Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 32 (2006) 568–589
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Modeling Master/Worker applications for automatic performance tuning ☆

E. Cesar *, A. Moreno, J. Sorribes, E. Luque

Autonomous University of Barcelona, Computer Architecture and Operating Systems, Edifici Q – Campus UAB, 08193 Cerdanyola, Barcelona, Spain

Received 10 January 2006; received in revised form 24 May 2006; accepted 6 June 2006
Available online 17 August 2006

Abstract

Parallel application development is a very difficult task for non-expert programmers, and therefore support tools are needed for all phases of this kind of application development cycle. This means that developing applications using predefined programming structures (frameworks/skeletons) should be easier than doing it from scratch. We propose to take advantage of the intrinsic knowledge that these programming structures provide about the application in order to develop a dynamic and automatic tuning tool. We show that using this knowledge the tool could efficiently make better tuning decisions. Specifically, we focus this work on the definition of the performance model associated to applications developed with the Master/Worker framework.

Keywords: Master/Worker performance model; Dynamic performance tuning; Load balancing

1. Introduction

This study is focused on the description of a methodology of parallel applications dynamic performance tuning based on the definition of optimization models associated to the application's structure, and in its application to the development of a performance model associated to Master/Worker.

Parallel programming constitutes a highly promising approach to improving the performance of many applications. However, in comparison to sequential programming, many new problems have arisen in all phases of the development cycle of these kinds of applications. One of the best ways to solve these problems would be to develop, as has been done in sequential programming, tools to support the analysis, design, coding, and tuning of parallel applications.

In the particular case of performance analysis and/or tuning, it is important to note that the best way of analyzing and tuning parallel applications depends on some of their behavioral characteristics. If the applica-

☆ Supported by the MEyC-Spain under contract TIN 2004-03388.
* Corresponding author. Tel.: +34 935812614; fax: +34 935812478.
E-mail address: eduardo.cesar@uab.es (E. Cesar).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2006.06.005

tion to be tuned behaves in a regular way, then a static analysis (predictive or trace based) would be enough to find the application's performance bottlenecks and to indicate what should be done to overcome them. However, if the application changes its behavior from execution to execution, or even dynamically changes its behavior in a single execution, then the static analysis cannot offer efficient solutions to avoid performance bottlenecks.

In this case, dynamic monitoring and tuning techniques should be used instead. However, in dynamic monitoring and tuning, decisions must be taken efficiently while minimizing intrusion on the application. This is very difficult to achieve if there is no information about the structure and behavior of the application, but if a programming tool, based on the use of skeletons or frameworks, has been used in the development of the application, then this information is available and a performance model associated to the structure of the application can be defined to be used by the tuning tool.

With the objective of developing these concepts, we have organized the rest of this paper in the following way: first, in Section 2, we present an overview of the related work in this field. Next, in Section 3 we describe the general tuning environment design. Then, in Section 4 we briefly describe the Master/Worker framework, its associated performance pitfalls, and the guidelines for its related performance model. After that, in Sections 5,5.1,5.2 we present the development of the Master/Worker performance model, as well as several experimental results that validate it. Finally, in Section 5.3 we set out the conclusions of this study.

2. Related work

There are many studies related to performance monitoring, analysis and tuning, which in turn could be broadly divided into those that propose a static (predictive or trace based) analysis, such as Kappa-Pi [10], EXPERT [30], or P^3T+ [11], and those that propose a dynamic one, such as Paradyn [18] and Dynamic Statistical Projection Pursuit (or PP) [29]. These tools are focused mainly on solving (or detecting) low level local problems, usually despising the performance information that is intrinsically provided by the knowledge about the structure of the application.

Nevertheless, the idea of using this information to optimize the application‘s performance has been developed by researchers whose main interest area has been the development of parallel programming tools, especially by those focused on developing tools based on skeletons or frameworks, such as P3L[27], llc [19], and eSkel [9].

P3L is a language that enables the development of parallel programs using paradigms, such as pipeline, farm (Master/Worker), loop, and so on. These paradigms are implemented by templates adapted to different architectures, which are selected at compilation time in accordance to the characteristics of the architecture and the profile of the application. Moreover, in a second optimization phase the chosen templates are tuned in order to minimize the use of the resources.

The llc group has developed analytical models for optimizing pipeline ([24] homogeneous and [1] heterogeneous) and Master/Worker ([28] homogeneous and [2] heterogeneous) applications. The proposed optimization process is static, and the main goal of these models is to find the processes to processors mapping that minimizes the application's execution time.

Finally, the eSkel group has based its approach to performance optimization on process algebras [5]. First, the application and the target architecture are modeled using the algebra, next, the model parameters are provided, and, finally, a workbench application is used to estimate its execution time and obtaining the best processes to processors mapping.

The main difference between our methodology and the ones mentioned before is that those are focused on static optimization while ours is designed for dynamic optimization. This means that, on the one hand, our model will be evaluated with the latest information about the application, while the aforementioned models are evaluated with information of previous executions of the application. Consequently, these models are more suitable for applications exhibiting a regular behavior from execution to execution than for those exhibiting an irregular behavior. On the other hand, our model must be evaluated at run time while those mentioned before are evaluated off-line. As a result, evaluation time is a minor problem for those models while it is of the utmost importance in our model.

However, the most noticeable consequence of the dynamic vs. static approach is that, whereas the final objective of minimizing the application's performance time is shared, the means to achieve this objective are completely different.

3. Environment design methodology

The main idea we have introduced is that it is possible to define a *performance model* associated to the most common *application frameworks* in order to simplify the *dynamic tuning* process. This model, represented in Fig. 1, includes a set of *performance functions* (aimed at the detection of performance drawbacks), but also suggests which parameters have to be measured (*measure points*) to evaluate this model and determine which parameters (*tuning points*) have to be modified to activate the actions that should be taken to overcome the detected performance drawbacks.

These definitions make up the static part of the environment. Then, there is a dynamic tuning environment (*Run-time*) which, at application execution time, uses this *performance model* to *monitor* the appropriate parameters to evaluate the performance functions (*performance analyzer*) and, based on the evaluation results, performs the required actions to improve the application's performance (*tuner*). This run-time phase have been called MATE (Monitoring, Analysis and Tuning Environment) [23], and it has been implemented as an independent tool.

To implement this development and tuning environment we have designed POETRIES [6] (Performance Oriented Environment for Transparent Resource-management, Implementing End-user parallel applications) an environment which integrates a framework based parallel programming environment with the performance model needed to carry out the dynamic analysis and tuning for these kinds of applications. The main goal of POETRIES is to offer programmers a set of tools to allow them to develop parallel applications concentrating solely on the computational problem to be solved. This goal poses two different, although related, problems. On the one hand, tools and structures (frameworks) to simplify application development have to be available; on the other hand, a methodology must be designed to enable dynamic and automatic performance tuning. The main issue of this methodology is to define the performance model associated with each framework.

Basically, any programming tool based on the application structure, such as CO_2P_3S [16] or eSkel [9], which fulfills the requirements of flexibility, usability and independence of the communication library could be used in our environment, the reason being that the performance model is not related to any specific implementation of the structure, but rather to the structure itself.

As we previously outlined, for the dynamic tuning tool to be efficient it needs to know the performance model associated to the structure of the application being tuned. In this sense, our main goal is to minimize

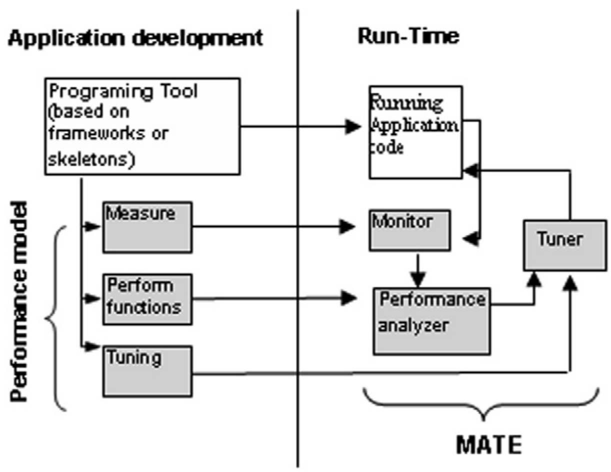

Fig. 1. Structure of the dynamic tuning environment POETRIES.

the execution time of the application, so the *performance functions* should describe the general behavior of the application as a function of the computing and communication times of the application processes and the structure of the framework used to implement it. Consequently, the general methodology used to define a performance model, useful for the dynamic tuning environment, consists of finding for each framework:

- A set of expressions (*performance functions*) that reflects the behavior of the application in terms of execution time and based on the structure of the framework used in its development. These expressions also enable the detection of performance drawbacks and the impact of dynamically introducing changes to the application.
- The parameters (*measure points*) that must be monitored at runtime to enable the evaluation of the performance functions. It is important to define where and when these parameters have to be measured.
- The actions that should be taken by the tuning tool, modifying some parameters (*tuning points*), to solve the application performance problems detected through the performance functions.

It is worth noting that the conditions that trigger an action by the tuning tool have to be persistent enough to be sure of the improvements that could be obtained from that action. This means that the tool will only react when the conditions that indicate a performance problem are continuously observed over a period of time. Solving transient problems does not worth the effort.

