Dynamic Tuning of the Workload Partition Factor and the Resource Utilization in Data-intensive Applications

Claudia Rosas[a], Anna Sikora[a], Josep Jorba[b], Andreu Moreno[c], Antonio Espinosa[a], Eduardo César[a]

[a]*Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona. 08193, Bellaterra. Spain*
[b]*Estudis d'Informatica, Multimedia i Telecomunicacio, Universitat Oberta de Catalunya, 08018 Barcelona, Spain.*
[c]*Escola Universitaria Salesiana de Sarria, 08017 Barcelona, Spain*

Abstract

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, in order 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 exploration (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 def-

Email addresses: crosas@caos.uab.es (Claudia Rosas), ania@caos.uab.es (Anna Sikora), jjorbae@uoc.edu (Josep Jorba), amoreno@euss.cat (Andreu Moreno), aespinosa@caos.uab.es (Antonio Espinosa), eduardo@caos.uab.es (Eduardo César)

inition 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 as case of study 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 tuning, Data-intensive applications, Divisible Load Theory (DLT)

---

1. Introduction

Nowadays, one of the biggest challenges in the computational field is the continuous growth of data that needs to be processed. The data flow coming from sensors, results of biological and physical experiments [1], and even from the information generated by users, are surpassing the capacities of the systems and algorithms recently designed. This led to a new type of applications known as *data-intensive applications* [2], or *big-data computing* [3].

In the era of data-intensive applications, computational systems are not only intended to compute but also to store and manage data. Given the current volume of data, those tasks increase the challenge and complexity of developing a suitable solution. Moreover, the efficient data processing is not only a matter of having a large number of processing units because it also depends on characteristics of the workload of the application.

In order to improve performance, there are many studies that have obtained good results, ranging from approaches that analyze the effectiveness of I/O systems, to the design of appropriate strategies to define and access data structures [4]. In many cases, it has been necessary to divide the workload of data-intensive applications into smaller data chunks (according to *Divisible Load Theory*, DLT [5]) to ensure that the workload of the application can be manageable. This has

been done to reduce the size of the workload and enable parallelism, but once the workload has been divided, other issues rises like those related to disk access or load balancing.

The execution of data-intensive applications that involves a large number of queries or iterations, may lead to variations in the overall execution time between iterations. For this reason, performance analysis and load balancing techniques must be adapted to particular characteristics of the application. In most cases, given the variability between (or within) iterations, performance analysis must be carried out at run time. This is an extremely complex process because it is carried out during the execution of the application without incurring in excessive overheads. If not, the proposed solution may be obsolete from one iteration to the other.

Most load balancing methods, such as *factoring* [6][7], are based on the idea of distributing the workload of the application in chunks of decreasing size. For the purpose of improving total computation time of scientific applications, these methods try to determine a good *partition factor* to obtain the chunks. When doing this, parameters such as computation time, communication time, and overall performance of the application are taken into consideration.

This work proposes a methodology that dynamically identifies and tunes load imbalances in parallel data-intensive applications. This proposal is oriented to: applications that perform several related explorations or queries[1] on a large workload; and the possibility of arbitrarily dividing or concatenating the workload of the application into data chunks of different size. These assumptions are sound because large-scale data processing usually consists on launching several related explorations on the data, and processing can be performed on data chunks of arbitrary size.

To improve performance in parallel data-intensive applications with arbitrarily divisible workloads, our methodology considers: (i) the adaptation of the partition factor for the workload to reduce overall execution time and avoid load imbalances; and (ii) the modification of the number of processing nodes that can be used efficiently. This proposal works for homogeneous clusters and uses an application performance model that allows for dynamically adjusting the tuning parameters according to the current application behavior.

The methodology is based on monitoring the computation time of generated

[1]We consider terms: exploration, query and iteration as synonyms; and they may be used interchangeably along this work.

data chunks to determine the order in which they should be scheduled in future explorations. The proposal includes the dynamic division and gathering of data chunks (when the partitioning cost is low); and the possibility of dynamically choosing among previously generated partitions (when the partition cost is too high). In both cases, the calculation of the partition factor will take into consideration the communication cost, memory use, and the number of available computing nodes (besides the computation time).

Our methodology assumes that a single execution includes multiple related explorations on the same partitioned workload. Thus, previously collected data for one exploration can be used to dynamically adapt the number of resources (processing nodes) for subsequent explorations. As our method is based on the execution of applications in homogeneous clusters of workstations, the computation capacity is constant and, in most cases, the disk and network latency are stable. Moreover, in order to make easier the initial design we used a *shared nothing* [8] processing approach. Under this approach, each node (consisting of processor, local memory, and disk resources) shares nothing with other nodes in the cluster.

Summarizing, our strategy proposes:

a) Generating multiple representative workload partitions prior to the execution of the application when the cost of partitioning data is too high.

b) Monitoring computation time of every exploration or query on every data chunk.

c) Ordering and allocation of data chunks along the execution of the application according to their associated computation times.

d) Tuning of the partition factor of the data chunks with the highest (partitioning) and lowest (grouping) associated computation time according to the observed efficiency (relation between computation time and the number of computation nodes).

e) Distributing newly generated data chunks in subsequent explorations.

f) Estimating the number of processing nodes to be effectively used by the application.

The evaluation of our proposal has been carried out in a real and widely used data-intensive application: the computation/data-intensive bioinformatics tool *Basic Local Alignment Sequence Tool* (BLAST) [9], as well as on a distributed merge

sort. Results obtained from both applications are encouraging in terms of total execution time reduction and efficient use of resources.

Moreover, an analytical simulator has been used to evaluate the analytical expressions of the methodology. These expressions are used to estimate the modifications in the size of the data chunks and in the number of processing nodes to be used. In order to analyze the behavior of the methodology we used the simulation for a wider range of scenarios.

The rest of the paper is organized as follows. First, Section 2 provides an overview of related work. Next, Section 3 describes the proposed methodology for balancing the load and improving performance of data-intensive applications. Section 4 shows the most relevance characteristics of the selected scenarios to evaluate our methodology: (i) a real data-intensive application, the bioinformatics tool BLAST; (ii) a synthetic application based on a distributed sorting algorithm; and (iii) an analytical simulator of data-intensive applications. In addition, Section 4 explains the reasons for using these applications to test the proposed methodology. Section 5 where the experimental evaluation is described and results are discussed. Finally, Section 6 shows the conclusions and outlines future work.

2. Related Work

A divisible workload is such that can be divided into several independent pieces or chunks of arbitrary size to be processed in parallel by a set of compute nodes. The Divisible Load Theory (DLT) [5] was introduced in the late 1980s. Later on, DLT has branched in many new directions covering scheduling problems and performance modeling for various types of computational environments, such as Grid and Cloud systems [10], systems with memory limitations [11] or with computation time restrictions [12].

Some approaches, as the presented in [13], consider models of adaptive divisible load based on Genetic Algorithms in order to improve scheduling tasks on large scale data grids. In their work, they have decided the size of the portion of workload to be allocated to each processor to minimize the turnaround time of a job.

Others works, such as [14], have performed a wide study on the complexity of multi-round divisible load scheduling. They have considered a Master/Worker approach using heterogeneous platforms and have answered how the load should be partitioned in order to minimize the time to complete the last unit of work.

Many of these studies have used the linear programming models proposed in the first publications of DLT to represent the scheduling problem as a system of

linear equations. These equations are evaluated at run time to decide the appropriate amount of load to deliver at each processing node. On the contrary, our proposal uses analytical expressions to estimate both the size of the load to be delivered and the appropriate number of resources to be used. Previous works also consider factors, such as: variable startup times of the processing nodes [12], different types of interconnection networks [15] or, in recent publications, Map Reduce computations [16] that are outside the scope of this work.

Regarding the dynamic tuning of performance parameters, the works proposed in [17] and [18] have introduced the use of a load balancing strategy named *factoring* [6][7] to modify the size of the data chunks distributed among the workers. In the first proposal, the authors have detailed a performance model for Master/Worker applications that comprises two stages: one stage for load balancing, and the other to adapt the number of workers.

