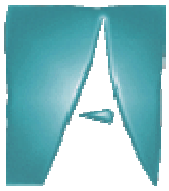


Universitat Autònoma de Barcelona

Definition of Framework-based Performance Models for Dynamic Performance Tuning

Departament d'Arquitectura
d'Ordinadors i Sistemes Operatius

Thesis submitted by Eduardo Cesar Galobardes in fulfilment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*.

Barcelona, February 20th 2006

Definition of Framework-based Performance Models for Dynamic Performance Tuning

Thesis submitted by Eduardo César Galobardes in fulfillment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*. This work has been developed in the Computer Architecture and Operating System Department of the *Universidad Autònoma de Barcelona* and was advised by Dr. Joan Sorribes Gomis,

Bellaterra February 20th 2006

Thesis Advisor

Joan Sorribes Gomis

To Elisa and our children David and Ruth
To my parents Juan and Ariana

Acknowledgements

A lot of people have made this work possible and I wish to express my gratitude to all of them for their inestimable help and for cheering me up in my weakest moments.

I would like to thank especially Joan Sorribes for being my advisor throughout this work and for spending so much time an effort on it. Furthermore, I also want to thank Emilio Luque for his invaluable advice and dedication to this work.

I would like to thank especially Tomàs Margalef and Ania Morajko for the long and profitable discussions about performance modeling.

My deepest gratitude goes to my wife Elisa and our children David and Ruth, for being so patience and sacrificing so many weekends, but especially for the encouragement I have received from them.

How to forget the amazing support received from Barton Miller and the Paradyn group of the University of Wisconsin at Madison, thanks to all of them. Thanks also to Barton Miller and his family, and to Miron Livny for the great moments we have shared in Madison, our second home.

Thanks to Andreu Moreno, Jose Gabriel Mesa, Paola Caymes, Judith Colome, and Francesc Noguera for their, sometimes ongoing, contributions to this project.

I would also like to thank Anna Cortés, Miquel A. Senar, Daniel Franco, Juan C. Moure, Josep Jorba, and Fernando Cores for their encouragement and friendship.

I do not want to forget the rest of the CAOS group and those who have been members of it during this long trip.

Last but not least, I want to thank my parents and siblings for their indefatigable support.

Table of Contents

Chapter I:
Introduction

Abstract

The main objective of this chapter is to introduce the motivations that have inspired this work, as well as this thesis' framework and background. In addition, an overview of other studies related to ours is included in order to illustrate the originality and soundness of our work.

1. Introduction

Parallel and distributed programming constitutes a highly promising approach to improving the performance of many applications. However, in comparison to sequential programming, many new problems arise in all phases of the development cycle of this kind of applications.

For example, in the analysis phase of parallel/distributed programs, the programmer has to decompose the problem (data and/or code) to find the concurrency of the algorithm. In the design phase, the programmer has to be aware of the communication and synchronization conditions between tasks. In the implementation phase, the programmer has to learn how to use specific communication libraries and runtime environments but also to find a way of debugging programs. Finally, to obtain the best performance, the programmer has to tune the application by using monitoring tools, which collect information about the application's behavior. Tuning can be a very difficult task because it can be difficult to relate the information gathered by the monitor to the application's source code. Moreover, tuning can be even more difficult for those applications that change their behavior dynamically because, in this case, a problem might happen or not depending on the execution conditions.

It can be seen that these issues require a high degree of expertise, which prevents the more widespread use of this kind of solution. 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/distributed 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/distributed applications depends on some of their behavioral characteristics. If the application 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 for avoiding 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, which means that the application's performance analysis outcome must be accurate and punctual in order to effectively tackle problems; at the same time, intrusion on

the application must be minimized because the instrumentation inserted in the application in order to monitor and tune it alters its behavior and could introduce performance problems that were not there before the instrumentation.

This is more difficult to achieve if there is no information about the structure and behavior of the application; therefore, blind automatic dynamic tuning approaches have limited success, whereas cooperative dynamic tuning approaches can cope with more complex problems at the cost of asking for user collaboration. We have proposed a third approach. If a programming tool, based on the use of skeletons or frameworks, has been used in the development of the application then much information about the structure and behavior of the application is available and a performance model associated to the structure of the application can be defined for use by the dynamic tuning tool. The resulting tuning tool should produce the outcome of a collaborative one while behaving like an automatic one from the point of view of the application developer.

In this chapter we want to summarize and review some of the most important structure-oriented tools for parallel/distribute development (section 2), as well as the most relevant approaches and tools for performance tuning (section 3). We conclude with the presentation of the proposal that is developed in this thesis (section 4), and an overview of related studies (section 5).

2. Parallel/Distributed Application Development

There is no doubt that developing parallel/distributed applications is the way to cope with many complex problems such as weather forecasting, genetic and medical research, physics of high energy simulation, and so on; however, developing this kind of applications involves dealing with many more problems than its sequential counterpart.

First, application designers must find out how to decompose a problem into sub-problems that can be solved concurrently. This decomposition must be devised taking into consideration the characteristics of the problem but also the characteristics of the computation and general programming model to be used.

The reason is that: it is not the same to design an application for a Multiple Instruction Single Data (MISD) computer, in which, theoretically a single memory is shared by multiple processors as to design an application for a Single Instruction Multiple Data (SIMD) one, where, theoretically, multiple processors synchronously execute the same code over different streams of data, or as to design it for a

Multiple Instruction Multiple Data (MIMD) computer, (the most general and powerful computation model) in which multiple independent code streams hosted in different processors operate asynchronously over different streams of data.

Moreover, it is not the same to design an application on a Message Passing programming model, where a set of processes with access to their local private variables interchange data among themselves by sending and receiving messages, as to design an application on a Shared Memory programming model, where a set of processes have access to their local private variables and also to a central pool of shared ones, as to design an application on a Data Parallel programming model, which is closely related to the SIMD computation model because it consist of applying the same instructions to different elements of a data structure.

Secondly, application programmers must deal with communication libraries, race conditions, poor debugging tools, and so on. The main problem, at this level, is that programming parallel/distributed applications using low level primitives, such as sockets o binary semaphores, is like developing a complex sequential tool using assembly language.

However, the solution to many computationally intensive problems exhibits a degree of commonality that can be exploited. This fact makes it possible to define patterns describing common problems and the core of their solution for parallel/distributed applications design and programming. Furthermore, this constitutes the basis for many developing tools and design methodologies that we summarize in the next subsections by discussing some relevant examples.

2.1. Parallel functional languages

The potential of functional languages for parallelism is supported by the abstraction mechanism of these languages, as well as their sophisticated type system, and high level coordination. However, the key advantage of the functional paradigm is that referential transparency guarantees considerable freedom of execution order without changing program semantics.

Several parallel functional languages have been defined, many of them based on Haskell [HH92] a standard lazy functional research language with a sophisticated type and class system and with a relatively mature development environment including compilers, interpreters, libraries, and profiling tools. A comprehensive summary of the parallel functional Haskell-based languages can be found at [TLP02]. These languages are classified in two major categories: Parallel Haskells

and Distributed Haskells, depending on whether additional processors are used to reduce program runtime (parallelism) or to allow machines to interact in a common virtual world.

Parallel Haskells are classified in:

- *Skeleton-based* Haskells, such as HDC [Her00], which is a subset of Haskell with skeleton-based coordination that supports two divide-and-conquer skeletons and a parallel map.
- *Data parallel* Haskells, such as Nepal [CK+01], which provides special syntax for array comprehensions and parallel implementations of basic functions over these arrays; in addition, data parallelism can be nested and latter flattered using a special flattering transformation.
- *Semi-explicit parallel* Haskells, such as Eden [LOP05], which coordinates parallel computation using explicit process creation and interconnection, enabling the programmer to define arbitrary process networks.
- *Haskell with a coordination language*, such as Caliban [KT99], which is a subset of Haskell plus a set of constructs for explicit partitioning of computation into threads and for assigning threads to processors in a static process network.

Finally, among Distributed Haskells we have Haskell with Ports [HN02], which extends concurrent Haskell with the port data type to allow distributed system development.

However, functional programming suffers two main problems: first, it is not a popular programming paradigm, though programs tend to be simpler and smaller than those written with imperative languages; secondly, and more importantly, the performance improvements obtained with these languages are quite limited when compared to imperative ones. A comprehensive study on the performance of programs written with some of these languages can be seen in [LR+03].

2.2. Pattern and framework-based methodologies

A *pattern* for parallel applications' design and programming can be defined as, "abstractions that capture the expertise needed to write parallel programs" [MA+02] or, more generally, as a "solution to a problem in a context" [MSM04]. The main idea is that patterns are applicable to different problems domains (each with different characteristics and concerns) and that they must be adapted for each particular problem.

Once a pattern has been selected, we can use a *framework* to implement the solution without going into the tedious and error-prone process of writing the code from scratch. A framework can thus be defined as a piece of code that implements the application-independent structure of a specific kind of program, including the flow of control between the offered operations. Usually, programmers only should provide the functionality of those operations that are specific to the developed application. A representative example of a framework-based tool is CO_2P_3S [MA+02], which offers the possibility of choosing an application pattern and parameterize it to obtain a framework in form of a class that has some hook methods that must be provided by the user in order to get the application's code.

Besides frameworks, there are also *skeletons*, which are conceptually similar to frameworks but with a more functional approach (it can be seen in the previous subsection that there is a Skeleton-based Haskells category). Among the skeleton-based tools, it is worth mentioning eSkel [Col04] and llc [DG+03]. eSkel is a library of C functions and type definitions that extend the standard C binding to MPI with skeletal operations while llc is an extension of the C language by means of pragmas that introduce parallel constructions, such as *data parallel* or *pipeline*, into the application.

We want to take a closer look at the *Pattern Language for Parallel Programming* (PLPP) presented at [MSM04] because it is a comprehensive approach to parallel application design. Moreover, we will use it in the next chapter when describing the frameworks we have modeled. The aim of this pattern language is to guide the application designer through the entire process of developing a parallel program assuming only that the designer has a good understanding of the actual problem to be solved. The idea is then to work through the pattern language in order to obtain a detailed application design.

PLPP is organized in four design spaces:

1. The *Finding Concurrency* design space shown in figure 1 is intended to help in finding the problem concurrency and decompose it in a set of sub-problems.
2. The Algorithm Structure design space shown in figure 2 contains patterns that help to find an appropriate algorithm structure to exploit the concurrency that has been identified.
3. The *Supporting Structures* design space includes patterns describing useful abstract data types and other supporting structures.

4. The Implementation Mechanism design space contains patterns that describe lower-level implementation issues.

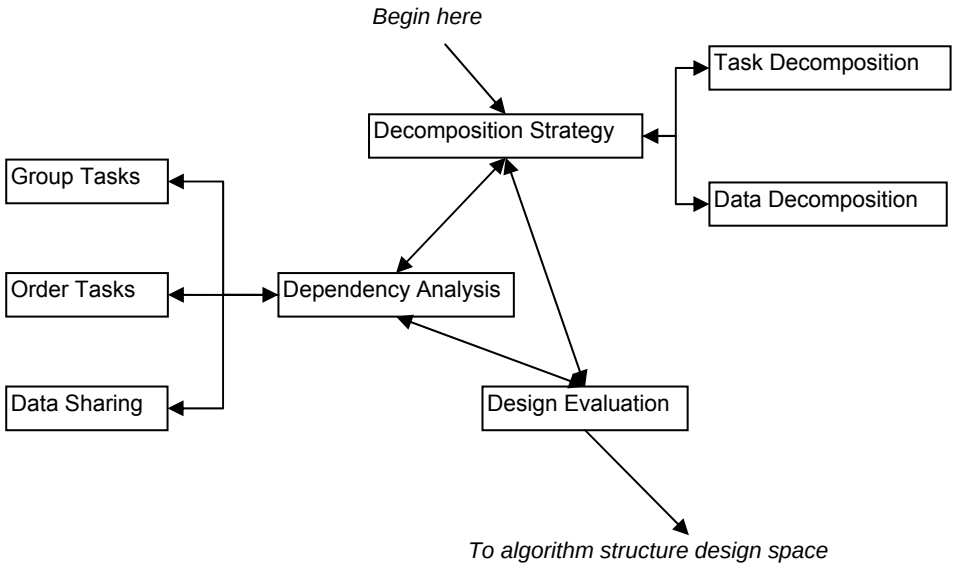


Figure 1. PLPP Finding Concurrency design space patterns and organization.