Moreover, in order to define these components, it is necessary to analyze the dynamic behavior of the applications developed with a certain framework. First, the framework related performance drawbacks must be detected with the objective of finding the *measure points*. Next, the strategies for overcoming these problems have to be defined in order to find the *tuning points*. Finally, the information given by the framework structure and behavior has to be used to define the performance functions that allow the detection of problems and the selection of appropriated tuning actions.

In the following sections we apply this methodology to the Master–Worker framework. First, we analyze its dynamic behavior, then we present a detailed description of the development of the performance functions, and, finally, we summarize the framework performance model.

4. Master–Worker framework analysis

The Master–Worker framework is a well-known parallel programming structure because it enables the expression, in a natural way, of the behavioral characteristics of a wide range of high-level parallel application patterns. Using the terminology defined in [17], we could mention, among others, *embarrassingly parallel*, *separable dependencies* or *geometric decomposition* patterns. Basically, this framework includes a Master process which sends tasks to a set of Worker processes, then each worker makes some kind of computation on the received tasks, a computation that generally requires a variable and unpredictable time, and finally it sends the results back to the Master.

Depending on the nature of the problem, the Master process might have to wait for an answer from all Workers before sending them new tasks, which means that we can divide the application execution into iterations. This is the case of *separable dependencies* or *geometric decomposition* patterns, because there are data dependencies among tasks. Otherwise, the Master just has to wait for the answer from each individual Worker before sending a new task to that Worker, as in the *embarrassingly parallel* pattern, because tasks are mutually independent.

In the first case (data dependencies among tasks), the performance of the application mainly depends on two factors; first, it is important to get a balanced computational load among workers, and secondly, it is important to decide on the appropriate number of workers. The order of these factors is important, because, as will be shown later, changing the number of workers makes little sense if the computational load has not been previously balanced. In the second case (independent tasks), the performance of the application only depends on the number of workers, because load balancing is dynamically and automatically obtained if there are enough tasks to distribute. Finally, in both cases the proper number of workers depends on the tasks' granularity, resource cost, and the relation between communication time and computation time.

Therefore, the Master/Worker performance model consists of a strategy for achieving a balanced computational load, a strategy for adapting the number of workers, and a linking expression that enables the combination of both.

5. Load balancing through data distribution

The execution time of a Master/Worker application with N workers and a set of tasks that can be sequentially processed in time T, can be roughly bounded by the expressions T/N (lower bound), and T (upper bound), though in both cases some communication time should be added. Getting an execution time closer to the lower bound mainly depends on good load balancing among workers, which in turn relies on a good data distribution policy.

We were aware of this problem from the very beginning of our research, but did not propose the first, mostly informal, solutions until [7] and [20], and finally we have recently studied it in depth [25]. Our main source of inspiration for coping with this problem was studies of the distribution of parallel loops, such as [15,12,3,4]. Many policies have been defined that aim to balance the computation load of a set of processors used to execute several instances of a parallel loop, which is a very similar problem to the one of task distribution for Master/Worker applications.

The general solution to the task distribution problem could be stated the following way:

Instead of distributing the whole set of tasks among workers and waiting for the results (with no control over load balancing), the master will make a partial distribution by dividing this set of tasks into different portions called batches. The number of tasks assigned to each batch depends on the distribution strategy, and it may be different from one batch to another. The idea is to distribute the first of these batches among workers in chunks of (roughly) the same number of tasks, then when a worker ends the processing of its assigned chunk the master will send it a new chunk from the next batch, the process continues until all batches have been completely distributed. This way, workers that have received tough tasks will not receive more load and workers that have received lighter tasks will be employed to do more work.

Different strategies can be used to determine the batch sizes with the objective of getting better load balancing with little computation and communication overheads. We have adapted and analyzed three different strategies that will be looked at in depth later in this work:

- *Fixed Size Chunking* (FSC), which consists of dividing the set of tasks into some number of equal sized batches (with the only possible exception of the last one). In this case, we must try to find the best number of batches to improve load balancing.
- *Dynamic Predictive Factoring* (DPF), which consists of building the first batch with some portion of the task set, the second with the same portion of the remaining tasks, and so on until some lower bound for the batch cardinality is reached. In this case, we must try to find the best factor for determining the portion of the remaining task set that will be distributed in each batch.
- *Dynamic Adjusting Factoring* (DAF), which is like DPF but with a variable factor that is recomputed from batch to batch in accordance with the current load balancing conditions.

5.1. Fixed Size Chunking (FSC)

This is the simplest strategy designed to improve the load balancing of the application. It consists of dividing the task set into a series of batches of the same size, and then the tasks of the first batch are distributed among workers in equal sized chunks. When a worker finishes the processing of its assigned chunk the master sends a new chunk from another batch until all batches have been distributed. We proposed this adaptation of the Fixed Size Chunking policy [15] to the M/W task distribution problem in [7]. The number of batches to be distributed depends on a partition factor, which is the proportion of the whole set of tasks to be included in each batch, for example, a factor of 0.2 means that the set of tasks will be divided into five batches, each of which includes 20% of tasks. This means that low partition factors will produce more batches with fewer tasks whereas high partition factors will produce fewer batches with more tasks.

As a consequence, a low partition factor leads to a finer grained distribution of tasks, but also to a higher communication overhead, since we are producing more messages with fewer tasks, while a higher partition factor leads to a coarser grained distribution of tasks, but also to a lower communication overhead. Therefore, it is better to choose higher partition factors in order to minimize communication overhead, but if the standard deviation of the task processing time is big enough, a lower factor must be used in order to minimize load unbalancing. In any case, one of the main drawbacks of this policy is its lack of sensitiveness to load unbalances in the last batches for relatively high partition factors.

We can try to estimate the best partition factor at execution time from the history of each worker, assuming that, in the near future, their behavior will not substantially change. The idea is to find a good partition factor by calculating the mean time invested by each worker to process the last x tasks that it has received, and then estimate what will happen to the next set of tasks by simulating for different values of the partition factor.

Let us specify in more detail how this estimation is accomplished:

1. For the first iteration of the application a predefined partition factor of 0.25 is used. With such a low partition factor we are being pessimistic about the task processing time standard deviation, but not too much in order to avoid a high communication overhead. During this iteration the number of chunks processed by each processor and the time spent on this processing are stored for use in the calculation of new partition factors.
2. For the second iteration we use the same partition factor of 0.25, and we also store the per processor number of processed chunks and processing time. However, at the same time we use the stored data to simulate what would had happened if other partition factors had been used. The algorithm of this simulation is the following:
   (a) Choose a test partition factor of 0.1.
   (b) Calculate the batches and chunks that will result from the application of this partition factor. This is possible because we know the number of tasks being processed and the number of workers in our application.
   (c) Using the historical data about the processing time spent by each processor on the chunks it has received, calculate the processor mean processing time per task.
   (d) With the calculated partition and mean time and using the expressions developed for the estimation of the execution time of a Master/Worker application, which will be discussed in the second part of this chapter, we can estimate the execution time of a whole iteration for this partition factor (considering communication).
   (e) If this estimated execution time is the best one so far, then the tested partition factor becomes the new proposed partition factor.
   (f) If the test partition factor is less than 1.0, increase it by 0.1 and go to (b), otherwise finish.
3. For the third and subsequent iterations we use the partition factor proposed by the simulation process and, in addition, we keep on storing processing data and repeating simulations to adapt the partition factor to possible variations of the processing times associated to the tasks.

5.2. Dynamic Predictive Factoring (DPF)

We have observed that the previously described FSC policy lacks the necessary sensitiveness to deal with load unbalances introduced by the last distributed chunks. This drawback appears because all chunks include the same number of tasks and because we prefer to have the biggest possible chunks to minimize both the communication overhead and load unbalance. With the objective of overcoming this problem, an adaptation of the Factoring policy [12], which is a partition policy based on assigning big chunks at the beginning of the iteration and small ones at the end, was proposed in [20] named the Dynamic Predictive Factoring (DPF) distribution policy.

The general idea of this policy consists of dividing the set of tasks into batches of decreasing size, then, as in the case of the FSC policy, the tasks of the first batch are distributed among workers in equal sized chunks, and when a worker finishes the processing of its assigned chunk the master sends a new chunk from another batch, until all batches have been processed. Again, the number of batches to be distributed depends on a partition factor, but for the DPF policy this factor is used for the whole set of tasks to calculate the number of tasks in the first batch, then for the tasks left out (those not included in the first batch) to calculate the number of tasks included in the second batch, and so on for the remaining batches, until some predefined lower limit for the per chunk number of tasks is reached.

Intuitively, with this policy we are assuming that sending bigger chunks at the beginning of an iteration is not only unlikely to make any worker go beyond the T/N target time but is also useful for minimizing the communication overhead. However, as we get closer to the end of the iteration it is more probable that processing a big chunk causes a significant deviation on the overall execution time. Consequently, as the iteration advances, the chunk size should be progressively reduced to minimize this effect, though increasing the communication overhead.

Although the DPF policy is more sensitive to load unbalances introduced by the last chunks, choosing the appropriate partition factor will determine the degree of success of the method. Choosing a high partition factor could lead to load unbalances if the processing time standard deviation is also high. On the contrary, being too conservative by choosing a low partition factor could produce an unnecessary increase in the communication overhead.