Later, they have defined a strategy for dynamically improving the performance of pipeline applications [19]. This strategy improves the throughput of applications by gathering the fastest pipe stages and replicating the slowest ones. In this work, we further extend the basis proposed by [17], [18] and [19] to parallel data-intensive applications with arbitrarily divisible loads.

Finally, there are some works that consider the processing characteristics of data-intensive applications for developing dynamic load balancing strategies, such as [20] and [21]. The first is focused on multicast problems for data-intensive applications on the Cloud, while the second developed resource allocation and scheduling strategy to minimize the total time spent on processing data. Although these works have considered partitioning data, it is mainly focused on data locality.

3. Methodology for improving performance in data-intensive applications

The main contribution of this work is the introduction of a methodology for improving the performance of data-intensive applications through dynamic load balancing and adaptation of the number of used resources. Specifically, the methodology determines: (i) the number of data chunks in which the workload is divided; (ii) the scheduling strategy for data chunks; and (iii) the number of processing nodes to be used. This methodology has been developed making the following assumptions:

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

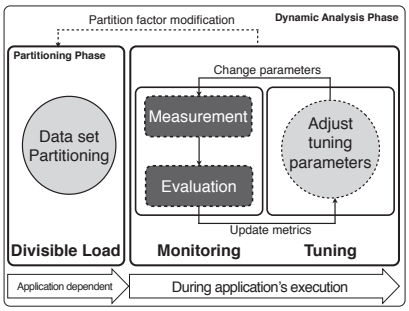


Figure 1: General description of the load balancing methodology.

2. The application performs a set of related explorations or queries on the workload, e.g. the application searches similarities for several related proteins on a large database, or looks for related strings on the web.
3. The performance of the application varies significantly (according to the input data), justifying the use of a dynamic performance analysis/tuning approach.
4. The characteristics of the input data are unknown at the beginning.

The methodology has been designed for homogeneous clusters because they provide steadiness in systems' parameters, such as processing capacity and disk and network latency, simplifying the model definition.

The proposed methodology, represented on figure 1, includes a partitioning phase and a dynamic analysis phase. In the partitioning phase, the initial workload is divided into smaller pieces, and multiple alternative partitions are generated whether the cost of generating new partitions during the application execution is too high. The aim of our approach is to take advantage of the adaptation made in [18] to the load balancing policy, named factoring [6][7]. The modification proposed in [18] 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 policy will be applied at execution time. However, the workload's partitioning cost can be high for data-intensive applications. If this happens, we propose to generate multiple partitions before executing the application and then, during the execution choose which of them is the most appropriated one. In the dynamic analysis phase, performance metrics are collected and the performance model is evaluated. Both tasks are carried out dynamically in order to determine which tuning parameters must be adjusted for the next explo-

Table 1: Summary of notation

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 expl. $\left(C_i = \sum_{j=1}^{N_f} C_{ij}\right)$
μ_i	average computation time (in secs) for the ith expl. $\left[\mu_i = (\sum_{j=1}^{N_f} C_{ij})/(N_f)\right]$
σ_i	standard deviation of computation time (in secs).
ρ_n	performance index for n number of workers.
Ts_i	total sequential computation time for the ith expl. $\left(\forall i \in N_q : Ts_i = \sum_{j=1}^{N_f} C_{ij}\right)$
T_{max_i}	maximum computation time for ith expl.
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.

ration. In this phase, the performance of the application is improved at run time by tuning three performance parameters: (i) the workload partition factor (number of data chunks); (ii) the scheduling policy; and (iii) the number of resources (processing nodes) used. In order to perform a dynamic tuning, measurements relative to execution time has to be collected at run time. Then, collected data is evaluated using the performance model and, based on the results of the evaluation, the performance parameters are tuned. The notation used in our methodology is described in Table 1.

The dynamic analysis phase is summarized in the flowchart presented in figure 2. Here, the main tasks performed in the dynamic analysis phase are highlighted: (i) measurement; (ii) model evaluation; and (iii) tuning. The execution of the

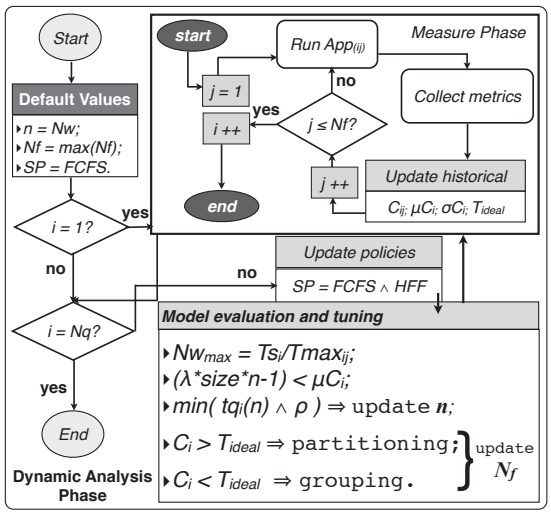


Figure 2: Dynamic analysis phase of the load balancing methodology.

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 subsection 3.1. These values can be modified at run time because they affect the overall performance of the application.

Data chunks associated computation time is collected in the *measurement* phase. In the first exploration, a *First Come First Serve* scheduling policy is used because there is no 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* approach (sending data chunks according to their processing times in decreasing order). 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 application's workload might have been partitioned prior to execution, but the tuning of these parameters will be done dynamically and continuously at run time.

3.1. Selection of the initial workload partition factor

In general, the workload of data-intensive applications can be split into smaller data chunks. Nevertheless, to select how many data chunks should be generated, i.e. the workload partition factor, is a non-trivial endeavor. The difficulty resides in how to choose a trade-off between a well balanced executions 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 each chunk may introduce some overhead in the total execution time.

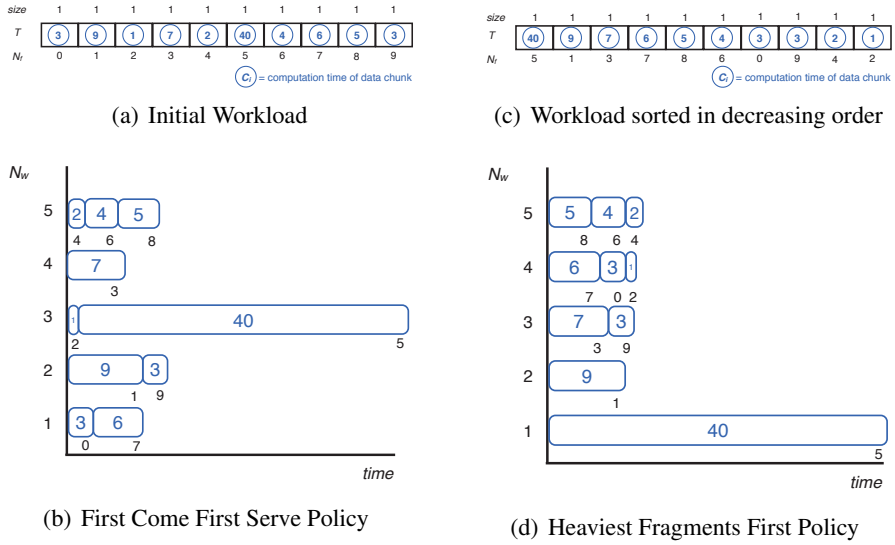
First, an initial common partition factor for all data chunks is established. This factor is based on: (i) hardware parameters, such as the available physical memory, the network bandwidth, and the number of available resources (nodes); and (ii) application parameters, such as the total size of the workload, its partitioning cost (time), and the number of explorations. Particularly, the partitioning cost determines whether all partitions or only the initial ones, are going to be generated before the execution of the application. In the first case, the proposed methodology will dynamically choose the best partition factor among those available, while in the second case it will generate the best partition dynamically.