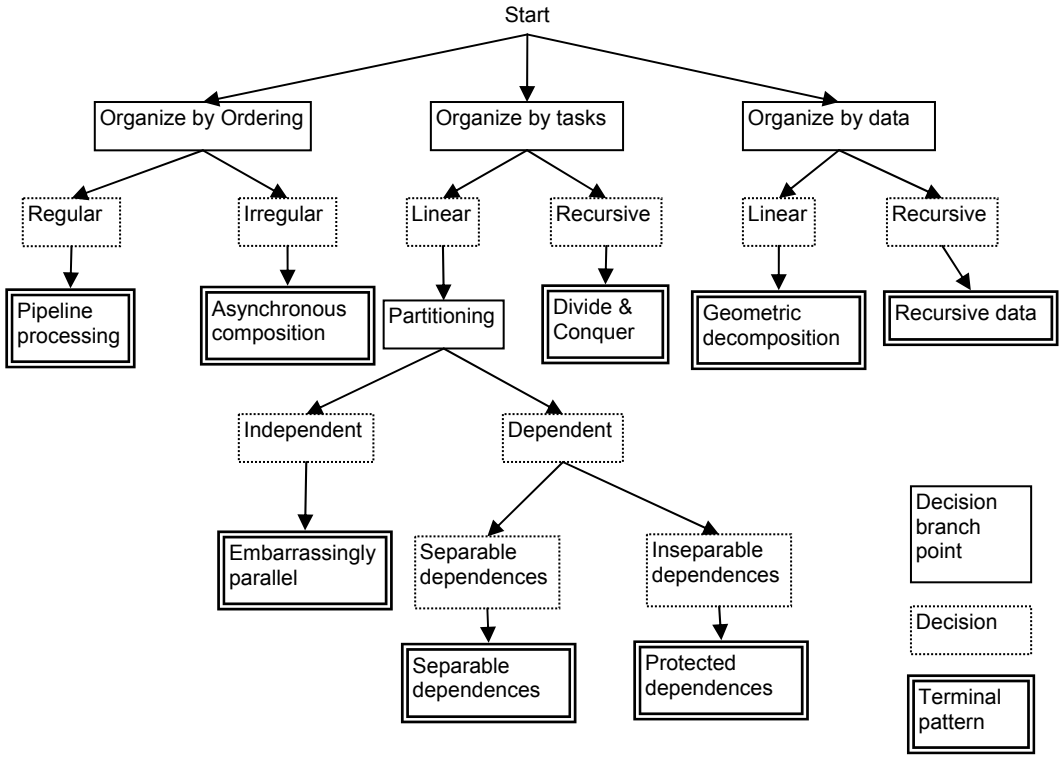


Figure 2. PLPP Algorithm Structure design space patterns and organization.

Once the problem has been subdivided, the concurrency disclosed, and the target platform constraints are known, the algorithm structure should be selected by first

considering if the sub-problems must be solved in an orderly way or not (Organize by Ordering), or if simply solving the tasks will do the job (Organized by Tasks), or when data decomposition is the major concurrent organizational principle (Organized by Data).

Finally, each of these branches leads to new patterns that are closer to implementation issues. In this way, if the organizational principle is the order then we can choose the *Pipeline* terminal pattern for calculations that can be orderly applied on different sets of data, or the *Asynchronous Decomposition* pattern for groups of tasks that interact through asynchronous events. If tasks are the organizational principle, which is the most common case, then we can choose the *Embarrassingly Parallel* pattern for totally independent tasks, or the *Separable Dependences* one when dependences can be pulled outside concurrent execution, or the *Protected Dependences* pattern when dependences cannot be pulled outside concurrent execution and must be managed during this execution, or the *Divide and Conquer* one when the sub-problems are found recursively. Finally, if data decomposition is the organizing principle then we can choose the *Geometric Decomposition* pattern if the problem space can be decomposed into discrete subspaces and a solution computed for each subspace and then each partial solution aggregated to the global one, or the *Recursive Data* one, if the problem is defined in terms of following the links of a recursive structure.

2.3. Representation-oriented approaches

Graphical interfaces are used in many tools, such as CO_2P_3S [MA+02], in order to facilitate the application's specification. However, for some tools, intended for use in the whole development process, researchers have adopted some graphical or other high-level representation of applications as their guiding principle.

This is the case of GRADE [KC+97], that was designed as an environment for specifying, executing, debugging, and monitoring parallel applications using the PVM library. This environment is based on a graphical language called GRAPNEL that allows a multi-layer representation of the application. The upper layer allows the representation of processes and communication channels, the second layer is for graphically representing the internal algorithm of each process plus its communication operations (sends & receives), and the third level is for including the process code in C. This approach has proved quite successful and lately it has evolved from parallel to GRID application development [LS+05].

Another recent example is the UML-based approach presented in [PF02]. This study proposes taking advantage of a popular (and well supported) modeling language such as UML and adding extensions to model the most important constructs of message passing and shared memory paradigms to it, plus performance annotations. In this way, it is possible to model distributed applications obtaining performance information at an early development stage of an application. This approach has, lately, been more concentrated on performance predictability issues than on design ones [PF05].

3. Monitoring and Tuning Parallel/Distributed Applications

The main reason for investing resources and effort in developing parallel/distributed applications is to increase their performance. However, in many cases the results, in terms of performance, of such difficult and usually long developments are rather disappointing.

Theoretically, Amdahl's law [Amd67] limits the performance gain to the parallelizable portion of the algorithm divided by the number of processors. This law states that the minimum execution time of an application running on n processors is: $T(n) = \alpha T(1) + (1 - \alpha)T(1)/n$, where T(1) is the time of the application running in one processor, and α is the non parallelizable portion of the application; hence, the speedup of the application (T(1)/T(N)) is limited to n/(α*(n-1)+1) that, as can be seen in figure 3, clearly shows that there is an efficiency loss for each new added resource. This result is quite shocking because it strongly limits performance gains due to application of parallelism and, in addition, it is not even considering parallelism overheads, such as the cost of message passing. Fortunately, Amdahl's law is based on the assumption that the algorithm to be parallelized is immutable (meaning that α is the same for any number of processors), which is overly simplistic because it does not take into consideration scalability issues.

Problem scalability was taken into consideration by [Gus88] and a new expression for speedup known as the Gustafson-Baris' law, was defined as: n – (1 – n)α. This, as can be seen in figure 4, is a linear expression that promises bigger performance gains but, again, without considering parallelism overheads.

In conclusion, a parallel application can run several times faster than its sequential counterpart, though the results, in terms of performance, are likely to be highly disappointing if the application is not accurately tuned. In order to be able to do the

appropriated performance tuning, it is necessary to have a thorough knowledge of the application and its running environment, as well as a high degree of expertise.

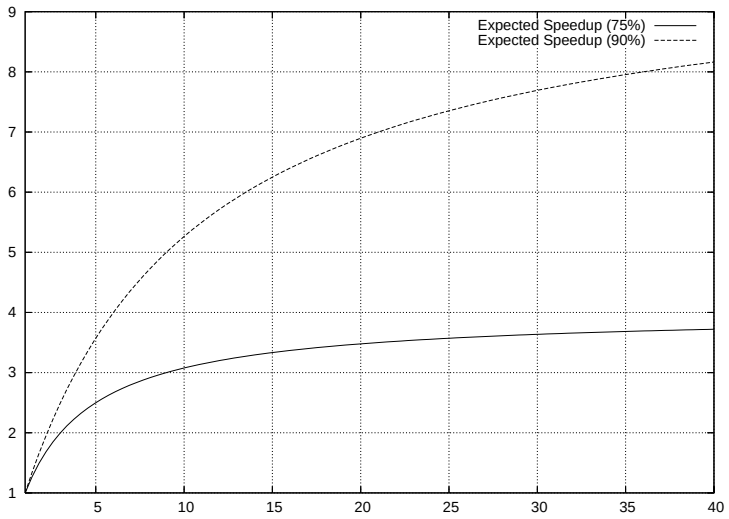


Figure 3. Expected speedup, according to Amdahl's law, for a 75% parallelizable application and for a 90% parallelizable one.

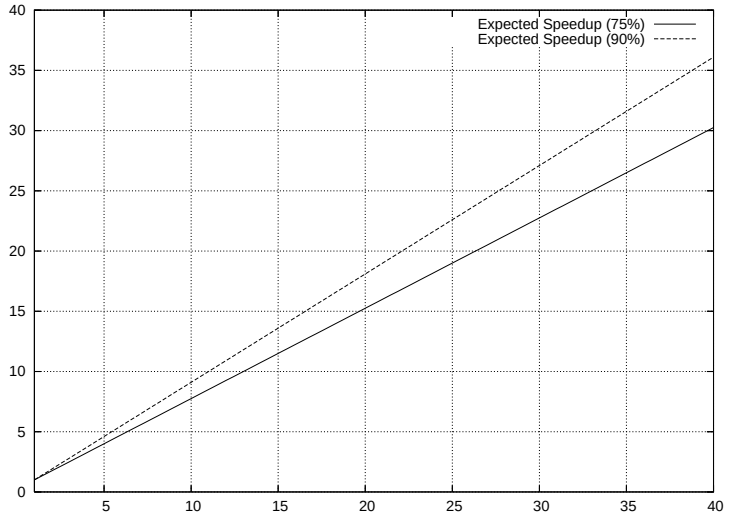


Figure 4. Expected speedup, according to Gustafson-Baris's law, for a 75% parallelizable application and for a 90% parallelizable one.

This is a very complicated task and, consequently, the aid of tools for monitoring the application's execution in order to track the most relevant performance parameters, as well as tools for analyzing the monitored data in order to find the application's performance bottlenecks, is usually very welcomed. The general performance tuning cycle shown in figure 5 consists, in the first place, of getting relevant performance data from the application's execution (monitoring). That data should then be analyzed (automatically or not) to discover the application's

performance bottlenecks. Finally, once these problems have been related to the proper application's portion of code, modifications can be introduced on the application to overcome these problems (automatically or not, dynamically or not).

Nonetheless, it also can be useful to have tools to carry on some performance analysis in advance (before the application is executed or even completely coded) in order to generate a tuned first version and avoid future time consuming code modifications.

Consequently, we can find predictive or trace-based performance analysis and/or tuning tools. Moreover, in the second case, the analysis can be performed when the application has finished its execution (post-mortem static analysis) or on the fly while the application is running (dynamic analysis), and the same can be said for the tuning process. According to this classification, a summary of some relevant tool examples is presented in the following subsections.

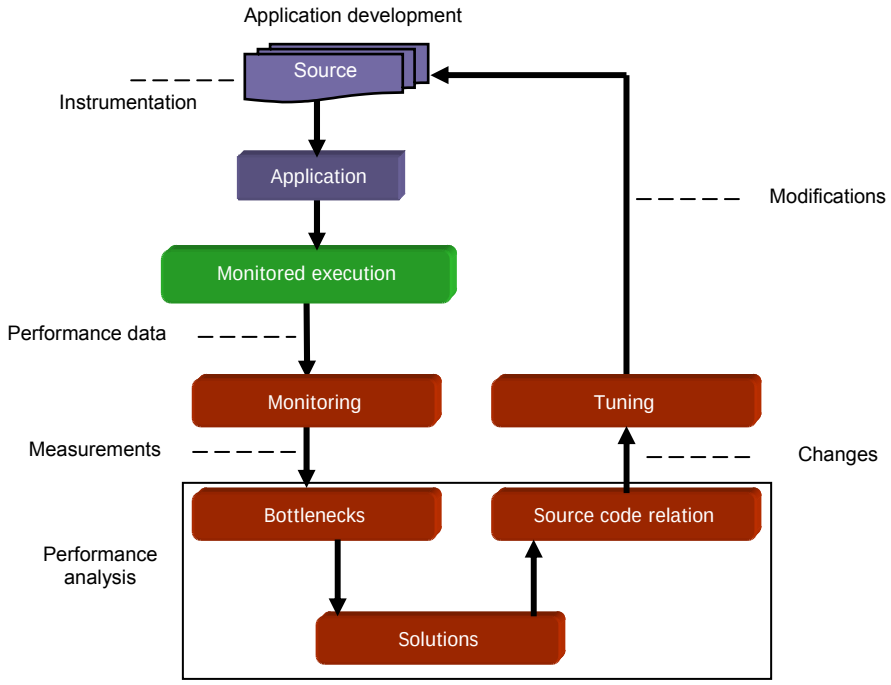


Figure 5. Performance analysis/tuning cycle.