As for the FSC policy, we can try to estimate the best partition factor by a simulation process based on the history of the execution of each worker, assuming that, for the next few succeeding iterations, they will behave steadily. The estimation process is similar to the one described for the FSC policy, only changing the way batches are defined.

5.3. Dynamic Adjusting Factoring (DAF)

For the policies analyzed so far, we are assuming that the application conditions that were used to estimate the partition factor are already present when it goes into use two iterations later. This means that for applications with significant variations from iteration to iteration, or even for applications with some "different" iterations, we could get results that are not so good from these distribution policies. Moreover, even for applications with steady behavior, we will see that the distribution policies present their worse results for higher values of the standard deviation of the task processing time. This is because for the estimation of the partition factor, only the mean processing time of tasks is used.

These were the reasons that motivated us to adapt the Dynamic Adjusting Factoring policy [4], which is a partition policy that dynamically tries to adapt the partition factor to the current conditions of the application, to the task distribution problem [24]. The original Factoring policy (intended for parallel loop scheduling) tries to assign to processors the biggest possible chunks of parallel loop iterations that minimize the probability of exceeding the expected optimal execution time (T/N). This can be easily adapted to Master/Worker applications by substituting parallel loop iterations by tasks.

This statement can be formalized mathematically in the following way:

Suppose that we have N available workers (processors) for executing M tasks ($M >> N$), each one modeled as an identical independent random variable with mean μ and standard deviation σ. Assuming that all workers are initially idle, we can model the execution of a batch of N chunks as an Nth order statistic, which is the maximum of N identically distributed random variables.

In general, an upper bound for the expected value of an Nth order statistic is defined by the expression $\mu + \sigma\sqrt{N/2}$. Consequently, supposing that the number of tasks included in each chunk of the first batch is F_0, we will have an execution mean time of μF_0 and a standard deviation of σF_0, and an upper bound for its processing of $\mu F_0 + \sigma F_0\sqrt{N/2}$.

As our target is not to exceed the optimal execution time, which can be expressed as $\mu(M/N)$, we can say that fulfilling the expression $\mu F_0 + \sigma F_0\sqrt{N/2} = \mu(M/N)$ should be the goal of our policy. To do so, we need to compute F_0, which is the portion of the set of tasks to be distributed in the first batch divided by the number of processors, thus it can be expressed as $M/(x_0 N)$, where x_0 is the inverse of the partition factor used by the policy to generate the first batch to be distributed.

Finally, we solve this expression for x_0, getting

$$x_0 = 1 + (\sigma\sqrt{N/2})/\mu \tag{1}$$

However, for the succeeding batches we cannot assume that all workers are idle, because when any worker finishes the processing of its first chunk the master is going to assign it a new chunk from another batch, independently of the state of the remaining workers.

Consequently, to calculate the number of tasks to be included in each chunk of batch j, a more conservative target than the optimal expected processing time of the currently remaining tasks (R_j) is proposed in order to compensate for the differences in the processing starting times of these chunks. The restriction on the target consists of trying to match the optimal expected processing time for the set of remaining tasks, but excluding the tasks that will be included in the current batch. Therefore, the resulting goal expression is

$$\mu F_j + \sigma F_j\sqrt{N/2} = \mu[(R_j/N) - F_j]$$

.

Obviously, this time we must solve this expression for x_j, which will be the inverse of the partition factor used to calculate the number of tasks of batch j, getting

$$x_j = 2 + (\sigma\sqrt{N/2})/\mu \quad (2)$$

Now, using expressions (1) and (2), we can define a new distribution algorithm, which adapts the partition factor to the application's current executing conditions:

1. With the goal of accumulating enough information to compute the adaptative factor, the first iteration will be executed using the DPF partition method with a fixed factor of 0.5. This initial factor is empirically chosen because it has generally behaved quite well.
2. At the beginning of the remaining iterations calculate x_0, using the information accumulated in the past and expression (1). Then, using the same data and expression (2), calculate x_1. Prepare chunks of batches 0 and 1 for being distributed among workers.
3. When the number of available chunks falls below a predefined threshold, which we have fixed at half the number of workers ($N/2$), use expression (2) to compute x_j (from $j = 2$), then calculate the number of tasks for batch j, and prepare the new chunks to be distributed among workers.
4. If the number of tasks per chunk reaches some predefined lower limit, the remaining tasks are distributed in the last N chunks, and the distribution process ends.

Finally, we have mentioned that for DPF and DAF policies the partition method will not be applied if the number of remaining tasks is about to fall bellow the number of workers. However, we are going to face performance inefficiencies long before reaching this limit because the defined policies do not always directly consider some relevant application features, such as communication cost and the Master's chunk managing capability. Communication cost is considered both by the FSC and DPF policies, because the estimation process uses performance expressions that include this cost, but not by the DAF policy, however, considering the Master's chunk managing capability will strongly minimize this problem.

We define the *Master's Chunk Managing Capability* (MCMC) as the maximum number of chunks that the Master can manage from the moment it sends a chunk to a worker until the moment it receives the answer for the same chunk from that worker. Task managing includes message-handling times and eventually, some computation made by the Master on the tasks. It is, ultimately, a communication capacity problem, which mainly depends on the number of workers, and the chunk size and mean processing time, but these depend on the number of tasks included in each chunk. We will return to this problem in a lot more detail when we discuss the estimation of the number of workers, but now we can say that if the number of tasks per chunk causes the Master to fall below its chunk managing capability it won't be able to feed all workers and, as a result, there will be some idle workers.

5.4. Policy comparison through experimentation

In the previous sections, we presented three possible task distribution policies aimed at solving the load unbalance performance problem for Master/Worker applications and we have discussed their theoretical pros and contras. Now, our goal is to provide stronger evidence about their performance through the use of extensive experimentation.

Before getting into the discussion of results, we are going to briefly describe the platform that has been used to execute all these experiments, the tools developed to generate the appropriate synthetic programs, as well as the set of experiments we are presenting in this section. We have executed our experiments on one of the clusters of the Computer Science Department of Wisconsin University in Madison. It is a 150 dual 933 MHz nodes connected to a 100 Mbit switch, which has a gig-uplink to the core of the network (six clusters). On this platform, we have developed a set of configurable programs to test our models. These programs have been developed in C plus MPI, and the ones that are used specifically to test the task distribution policies accept the following parameters: distribution policy (none, FSC, DPF, or DAF), task size (number of bytes sent and received to and from the worker), network parameters (message overhead and communication speed per byte), and processing time matrix (generated by a statistical tool in accordance with a given mean and standard deviation).

We have executed each experiment using the three described policies and also distributing the whole set of tasks from the beginning, i.e., without any distribution policy. Each execution has been made with 10,000 tasks and for 15 iterations (Figs. 2–4).

We can see that, in general, we have achieved the expected results: the Dynamic Adjusting Factoring policy (DAF) leads to the best results and is less affected by a high standard deviation, the factoring policy with Dynamic Predictive Factoring (DPF) is quite good for a small or medium standard deviation only, and the

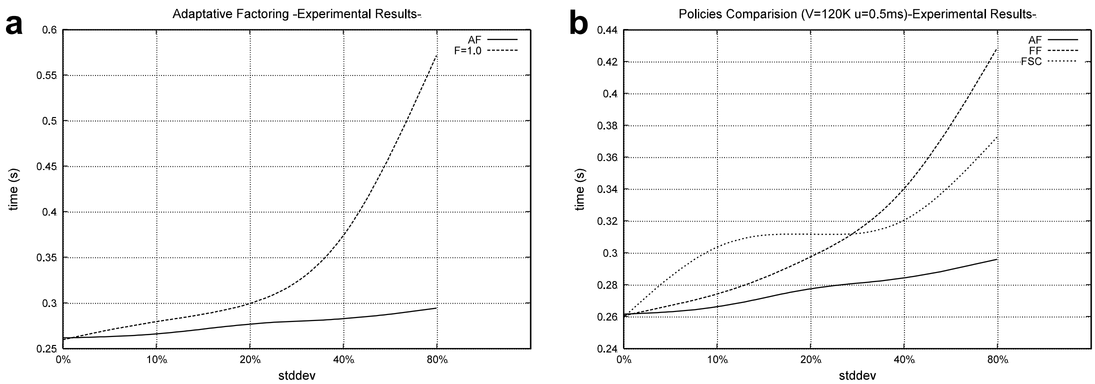

Fig. 2. Execution of a 25 worker application with an associated mean processing time of 0.5 ms per task, a communication volume of 240 Kbytes, and for a standard deviation ranging from 0% to 80%. Distribution policy used (a) DAF vs. none, (b) FCS, DPF, and DAF.