Once data chunks has been 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 have the same size, the average computation time by data chunk may vary. We use this variation to determine the modifications of 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. Selecting a scheduling policy

The application starts with predefined default values for the tuneable parameters: a high number of workers, limited by the number of nodes available in the cluster, and a high workload partition factor (as shown in figure 3(a)). As we mentioned before, our methodology assumes that the explorations are related to each other and they are processed sequentially. Once an exploration has been processed, the execution time for each data chunk is stored and historical statistics updated.



(a) Initial Workload

(b) First Come First Serve Policy

(c) Workload sorted in decreasing order

(d) Heaviest Fragments First Policy

Figure 3: Comparison between scheduling policies.

In the first iteration (exploration), a *First Come First Serve* scheduling policy is used because there is no information about data chunks' computation time (as shown in figure 3(b)). Then, starting from the second exploration, data chunks are ordered and scheduled according to their associated processing time. Data chunks are ordered from the heaviest (those with highest processing time) to the lightest ones (with the lowest processing time) as shown in figure 3(c). When a worker requests a chunk, it receives the heaviest non-processed chunk. This scheduling method is known as HFF (*Heaviest Fragments First*) [22].

Additionally, when monitoring the behavior of the application using the initial workload partition factor, two kinds of data chunks are 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. Data chunk with the maximum computation time is labeled as T_{max_i}. In a parallel execution 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, resulting in an inefficient execution (as shown in figure 3(d), where total execution time will not be lower than the time of the data chunk number 5).

3.3. Adjusting the Partition Factor

Besides scheduling, the proposed methodology also adapts dynamically the partition factor with the aim of balancing the load among workers (this strategy has been named ***HFF + factor***). The methodology considers: (i) repartitioning the data chunk(s) with highest associated computation time; and (ii) gathering data chunks with low associated computation times. The main criteria 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 (1)), is given by the relation between the serial computation time of the entire workload, Ts_i; and the total number of available processing nodes N_w.

Consequently, monitoring the execution time of the data chunks C_{ij} allows to calculate 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.

$$T_{ideal} = \frac{T_{s_i}}{N_w} = \frac{(\mu_i * N_f)}{N_w} \tag{1}$$

3.3.1. Partitioning

When executing a data-intensive application in parallel, the total execution time C_i is given by the last worker that finishes processing. Usually, this delay is given by processing data chunks with large execution times. In order to reduce this time and balance the execution, we propose to break this (or these) data chunk(s) into smaller pieces, and reallocate them among the available processing nodes.

In this work, we chose a conservative approach to partition data chunks in order to prevent unnecessary reallocation of data chunks with short execution times. A threshold, defined by expression (2), is used to determine whether a data chunk should be partitioned or not. We defined this restriction because, in some cases, after repartitioning 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 observed through experimentation, and this non-linearity characteristic depends on both the algorithm and the data.

$$C_{ij} > T_{ideal} \tag{2}$$

With the objective of reducing the gap between data chunks with large execution time, and the ideal time T_{ideal}, there is a need to estimate the computation time for partitioned data chunks. In this sense, a statistic of order N_w (shown in expression

(3)) is used to estimate 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.

$$E = \mu_i + \sigma_i * \sqrt{N_w/2} \tag{3}$$

For statistical purposes and with the objective of enabling the estimation of the number of new partitions, we have introduced appropriate modifications to ensure consistency of 3), resulting on the expression (4).

$$E = \left(\frac{C_{ij}}{y}\right) + \left(\frac{\sigma_i}{y}\right) * \sqrt{\frac{N_w}{2}} \tag{4}$$

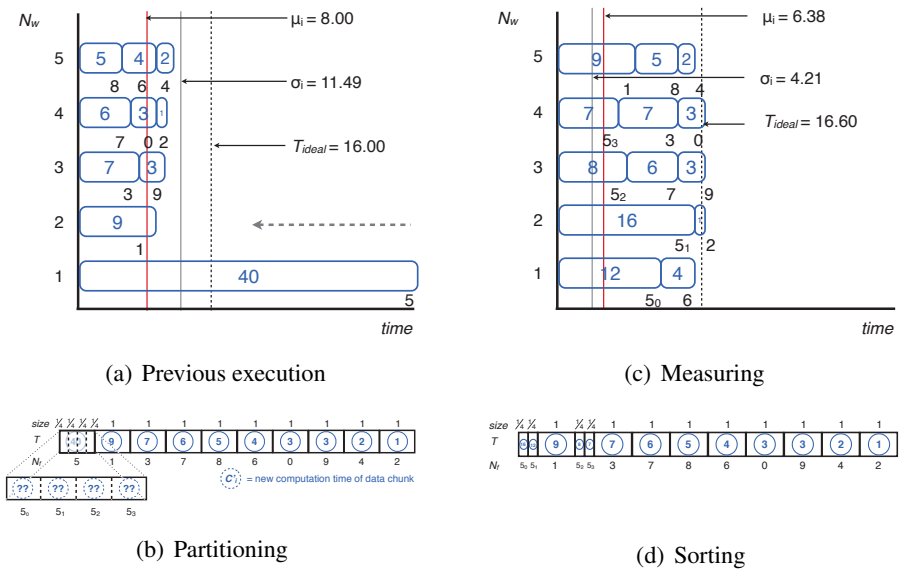
In this expression, the average computation time of data chunks which meets the time restriction (2) will be given by the relation between their associated computation time C_{ij} and the number of new data chunks generated y. Similarly, standard deviation of new data chunks will be represented as the 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 exactly the same as the population mean; and the variance of the sampling distribution of variances equals the population variance divided by the sample size.

$$\left(\frac{C_{ij}}{y}\right) + \left(\frac{\sigma_i}{y}\right) * \sqrt{\frac{N_w}{2}} \leq T_{ideal} \tag{5}$$

Finally,to define the number of new data chunks to be generated, y must be calculated from expression (5), leading to expression (6).

$$y = \frac{N_w * \left(C_{ij} + \sigma_i * \sqrt{\frac{N_w}{2}}\right)}{(\mu_i * N_f)} \tag{6}$$

For example, if an exploration is executed using the following values for the number of processing nodes and partition factor, $N_w = 5$ and $N_f = 10$, the resulting scheduling may look as the one in figure 4(a). 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 $\mu_i = 8$, $\sigma_i = 11.49$, and $T_{ideal} = 16$.



(a) Previous execution

(c) Measuring

(b) Partitioning

(d) Sorting

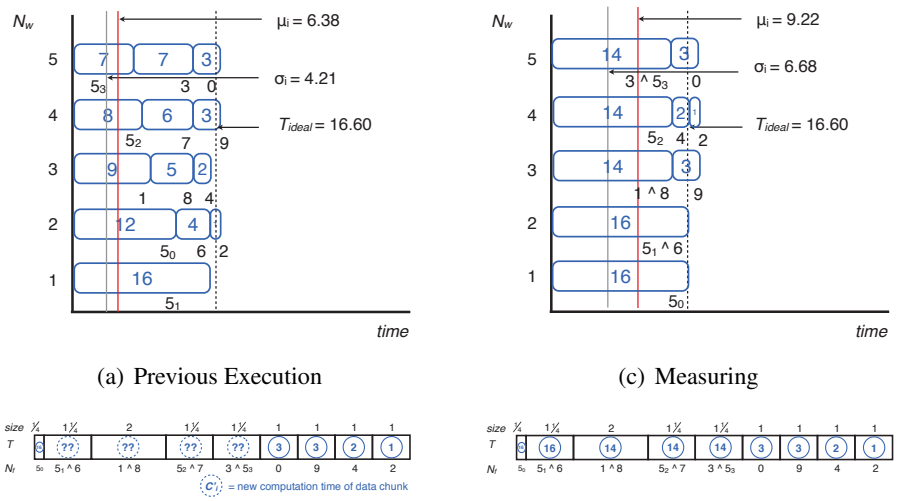
Figure 4: Partitioning data chunks with the highest execution times.