3.1. Predictive performance analysis/tuning

Obtaining an early insight of the performance behavior of an application from a model of the application by means of a predictive performance analysis tool can be very helpful for design decisions and also for avoiding time-consuming code modifications.

A good example of such a tool is the Performance Prophet [PF05], which using a model of the application specified in the modified UML language defined in [PF02], can quickly generate and evaluate many application's performance models. The tool extracts the application's significant performance characteristics from its specification, neglecting those parts of the specification that are not relevant in performance terms, thereby simplifying the performance model definition and analysis. Then it operates by mixing analytical and simulation evaluation models by using mathematical expressions to model code blocks involving a single processing unit and event driven simulation to model code blocks involving multiple processing units. As a result, the authors claim that this tool offers comparative accuracy between parallelization strategies despite the specific difference between the real execution of a parallelization strategy and the tool outcome for the same strategy, which means that the results given by the tool for two different parallelization strategies are likely to keep the same relation than their real implementations.

Another approach consists of carrying out the performance analysis at compile time. This is the case of P3T+ [FP00], which predicts application performance based on information gathered at compiler time, plus sequential simulation and architecture parameters.

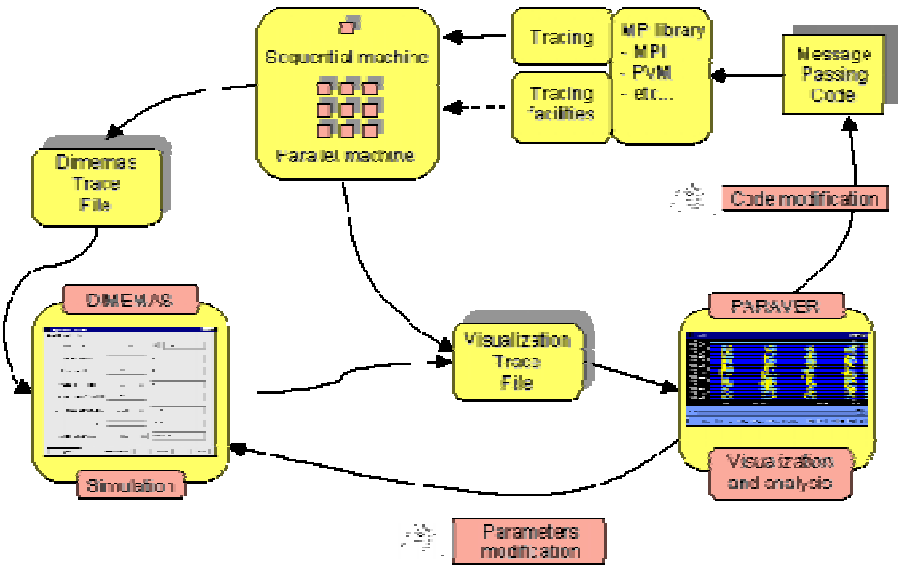


Figure 6. Dimemas and Paraver interoperation scheme.

A third approach, aimed at allowing the study of the application's performance on different platforms or parallel machines, consists of using one or more trace files (some from real executions, but mainly from simulations) to predict the behaviour of

the application under different circumstances. This approach is implemented in Dimemas [BR+03] and Paraver [Paraver], a couple of tools that allow an accurate predictive performance analysis and tuning of parallel applications without actually using a parallel machine. A schematic representation of the Dimemas-Paraver performance analysis and tuning operative model is shown in figure 6. We can see, there, that the application's trace file for visualization can be obtained from a real execution on a parallel machine or from a simulated one from Dimemas.

Finally, a very complete set of predictive tools was integrated in the POEMS project [DB+99], which was a very ambitious project for performance prediction of large scale adaptative parallel applications. The idea was to create an environment based on the composition of components represented by compositional objects; those objects would be stored in a database in order to be available to the users. The components were defined as models of application and system elements (OS and hardware). Consequently, using those components plus a performance knowledge base responsible for carrying on the performance analysis, the task dependence graph of the application (automatically generated by the compiler), and the execution description of each task (obtained from simulation), users would be able to comprehensibly analyse the application's performance at different levels.

3.2. Static trace-based performance analysis/tuning

In the classical approach to performance analysis and tuning, both the analysis and the tuning were not automatic, which means that the only available supportive tools were monitors, that were responsible for gathering information and registering an ordered trace of all relevant performance events, and visualization tools responsible for showing the information gathered in the most friendly and meaningful way. Many tools from this group can be mentioned, for example: Vampir [NA+96], Tape/PVM [Ma95] and XPVM [GB+94], ParaGraph [HF03], or Pablo [RR+93].

The main problem of this approach is that a high expertise degree is required to be able to detect performance problems, and it must be still higher to be able to relate those problems to the application code. Consequently, a more comprehensive approach to post-mortem performance analysis consisting of carrying out an automatic performance analysis, by means of adding some degree of knowledge to the tool, has been proposed in several studies, such as KappaPi [EM+00], Paradise [KK96], Expert [WM00], or AIMS [Yan94]. Usually, these tools work through a trace

file and use some heuristic knowledge to discover simple, and sometimes not so simple, problems such as sends that are called too late and make the receiver process wait too long (late-send). Finally, the main problems related to this approach are the generation and storage of possibly huge trace files and controlling the overhead introduced by the instrumentation needed to gather the performance data, which many times must be directly introduced by the programmer.

3.3. Dynamic performance analysis/tuning

In order to eliminate the need for generating and storing huge trace files and to take control of the amount of instrumentation introduced (and hence of the intrusion degree), tools with the ability to dynamically analyze the application's performance have been developed.

Some of them, such as Paradyn [MC+95] and Dynamic Statistical Projection Pursuit (PP) [VR99], are still only focused on analyzing performance. In the specific case of Paradyn, performance bottlenecks are sought using the W^3 search model (Why is there a performance bottleneck? Where is it located? When did it happen?), whereas PP dynamically analyzes performance and minimizes instrumentation intrusion. It identifies through projection indexes the most important metrics that reflect the application's performance.

However, if a tool is able to dynamically insert instrumentation in an application and also to detect the application's performance bottlenecks and their causes on the fly then it can also be possible to extend the tool to automatically solve these problems (at least some of them). This approach has been called dynamic performance tuning and has been implemented by many tools, such as the *Monitoring Analysis and Tuning Environment* (MATE) [Mor03], the *Mirror Object Steering System* (MOSS) [ES98], *Autopilot* [RS+01], *Java HotSpot* [JHS02], and *Active Harmony* [TC+02].

We will discuss in more detail in the next chapter the dynamic performance tuning approaches and related problems, and especially the MATE tool because it is the frame in which the work presented in this thesis has been developed. However, we can say that dynamic tuning is some times the only way to improve the application's performance because the static approach discussed before is useful for those applications that show a steady behavior from execution to execution but not for applications that show significant differences between executions depending, for

instance, on the input data or even for those applications that show a highly dynamic behavior in a single execution.

Nevertheless, taking tuning decisions dynamically requires a highly efficient analysis because performance problems must be detected quickly in order to get greater improvements and with as little instrumentation as possible in order to minimize intrusion. This means that much information about the behavior of the application must be available to the tuning tool with little intrusion, which is difficult to achieve if all information must be blindly gathered at execution time.

Three different approaches have been proposed to solve this problem, in the first place, the cooperative approach implemented in tools such as MOSS [ES98] in which human users (programmers, final users) actively participate in the tuning process (completely or partially) by carrying out tasks such as inserting instrumentation into the application, analyzing gathered information, or taking tuning decisions. This approach is highly effective but demands a high degree of user expertise. In the second place, the automatic approach, which is implemented in tools such as Java HotSpot [JHS02], consists of searching for performance inefficiencies without any specific knowledge on the application. This approach is transparent to the users but is strongly limited because generating enough knowledge to take complex decisions needs too much time to be effective. Finally, the semi-automatic or automatic with knowledge approach, used in tools such as Autopilot [RS+01], Active Harmony [TC+02], and MATE [Mor03], consists of carrying out an automatic tuning (without user intervention) but with previous knowledge about the application.

In the case of Autopilot, programmers have to explicitly insert instrumentation in the application, and then the application is steered, interactively or automatically, through a set of sensor processes responsible for gathering performance information. There is also a fuzzy logic engine responsible for selecting resource managing policies based on the gathered data and a set of actuators responsible for invoking local functions or modifying application variables. Active Harmony is a tool focused on the automatic selection of the most appropriated algorithm to perform a calculation, providing a Library Specification Layer that allows the integration of different libraries with similar functionality. Developers must explicitly introduce calls to the system API to indicate the places where decisions have to be taken and which are the tunable parameters. At run time, the Adaptation Controller is

responsible for evaluating the application's behavior and heuristically exploring the tunable parameters value space in order to find the best configuration.

Finally, MATE is an environment that provides a general performance analysis and tuning framework for developing dynamic tuning tools. In this environment, performance knowledge is provided from outside allowing the application of tuning techniques at different levels (system calls, libraries, the application code), and by different approaches (automatic, automatic with knowledge). Moreover, instrumentation of applications is not explicit, which allows tuning tools to be built that are completely transparent to the application developers. Regarding the structure and specification of performance knowledge, MATE requires an specification of the parameters that should be monitored at run time (*measure points*), a set of code pieces specifying an analysis model to be applied to the data gathered (*performance expressions or strategies*), and a set of actions to be applied if a performance drawback is discovered during the analysis period (*tuning points*).

4. Our Proposal

We saw in the previous sections that two kind of supportive tools are of utmost importance for overcoming the difficulties of developing parallel/distributed applications: in the first place, design and programming tools aimed at accelerating the implementation process and reducing the possibility of costly to prune programming errors; in the second place, performance analysis and tuning tools aimed at obtaining the best performance (usually in terms of execution time) for the applications developed.

In the first group of tools, a very common approach consists of providing the programmers with predefined structures that partially implement and hide several functional and structural aspects of the most popular parallel/distributed programming constructions. This is the case of skeletons and frameworks, which allow programmers to focus on the functionality of the problem being solved rather than on synchronization and communication issues at the price of some loss of performance. However, using this kind of supportive tools has another result: the overall structure of every application developed with them, as well as the interrelations between its components can be known in advance, and that can be very useful for modeling broad sets of applications without knowing their specific functionality.

In the group of performance and tuning tools, the most sophisticated ones are those able to provide advice, or even directly apply corrections, as to what changes must be carried out on an application in order to improve its performance. These tools can be divided in those that make a post-mortem trace-based analysis of the application and those that make a dynamic on the fly performance analysis and possibly tuning. The former should be used for applications with regular behavior because the performance analysis and tuning process must be done only once, and consequently, the overhead associated with this process will happen only once. However, for those applications that present different behaviors from one execution to another, or even in the same execution, the dynamic performance tuning approach can lead to better results.

In addition, the main requirements for a successful dynamic performance tuning process are low intrusion and having as much previous knowledge about the application as possible. Some times, this knowledge must be completely provided by the user (collaborative approach); some times, it should be partially provided by the user (automatic with knowledge approach); however, in both cases, there is a demand for significant user expertise, as well as the requirement of learning how to use the tool.

The objective of this thesis and our main contribution is to demonstrate that it is possible to define performance models associated to the application's structure suitable for integration in a dynamic performance tuning tool. This way, users can take advantage of developing applications using a supportive tool and can also transparently use a dynamic tuning tool without having to specify complex model parameters or interpret difficult analysis data and still get highly efficient results. From the point of view of final users, they are using a very efficient dynamic automatic tuning tool while from the tuning tool point of view (if it had one) the user is using an automatic with knowledge approach.

To fulfill this objective, we have chosen to develop the performance model for the Master/Worker and Pipeline frameworks because their popularity and usefulness. In addition, the models have been designed for the MATE environment. Therefore, we have analyzed the structure and functionality of those structures in order to determine the main performance drawbacks associated with these frameworks. Then we have developed the corresponding set of performance modeling strategies and expressions following MATE's subjacent performance model architecture (measure points + performance expressions + tuning points).

Finally, though the general idea has been very innovative and original there are several studies regarding definition of performance models associated with the application's structure. In the next subsection, we include a summary of the most relevant ones. Next, in section 4.2, we conclude this chapter with a description of the organization of this thesis.