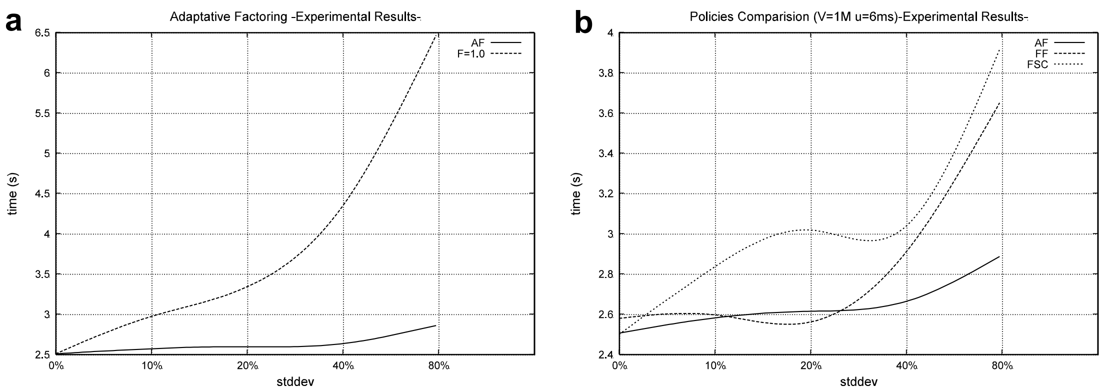

Fig. 3. Execution of a 25 worker application with an associated mean processing time of 6 ms per task, a communication volume of 1 Mbytes, and for a standard deviation ranging from 0% to 80%. Distribution policy used (a) DAF vs. none, (b) FCS, DPF, and DAF.

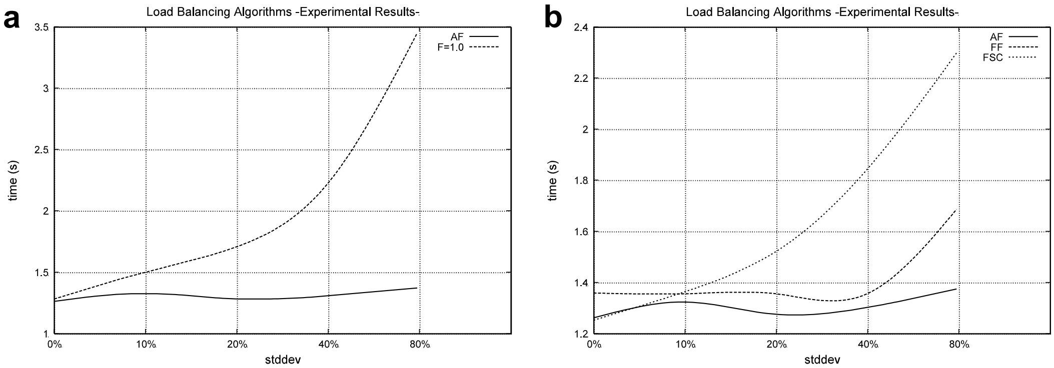

Fig. 4. Execution of a 50 worker application with an associated mean processing time of 6 ms per task, a communication volume of 240 Kbytes, and for a standard deviation ranging from 0% to 80%. Distribution policy used (a) DAF vs. none, (b) FCS, DPF, and DAF.

Fixed Size Chunking policy (FSC) is, usually by far, the worst one. As a result, we adopt the DAF as the data distribution policy for our performance model.

6. Adapting the number of workers

As we mentioned in the previous section, in an ideal Master/Worker application the total execution time would be equal to the sequential execution time divided by the number of workers, but this is assuming that there is no communication cost that the application is executing on a dedicated and homogeneous platform that we have achieved a perfect load balancing, and that the computation also scales ideally. In this ideal world, any available resource that can be assigned to the application must be assigned, because it will be efficiently used to improve the application's performance.

In the real world however, it can be observed that the speedup of the application usually decreases as new resources are assigned to it, indicating a loss in efficiency. Moreover, at some point, assigning more resources to the application will produce drops in performance because the costs introduced are greater than the advantages brought about by the new resources.

Consequently, we must take all these parameters into consideration in order to be able to decide how many resources must be used to optimize the application's performance ensuring, at the same time, an efficient use of these resources. With this objective, we have developed an analytical model, based on the behavior of a Master/Worker application.

We presented a first version of this analytical model in [8], and successive extensions to it were presented in [6,7]. Finally in [21,22] we presented a couple of implementations of a tuning tool that uses part of the model to dynamically improve the performance of a real application. This model is employed to evaluate the behavior of the application when it is executing and decide if it would be worthwhile to change the number of workers in order to improve its performance. With the objective of defining a useful model for making the best possible predictions, we have tried to take into consideration all the relevant parameters, but at the same time, we have tried to keep the model as simple as possible.

The only assumptions we have made when defining the model are that there is only one worker executing in each processor, and that the application is balanced. The former can be justified for reasons of efficiency because having several workers in the same processor is only useful in terms of performance if they can get blocked in I/O operations. On the other hand, we have previously shown that it is possible to get a pretty good balance of the application load by using task distribution methods; thus we can simplify the complexity and increase the efficiency of the model by assuming that there has been a load balancing stage before deciding on the number of workers. In addition, although we do not make any specific assumption about the hardware platform used for the execution of the application, we have not dealt with heterogeneous platforms and, consequently, the performance model makes more sense on homogeneous ones. The detailed definition and discussion of the model, and then the results of the extense experimentation we have done are shown and commented in the following subsections.

6.1. Expressions for modeling a balanced Master/Worker

Before defining the expressions that describe the behavior of a Master/Worker application we have to indicate the parameters that we have taken into consideration and the terminology that will be used from now on. In the first place, we have characterized the interconnection network with the classical message start up time plus communication time formula, which is quite simple; although [13] claims that it is not very accurate because it could lead to many pitfalls, such as ignoring the contention with jobs not related to the application, or ignoring the synchronization component of the communication. Nevertheless, we use this expression because, since our model will be used in a dynamic tuning environment, we will be able to monitor the network conditions and adapt the expression parameters accordingly.

Secondly, in order to be able to evaluate the model expressions, we need to know the time each worker is making useful computations, the time the master invests in building new sets of data, the amount of data sent and received to/from each worker, and the communication protocol (synchronous or not).

Finally, we will use the following terminology to identify the different parameters that are part of our performance model:

- m_o: per message start up time, in ms.
- λ: per byte communication cost (inverse bandwidth), in ms/byte.
- $v_i^{m/w}$: size sent/received to/from worker i, in bytes.
- V: total communication volume, in bytes.
- n: current number of workers of the application.
- α: portion of V sent to the workers, $\alpha V = \sum_{i=0}^{n-1} v_i^m$ and $(1-\alpha)V = \sum_{i=0}^{n-1} v_i^w$.
- μ_i: processing time worker i has spent in processing its assigned tasks, in ms.
- μ_m: processing time spent by the master on the preparation of a new set of tasks.
- T_c: total processing time o workers $(\sum_{i=0}^{n-1} \mu_i)$, in ms.

From now on, the idea is to define, in the first place, a set of general performance expressions that clearly reflect the framework functional behavior, and then to derivate a more specific expression set that must be useful for improving the application's performance.

In Fig. 5 we show a graphical representation of the execution trace of an iteration of a Master/Worker application; there we can see that the iteration begins with the master distributing tasks among workers, this step implies a communication phase between the master and each worker; then each worker goes on processing the set of tasks it has received; next, each worker sends the results of its calculations back to the master, implying another communication phase between each worker and the master; finally, the master could spend some processing time on generating a new set of tasks for another iteration. How long the first communication phase (data distribution) lasts depends on the interconnection network parameters, the amount of data to be sent, the number of workers, and the communication protocol; consequently, we can define it as function $S(\lambda, m_o, \alpha V, n)$. The duration of the worker computation phase depends on the total processing time and the number of workers; then it can be defined as a function $C(T_c, n)$. Finally, the amount of time

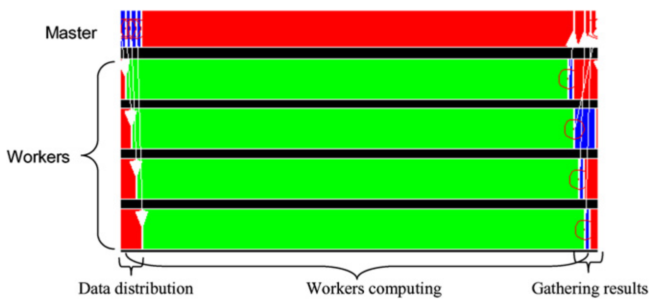

Fig. 5. Typical execution trace of an iteration of a Master/Worker application.

the second communication phase (results gathering) takes depends again on the network parameters, the amount of data being transferred, and the number of workers; thus it can be defined as a function $R(\lambda, m_o, (1-\alpha)V, n)$.

It should be noted that, when defining these functions, we must be aware of the existing overlapping times among them. In particular, we could graphically see in Fig. 5 that there is a time overlap between the data distribution phase and the calculation phase, and another one between the calculation phase and the results gathering phase. Based on this analysis, we are able to define a general performance expression for the execution time of an iteration of a Master/Worker application (T_t) as:

$$T_t = S(\lambda, m_o, \alpha V, n) + C(T_c, n) + R(\lambda, m_o, (1-\alpha)V, n) + \mu_m \tag{3}$$

As of now, we want to specify these functions for different execution. In the first place, we will develop the data distribution function (S) considering the communication protocol. As we mentioned before, we can use the $m_o + \lambda v_i^m$ expression to model the transference of v_i^m bytes from the master to worker i.

