneous network. *IEEE Transactions on Software Engineering*, 17(9):922–933, Sep. 1991. ISSN 0098-5589. doi:10.1109/32.92912.

[76] C. Rosas, A. Morajko, J. Jorba, A. Espinosa, and T. Margalef. *Performance Modelling for a Bioinformatics Parallel Application*, chapter Evaluación de Prestaciones, pages 459–465. Ibergaceta Publicaciones, S.L., 1 edition, September 2010.

[77] C. Rosas, A. Morajko, J. Jorba, and E. César. Workload Balancing Methodology for Data-Intensive Applications with Divisible Load. In *Proceeding of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 48–55. IEEE Computer Society, Oct. 2011. doi:10.1109/SBAC-PAD.2011.15.

[78] C. Rosas, A. Sikora, J. Jorba, A. Moreno, and E. César. Improving performance on data-intensive applications using a load balancing methodology based on divisible load theory. *International Journal of Parallel Programming*, –:–, 2012.

[79] C. Rosas, A. Sikora, J. Jorba, A. Moreno, and E. César. Dynamic tuning of the workload partition factor in data-intensive applications. In *In Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Jun. 2012.

[80] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009. doi:10.1093/bioinformatics/btp236.

[81] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. ISBN 0818665874.

[82] A. Shokripour and M. Othman. Survey on divisible load theory and its applications. In *Proceedings of the International Conference on Information Management and Engineering*, ICIME '09, pages 300 –304, 2009. doi:10.1109/ICIME.2009.59.

[83] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981. ISSN 0022-2836.

[84] S. Suresh, H. Kim, C. Run, and T. Robertazzi. Scheduling nonlinear divisible loads in a single level tree network. *The Journal of Supercomputing*, –:1–21, 2011. ISSN 0920-8542. doi:10.1007/s11227-011-0677-2.

[85] A. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, J. Heasley, T. Hey, M. Nieto-SantiSteban, A. Thakar, C. van Ingen, and R. Wilton. Graywulf: Scalable clustered architecture for data intensive computing. In *42nd Hawaii Interna-*

tional Conference on System Sciences, 2009., HICSS '09., pages 1 –10, Jan 2009. doi:10.1109/HICSS.2009.234.

[86] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 57–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi:10.1109/SC.2005.52.

[87] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467:Nature Publishing Group, a division of Macmillan Publishers Limited., 2012. doi:10.1038/nature09534.

[88] B. Veeravalli and S. Ranganath. Theoretical and experimental study on large size image processing applications using divisible load paradigm on distributed bus networks. *Image and Vision Computing*, 20(13–14):917 – 935, 2002. ISSN 0262-8856. doi:10.1016/S0262-8856(02)00090-2.

[89] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, Sep. 1993. ISSN 1045-9219. doi:10.1109/71.243526.

[90] C. Xu and F. C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Number 381 in Springer International Series in Engineering and Computer Science Series. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 079239819X.

[91] Y. Yang, H. Casanova, M. Drozdowski, M. Lawenda, and A. Legrand. On the Complexity of Multi-Round Divisible Load Scheduling. Research Report RR-6096, INRIA, 2007. URL http://hal.inria.fr/inria-00123711.

[92] C. Yu and D. Marinescu. Algorithms for divisible load scheduling of data-intensive applications. *Journal of Grid Computing*, 8:133–155, 2010. ISSN 1570-7873. doi:10.1007/s10723-009-9129-0.

Index