4.1. Related studies

The group of parallel computing of the Statistics and Computer Science Department of La Laguna University has great experience in the development of programming tools based on frameworks, parallelizing compilers, and performance modeling at different application levels. In particular, they have developed models for Master/Worker applications on homogeneous [RR+98] and heterogeneous [AG+03] networks of workstations and also models for Pipeline applications on homogeneous [MA+01] and heterogeneous [AG+02] networks of workstations.

Their study for Master/Worker applications [RR+98] is mainly focused on explaining why there are differences in the message latency of an Ethernet network when using one to all communications in TCP and UDP protocols. With the objective of illustrating the theoretical model presented, a Master/Worker matrix multiplication application is used and its execution time modeled with the following expression:

$$P(\alpha_{BR} + \beta_{BR} N_i) + P(\alpha_{BR} + \beta_{BR}(N_i / P)) + DN^3 / P + (\alpha_{PP} + \beta_{PP}(N_i / P))$$

Where P is the number of processors (one worker per processor), N_i is the size of one matrix, N_i/P is the part of the second matrix sent to each worker, DN^3/P is the computation time of each worker (assuming homogeneous workstations), and α and β are the network latency and inverse bandwidth respectively (sub-indexes BR and PP meaning broadcast and point to point respectively).

It can be seen that this expressions has the form: Time for sending data (first two terms) + Time for computing (third term) + Time for receiving results (fourth term), which is basically the one we have used to define our Master/Worker performance model for balanced (almost homogeneous) Master/Worker applications. Obviously, we have a different objective: dynamic performance tuning vs. an example for a specific case of communication modeling; hence, we have analyzed many more cases and studied other aspects of the problem. The main problem is to estimate (dynamically) how many workers (processors) can be efficiently used by the application.

The people of La Laguna University have generalized this study to the case of heterogeneous Master/Worker applications [AG+03]. This time, the more ambitious objective was to assign the heaviest tasks to the fastest processors and communication channels. The basic idea is to define a more general expression that includes the execution time differences of workers; the resulting expression is the following one:

$$R_1 + C_1 + \sum_{i=1}^{p} S_i + \sum_{i=1}^{p} \max(0, d_i)$$

Where R_1 is the receiving time of worker 1 (one worker per processor), C_1 is the computation time of worker 1, S_i is the sending time of processor i, and d_i is the accumulated delay between the first i-1 workers and worker i. Communication times are calculated as in the expression for the homogeneous case by using the latency plus inverse bandwidth approximation. Computation times (C_i) should be estimated because this models is not intended to be used in a dynamic tuning environment.

It is clear that in order to get the best execution time, this expression has to be minimized, which means finding the appropriate mapping of tasks to processors and communication channels. However, this expression is too complex (basically due to the fourth term) to be solved analytically, and numerical approximations must be used instead.

We have adopted a radically different approach to heterogeneous Master/Worker applications modeling. Actually, we do not try to model this kind of application because heterogeneity means, in this case, inefficiency, and assigning heaviest tasks to the fastest processors will hardly eliminate it (only relieve it). We have designed, instead, some strategies to dynamically balance the workers' load, which result in significant performance improvements and in nearly homogeneous applications that can be treated following the homogenous model in order to adapt the number of workers.

Regarding their studies on performance models for Pipeline applications (homogeneous [MA+01] and heterogeneous [AG+02]) they have defined expressions that model the whole application live (filling in the pipe, all stages working, draining the pipe), which makes sense because their objective is to predict the application's behavior in order to make the best possible mapping of stages in the available processors. The specific tuning parameters considered in their model are the number of processors, the granularity (number of stages) to be assigned to each processor, and the size of communication buffers. The general target of their

model is to minimize the start up time (filling in the pipe), and balance processors load by grouping stages (calculating the granularity for each processor).

Although we share the main objective of eliminating load unbalances, we consider that filling in and draining phases are transient activities and, consequently, that performance inefficiencies associated with these phases cannot be solved dynamically and should be considered design problems (developers must assure that the number of data sets to be processed is significantly greater than the number of stages). Moreover, we have proposed replication of stages as the main solution to load unbalances instead of stage grouping, which we consider mainly as a mean of freeing processors. Therefore, our model, designed from the beginning to be used in a dynamic tuning environment, eliminates the complexity of modeling the start up phase of the application because it is not suitable to be tuned dynamically, but on the other hand, adds the complexity of modeling replicated stages. It is worth noting that the Pipeline performance model we present in this work is not as completely defined as the Master/Worker one and it does not consider grouped stages yet.

Another highly relevant group in this area is the Murray Cole's group of the School of Informatics of the University of Edinburgh. This group has a long-term experience in development programming tools based on skeletons (we have previously mentioned eSkel [Col04] library). Lately, they have developed an extensive work on performance modelling of skeleton-based parallel programs [BC+04]. They have realized that using skeletons carries with it considerable information about implied scheduling dependences and have decided to use process algebras (specifically PEPA [Hill96]) for modelling them.

In the study referred to, the pipeline skeleton is used to illustrate the overall idea and the developed tool. Although describing PEPA is beyond the scope of this work, we believe that it is intuitive enough and that it is worthwhile to include the description of the pipeline application model. The idea is to algebraically define all the application components (stages, processor, and network) in the following way:

- $Stage_i \overset{def}{=} (move_i, T) \bullet (process_i, T) \bullet (move_{i+1}, T) \bullet Stage_i$
- $\Pr ocessor_i \overset{def}{=} (process_i, \mu_i) \bullet \Pr ocessor_i$ if there is one stage per processor

  $\Pr ocessor_i \overset{def}{=} \sum_{j=k}^{l} (process_j, \mu_j) \bullet \Pr ocessor_i$ if from stage k to stage l are assigned to processor i
- $Network \overset{def}{=} \sum_{i=0}^{n} ((move_i, \lambda_i) \bullet Network)$

Where n is the number of stages, T (processing and moving times), μ_i (processor characterization), and λ_i (characterizes connections among stages and between the first and last stages and the user). Next, based on the previous constructions, the more general Pipeline and Processors structures are defined in the following way:

- $Pipeline \overset{def}{=} Stage_0 \underset{\{move_1\}}{><} Stage_1 \underset{\{move_2\}}{><} \ldots \underset{\{move_{n-1}\}}{><} Stage_{n-1}$
- $\Pr ocesors \overset{def}{=} \Pr ocessor_0 \| \Pr ocessor_1 \| \ldots \| \Pr ocessor_m$

Finally, an algebraic expression is built to define the mapping of the pipeline on the available processors and network (using the collaboration operator $><$):

- $Mapping \overset{def}{=} Network \underset{Lm}{><} Pipeline \underset{Lp}{><} \Pr ocessors$, where

  Lp = {process$_i$} (i=0..n-1) and Lm = {move$_i$} (i=0..n)

Once the model has been defined, the PEPA Workbench [Hae03] is used to calculate the application throughput providing μ and λ, and determining T. This way, it is possible to predict the best mapping of stages on a given network of processors.

The main difference between this approach and ours, besides the use of process algebras to define the programming structure, is that, at the moment, this is not intended for dynamic tuning. Therefore, their study is focused on improving performance by finding the best mapping for the pipe stages, which is a task that should be carried out before starting the execution of the application. In fact, their current objectives are to provide a tool for automatically generate the algebraic description of a specific application and to find the way of getting a good estimation of the model parameters.

4.2. Organization of this thesis

We have organized the contents of this thesis into 5 chapters, being the first one this introduction.

In Chapter II, we introduce a more detailed description of the dynamic tuning environment MATE and the Master/Worker and Pipeline frameworks with the objective of formalizing the structure and objectives of the performance models that will be presented later on.

Next, in Chapter III, we present a very detailed description of the performance model developed for dynamically tuning Master/Worker applications. This is a two-phase model consisting of a strategy for balancing the workers' load, and an analytical model for adapting the number of workers of the application. The results

of a wide set of experiments are presented in order to validate this performance model.

Then, in Chapter IV, the performance model proposed for dynamically tuning Pipeline applications is introduced. The main objective of including this model in this study is not only the analysis of Pipeline applications but also to demonstrate that it is possible to define models for frameworks other than Master/Worker.

Finally, in Chapter V, we present the conclusions and summarize the main achievements of this thesis and indicate what are, in our opinion, the most relevant open issues and challenges to be faced in the future.

Chapter II: Dynamic Automatic Performance Tuning Based on Application Structure

Abstract

The aim of this chapter is to describe the structure of the performance models that will be presented later and the aims of those performance models. To achieve these objectives, we introduce a brief description of the MATE dynamic tuning environment, a tool that provides a framework for developing dynamic tuning tools and, consequently, requires the performance knowledge to be provided from outside. We also introduce a description of the structure and functionality of the Master/Worker and Pipeline frameworks in order to highlight their advantages, but also their performance bottlenecks suitable to be dynamically overcome.

1. Introduction

The main objective of this work, as we mentioned in the previous chapter, is to demonstrate that it is possible to develop a performance model associated with the structure of the applications suitable for use in a Dynamic and Automatic Tuning Environment. Consequently, before introducing the performance models developed for the selected structures, we want to describe in detail the underlying monitoring, analyzing, and tuning model, as well as the structural and functional characteristics of the Master/Worker and Pipeline frameworks, because they are the determinant elements for the definition of the proposed performance models.

On the one hand, the monitoring, analyzing, and tuning model determines the structure of the performance model and its capabilities and limitations. It is not the same to use a static approach as to use a dynamic one. In the former, a longer and deeper analysis can be done, but in the latter performance improvements are obtained earlier. Moreover, it is not the same to use a model based on dynamically selecting the appropriate problem solving strategy (like Active Harmony [TC+02]), or another based on dynamically discovering and overcoming the application bottlenecks (like MATE [MM+03]). On the other hand, the specific contents of the performance model will depend on the framework used to develop an application because there will be different performance targets for each framework depending on its dynamically solvable performance associated bottlenecks.

In conclusion, we have to describe the selected monitoring, analysis, and tuning model in order to establish the general structure of the performance models we are going to develop later. In addition, we have to describe the selected frameworks in order to identify the objectives that will guide the development of those performance models, and also justify why we have chosen these frameworks. Consequently, in this chapter, we include in section 2 a detailed description of our target model for dynamic and automatic performance tuning. In section 3, one finds the description of the Master/Worker framework. Section 4 gives the description of the Pipeline framework. Finally, in section 5, we summarize the structure and requirements for the performance models that will be defined in the following chapters.

2. Monitoring, Analysis, and Tuning Model

In this section, we describe the automatic performance analysis and dynamic performance tuning model of the *Monitoring Analysis and Tuning Environment*

(MATE) [Mor03] that determines the framework for the structure of the performance models that are going to be described in the next chapters. This model has been designed and implemented by the *Parallel and Distributed Applications Performance Group of the Computer Architecture and Operating Systems Department of the Autonomous University of Barcelona* and, as well as other projects, such as *Mirror Object Steering System* (MOSS) [ES98], *Autopilot* [RS+01], and *Active Harmony* [TC+02], is inspired by the necessity of improving the performance of some applications dynamically, and it is based on the possibility of dynamically inserting instrumentation into the application code without re-compiling or re-linking it.

As previously mentioned, there are applications with a dynamic behavior that cannot be successfully improved through a static performance analysis and tuning approach. These applications can be instrumented on the fly using an instrumentation library like *Dyninst* [HB02] in order to obtain some relevant measures. These measures can be used for analyzing the application performance in parallel with its own execution and, if the analysis discovers any performance drawback, some application parameters can be modified to overcome it (or them).

Classically, it has been considered that this process can be performed using a *cooperative* or an *automatic* approach. In the former, the application developer collaborates with the tuning process by providing information online about what should be measured, how it should be analyzed, and what can be changed. This approach is simpler, powerful, and the instrumentation can be static, however, it demands a high degree of expertise from the developer and the application must be prepared for being tuned. In the latter, the application is treated as a black-box and the tuning application tries to discover and solve some common problems. This approach has the advantage of being completely independent of developers, but the lack of knowledge about the application being tuned usually limits the complexity of the problems that can be tackled, and, consequently, the tool's potentiality for getting significant performance improvements. Moreover, this approach needs dynamic instrumentation and, in consequence, introduces some degree of intrusion that can modify the application behavior, making it difficult to know if the problem that is being solved has been introduced by the tuning tool.