However, we want to calculate the time needed to perform the whole data distribution, which is not always the sum of the times spent sending messages to each worker. It depends on the communication protocol (synchronous or asynchronous) that is in use when the communication takes place. Consequently, we have the following cases:

- If a synchronous communication protocol is being used then the master has to finish each transference (the data is received by the worker) before starting the next one. Thus resulting in:

$$S(\lambda, m_o, \alpha V, n) = \sum_{i=0}^{n-1} (m_o + \lambda v_i^m)$$

- If an asynchronous communication protocol is being used then when the master makes a send operation the data is stored in some intermediate buffer and the next transference can begin before the previous one has ended. Consequently, as sending operations overlap in time, only the greater of the two components of the communication expression has to be added for each worker. Hence the resulting expressions are:

$$S(\lambda, m_o, \alpha V, n) = n m_o + \lambda v_{n-1}^m \quad \text{if } m_o \geqslant \lambda v_i^m \{\forall i | 0 \leqslant i \leqslant n-1\}; \quad \text{or}$$
$$S(\lambda, m_o, \alpha V, n) = m_o + \sum_{i=0}^{n-1} \lambda v_i^m \quad \text{if not}$$

Secondly, we will define the computation function (C) considering that there is an overlapping of time among the worker processing phase and the distribution phase. Specifically, when the last chunk of tasks sent by the master has been received by the destination worker, an event that marks the end of the communication phase, every other worker has already started processing its tasks. As we have assumed that we have a balanced execution, which means that the mean processing times of all workers will be very similar, we can assume that the last worker to start will also be the last one to finish; therefore, we can define function $C(T_c, n)$ as μ_{n-1}.

Finally, we will define the results gathering function (R) considering the overlapping of time with the computation phase. There is time overlapping with the computation phase because when the last worker to end its processing starts sending results back to the master, an event that marks the end of the computation phase, every other worker has already sent back its results. Consequently, it seems that we only have to be aware of this last data transference in order to define function R, but there is a final consideration to be made, which is that there cannot be overlapping of time between the data distribution and results gathering phases. This time overlap is possible if a worker sends its answer back before the master has ended the data distribution, in this case, there will be a delay in the reception of results by the master and the iteration will last longer. The problem is the same as that introduced when explaining the Dynamic Adjusting Factoring (DAF) data distribution policy, which led us to the definition of the *Master's Chunk Managing Capability* (MCMC) concept (see Section 5.3). Summarizing, function $R(\lambda, m_o, (1-\alpha)V, n)$ can be defined as $m_o + \lambda v_{n-1}^w$, provided that the application is below the MCMC.

The analysis of the execution conditions of the application leads to the derivation of the following set of expressions from expression (3):

	Communication protocol	
	Asynchronous	Synchronous
$m_o \geqslant \lambda v_i^m$	$T_t = (n+1)m_o + \lambda v_{n-1}^m + \mu_{n-1} + \lambda v_{n-1}^w + \mu_m \quad (4)$	$T_t = \sum_{i=0}^{n-1}(m_o + \lambda v_i^m) + \mu_{n-1} + m_o + \lambda v_{n-1}^w + \mu_m \quad (6)$
$m_o \leqslant \lambda v_i^m$	$T_t = 2m_o + \lambda \sum_{i=0}^{n-1} v_i^m + \mu_{n-1} + \lambda v_{n-1}^w + \mu_m \quad (5)$	

Next, we analyze expressions (3)–(5) with the objective of finding a simpler set of expressions more suitable to be used by a dynamic tuning tool. First, as we have assumed that the application is balanced, we can say that the overall processing time and data will be fairly distributed among all workers. Therefore, we can define v_i^m as $\alpha V/n$, v_i^w as $(1-\alpha)V/n$, and μ_i as T_c/n. Clearly, it will be easier to store and use only the overall amount of data transferred between the master and the workers (V) and the overall processing time of the workers (T_c), than the specific values of each worker. Secondly, it is necessary to define the performance expressions in terms of the number of workers to fulfill the objective of being able to predict the performance of the application for a different number of workers. Applying these substitutions to expressions (4)–(6) results in the following set of expressions to describe the performance of a Master/Worker application:

	Communication protocol	
	Asynchronous	Synchronous
$m_o \geqslant \lambda v_i^m$	$T_t = (n+1)m_o + \frac{(T_c + \lambda V)}{n} + \mu_m \quad (7)$	$T_t = (n+1)m_o + \frac{[((n-1)\alpha + 1)\lambda V + T_c]}{n} + \mu_m \quad (9)$
$m_o \leqslant \lambda v_i^m$	$T_t = 2m_o + \frac{[((n-1)\alpha + 1)\lambda V + T_c]}{n} + \mu_m \quad (8)$	

Finally, it is now possible to introduce a more formal definition of the concept of the *Master's Chunk Managing Capability*. We have said (Section 5.3) that it is the maximum number of data chunks the Master can manage from the moment it sends a chunk to a worker to the moment it receives the answer for the same chunk from that worker. If this capability is persistently exceeded then there will be workers answering quicker than the Master's capability to send data to all of them and there can be two different consequences: if the Master prioritizes receives over sends then some workers will not receive data and will be idle; otherwise, if the Master prioritizes sends over receives then workers will be idle waiting for the Master to end the data distribution.

In order to calculate the *Master's Chunk Managing Capability* (MCMC) and, consequently, be able to detect when it is being exceeded, we must be able to know how many chunks can be sent by the Master from the moment it sends a chunk to a worker to the moment the answer of that worker for the same chunk is available. The expression to calculate this value can be obtained by equating the time needed by the Master to distribute tasks with the time needed by a Worker to get a chunk, process the received tasks, and send results back to the Master. Considering different communication conditions, and solving the expressions for the number of workers, we get the following equations for calculating the MCMC exceeding point:

	Communication protocol	
	Asynchronous	Synchronous
$m_o \geqslant \lambda v_i^m$	$\left\lfloor \frac{\sqrt{m_o^2 + m_o((1-\alpha)\lambda V + T_c)}}{m_o} + 1 \right\rfloor$ (10)	$\left\lfloor \frac{(2m_o - \lambda\alpha V) + \sqrt{(\lambda\alpha V - 2m_o)^2 + 4m_o(\lambda V + T_c)}}{2m_o} \right\rfloor$ (12)
$m_o \leqslant \lambda v_i^m$	$\left\lfloor \frac{\lambda V + T_c}{\lambda\alpha V - m_o} \right\rfloor$ (11)	

6.2. Efficiency indexes

From expressions (7)–(9) we can find, through differentiation, new expressions to calculate the number of workers that would lead to the lowest execution time. However, this optimization is not taking into consideration the efficiency in the use of the resources. Nonetheless, it can be seen intuitively in the example shown in Fig. 6 that adding more workers to the application when the execution time is close to its minimum is, from the point of view of the use of the resources, highly inefficient. In this figure, we show the expected execution times (using expression (7)) of an application with a communication volume (V) of 4 Kbytes and an overall processing time (T_c) of 1.6 s, for a number of workers ranging from 10 to 60. A simple calculation shows that going from 15 to 20 workers means a gain of 17.68% in the execution time with an increase of 33% in the number of resources, while going from 30 to 40 (lowest execution time) workers means a gain of 3.99% for the same relative increase in the number of resources. Since we are increasing the number of resources by the same proportion, the gain lost in the execution time must be the result of a drop in efficiency.

Consequently, the definition of a performance index, which takes into consideration both the performance gain and efficiency in the use of resources, is a requirement that must be fulfilled in order to complete the definition of our Master/Worker performance model.

With the objective of making the best decision when changing the number of workers, several efficiency indexes could be defined, from simple ones, such as fixing a lower bound for the relation between the observed speedup and the ideal one, to more complex ones, such as relating the speedup or execution time changing rate with the amount of resources needed to achieve it. However, users must indicate a threshold for such indexes, which implies that they should know exactly what any value of the index means. It usually demands a high degree of knowledge about the application and its execution platform. On the other hand, it is possible to define a performance index that directly relates performance with efficiency in the use of resources, like the

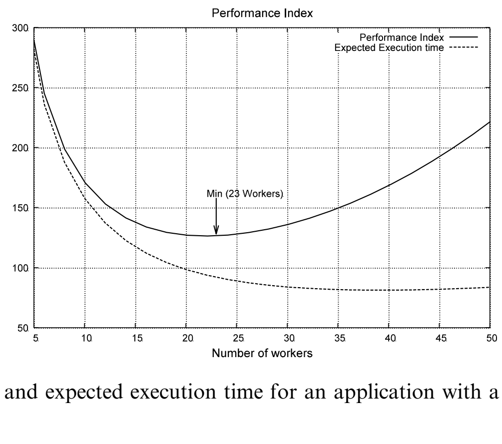

Fig. 6. Performance Index (13) and expected execution time for an application with a T_c of 1.6 s and a V of 4 Kbytes.

one defined in [14]. The main advantage of such an index is that it can be automatically optimized because we can find the best possible relation between efficiency and performance gain.