After evaluating the restriction given by expression (2), and solving y from expression (6), the resulting value for the number of pieces in which data chunk $j = 5$ should be partitioned is $y = 4$ (as shown in figure 4(b)). However, the computation times for the new data chunks are unknown. Then, a subsequent exploration is used for labeling new data chunks with their associated computation times. In order to avoid possible load imbalances, new data chunks are scheduled when their original data chunk was expected to be sent (figure 4(c)). The new chunks can be labeled and rearranged in decreasing order of computation time, for the next exploration (as shown in figure 4(d)).

3.3.2. Grouping

Partitioning the workload in small data chunks may improve load balancing but can also produce overheads on scheduling, communication and computing. Consequently, we propose to evaluate the application performance at run time and, whether is convenient to group or distribute bigger data chunks to avoid these overheads.

For every data chunk that can be *grouped*, the grouping strategy will stop when the sum of the associated computation time of the data chunk exceeds T_{ideal}. However, by doing this, too many data chunks with a similar computation time may be



(a) Previous Execution

(c) Measuring

(b) Grouping

(d) Sorting

Figure 5: Grouping data chunks with the shortest execution times.

generated. If this happens, the number of data chunks with short computation time will not be enough to fill the *blanks* left by imbalances along the exploration. Due to this situation, we considered a more precise approach to estimate the total time for the grouped data chunks. This has been achieved by defining a time threshold for such data chunks (as shown in expression (7)). 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.

$$T_{group} \leq T_{ideal} \tag{7}$$

For the sake of clarity in the example, data chunks are left in the same order as in previous execution (as shown in figure 5(a)). Then, bigger data chunks are created by grouping "smaller" data chunks (those data chunks with the lowest execution times) while the restriction defined in expression 7 is not met. We evaluate the possible resulting computation time when grouping, i.e. T_{group}, by adding the computation time of the selected data chunks. For this particular example, we group data chunks 5_1 and 6, 1 and 8, 5_2 and 7, and 3 and 5_3 (as shown in figure 5(b)) because the sum of their computation times is below the defined threshold. Later, similarly to *partitioning* strategy, new data chunks are executed when their maximum of their original data chunks would have been executed (as represented

in figure 5(c)) to label the data chunks with their new associated computation time. In this manner they can be sorted in decreasing order for the following explorations (figure 5(d)).

3.4. Estimation of the number of resources being used

After tuning the partition factor, and once the load is as balanced as possible, we can assess the number of resources (processing nodes) that are going to be used. It is possible to determine the maximum number of processing nodes n that can be used, accordingly to the measured processing time for each data chunk, and expressions (8) and (9).

$$n \leq \frac{Ts_i}{T_{max_{ij}}} \tag{8}$$

Expression (8) is used to calculate the maximum number of workers which may be processing efficiently. It is possible to infer from (8) that the minimum execution time for an exploration is limited by the chunk with the maximum processing time. Expression (9) is used to calculate the maximum number of processing nodes (workers) that can be managed by the master. If a worker finishes before the master has distributed chunks to every other worker that worker must wait.

$$n \leq \frac{\sum_{j=1}^{N_f} C_{ij}}{\lambda * size * N_f} \tag{9}$$

In addition, the methodology must be able to estimate the application execution time for an exploration using a certain number of workers, in order to decide if this parameter should be changed. This estimation is done through expression (10), which takes into consideration the time needed for sending the first chunk to all workers ($\lambda * size * (n-1)$) and the computation done by one worker ($(\lambda * size) + \mu_i$) on every chunk it expects to receive (N_f/n).

$$tq_i(n) = [\lambda * size * (n-1)] + (N_f/n) * [(\lambda * size) + \mu_i] \tag{10}$$

The criteria for deciding the appropriated number of workers has been defined as an index relating the estimated execution time ($tq_i(n)$) and the efficient use of the resources. Efficiency is defined by (11) 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 ($tq_i(n)$).

$$Ef_n = \frac{\mu_i * N_f}{n * tq_i(n)} \tag{11}$$

Consequently, expression (12) was designed to find the number of workers that minimizes both the exploration execution time and the efficiency loss.

$$\rho_n = \frac{tq_i(n)}{Ef(n)} = \frac{n * tq_i^2(n)}{\mu_i * N_f} \tag{12}$$

Summarizing, the number of processing nodes (n, workers) used by the application is delimited by the minimum values of (8), (9) and (12).

In order to be able to dynamically tune these factors, the following parameters must be monitored: network parameters, such as bandwidth and setup overhead; communication parameters, such as message size; and computation parameters, such as CPU utilization. In this way, the loop shown in figure 1 is closed, covering the process of dynamic monitoring, analysis, and tuning a data-intensive application with divisible workload.

4. Case studies of Data-Intensive Applications

The proposed methodology has been designed considering dedicated clusters working under a *shared nothing* [8] processing approach. This means that each node consisting of processor, local memory, and disk resources shares nothing with other nodes in the cluster. 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.

The methodology has been evaluated using: (i) a real and widely used data-intensive application, BLAST (subsection 4.1); (ii) a synthetic application based on the merge sort algorithm (subsection 4.2); and (iii) an analytical simulator (subsection 4.3). Each scenario has been selected to analyze different stages of the methodology. First, BLAST was used as main case of study to check the performance gain for real applications when applying our proposal. Then, we used a synthetic application to facilitate the analysis of the workload partition factor adaption at run time (because, in comparison with BLAST, the partitioning process is faster). Finally, the analytical simulator evaluates the methodology in a wide range of scenarios.

4.1. Basic Local Alignment Search Tool

As a real application, we choose BLAST (Basic Local Alignment Sequence Tool) [9] for assessing our proposal because: (a) it is one of the most widely used bioinformatics tools, and (b) it satisfies the assumptions presented in Section 3.

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 or Swiss-Prot. Based on heuristics, BLAST algorithm improves up to 10 times the exact match Smith-Waterman Algorithm [23].

BLAST is both a CPU and a data-intensive application. This application presents long and irregular processing times due to data characteristics, and it processes biological databases up to 50 GB size –and growing– that can be arbitrarily divided into non-dependent data chunks (in BLAST literature these chunks are called fragments).

Most parallel BLAST versions, such as mpiBLAST [24] and ScalaBLAST [4], have been developed using the Master/Worker paradigm and they take advantage of the parallelism of the systems using database partitioning. In general, the 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. 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. In consequence, BLAST represents a good candidate to benefit for our proposal. Overall performance improvements obtained from applying our methodology to this real scenario are shown in Section 5.1.

4.2. Distributed sorting algorithm

The main characteristic of any sorting algorithms is the difference in computation time when processing unsorted and sorted files. Additionally, many algorithms as *merge sort* include modifications to enable sorting large input files in a short period of time. These two characteristics, 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 is used to analyze the effect of dynamically modifying: (i) the scheduling policy; (ii) the size of the data chunks[2]; and (iii) the number of processing nodes used. This application has been developed as a Master/Worker

[2]By repartitioning or grouping those meeting the restrictions defined in subsection 3.3

and its input data files (of unsorted items) are generated using the *gensort* program [25]. These files contain items represented as lines of 100 ASCII characters. The size of generated files is of up to 32 GB.

In order to introduce variability in the computation time associated to some of the data chunks in the workload, unsorted and sorted data chunks has been randomly combined in the workload. The distribution was performed following the scheduling policies described in subsection 3.2. Moreover, to keep integrity in the results, after each worker performed a distributed merge sort in the received data chunks, and once all its data chunks has been sorted, the workers merge their processed data chunks into a bigger file.

This application 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.

4.3. Analytical simulator

The aim of the proposed methodology is to achieve balanced executions for data-intensive applications by tuning performance parameters such as, the workload partition factor and the number of processing nodes used. In order to do this, some analytical expressions were defined to choose the appropriate values for these parameters. Nevertheless, the evaluation of the appropriate functioning of the methodology is a big challenge. In many cases, the evaluation of a performance improvement proposal requires a lot of work to implement, debug and execute the analysis environment.