We have introduced a third approach that we have called *the automatic with knowledge approach* because, on the one hand, there is no collaboration demand for the developer and the application is still a black-box, but on the other hand, the application architecture and functional structure is known because it has been

developed using a known framework. This approach can lead to significant performance improvements because it can deal with complex problems: in addition, it can be implemented with or without dynamic instrumentation, though in the second case intrusion can be minimized. Nevertheless, as there are several frameworks and also several programming environments based on frameworks, it is required that the tuning tool be flexible enough to allow it to be adaptable to these different possibilities.

The MATE environment fulfills this requirement by forcing an environment independent specification of the performance knowledge. This means that MATE implements the architecture of a dynamic analysis and automatic tuning tool, defines the components of the performance knowledge, and how it should be specified, but MATE does not provide any performance knowledge by itself. This makes it suitable for developing several different strategies of dynamic tuning, depending on the approach (cooperative, automatic, or automatic with knowledge), and also on the element on which it is applied (the application itself, the application framework, the libraries used by the application, or the system libraries).

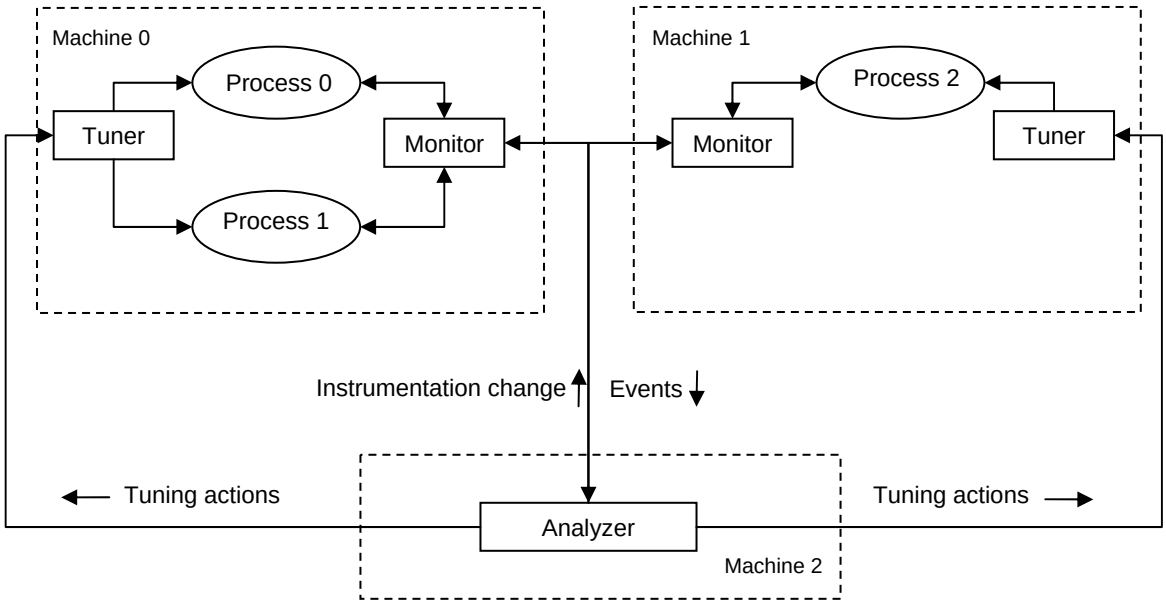


Figure 1. MATE steering loop.

The architecture or steering loop implemented in MATE is composed of three main modules: the *Monitor*, the *Analyzer*, and the *Tuner*, as shown in figure 1. The Monitor module (also called Tracer) is responsible for gathering all the events produced during the application execution; in consequence, there must be a monitor

in any machine hosting application processes. The Analyzer module is responsible for the automatic and dynamic analysis and tuning of the application and also it is responsible for informing the user about problems detected and actions undertaken. Finally, the Tuner is responsible for inserting modifications into the running processes for overcoming performance bottlenecks; consequently, as in the case of the monitor, there must be a tuner in every machine hosting application processes.

With the aim of presenting the performance model architecture associated with MATE, we describe in the next few subsections the functionality of those modules, their interrelations, and the parameters that should be defined for each one.

2.1. Monitors

This module has to gather all the relevant events produced during application execution, in order to monitor each application process, which means that there should be an instance of the monitor in every machine hosting an application process.

Consequently, the environment must be aware of every newly created process, as well as every new machine available to the application. In the case of the current MATE implementation on the PVM library, it takes advantage of the *tasker* and *hoster* services of this library in order to catch every request for creating new processes or adding new hosts to the virtual machine. That way, when the creation of a new application process is requested, the Monitor residing at the corresponding host catches it and performs the appropriate steps to create an *instrumented* process, which is the requested process plus extra calls to monitor the desired events. Moreover, when the addition of a new host is requested for the master PVM daemon, the Monitor residing in the same host (the *Master Monitor*) catches it and performs the appropriate steps to add a *monitored* host, which is the requested host with a Monitor module running on it.

Using the *tasker* and *hoster* services solves the problem of tracking all the application processes and available resources, but there are still a couple of problems to be solved. In the first place, temporal relationships between events gathered by different monitors must be preserved in order to reach right conclusions in the analysis regardless of any possible differences between local clocks. MATE implements a global timestamp scheme where each local Monitor is responsible for synchronizing its local clock with the one of the Master Monitor. Secondly, there must be a mechanism to indicate the events that should be monitored. Monitors use

the DynInst library to insert instrumentation dynamically in the monitored process code. This instrumentation is a piece of code (snippet) consisting of a call to an interface library for describing what the event is, when it happened, and where it was registered. In addition, Monitor is responsible for sending the gathered information to the Analyzer, thereby minimizing the intrusion into the network by means of event buffering and aggregation.

Finally, as MATE does not provide performance knowledge by itself, the definition of the points of the application process that should be instrumented by the Monitor depends on the performance model provided. These points are called *measure points* and the Monitor is instructed by the Analyzer about which points should be instrumented upon the creation of an application process. Moreover, it is possible to change these measure points during runtime, depending on the execution conditions and performance drawbacks detected. Consequently, any performance model defined for MATE must include a set of *measure points* that indicate what information should be gathered to evaluate the model.

2.2. Analyzer

The Analyzer is the module responsible for the dynamic performance analysis of the parallel/distributed application. It must be able to examine the application behavior, identify persistent performance bottlenecks, and provide the solutions for overcoming them. In addition, the Analyzer must inform users about the problems detected and actions undertaken.

The functionality of this module is composed of two steps: an initialization one and the evaluation one. In the first one, the Analyzer uploads the performance knowledge provided and sets up the necessary Monitor and Tuner modules. In the second, the Analyzer receives the events gathered by Monitors and evaluates the application behavior using a set of *evaluation expressions and/or strategies*; if it detects a persistent performance problem (transient ones not worth solving), then it tracks the causes of the problem down (it may need to instruct Monitors to insert more instrumentation), and finally it sends to the appropriate Tuner(s) the changes that have to be made in the application process(es) to overcome the problem.

The current MATE implementation has only one Analyzer process that is responsible for doing a centralized performance analysis of the whole application. This approach works well for medium-sized applications but is difficult to scale for large ones; in addition, it is likely to be highly intrusive if the Analyzer is hosted in the

same machine as other application process(es). To solve the first problem, a distributed analysis scheme is being designed and, to solve the second problem, the Analyzer is executed on a dedicated node.

Finally, the *evaluation expressions and/or strategies* used to evaluate the application's performance must be part of any performance model defined for MATE and are provided in the form of evaluation routines written in C++ called *tunlets*.

2.2. Tuner

Tuners are responsible for dynamically inserting modifications into the application processes. Consequently, a Tuner instance must be present in every machine hosting an application process. Actually, in the current MATE implementation the Monitor and Tuner modules are integrated in a single process controlling all the application processes executed on a given node.

When the Analyzer detects a performance problem and finds its solution, it sends a tuning requirement to the appropriate Tuner instance(s), then each Tuner applies the modification dynamically to the corresponding process(es). A tuning requirement is composed of a target process, a tuning point, a tuning action, and a synchronization method. The tuning point specifies what must be changed in the given process, the tuning action is the command to be performed on that point, and the synchronization method specifies the conditions that must hold for the tuning action to be performed in order to keep the application's consistency.

The tuning actions offered by MATE are:

- Changing the value of an application variable.
- Function replacement. A function call can be replaced with another with the same signature.
- Function invocation. A function call is inserted into the application code and it will be invoked each time this code section is reached.
- One-time function invocation. A function call is done at the specified point but the invocation code is not inserted in the process code.
- Changing the arguments of a function call.

Finally, as in the cases of the Analyzer and Monitors, the performance knowledge components of the tuner, the *tuning points*, the *tuning actions*, and the *synchronization method* must be specified by the performance model defined for MATE.

In summary, the MATE architecture determines that any performance model defined for this dynamic tuning environment must consist of a set of *measure points*, which are the inputs to the model to be monitored; a set of *evaluation strategies and/or expressions* used by the Analyzer to find and solve performance bottlenecks through their evaluation on the inputs; finally, a set of *tuning points and actions* that the Analyzer sends to the Tuners for dynamically introducing changes into the application in order to overcome the performance problems detected.

3. Master/Worker Framework

3.1. Framework structure and functional description

The Master/Worker framework consists of a Master process and a farm of Worker processes. The Master is responsible for decomposing a problem into a set of tasks, for distributing them among the Workers, and for gathering the results produced by those Workers. Each Worker process gets messages with tasks, processes these tasks, and sends back the results to the Master.

The Master/Worker model is used in many scientific, engineering and commercial applications, such as software building and testing, sensitivity analysis, parameter space exploration, image and movie rendering, high energy physics event reconstruction, processing of optical DNA sequencing, training of neural networks and stochastic optimization among others [Can98, WW95, AI98].

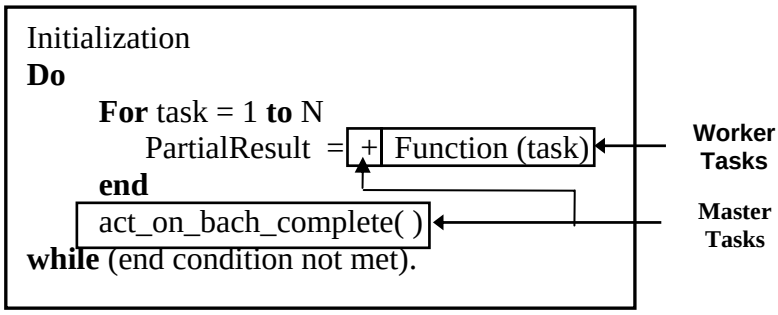


Figure 2. Generalized Master/Worker algorithm

There are several reasons that made this framework quite popular; among them as we will show later, it can be used to implement many of the patterns described in the introductory chapter; or that it is suitable for programs to be executed in networks of workstations because with this framework it is possible to implement programs that dynamically adjust the computation load between units of execution,

or that Master/Worker programs work particularly well on heterogeneous networks, since faster or less-loaded processors naturally take on more work.

The general Master/Worker functionality can be summarized with the algorithm of figure 2. It can be seen there, that the process of distributing tasks among workers and waiting for results can be repeated several times in an iterative way. This functionality, represented graphically in figure 3, makes this framework suitable to implement many of the parallel patterns discussed in the previous chapter.

This way, the *Separable-Dependences* pattern, which can be used for task-based decompositions, in which dependences between tasks exist but can be pulled outside of the concurrent computation of tasks, can be implemented with this pattern by letting the Master do some data replication before distributing tasks and some results aggregation at the end the iteration. Moreover, the *Geometric-Decomposition* pattern, which can be used for data-based decompositions, in which a core date structure is decomposed in not completely independent chunks that can be updated concurrently, can also be implemented with the Master/Worker framework if the Master can cope with dependences at the beginning of each iteration. Finally, the *Embarrassingly-Parallel* pattern, which can be used for task-based decomposition as well as data-based ones in which tasks are completely independent of each other, is naturally implemented with this framework.