This efficiency index is defined as the portion of time that workers do useful work over the time they have been available for doing useful work. More formally, we define the efficiency index for x workers $E(x)$ as T_c/T_{avail}, where T_{avail} is $\sum_{i=0}^{x-1}\text{tavail}_i$, and tavail_i is the time worker i has been available for doing useful work, which for an application like the ones we are modelling, will be the whole iteration time (T_t). Consequently, the efficiency index will be defined as $T_c/nT_t(x)$, and the performance index as

$$\text{Pi}(x) = \frac{T_t(x)}{E(x)} = \frac{xT_t(x)^2}{T_c} \tag{13}$$

The performance index of the example application can also be seen in Fig. 6, as well as how it reaches its minimum value at the point from where adding more workers to the application is not expected to significantly improve its performance. There is only a 13.5% performance gain from 23 workers (93.7433 ms) to 40 workers (lowest expected execution time of 81.1024), increasing the amount of resources by 74%.

In conclusion, the performance index (Pi), which relates execution time with resource efficiency, allows us to automatically find the number of workers that maximizes performance (minimizing execution time) without wasting resources for every Master/Worker application, independently of the value of the parameters that characterize them.

6.3. Experimental evaluation on a real platform

In the previous sections we have defined and analyzed a set of expressions aimed at modeling the performance of Master/Worker applications in a dynamic performance-tuning environment. Now, our goal is to validate this analytical model through the execution of a wide range of synthetically generated applications on the same platform described in Section 5.4. The examples cover a wide range of possibilities, from low-compute low-communication to compute and communication intensive applications, in order to reach this target of our validation. Moreover, we have executed configurations using an asynchronous communication protocol, and others using a synchronous communication protocol. Through the experiments shown in Fig. 7 we show the results of executing an application with a low associated processing time of only one second and for different communication loads using asynchronous sends. While it can generally be seen that the model matches the application's behavior, there are some cases that deserve more detailed comment.

First, it can easily be seen that minor differences between the real execution value and the expected one lead to significant differences between the real performance index value and the expected one. This happens because of the multiplicative effect that the differences between the expected and the observed execution time ($T_t(x)^2$) introduces to the performance index. However, we can trust the expected value because the index tendency is generally preserved.

In the second place, a sudden discontinuity can be seen in Fig. 7(a) for 32 workers, which is due to implementation of the communication library. The MPI library uses different communication protocols depending on the message size and buffers availability [26].

Briefly, if programmers do not choose a particular communication mode (blocking, non-blocking, buffered, etc.) the library uses the *standard send*, which means that depending on the current execution conditions and message characteristics a *ready*, a *blocking*, or even a *synchronous send* may be issued. A *ready send* can be used when the matching receive has been posted and allows for the removal of some hand-shake messages; thus improving performance. A *blocking send* does not return until the message has been stored away and the sender is free to reuse the send buffer, this happens when the message is copied in the matching receiving buffer or in a system buffer. Finally, a *synchronous* send only returns when the matching receive has started to receive the message. Usually, MPI implementations use ready sends for small messages and blocking or synchronous ones for long messages.

Consequently, as far as we have used the standard communication mode in the synthetic applications, what has been reflected in the graph is the change between *blocking* communication mode and the *ready* one. The possibility of changes like this occurring should be taken into consideration by the tuning tool because they

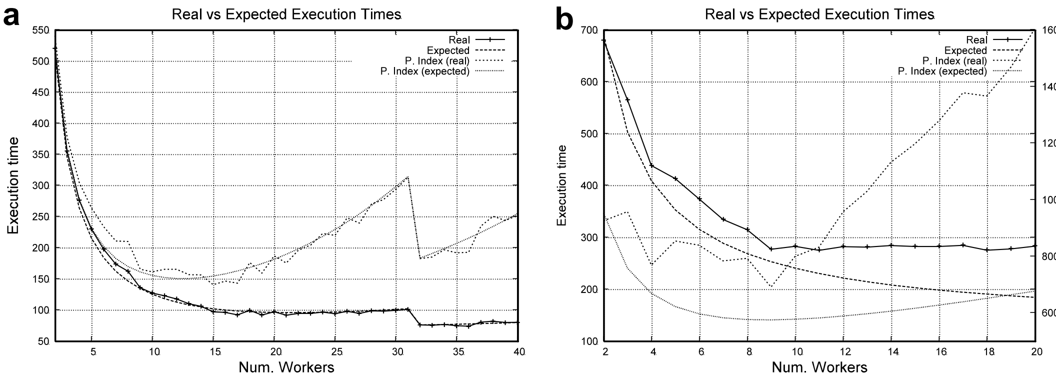

Fig. 7. Execution times and Performance Index values of a Master/Worker application with an associated processing time (T_c) of 1 s and communication volume of 100 Kbytes (a) and 2 Mbytes (b). Ranging from 2 to 40 workers for case (a) and from 2 to 20 workers for case (b). Using asynchronous communication.

could cause significant mismatches between the observed results and the predicted ones. Finally, the differences between the observed and the predicted values of case in Fig. 7(b) should be highlighted, in order to emphasize the relevance of calculating the Master's Chunk Managing Capability (MCMC) exceeding point (Table 1). It can be seen that from 9 workers onwards, the divergence between the observed and expected values constantly gets bigger and bigger.

In the graphs in Fig. 8, we show the results of the execution of an application with a medium associated processing time of five seconds and for different communication loads. In this case, we can see that both the lowest execution value and the lowest performance index value are reached with a significantly greater number of workers than those of Fig. 7; nevertheless, it must be noted that this increase is far from proportional to the one for processing time. Moreover, it seems that there are less mismatching points between the

Table 1
Relevant real and expected magnitudes associated with applications of Fig. 7

		Number of workers for the lowest execution time	MCMC exceeding point	Number of workers for the performance index lowest value
Fig. 7(a)	Real	36	–	15
	Expected	32	–	12
Fig. 7(b)	Real	11	9	9
	Expected	64	10	9

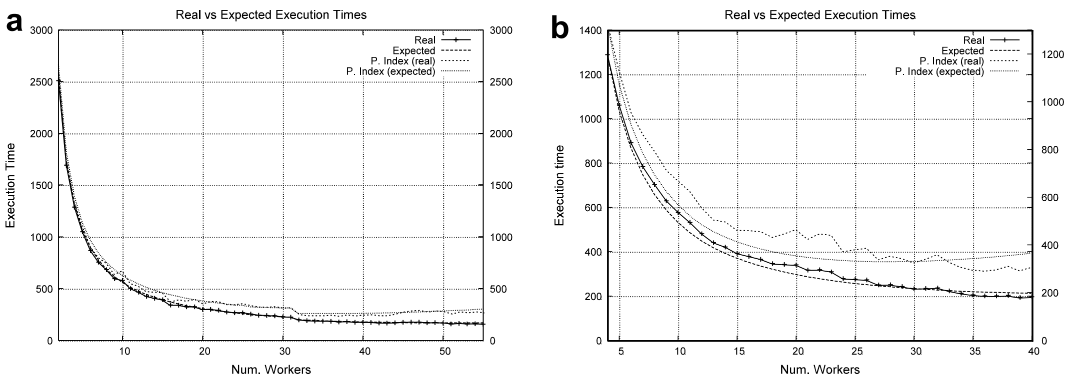

Fig. 8. Execution times and Performance Index values of a Master/Worker application with an associated processing time (T_c) of 5 s and communication volume of 100 Kbytes (a) and 512 Kbytes (b). Ranging from 4 to 55 workers for cases (a) and from 4 to 40 workers for case (b). Using asynchronous communication.

observed and the expected results than for the previous figure, because in the previous case the application executions were more sensitive to any external influences (such as context switching) due to the low computing time associated to each worker. A very good example of this can be seen in Fig. 8(a), where the point of communication mode switching of the communication library at 32 workers is barely reflected in the graph.

Finally, in Table 2 we can see significant differences between the expected and the observed number of workers where the lowest performance index is reached. We can partially blame the multiplicative effect of the total expected and observed execution time ($T_t(x)$) differences in the performance index expression of this occurrence. Although there are exceptions, the absolute differences between the observed index value for the number of workers that lead to the lowest expected value and the observed lowest index value are usually not particularly high. However, in the particular case of Fig. 8(b), we find significant differences of 20.9% because in this case the observed execution time has experienced a slight increase in relation to the expected value just for the number of workers that should lead to the lowest index value, while next to it we have obtained a few observed values that are below what was expected.

In the graphs in Fig. 9, we show the results of the execution of an application with a high associated processing time of 15 s and for different communication loads. Again, a close match between observed and predicted values is confirmed because this application is still more immune from external influences than the previous ones. Moreover, we can see that again more workers can be added to the application (Table 3) than for the previous case (Table 2) before reaching the application's lowest execution time and the lowest value of the performance index, but also that this increase is not proportional to that of the processing time. In addition, the performance index values are closer to those of the total execution time in the range shown than in the previous cases; this is because the communication overhead is very low in relation to the processing time and, consequently, the total execution time is closer to the processing time ($\mathrm{Pi}(x) = T_t(x)^2/T_c \approx T_t(x)$).

In this case, there is also a closer match between the expected and observed values of the performance index than the one obtained for the application of Fig. 8. In particular, there are differences of only 1.3% and 1.9% between the observed and the expected lowest index values for cases (a) and (b) respectively.