In this sense, an analytical simulator has been implemented to evaluate the load balancing methodology (described in Section 3) in a wide range of scenarios. The developed tool allows us to observe and analyze the influence of the performance parameters in the execution of the application. For example, we were able to see how certain variations in data chunks processing time, and how changes in the partition factor affect the performance of the application. Thus, the simulator is able to reproduce such situations.

As a first step, the simulator has been fed with different scenarios of input data. These initial 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, we defined 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. The communication time was modeled as the product of the size of the data chunk by the communication time per MB. Thus, it is possible to estimate the time for sending every data chunk.

The expression (10) 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) was not 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 in a great number of scenarios; (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 subsection 5.3.

5. Case studies evaluation

In order to evaluate the two phases of the methodology described in Section 3 a set of experiments using representative scenarios of data-intensive applications were designed. First, the proposed methodology was evaluated using real bioinformatic (BLAST) and merge sort applications, and the corresponding results are presented and discussed in subsections 5.1 and 5.2. Then, in subsection 5.3 we present the main simulation results.

5.1. Applying the methodology to BLAST

The aim of these experiments is to evaluate the performance of real executions of BLAST when: (i) changing the scheduling policy to *Heaviest Fragments First* (subsection 5.1.1); (ii) tuning the size of data chunks according to the performance of the application (subsection 5.1.2); and (iii) varying the number of processing nodes to determine the best number of nodes (subsection 5.1.3).

Experimental environment: The experiments were carried out in a cluster of workstations with 32 processing nodes, with 12GB of memory per node. Each node consists of two dual core Intel Xeon 5160 at 3 Ghz, 667Mhz FSB, with 4MB of L2 Cache. The version of BLAST used is the ncbi-blast-2.2.23 [26].

Scenarios: BLAST has been executed using three different workloads: a heavy workload (tagged *Slow*), using queries with many similarities with the database; a medium workload (tagged *Common*), decreasing the number of similar-

ities; 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 in the order of minutes.
- *Fast*: a sequence of 1,015,156 chars, taken from the *yeast* DNA database. This sequence has fast associated execution times of only a few seconds.

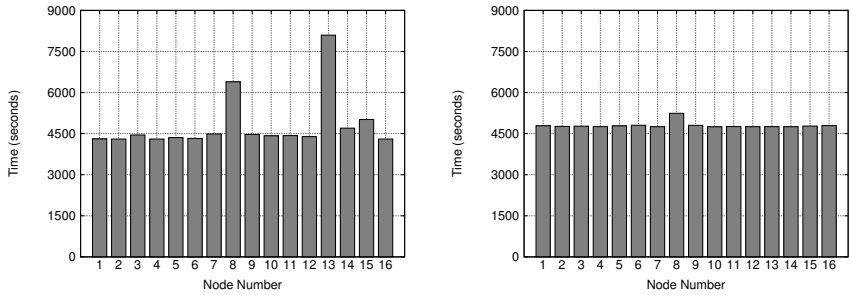
5.1.1. Selection of the scheduling policy

According to our methodology, the time spent on each fragment should be recorded because we need to use this information for scheduling data chunks for the following sequences of the query. Here, the partition factor was set to $N_f = 128$, 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.

Results obtained for 16 workers, $N_w = 16$ using the selected partition factor and applying the FCFS scheduling are shown in figure 6(a). Load imbalance can be clearly observed, and it is caused by a data chunk with large computation time that have been scheduled at the end of the execution. At the right side of the graphic (figure 6(b)) are presented the results of a similar execution of BLAST applying the *Heaviest Fragments First, (HFF)* scheduling policy. The advantage of using the *HFF* instead of FCFS is clear because HFF would enable reductions of up to 40% in total execution time.

5.1.2. Adjusting the Partition Factor

The objective of these experiments is to evaluate the performance of a real scientific application when tuning the workload partition factor. For the initial exploration, $N_f = 128$ and $N_w = 32$ were chosen, obtaining 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 BLAST computations over each data chunk are shown in figure 7(a). Figure 7(b) presents results of a new partitioned workload after gathering and dividing the data chunks. In this case,



(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 6: Performance of BLAST using Heaviest Fragment First (HFF) scheduling policy.

N_f is reduced from 128 to 64 and the total execution time is equal to $2,910.03$ seconds (a reduction in the total execution time of up to 55%).

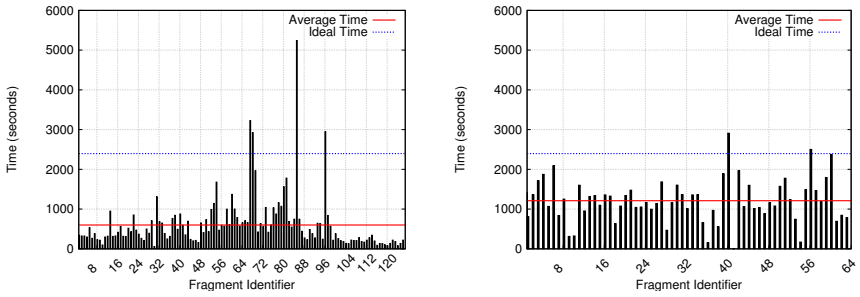
The behavior of the HFF scheduling policy was evaluated with and without adapting the size of the data chunks distributed to the BLAST execution (workload partition factor modification). The evaluation was performed ranging the number of workers, N_w, from 4 to 32. Obtained total execution times were compared with the expected ideal time for each case; results are showed in figure 8. The difference between tuning or not the size of the data chunks is clearly shown in this figure. When applying partitioning and grouping strategies, BLAST's total execution is greatly reduced.

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

5.1.3. Estimating the Number of Resources

To show the advantages of tuning the number of workers used by the application, BLAST was 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 was 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 for a Gigabit Ethernet network). The average sizes of the data chunks for the selected partition factors are shown in Table 2.

Figure 9(a) shows the evaluation of expression tq_i (10) for each value of N_w using the *Slow* scenario, figure 9(b) shows the real execution time of BLAST, and



(a) Initial processing time by data chunk (b) Processing time by tuned data chunks

Figure 7: Variation of processing time by data chunk.

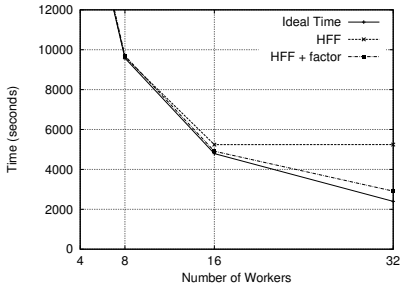
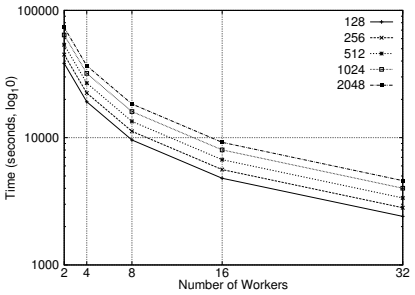


Figure 8: Performance improvement in total processing time when using *HFF* scheduling policy + workload partition factor adaptation (*HFF + factor*).

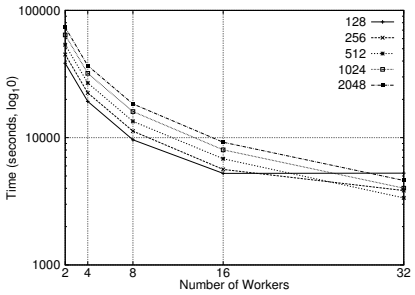
figure 9(c) shows the real execution time after tuning the partition factor. It can be observed that when the restrictions defined by expressions (8) and (9) are met, differences in total execution time are lower than a 5% (for most of the cases). Therefore, as expected, expression (10) can be used to estimate the execution time of the application. Results shown in the figures present the improvement that can be obtained from tuning the number of workers up to 87.5% when varying from 2 to 16 workers in all the cases. These results are sound because they are similar to those that will be shown in subsection 5.3.3 where the use of more than 16 workers do not reduce total execution time.