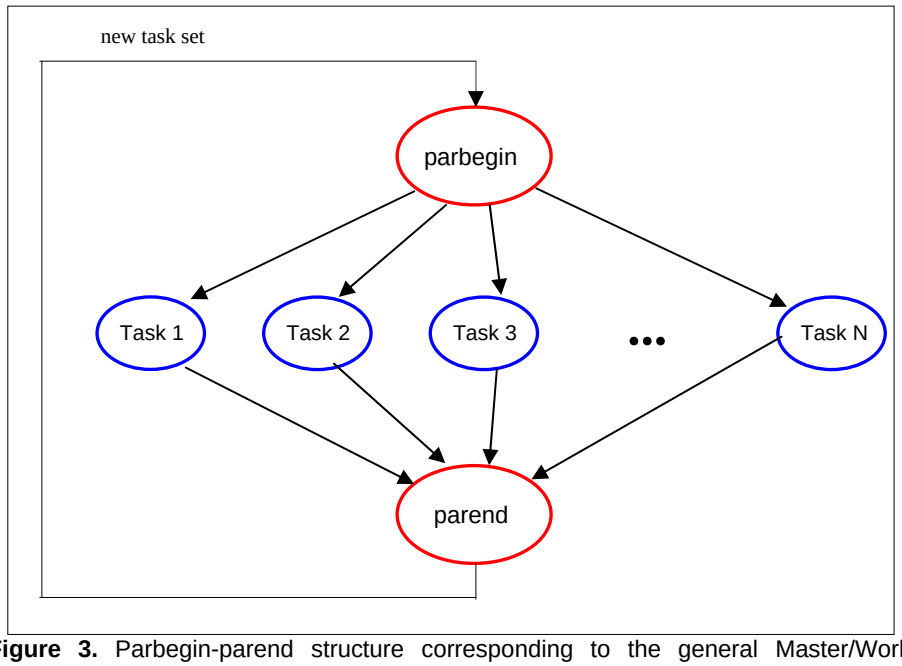


Figure 3. Parbegin-parend structure corresponding to the general Master/Worker framework.

Some simple real application examples of each pattern could be mentioned, such as numerical integration of a given function over a given interval using the trapezoid

rule for the *separable-dependences* pattern because each worker can calculate the integral of some sub-interval while the Master aggregates the results, or matrix multiplication using a block decomposition for the *geometric-decomposition* pattern because matrixes are decomposed in blocks that are sent to workers that perform a classical matrix multiplication on them while the Master aggregates the results, or vector addition for the *embarrassingly-parallel* pattern because the vector to be added can be decomposed in chunks that can be independently added.

The main reason we have principally focused on this framework in developing our work is because it is so useful and widely used. Actually, we have developed not only its performance model, but also an implementation of the framework [Mes04] aimed to be used for carrying out experimentation on real applications [MCe+05, MC+05], such as the classical N-body and a forest fire propagation simulator named xFire [JM+98].

3.1. Framework associated bottlenecks

As previously explained, one of the advantages of using the Master/Worker framework for developing an application is its flexibility in dynamically adjusting the computation load between the execution units, which is a significant characteristic in order to get good performance from the assigned resources, though this adaptation depends on a careful design and the characteristics of the application. Furthermore, a second relevant performance parameter for this kind of applications is to determine the appropriate number of execution units (processors) that should be associated with them.

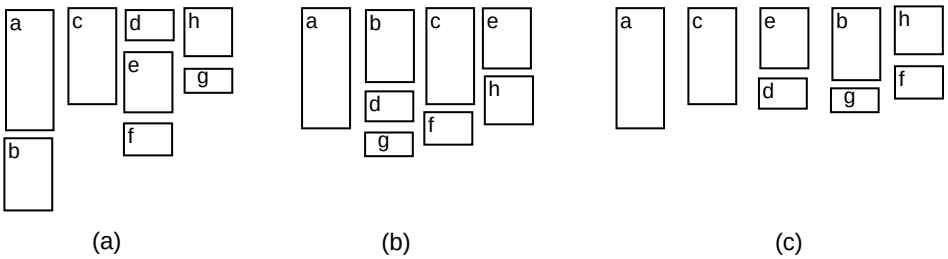


Figure 4. 8 independent tasks distributes among (a) 4 execution units with poor load balancing, (b) 4 execution units with good load balancing, and (c) five execution units with good load balancing.

Both characteristics are illustrated schematically in figure 4. In this figure, each box represents a task and the size of each box represents the computational requirements of the task: hence, it can be seen that a good load balancing is likely to improve the application's performance especially if tasks are of different lengths,

and also that adding more workers may lead to performance improvements especially if the computational requirements are high. However, adding too many workers could lead to wasted resources without improving the application's performance, as can be seen in the figure for case (c) where adding more workers cannot improve the application's performance due to the length of task *a*.

Moreover, the number of tasks and its associated computational needs are likely to vary during the execution of the application: hence, making these problems more difficult to solve. Consequently, load unbalance and an inappropriate number of workers seems to be the most relevant performance bottlenecks of these applications, and dynamically adjusting the workers' load, and dynamically adapting the number of workers the most significant challenges to improve their performance.

Nevertheless, there are other requirements for getting good performance from a Master/Worker application: in the first place, the number of tasks should be much higher than the number of workers because, on the contrary, balancing the load could be significantly more difficult to achieve; in the second place, the cost of initializing and sending a task to a worker must be much less than the cost of computing the task because, on the contrary, the advantages of concurrent processing are lost due to the cost of setting tasks up. These performance problems, however, are unlikely to be solved dynamically because they are produced by design pitfalls and not due to execution conditions.

4. Pipeline Framework

4.1. Framework structure and functional description

The Pipeline framework is a well-known parallel programming structure used as the most direct way to implement algorithms that consist of performing an orderly sequence of essentially identical calculations on a sequence of inputs. Each of these calculations can be broken down into a certain number of different stages, and these stages can be applied concurrently to different inputs. Many image treatment programs and the computation of the Fast Fourier Transformation (FFT) are suited to be implemented using this framework.

In figure 5 this description is schematically illustrated for a pipeline consisting of four stages, supposing that the calculations to be performed are named C1, C2, and so forth. Then the pipeline operation begins with the first stage performing its part of C1. When that is completed, the second stage of the pipeline performs its part of

C1, while the first stage simultaneously performs its part of C2. Next, stage 3 performs its part of C1, while stage 2 concurrently performs its part of C2 and stage 1 its part of C3.

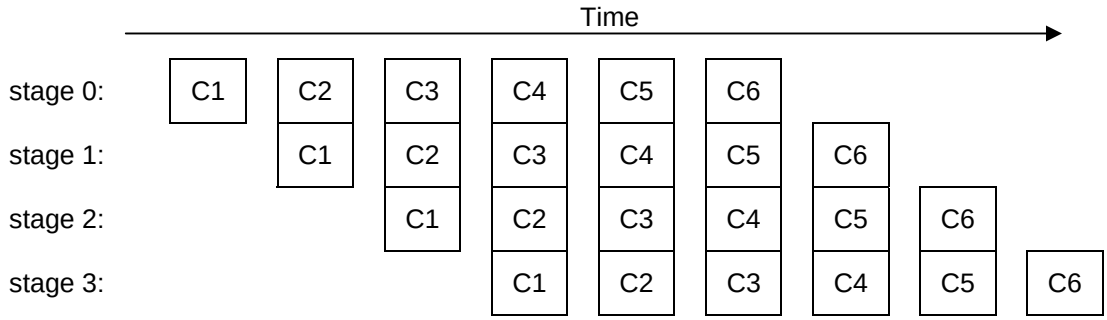


Figure 5. Schematic representation of the functionality of a four-stage pipeline over six calculations.

The structure behind this functionality is a linear arrangement of processes, like the one shown in figure 6 (a), in which each one is responsible for performing one or more stages. However, this structure can be extended, as in the example shown in figure 6 (b), to include situations in which some operations can be performed concurrently over different calculations. This suggests that a pipeline can be represented as a directed acyclic graph, with vertices corresponding to stages (or elements of calculation) and edges indicating dataflow. Clearly, this structure is suited to be implemented in a message-passing environment rather than in a shared-memory one.

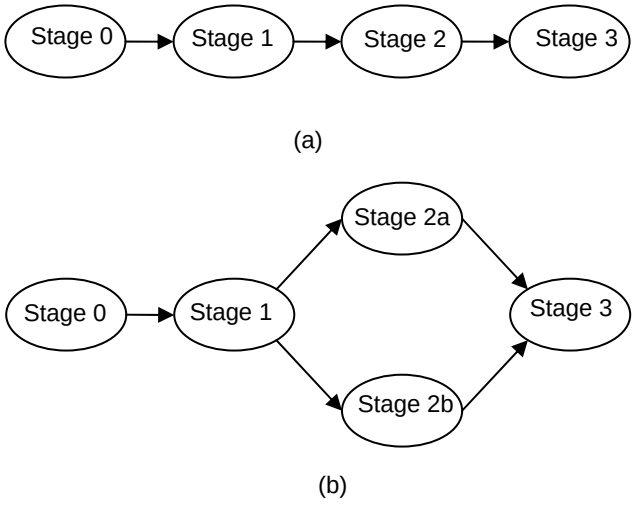


Figure 6. Structure of a four-stage linear pipeline (a) and four-stage non-linear one (b).

Finally, it can be seen that there are three different kinds of processes in a Pipeline structure, the *first stage process* that has a succeeding stage but not a preceding one, the *last stage process* that has a preceding stage but not a

succeeding one, and the *intermediate stage process* that has both a preceding and a succeeding stage.

4.1. Framework associated bottlenecks

It can be seen in figure 5 that at the beginning, while the Pipeline is being filled and the number of calculations is less than four, some stages are idle. In general, in an

n-stage Pipeline there are idle stages at least until n calculations have been pushed into the pipe. It can also be seen in the same figure that at the end, when the Pipeline is being drained processing the last four calculations, there are also some idle stages. In general in an n-stage Pipeline there are idle stages at least while processing the last n-1 calculations.

These inefficiencies in the filling and draining phases (also known as ramp-in and ramp-out phases) of the execution of a Pipeline application cannot be avoided in the straightforward implementation of the structure but can be minimized if the number of calculations is large compared to the number of stages. Clearly, ensuring this condition is a design problem that cannot be dealt with dynamically because it is, in any case, a transient problem.

However, there is a second source of performance inefficiencies in Pipeline applications. Suppose, as shown in figure 7, that in an n-stage pipe every stage except one takes roughly the same amount of time to perform its part of calculation and the exception takes three times more than the rest. It is easy to see that all the stages that come after the slow one are idle two thirds of the time.

Consequently, as the number of calculations per unit of time or throughput of the Pipeline application is determined by the pace of the slowest stage of the pipe, it is important to avoid significant differences between the computational efforts of pipe stages. This is a load balancing problem that can be solved in the design phase of the application (statically) if the calculations to be performed are known in advance and their associated operation time even. What should be done in this case is to group faster operations in the same processor, in order to increase the resource availability, while replicating slower ones in other processors in order to increase their throughput. On the contrary, if the operation time depends on the calculation and varies along the application execution the problem can be solved the same way but dynamically at execution time.

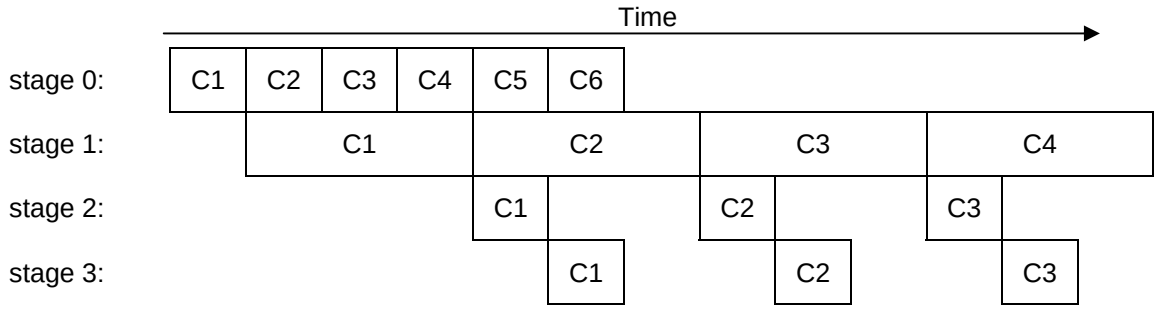


Figure 7. Unbalanced Pipeline execution.

5. Structure and Objectives of the Developed Performance Models

In the first place, this work has been developed in the same research group that has designed and implemented the MATE dynamic tuning environment and it is closely associated with MATE. Therefore, the structure of the defined performance models is the one required by this environment; hence, the models that have been developed and that will be presented in the next chapters consist of three main parts:

- The **measure points** or set of application parameters that should be monitored in order to evaluate the performance expressions and/or strategies.
- **Performance expressions and/or strategies** are the main components of the performance model and must be designed with the aim of describing the behavior of the application in order to dynamically detect its performance bottlenecks and generate the appropriated actions to overcome them.
- The **tuning points and actions** are the set of parameters and changes of these parameters used for improving the application's performance. These actions should also include a set of safety conditions that indicate when these actions can be applied without altering the application's consistency.