Table 2
Relevant real and expected magnitudes associated with applications of Fig. 8

		Number of workers for the lowest execution time	Number of workers for the performance index lowest value
Fig. 8(a)	Real	–	37
	Expected	62	35
Fig. 8(b)	Real	–	36
	Expected	49	28

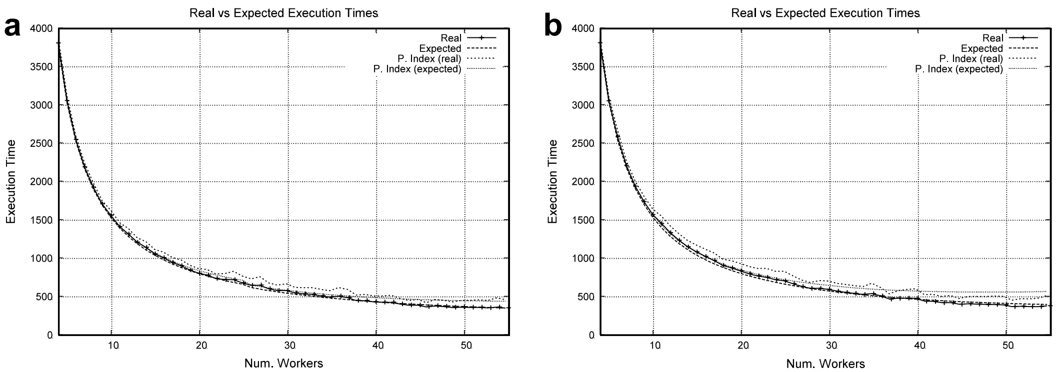

Fig. 9. Execution times and Performance Index values of a Master/Worker application with an associated processing time (T_c) of 15 s and communication volume of 100 Kbytes (a) and 512 Kbytes (b). Ranging from 4 to 55 workers for all cases. Using asynchronous communication.

Table 3
Relevant real and expected magnitudes associated with applications in Fig. 9

		Number of workers for the lowest execution time	Number of workers for the performance index lowest value
Fig. 9(a)	Real	–	–
	Expected	107	61
Fig. 9(b)	Real	–	45
	Expected	84	48

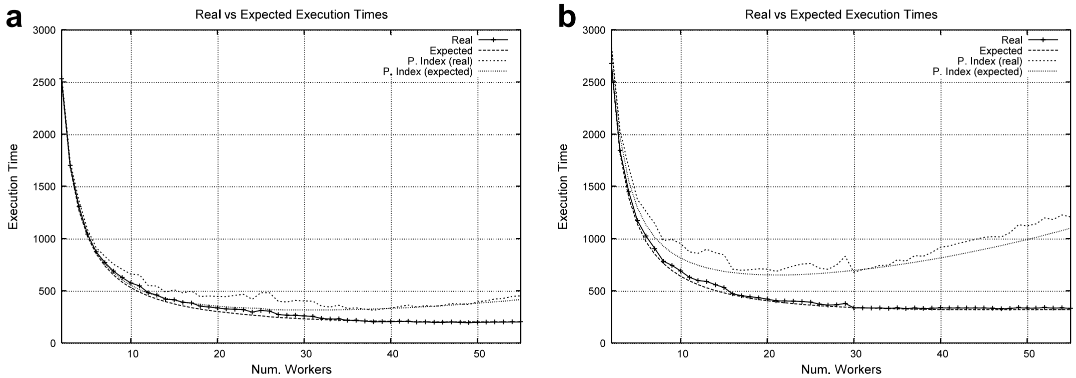

Fig. 10. Execution times and Performance Index values of a Master/Worker application with an associated processing time (T_c) of 5 s and communication volume of 100 Kbytes (a) and 2 Mbytes (b). Ranging from 2 to 40 workers for case (a) and from 2 to 30 workers for case (b). Using synchronous communication.

Table 4
Relevant real and expected magnitudes associated with applications in Fig. 10

		Number of workers for the lowest execution time	MCMC exceeding point	Number of workers for the performance index lowest value
Fig. 10(a)	Real	–	–	38
	Expected	55	–	31
Fig. 10(b)	Real	48	30	21
	Expected	46	30	21

In the graphs of Fig. 10, we show the results of the execution of an application with a medium associated processing time of 5 s and for different communication loads using a synchronous communication protocol, which in an MPI environment means, as we mentioned before, that a send will not finish until the matching receive begins.

We can see that there is also a close match between the observed values and the expected ones, and we can say that, by comparing these results with the ones shown in Fig. 8, using a synchronous communication protocol produces a significant decrease in the application's performance, and consequently, the lowest execution time and lowest performance index values are reached with less workers. The differences become greater when more workers (thus more communications) are added. For instance, with a communication volume of 100 Kbytes (a) the differences range from 0.9% for 2 workers to 17% for 40 workers (Table 4).

7. Summarizing Master/Worker model and final considerations

Our objective in this section is to sum up the performance model for Master/Worker applications in accordance with the general model for dynamic performance tuning presented in Section 3, first summarizing the set of expressions and strategies that must be used by the dynamic tuning tool to evaluate the performance of the application and predict what will happen if some conditions change, then indicating what application parameters must be monitored at run time in order to be able to detect performance bottlenecks, and finally, indicating the parameters that can be changed at run time to improve the application's performance and when these changes can take place. In addition, given that the proposed performance model includes two phases (load balancing and adjusting the number of workers), some considerations are included in this summary about the conditions that must be met to guarantee that both phases can be safely combined.

First, we showed that, in order to improve the performance of a Master/Worker application, we should be able to balance the worker load and then determine the appropriate number of workers to do the work. Therefore, we proposed a two phase strategy for the automatic performance tuning of this kind of application, the first phase consisting of applying a load balancing strategy and the second one of using an analytical model to evaluate and predict an appropriate number of workers for the application.

With the objective of balancing the worker load we have adopted a partial task distribution policy, which basically consists of dividing the set of tasks to be processed into sub-sets called batches and then distributing them among workers one by one in units called chunks. We have proposed a distribution balancing strategy that has been called Dynamic Adjusting Factoring (Section 5.3), which is based on making a self-adaptative partial distribution of the tasks to be processed through expressions (1) and (2).

A balanced application makes good use of its assigned resources and, therefore, achieves the best possible performance for those resources. However, it should be evaluated whether it is possible to get further improvements in performance by changing (usually adding) the number of assigned resources. This evaluation must be based on dynamic predictions of the number of resources needed by the application to achieve its ideal performance and, if the predicted value is different form the current one, then decide if it will be profitable to assign (or liberate) the extra resources. We have defined an analytical model for balanced Master/Worker applications aimed at performing this evaluation, this takes into consideration the current and expected application characteristics to determine the number of workers that should be used in order to efficiently complete the work in the minimum amount of time.

This analytical model includes expressions for modeling the execution time of an application iteration depending on the communication protocol: expressions (7) and (8) if an asynchronous communication protocol is used, and expression (9) if not. In addition, the model includes an expression to determine the Master's worker management capacity, which is the highest number of workers that the master can send tasks to before receiving the first answer, this has been called the Master's Chunk Managing Capacity (MCMC) because it is also useful for the distribution strategy in order to establish a lower bound for the number of tasks to be included in a batch; again the expressions depend on the communication protocol: expressions (10) and (11) if an asynchronous communication protocol is used, and expression (12) if not. Finally, the model includes a performance index (Pi-expression (13)) intended for figuring out the number of workers that would lead to the best execution time – resource efficiency ratio.

Both phases, load balancing and adapting the number of workers, have been individually validated by experimentation. However, assuming that we start from a balanced execution, we do not know in advance whether the distribution strategy will able to succeed in balancing the load for a different number of workers.

There are some conditions that must be met in order to make this possible. For example, the number of tasks should be considerably greater than the number of workers, and a small portion of tasks should not concentrate a major portion of processing time. Moreover, in so far as it is possible to monitor the number of tasks and calculate the mean processing time associated with each task and its standard deviation, it is possible to define an index to measure the quality of the outcome of the distribution strategy and predict whether it is likely to succeed in balancing the load for a different number of workers.

One possible way of evaluating the quality of the outcome of the distribution policy is by relating the mean idle time per processor with the total execution time through an expression like: $1 - \left(\frac{1}{x}\sum_{i=0}^{x-1}\mu_i/T_t(x)\right)$ (where x is the number of workers, μ_i the processing time spent by worker i, and $T_t(x)$ the total iteration time). For this expression, a zero value indicates a perfectly balanced application (all workers have been busy throughout the whole iteration), while a value closer to one indicates that there is a significant load unbalance in the application. Nevertheless, this expression cannot be used for predicting the quality of the outcome of a distribution strategy for a different number of workers because we cannot apply expressions (7), (8), or (9) to a prediction of the total execution time as they are only valid for balanced applications.

On the other hand, as mentioned in Section 5.3, an upper bound for a Pth order statistic (P independent random variables with mean μ and standard deviation σ) is defined by the expression $\mu + \sigma\sqrt{P/2}$, which leads to the following relationship: $T_t(x) \leqslant \mu(N/x) + \sigma N/\sqrt{2x}$, for a Master/Worker application with N tasks to process, a mean processing time of μ, and a standard deviation of σ. Then, substituting $T_t(x)$ by its upper bound in the previously defined quality index leads to the following upper bound: $1/\left(1 + \frac{\mu}{\sigma}\sqrt{\frac{2}{x}}\right)$. As a result, this expression can be used as a guide for deciding whether it is worth changing the number of workers depending on how easy it will be to balance the application load.