Table 2: Data chunks and chunks sizes

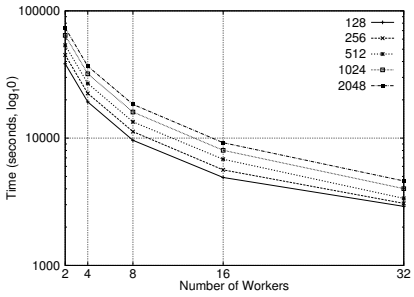
N_f	128	256	512	1024	2048
size [MB]	55.8	28.2	13.9	7.0	3.5



(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 9: Comparison between Expected and Real execution times.

The cases where the estimated execution time greatly differs from the real execution time can be explained through the constraints indicated by expressions (8) and (9). For example, in the case of 128 data chunks, the chunk with the highest processing time has an associated computation time of 5,245.61 seconds ($Tmax_{ij}$, in Table 3). Then, when the number of workers is changed from 16 to 32 there is no improvement in total execution time because of the data chunk with the highest execution time. Consequently, increasing the number or processing nodes will not reduce the total execution time.

For a partition factor $N_f = 128$, the result from evaluating the expression (8) is: $(76,598.19/5,245.61) = 15$. This value (the maximum number of workers that should be active) indicates that beyond 15 workers the total execution time

Table 3: Maximum number of workers (Nw_{max}) for Slow queries.

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**

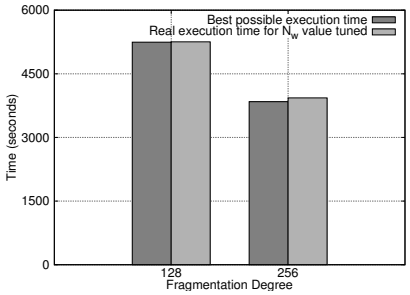


Figure 10: Execution for *Slow* query, using $N_w = 15$ for $N_f = 128$, and $N_w = 23$ for $N_f = 256$.

will remain the same. For the case of 256 data chunks, the maximum number of workers that can be used is 23 as shown in Table 3. Figure 10 illustrates this discussion by showing for $N_f = 128$ and $N_f = 256$ the best possible execution time and the real (measured) execution time.

In addition, the *Slow* scenario and a partition factor $N_f = 128$ were used to illustrate the use of expression (12) for tuning the number of workers. Figure 11 shows the performance index ρ_n for 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 scenarios, and therefore more workers could be added. However, if the number of workers is increased, there will be no gain in total execution time (because of the data chunk with the highest computation time) and more workers will remain idle for a longer period of time. This idleness is translated as an efficiency loss.

Once the workload partition factor is adjusted the time limitations imposed by data chunks with large computation time are softened. In consequence, more processing nodes can be used to execute BLAST without losing efficiency. For 32 workers and a partition factor $N_f = 128$, obtained results have reported up to 50% of reduction in the overall execution time when applying the proposed

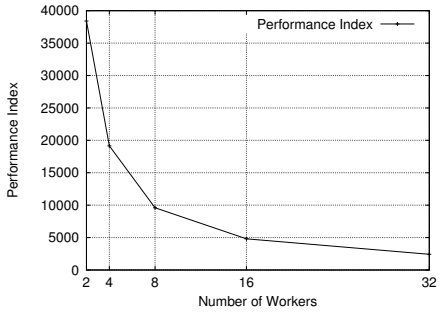


Figure 11: Performance index for *Slow* query and $N_f = 128$.

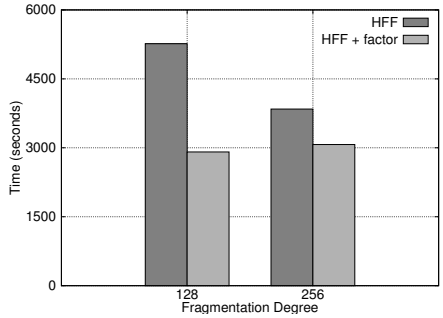


Figure 12: Execution with *Slow* query, for $N_w = 32$ using *HFF* and *HFF + factor*.

methodology (as shown in figure 12). However, for 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. This overhead appears from the serial fraction of the original data chunks. An implicit serial fraction, coming from BLAST's algorithm, is replicated when the computation times of the data chunks are summarized. As more data chunks are, the more overhead is introduced.

5.2. Evaluation of the methodology for a distributed merge sort

To evaluate the influence of the workload partition factor in the performance of the application we implemented a distributed sorting algorithm using merge sort. The implementation follows the assumptions made in Section 3 about data-intensive applications. This application can sort medium size input files (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 experimentation evaluates the behavior of the application when: (i) changing the initial distribution strategy (subsection 5.2.1); (ii) modifying the size of the data chunks (subsection 5.2.2); and (iii) adding more processing nodes (subsection 5.2.3). An input file of up to 32 GB was generated using the *gensort* program [25] and was divided into 128 data chunks. This number of pieces was selected because of the experimental environment described in section 5.1, which is large enough to guarantee that all workers will have data chunks to process. Additionally, a 25% of data chunks were already sorted when introduced in the workload. The combination of sorted and unsorted data chunks generates variability among their associated computation times.

5.2.1. Selection of the scheduling policy

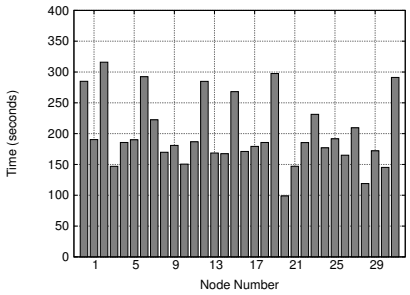
As a first step, the behavior of the application was analyzed when changing the scheduling policy. Data chunks were distributed using the *First Come First Serve* (FCFS) and the *Heaviest Fragments First* (HFF) approaches. For a number of worker $N_w = 32$ resulting total execution times are shown in figures 13(a) and 13(b). In figure 13(a) data chunks were distributed without any pre-defined order; while in figure 13(b), data chunks with highest execution times were delivered first.

From the reported results, it can be concluded that changing only the distribution policy will 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. A 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.

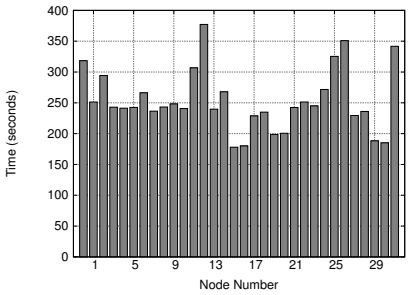
5.2.2. Adjusting the Partition Factor

The purpose of this experiment is to evaluate the performance of the application when adapting the size of the data chunks. This functionality was introduce with the aim of: (i) reduce the execution time of data chunks with high processing times; or (ii) reduce the number of data chunks with low computation times by sending less pieces of greater size. The experiment has been performed using the scenario described in the previous section. This has been done to allow comparisons between the results presented in figure 13 and figure 14.

Results presented in figure 14 show a more balanced execution in comparison with non-adapting the workload partition factor. This improvement is obtained



(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 13: Load imbalances with FCFS and HFF scheduling policies.

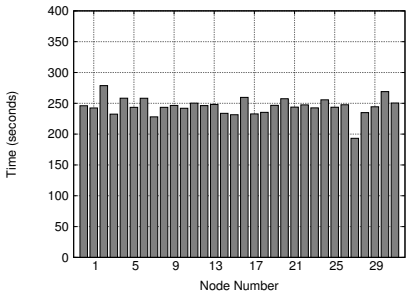


Figure 14: Execution for distributed merge sort, for $N_f = 128$ and $N_w = 32$ using *HFF + factor*.

after grouping and allocation data chunks of almost the same size for each worker. By doing this, the variability introduced by the final merge becomes constant and the workers are processing almost the same amount of time.

5.2.3. Estimating the Number of Resources

The intent of this experiment is to analyze the behavior of the distributed sorting algorithm when adding more processing nodes. This experiment was carried out using the initial workload described before and total execution time when changing the number of workers was collected. The value of N_w shown ranges from 16 to 58 workers to facilitate interpretation and the results are presented in figure 15.