In the second place, we have chosen to develop the performance model of two popular and well-known frameworks: the Master/Worker and the Pipeline. The Master/Worker framework is widely used because of its adaptability to clusters of workstations and its flexibility in implementing several application design patterns, while Pipeline framework is very useful for exploiting the concurrency of algorithms

consisting of the application of an orderly sequence of calculations over a stream of data.

Moreover, the main objective of the developed performance model associated with the Master/Worker framework must be to balance the load of workers in order to obtain the best performance from the available resources, but also to determine the number of workers that should be used to improve the application's performance, whereas the main objective of the performance model developed, associated with the Pipeline framework, must be to determine what stages should be replicated and how many replicas of each one should be introduced in order to improve the application's throughput and, consequently, its performance.

Finally, we want to highlight that, as a consequence of focusing on dynamic tuning and the performance model architecture we are using, a generic framework analysis methodology can be established. This methodology consists firstly in identifying those performance problems suited for being solved dynamically, then in finding or defining the magnitudes that should be monitored in order to identify those problems (inputs) and those that had to be changed in order to overcome them (outputs); finally, in building the set of performance analysis strategies or expressions that make it possible to determine, based on the defined inputs, the best performance improving actions (outputs).

Chapter III:
Master/Worker Framework Performance Model

Abstract

In this chapter, we present the performance model we have defined to dynamically improve the performance of applications developed with the Master/Worker framework. The objective of this model is to improve the application's performance in two phases in the first one the worker's loads are balanced in order to make efficient use of the available resources, while in the second phase the number of workers associated to the application is evaluated in order to determine if the application's performance can be significantly improved by adding more workers. We also present a set of experiments that validate both phases of the model.

1. Introduction

In this chapter, we will introduce our proposal for a performance model associated with the Master/Worker framework. To fulfill this objective we will first recall the framework associated performance problems presented in the previous chapter, and then discuss in which order they should be solved to improve the overall application execution time. Finally, we will devote the rest of the chapter to define, according to the general performance model introduced in the previous chapter, the strategies and expressions that are part of our performance model and the experimentation that has been carried out to validate them.

In Chapter II, we have shown that Master/Worker applications could suffer from two main performance bottlenecks related to the framework structure and functionality: the first one was the load unbalance of the workers, which could produce long idle times for quick or lightly loaded workers, and the second one was the use of an inadequate number of workers (too short or too large) to process the set of tasks at hand.

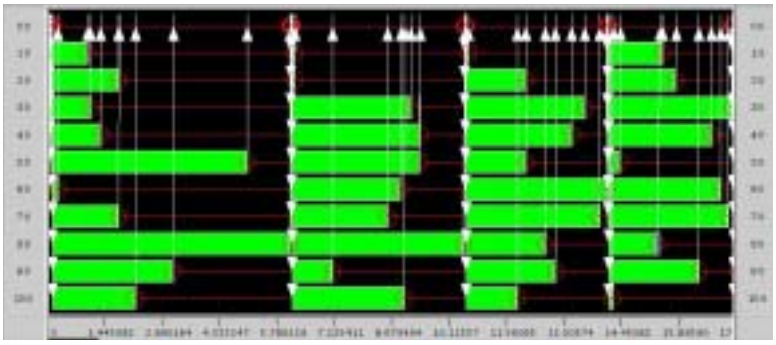
It is worth noting that both problems may depend on dynamic conditions, such as the amount of available tasks or the processors' load; thus, they are suitable for being solved dynamically.

In figure 1, we can see that balancing workers' load leads to significant performance improvements. Here, we show the execution trace file of a synthetically generated Master/Worker application with 10 workers, processing 10000 tasks per iteration, and each task with an associated mean processing time of 2 ms. and a standard deviation of 1.6 ms (80%).

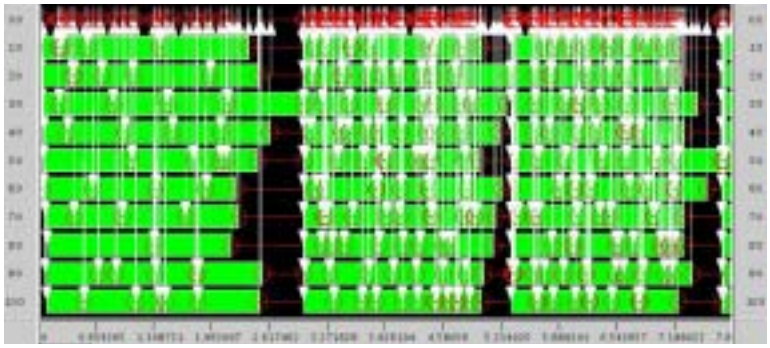
For the first execution trace (a), the Master is distributing the whole set of tasks at the beginning of each iteration, and we can see how several workers are resting idle too much time (green areas mean worker busy while black ones mean worker idle). However, we can see in (b) how a simple change in the task distribution policy leads to a better behavior of the application.

In addition, when the load unbalance between the different workers of the application is high, adding more workers has a limited positive impact on the application performance (actually it is likely to have a negative one) as we can observe in figure 2, where we show the execution time of a Master/Worker application, which processes 1024 1Kbyte tasks each iteration, with each task having an associated processing time distributed accordingly to table 1, and using an increasing number of workers (from 8 to 52). To allow comparison, we have also

included the best expected execution time for a balanced application of the same characteristics, which is basically the overall execution time divided by the number of workers plus a communication overhead.



(a)



(b)

Figure 1. Unbalanced (a) and balanced (b) Master/Worker application. It is an application with 10 workers, processing 10,000 50byte tasks with an associated mean processing time of 2 ms. each and a standard deviation of 1.6 ms. (80%)

It is clear that we are only able to take advantage of the extra available resources up to 20 processors; this is because the tasks that are assigned to each worker are chosen randomly among the groups with available tasks, meaning that there is a uniform probability of choosing a task from any group. When the number of processors is low, each one receives many tasks given more chance to be well assorted with every kind of them (short ones and large ones), in a proportion similar to the one of table 1. On the contrary, when the number of processors is high the number of tasks received is lower and it is likely that a processor with bad luck, i.e. one that receives several large tasks, would stall the whole application.

In contrast, it is possible to get large improvements from adding workers to a balanced application, as the best expected execution time function of figure 2 suggests. Moreover, we will show that it is possible to predict this improvement quite accurately. As a consequence, it is clear that, in order to improve the efficiency of

the decision about the number of workers of our application, this step should be preceded by a load balancing phase.

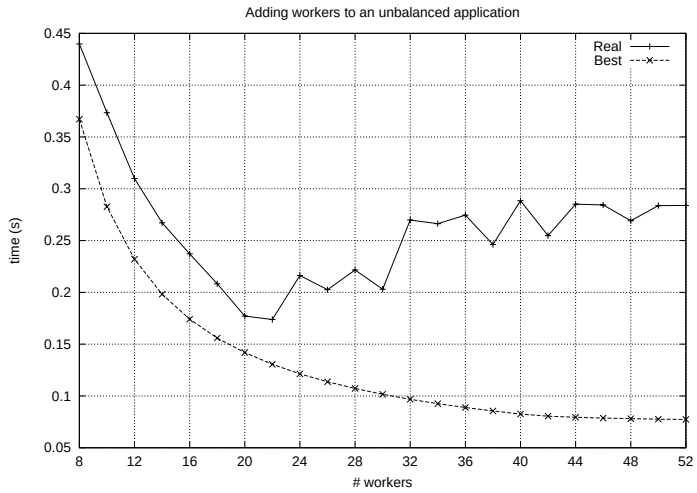


Figure 2. Real and best expected execution time of a Master/Worker application using from 8 to 52 processors, processing 1024 1Kb tasks each iteration, and task associated processing time distributed accordingly to table 1.

Number of tasks	% Over total (1024 tasks)	Associated computation time (ms/task)	% Over iteration execution time (2 s)
205	20	0.6	10
154	15	1.2	10
307	30	1.7	25
154	15	2.3	15
102	10	3.4	15
102	10	5	25

Table 1. Distribution of task processing time for the application shown in figure 2.

Consequently, in the rest of this chapter, we introduce in the first place the discussion of different strategies for achieving a good load balance for the application, including the experimental results that validate them; next, we present the set of expressions that model the performance of a homogeneous Master/Worker application, making it possible to dynamically calculate the appropriate number of workers to be used by the application; then, we present the formal analysis that works as a glue among the load balancing strategies and the model for calculating the appropriate number of workers; finally, we include a summary of the Master/Worker model for dynamic performance tuning.

2. 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 a 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, nevertheless, we did not propose the first, mostly informal, solutions until [CM+04] and [MCa+05]. Lately, we have studied it in much more depth [MoC+05]. Our main source of inspiration for coping with this problem has been the works about the distribution of parallel loops, like [KW85], [FSF92], [BV01], and [BV02]. They define many policies aimed at balancing 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 of the task distribution problem could be stated in 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 in 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 which have received tough tasks will not receive more load and workers which 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 a better load balancing with little computation and communication overheads. We have adapted, implemented and analyzed three different strategies that will be treated in deep later in this chapter:

- *Fixed Size Chunking (FSC)*, which consists of dividing the set of tasks in 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 to

determine 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 depending on the current load balancing conditions.

2.1. Fixed Size Chunking (FSC)

This is the simplest strategy intended to improve the load balancing of the application. As we illustrate in figure 3, it consists of dividing the set of tasks into a series of batches of the same size then the tasks of the first batch are distributed among workers in equal size 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 [KW85] to the M/W task distribution problem in [CM+04].

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 5 batches, each one including 20% of the 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 tasks processing time is large enough, a lower factor must be used with the objective of getting a finer grained distribution of tasks in order to minimize load unbalancing.

In figure 3, we show a hypothetical case of a Master/Worker with three workers and 12 tasks to be computed. Tasks are distributed in two batches of 6 tasks each, implying chunks of two tasks for each worker. For this example, we get a performance improvement of 16.67% from the application of the policy because a complete distribution of tasks would have produced a total execution time of 12 time units, since the tasks labeled 4, 1, 5, and 2 (total execution time of 12 time units) would have been assigned to worker 0.