Finally, it can be deduced from the index expression that increasing the number of workers (x) without changes in μ and σ causes the index to increase, this means that increasing the number of processors makes it more difficult to get a balanced execution; in addition, it can also be seen that, for the same reason, it will be difficult to balance applications with higher standard deviations. Both observations agree with the conditions required for balancing the load of an application as stated earlier: a number of tasks significantly greater than the number of workers and a small number of tasks should not concentrate a major portion of the processing time.

Secondly, in order to be able to apply the strategies and calculate the performance expressions several application parameters must be monitored. These parameters have been called the *measure points* of the performance model and are as follows:

- *Network parameters*: m_o and λ, which could be calculated at the beginning of the execution and should be re-evaluated periodically enabling the adaptation of the system to the network load conditions.
- *Message sizes* ($v_i^{m/w}$) have to be captured when the master sends/receives data to/from workers in order to calculate the total communication volume (V) and the portion of it that is sent by the Master (α).
- *Worker processing times* (μ_i) have to be measured in order to calculate the mean processing time μ, the standard deviation σ, and the total computing time (T_c). This parameter could be obtained by measuring the time spent by the *Do_Work* function of each worker. Finally, when applying the strategy for balancing the worker load or the expression for adapting the number of workers, some changes might be introduced to the application at run time. It is very important to know exactly what parameters should be changed and when these changes can take place. These parameters have been called the tuning points of the performance model and are as follows:
- *Partition factor of the Dynamic Adjusting Factoring strategy* (x_o): one of the major advantages of this data distribution strategy is that the factor is always being recalculated and can be changed at any time.
- *Number of workers*: this parameter is more sensitive than the previous one, the number of workers can only be changed at the beginning of an iteration, but only if the added workers (if the number of workers has been increased) have already been set up and are ready to receive tasks.

8. Conclusions

Dynamic tuning techniques are sometimes the only way of improving the performance of parallel applications, but decisions should be taken quickly and with as little intrusion as possible for this strategy to be useful. The main general goal of our study was to demonstrate that knowledge of the structure of a parallel application can be used to define a simple and highly efficient performance model aimed at allowing it to be used use by a dynamic tuning tool.

In particular, based on the analysis of the framework related drawbacks, we have described a detailed and comprehensive model for Master/Worker applications that comprises two phases, the load balancing phase and that adapting the number of workers. For the first phase we have discussed, analyzed, and tested different balancing strategies, all based on the partial distribution of tasks. Moreover, we have shown that the best outcome is obtained from the Dynamic Adjusting Factoring (DAF) policy and, in fact, its results are fairly close to the theoretical bests. On the other hand, we have presented, analyzed, and tested a complete set of expressions for modeling the behavior of a balanced Master/Worker application in accordance with the number of workers being used, the overall computation time, and the communication cost. These expressions are aimed at being used to predict an adequate number of workers for the application under the current execution conditions.

Finally, we have summarized the Master/Worker performance model, indicating what parameters should be monitored in order to evaluate the model expressions and apply its strategies, and also the parameters that have to be modified in order to trigger the actions that will overcome the detected drawbacks. In addition, we have provided a criterion, based on the succeeding possibilities of the balancing strategy, to decide whether it is advisable or not to enter the phase that adapts the number of workers at the end of the balancing stage.

References

[1] F. Almeida, D. González, L.M. Moreno, C. Rodríguez, An analytical model for pipeline algorithms on heterogeneous clusters, in: Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting, 2002, pp. 441–449.

[2] F. Almeida, D. Gonzalez, L.M. Moreno, C. Rodriguez, J, Toledo, On the prediction of Master–Slave algorithms over heterogeneous clusters, in: Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003, pp. 433–437.

[3] Ioana Banicescu, Vijay Velusamy, Performance of scheduling scientific applications with adaptative weighted factoring, in: Proceedings of the 15th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2001, p. 84.

[4] Ioana Banicescu, Vijay Velusany, Load balancing highly irregular computations with the adaptative factoring, in: Proceedings of the 16th International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2002, p. 195.

[5] Anne Benoit, Murray Cole, Stephen Gilmore, Jane Hillston, Evaluating the performance of skeleton-based high level parallel programs, Lecture Notes in Computer Science (LNCS), vol. 3038, Springer-Verlag, 2004, pp. 289–296.

[6] E. Cesar, J.G. Mesa, J. Sorribes, E. Luque, POETRIES: performance oriented environment for transparent resource-management, implementing end-user parallel/distributed applications, LNCS, vol. 2790, Springer-Verlag, 2003, pp. 141–146.

[7] E. Cesar, J.G. Mesa, J. Sorribes, E. Luque, Modeling Master–Worker applications in POETRIES, in: Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), IEEE Computer Society, Santa Fe, New Mexico, April 2004, pp. 22–30.

[8] E. Cesar, A. Morajko, T. Margalef, J. Sorribes, A. Espinosa, E. Luque, Dynamic Performance Tuning Supported by Program Specification, Scientific Programming, vol. 10, IOS Press, 2002, pp. 35–44.

[9] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.

[10] A. Espinosa, T. Margalef, E. Luque, Integrating Automatic Techniques in a Performance Analysis Session. LNCS, Vol. 1900, Springer-Verlag, 2000 (Euro-Par 2000), pp. 173–177.

[11] T. Fahringer, A. Požgaj, P^3T+: A Performance Estimator for Distributed and Parallel Programs. Scientific Programming, vol. 8, no. 2, IOS Press, The Netherlands, 2000.

[12] Susan Flynn Hummel, Edith Schonberg, Laurence E. Flynn, Factoring: a method for scheduling parallel loops, Communications of the ACM 35 (8) (1992) 90–101.

[13] William Gropp, Ewing Lusk, Reproducible Measurements of MPI Performance Characteristics, Technical Report ANL/MCS-P755-0699. Mathematics and Computer Science Division, Argonne National Laboratory, 1999.

[14] E. Heymann, M.A. Senar, E. Luque, M. Livny, Efficient resource management applied to Master–Worker applications, Journal of Parallel and Distributed Computing 64 (2004) 767–773.

[15] Clyde P. Kruskal, Alan Weiss, Allocating independent subtasks on parallel processors, IEEE Transactions in Software Engineering 11 (10) (1985) 1001–1016.

[16] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, K. Tan, From patterns to frameworks to parallel programs, Parallel Computing 28 (12) (2002) 1663–1683.

[17] B.L. Massingill, T.G. Mattson, B.A. Sanders, A Pattern Language for Parallel Application Programs. LNCS, vol. 1900, Springer-Verlag, 2000 (Euro-Par 2000), pp. 678–681.

[18] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, The Paradyn parallel performance measurement tool, IEEE Computer 28 11 (November) (1995) 37–46.

[19] D. Morales, F. Almeida, F. Garcia, J. Gonzalez, J. Roda, C. Rodriguez, A Skeleton for Parallel Dynamic Programming In: Proceedings of Euro-Par'99 Parallel Processing, Springer-Verlag, Toulouse France, 1999.

[20] A. Morajko, P. Caymes, Tomàs Margalef, Emilio Luque, Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications. LNCS, vol. 3515/2005, Springer-Verlag, 2005, pp. 132–139.

[21] A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, Joan Sorribes, Emilio Luque, Automatic Tuning of Master/Worker Applications. LNCS, vol. 3648, Springer-Verlag, 2005, pp. 95–103.

[22] Anna Morajko, Eduardo César, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, Emilio Luque, Development and Tuning Framework of Master/Worker Applications Invited paper, Journal of Computer Science and Technology (JCST) 5 (3) (2005) 115–120.

[23] A. Morajko, O. Morajko, J. Jorba, T. Margalef, E. Luque, Automatic performance analysis and dynamic tuning of distributed applications, Parallel Processing Letters, World Scientific, 2003, pp. 169–187.

[24] L.M. Moreno, F. Almeida, D. González, C. Rodríguez, The Tuning Problem on Pipelines. LNCS, vol. 2150, Springer-Verlag, 2001, pp. 117–121.

[25] A. Moreno, E. César, J. Sorribes, T. Margalef, E. Luque, Balanceo de carga en sitemas distribuidos Master–Worker. XVI Jornadas de Paralelismo 2005, Granada, 2005, pp. 443–450.
[26] MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. June 1995.
[27] Susanna Pelagatti, Task and data parallelism in P3L, in: F. A Rabhi, S. Gorlatch (Eds.), Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2002, pp. 155–186.
[28] J. Roda, C. Rodríguez, F. Almeida, D. Morales, Prediction of parallel algorithms performance on bus based networks using PVM, in: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998, pp. 57–63.
[29] J.S. Vetter, D.A. Reed, Managing performance analysis with dynamic statistical projection pursuit, in: Proceedings of SC'99, Portland, 1999.
[30] F. Wolf, B. Morh. Automatic performance analysis of SPM cluster applications, Technical Report IB-2001-05, 2001.