Obtained results show a reduction in the total execution time when adding more processing nodes. Additionally, it can be ascertain a constant reduction in total execution times when applying all the stages of the proposed methodology:

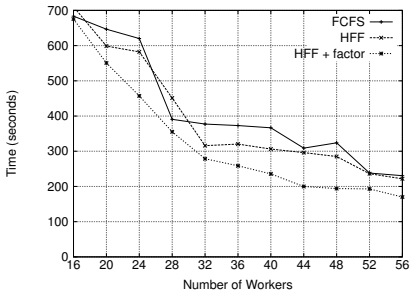


Figure 15: Comparison between *FCFS*, *HFF* and *HFF + factor*, when varying the number of processing nodes for $N_f = 128$.

(i) changing distribution policy, (ii) adapting the size of the data chunks, and (iii) adding more processing resources. Consequently, a fast and efficient execution is achieved.

5.3. Evaluation through Analytical Simulation

The analytical simulator has been implemented using the model described in Section 4.3. In subsection 5.3.1 is evaluated 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. Then, the effect of tuning the workload partition factor in the overall performance of the applications is described in subsection 5.3.2. Finally, in subsection 5.3.3 is shown how the methodology leads to a more efficient use of resources.

5.3.1. Introducing error in data chunks processing time predictions

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.

The purpose of these experiments is to 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 chunk. In order to do this, the simulation environment was set to consider a partition factor equal to 128 and a number of workers equal to 64.

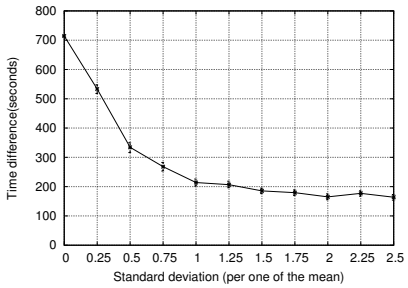


Figure 16: Total execution time differences between the *FCFS* and the *HFF* data scheduling policies when introducing errors.

The input data of the simulator has been generated following a normal distribution from an initial dataset obtained from real measurements. Then a certain degree of variation is introduced in the computation times of each data chunk. The greater this degree of variation, the greater the variability in the time associated to the data chunk.

The new dataset has been evaluated for two different scenarios: (i) FCFS: the data chunks are not sorted; and (ii) HFF: the data chunks are sorted by processing time in decreasing order. The simulation process was repeated 500 times and the results are the average values of execution time for both cases.

Introducing variability in the processing times of each data chunk tends to degrade the performance of the HFF strategy. The variability causes the chunk disorder and hence loads imbalances for the next exploration (query or iteration).

We have estimated the average and its corresponding 90% confidence intervals (where the probability to be inside is 90%). Confidence intervals cannot been obtained with traditional parametric methods (like the t-Student based one [27]) because results usually do not have a normal distribution. We have verified it with the normality test of Anderson-Darling [27] and we have used the nonparametric statistics to obtain the confidence intervals. To this aim we have chosen the Efron variant based on percentile from the Bootstrap methods [28].

Figure 16 shows the average of the differences between the executions times for both strategies. It can be seen that as the variability increases the performance of the HFF strategy collapses.

5.3.2. Evaluating performance when adjusting the partition factor

In order to show the performance improvement when changing the size of the data chunks, the simulator was used to evaluate the analytical expressions

presented in subsection 3.3. Simulations were performed under the following conditions:

- An initial partition factor $N_f = 128$.
- 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 were generated following a normal distribution based on measurements obtained from real executions of BLAST.

The results presented in figure 17 show the behavior of the simulator for the selected scenarios. It can be seen the difference between the maximum execution time $T_{max_{ij}}$ obtained with and without applying the tuning strategy for the workload partition factor. These results show that the execution time limitation imposed by data chunks with large computation times can be soften or diminished through the division of those data chunks. It is worth noticing that once this barrier has been removed, the total execution time of the application can be reduced by adding more workers.

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

As previously done, the variation in total execution time is analyzed. This variation is caused when a certain degree of error is introduced (changing the size of data chunks). The simulation environment was 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 was introduced. The greater this percentage of variation introduced in the computation time of each data chunk, the greater the variability obtained for each new data chunk. The generated dataset is evaluated for the scheduling policy HFF with and without adapting the workload partition factor. The simulation process was 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 18 we can observe that time degradation is significantly reduced when adapting the size of the data chunks. In general, when data chunks with large computation times are divided into smaller data chunks, their associated computation time is greatly reduced.

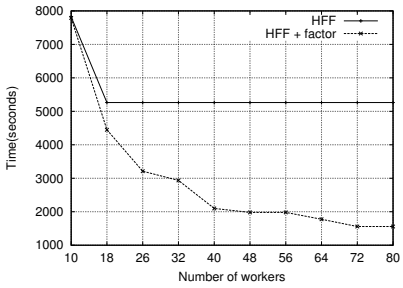


Figure 17: Performance improvement when changing data chunks sizes

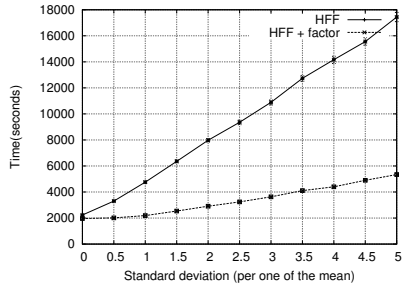
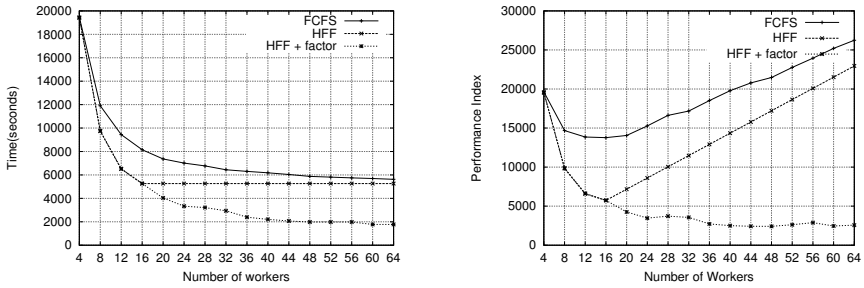


Figure 18: Performance improvement when introducing variability

5.3.3. Estimating the Number of Resources

In the previous subsections, three scheduling policies were analyzed when the variability of the execution time increases: (i) FCFS; (ii) HFF; and (iii) HFF + factor. In this section is analyzed the behavior of all strategies when varying the number of workers. Simulations were performed under the same conditions described in the previous section, and the results are shown in figure 19.

In figure 19(a) it can be seen the reduction in total execution time when applying the scheduling policies. From the comparison of all the strategies it can be observed that FCFS reports the worst performance results and HFF + factor reports the best results. Since data chunks with large computation times are not a major restriction for the HFF + factor strategy, it shows a more flexible behavior and scales well when adding more processing resources (as shown in figure 19(b)). On the contrary, the performance of FCFS and HFF scheduling strategies is quickly degraded by the time restriction imposed for data chunks with large computation time.



(a) Total execution time when varying the number of workers for the *FCFS*, *HFF* and *HFF + factor* scheduling policies

(b) Performance index when varying the number of workers for the *FCFS*, *HFF* and *HFF + factor* scheduling policies

Figure 19: Performance evaluation of the *FCFS*, *HFF* and *HFF + factor* data scheduling policies when varying the number of workers

Finally, if we compare the simulation results and experimental results are compared, it can be seen a consistent improvement in the performance of data-intensive applications given by: (i) changing the scheduling policy to HFF; (ii) adapting the size of the data chunks; and (iii) tuning the number of processing nodes that can be used. Also, the simulator enables the analysis of the methodology behavior for a larger range of situations. In the same way, when adjusting the number of workers, it is possible to identify the limit in which the addition of resources has to be stopped in order to keep an efficient execution.

6. Conclusions and Future Work

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 also has increased.

We have addressed the problem of load balancing through a methodology for balancing the load of a subset of data-intensive applications. In particular, we considered applications that perform multiple related explorations (queries) on the same workload. The methodology includes the dynamic adaptation of the partition factor or, for the case of high partitioning costs, the generation of mul-

tiple workload's divisions using different partition factors before executing the application, and then the dynamic selection of the most adequate one according to the current conditions. This phase of the methodology proposes to change, at run time, the size of the data chunks by dividing or gathering specific pieces according to application performance. This has been achieved by monitoring each exploration, and by using collected data to determine the corresponding modifications in the partition factor.

The tuning parameters included in the methodology are the workload partition factor, the distribution of generated data chunks among the application processes, and the number of resources (nodes) to be used by the application. This work introduces a detailed discussion about all the parameters.

The methodology has been tested using a real and a synthetic data-intensive application: the widely known bioinformatics tool BLAST and a distributed version of merge sort. BLAST handles a broader number of queries over large-scale biological databases and the Merge sort can order large text files. In addition, the main aspects of the proposal were implemented and evaluated through an analytical simulator. Also using simulation, it was possible 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; such as BLAST.

The improvements achieved were determined by three main reasons. First, by changing from a *First Come First Serve* (FCFS) scheduling policy, to a *Heaviest Fragments First approach* (HFF). Secondly, by adapting the partition factor of the workload at run time to reduce the time constraint imposed by data chunks with highest computation times. Third, by adapting the number of workers being used, to avoid inefficient executions in which there are idle workers 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 show that the proposed methodology can be widely use to improve performance in real data-intensive applications.

This work, as any work covering several aspects within a certain field of science gives rise to a wide range of affordable discussion lines and future work. One of the most interesting lines is the applicability and extension of the methodology to be used in virtualized environments, specifically those as Cloud. In MapReduce [29], jobs can be chained or may operate on multiple data sets, presenting a similar approach to the iterations in the studied applications. Consequently, these scenarios match our proposal.

Hadoop, the open source implementation of MapReduce, has several ways of

coordinating multiple jobs together, including sequential chaining and executing them according to predefined dependencies [30], which might be enriched by using dynamic tuning techniques such as the one we are proposing.

Additionally, implementing cost functions to ease the estimation of the number and characteristics of the processing nodes to be contracted. Since virtualized environments offer highly heterogeneous systems, and certain economic restrictions, we should take into consideration these factors before launching data-intensive applications. In general, platforms such as Hadoop and Amazon Elastic MapReduce [3] have shown an efficient and powerful system to create highly scalable applications with many aspects of parallel computing automatically provided. In this sense, our proposal could take advantage of the mechanisms for managing node failures, data and jobs allocation provided by current Cloud platforms to avoid load imbalances, and to use available resources efficiently; thus, extending the usability of our model. At this stage, heterogeneity and scalability issues have not been considered in our work.

Acknowledgment

This research has been supported by the MICINN Spain, under contract TIN2007-64974 and TIN2011-28689.

References

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

[2] R. E. Bryant, Data-Intensive Supercomputing: The Case for DISC, Tech. rep., Carnegie Mellon Univ. (May 2007).

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

[4] C. Oehmen, J. Nieplocha, ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis, IEEE Trans. Parallel Distrib. Syst. 17 (2006) 740–749.

[3] http://aws.amazon.com/elasticmapreduce/

[5] V. Bharadwaj, D. Ghose, T. G. Robertazzi, Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems, CLUSTER '03 6 (2003) 7–17.

[6] S. F. Hummel, E. Schonberg, L. E. Flynn, Factoring: a method for scheduling parallel loops, Commun. ACM 35 (8) (1992) 90–101. doi:10.1145/135226.135232.

[7] I. Banicescu, V. Velusamy, Load balancing highly irregular computations with the adaptive factoring, Parallel and Distributed Processing Symposium, International 2 (2002) 87–98. doi:10.1109/IPDPS.2002.1015661.

[8] A. Middleton, Hpcc systems: Data intensive supercomputing solutions, White paper, LexisNexis Risk Solutions (2011).

[9] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, Basic Local Alignment Search Tool, J Mol Biol 215 (October 1990) 403–410(8).

[10] S.-S. Boutammine, D. Millot, C. Parrot, An adaptive scheduling method for grid computing, in: Euro-Par 2006 Parallel Processing, Vol. 4128, Springer Berlin, Heidelberg, 2006, pp. 188–197.

[11] M. Drozdowski, P. Wolniewicz, Divisible load scheduling in systems with limited memory, Cluster Computing 6 (1) (2003) 19–29. doi:10.1023/A:1020910932147.

[12] S. Chuprat, S. Baruah, Scheduling divisible real-time loads on clusters with varying processor start times, in: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 15–24. doi:10.1109/RTCSA.2008.23.

[13] M. Othman, M. Abdullah, H. Ibrahim, 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 2007, ICCS '07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 446–453. doi:10.1007/978-3-540-72584-8_59.

[14] Y. Yang, H. Casanova, M. Drozdowski, M. Lawenda, A. Legrand, On the Complexity of Multi-Round Divisible Load Scheduling, Research Report RR-6096, INRIA (2007).

[15] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, Y. Yang, Scheduling divisible loads on star and tree networks: Results and open problems, IEEE Trans. Parallel Distrib. Syst. 16 (3) (2005) 207–218. doi:10.1109/TPDS.2005.35.

[16] J. Berlińska, M. Drozdowski, Scheduling divisible mapreduce computations, Journal of Parallel and Distributed Computing 71 (3) (2011) 450 – 459. doi:10.1016/j.jpdc.2010.12.004.

[17] E. César, A. Moreno, J. Sorribes, E. Luque, Modeling master/worker applications for automatic performance tuning, Parallel Comput. 32 (7) (2006) 568–589. doi:10.1016/j.parco.2006.06.005.

[18] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef, E. Luque, Dynamic Pipeline Mapping (DPM), in: Proceedings of the 14th international Euro-Par conference on Parallel Proc., Euro-Par '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 295–304. doi:http://dx.doi.org/10.1007/978-3-540-85451-7_32.

[19] A. Moreno, E. César, J. Sorribes, T. Margalef, E. Luque, Task distribution using factoring load balancing in master–worker applications, Inf. Process. Lett. 109 (16) (2009) 902–906. doi:10.1016/j.ipl.2009.04.014.

[20] T. Chiba, M. den Burger, T. Kielmann, S. Matsuoka, Dynamic load-balanced multicast for data-intensive applications on clouds, in: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 5–14. doi:10.1109/CCGRID.2010.63.

[21] L. Glimcher, V. Ravi, G. Agrawal, Supporting load balancing for distributed data-intensive applications, in: High Performance Computing (HiPC), 2009 International Conference on, 2009, pp. 235 –244. doi:10.1109/HIPC.2009.5433204.

[22] C. Rosas, A. Morajko, J. Jorba, E. César, Workload Balancing Methodology for Data-Intensive Applications with Divisible Load, in: SBAC-PAD '11, IEEE Computer Society, 2011, pp. 48–55. doi:http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2011.15.

[23] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences., J Mol Biol 147 (1) (1981) 195–197.

[24] A. E. Darling, L. Carey, W. Feng, The Design, Implementation, and Evaluation of mpiBLAST, in: Inproc. of ClusterWorld 2003, no. 0, 2003.

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

[26] NCBI, Blast homepage, http://blast.ncbi.nlm.nih.gov/ (March 2012).

[27] M. F. Triola, Elementary Statistics, Addison-Wesley Longman, Incorporated, 2006.

[28] A. C. Davinson, D. Hinkey, Bootstrap Methods and their Application, Cambridge Series in Statistical and Probabilistic Mathematics, 1997.

[29] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
URL http://doi.acm.org/10.1145/1327452.1327492

[30] C. Lam, Hadoop in action, Manning Publications Co., 2010.