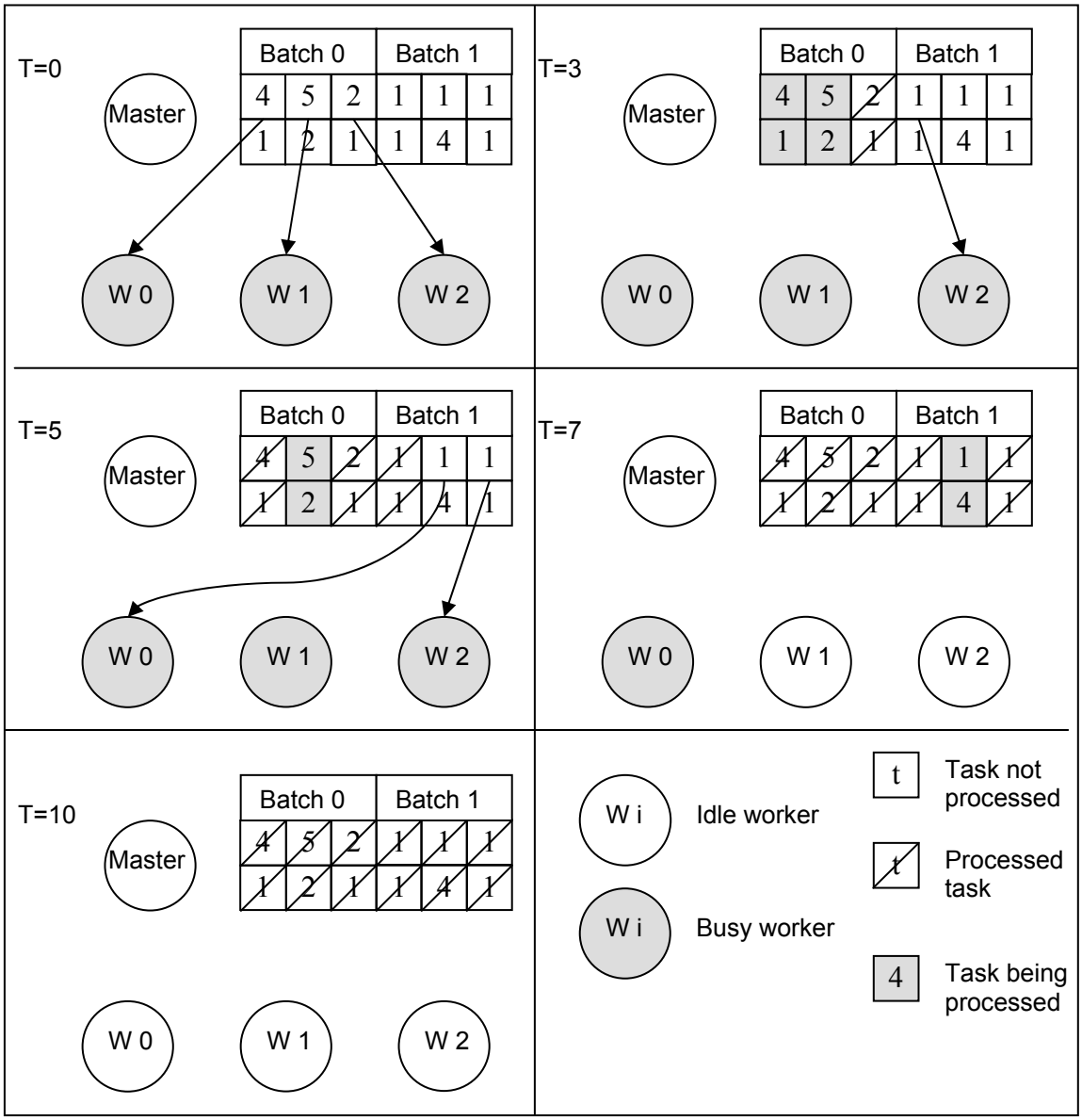


Figure 3. Schematic execution of an application using the FSC distribution policy. In this case, a factor of 0.5 has been used to generate two batches with three chunks of two tasks each.

However, we can also see in this example one of the main drawbacks of this policy, which is its lack of sensitivity to load unbalances in the last batches for relatively high partition factors. We can observe that if we had chosen a factor of 0.25, which would have lead to the distribution of one-task chunks, the total execution time, not including communications, would have been of 9 time units, only one unit over the best possible execution time of 8 (24 total processing time associated to tasks/ 3 workers).

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 different values of the partition factor.

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

1. For the first iteration of the application an arbitrary predefined partition factor of 0.25 is used. With such a low partition factor we are being pessimistic about the tasks 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 their future 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 have 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 of 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. Using the calculated partition and mean time and 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, the process is done.

3. For the third and subsequent iterations we use the partition factor proposed by the simulation process described above 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 with the tasks.

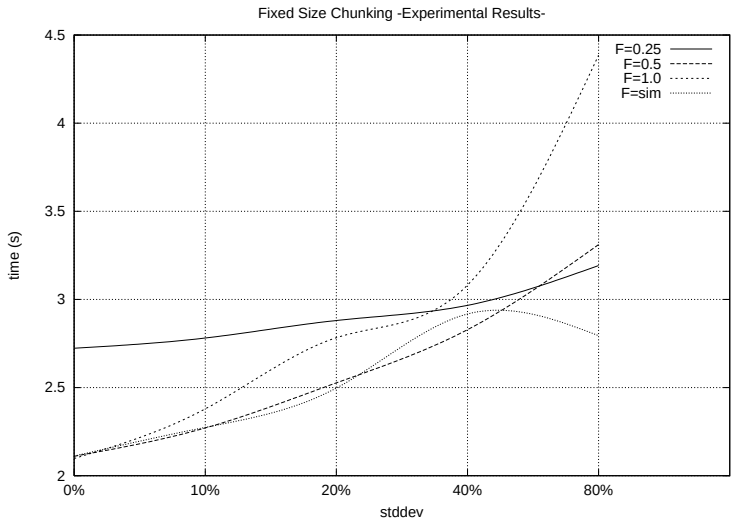


Figure 4. Execution times of a 10-worker application using the FSC distribution policy with different degrees of standard deviation and different partition factors.

Some experimental results of the application of this policy are shown in figure 4. In this figure, we show the execution times of a configurable Master/Worker application with 10 workers for different standard deviation values for the processing time associated with tasks. The application executes 15 iterations of 10,000 tasks each, the mean processing time associated with each task is 2 ms. which leads to a global processing time of 20 sec/iteration and a lower bound of 2 sec/iteration with 10 workers. The size of each task is 50 bytes, which means that the master is sending 50 bytes/task to the workers and the workers are replying with 50 bytes/task to the master, this makes a total communication volume of 10^6 bytes. Finally, we have included in the figure the results of using a constant partition factor of 1.0, 0.5, 0.25, and also those obtained from the use of the estimation algorithm described above.

We can see in this figure that for higher values of the standard deviation we are getting, in general, higher execution times, and also that higher partition factors lead to better results for lower values of the standard deviation but worse for higher ones, while lower partitions factors are working well for higher values of the standard deviation and poorly for lower ones. Finally, we can see that the results obtained by the dynamic estimation of the partition factor are as good as, or better than, the best ones of the fixed partition factors.

2.2. Dynamic Predictive Factoring (DPF)

We have observed that the previously described FSC policy lacks the necessary sensitivity to deal with load unbalances introduced by the lasts 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 [FSF92], which is a partition policy based on assigning large chunks at the beginning of the iteration and small ones at the end, was first proposed in [Mor03] and further developed in [MCa+05] with the name of Dynamic Predictive Factoring (DPF).

The general idea of this policy, as shown in figure 5, 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-size 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 on the whole set of tasks to calculate the number of tasks of the first batch, then on 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, we are assuming with this policy that sending bigger chunks at the beginning of an iteration not only is not likely 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 large chunk causes a significant deviation in the overall execution time. Consequently, as the iteration advances, the chunk size should be progressively reduced to minimize this effect, although the communication overhead is increased.

We illustrate this in the example in figure 5, there we can see that applying this policy to the same example of figure 3 with a partition factor of 0.5, we are getting the same total execution time than that of the best case of FSC (partition factor of 0.25), but with 3 (25%) messages less.

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 increment of 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 show a steady behavior. The estimation process is like the one described for the FSC policy, changing only the way batches are defined. Actually, this estimation process was first proposed in [Mor03] for the DPF policy.

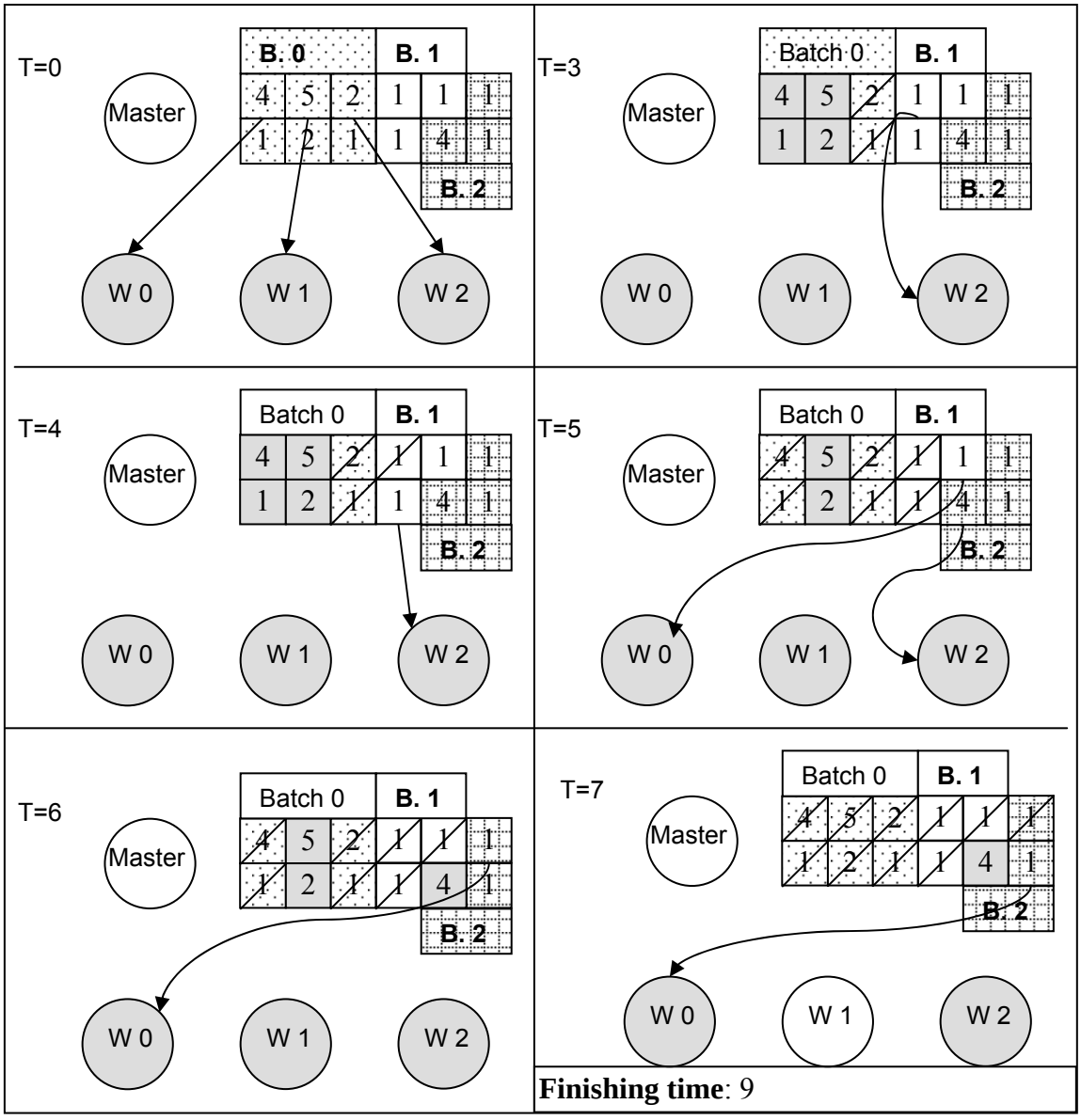


Figure 5. Schematic execution of an application using the DPF distribution policy. In this case a factor of 0.5 has been used to generate a first batch with three chunks of two tasks each, and two batches with three chunks of one task each.

Finally, in figure 6 we show some experimental results of the application of the DPF policy. The executed application has exactly the same characteristics as the one used for the FSC policy, i.e., same number of workers, same amount of data, same mean processing time and standard deviation, etc., with the only change being the distribution policy. This does not mean that all workers will process the same tasks as in the previous example because it depends on the distribution policy; actually, the set of tasks will differ because only the mean processing time and standard deviation is preserved.

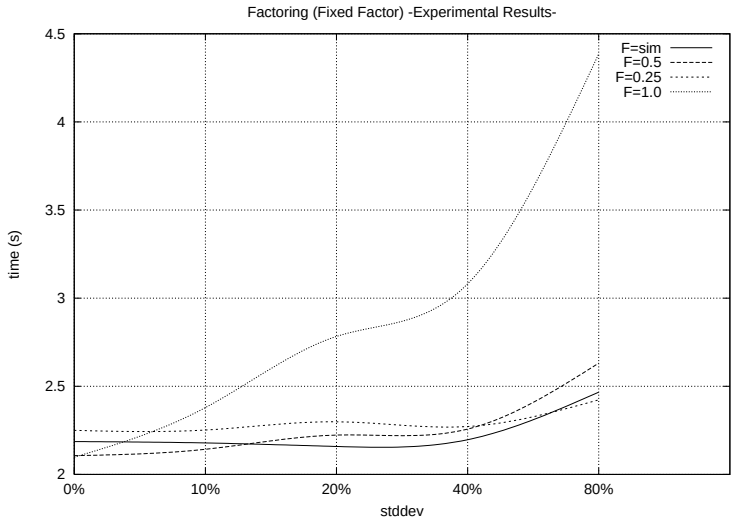


Figure 6. Execution times of a 10-worker application using the DPF distribution policy with different degrees of standard deviation and different partition factors.

We can see in figure 6 that we are getting acceptable results even for high values of the standard deviation in the task processing time. Even so, we could see that dynamic estimation of the partition factor seems worthwhile because it usually leads to better results for all displayed values of the standard deviation.

2.3. Dynamic Adjusting Factoring (DAF)

For the policies analyzed so far, we have seen that estimating the partition factor appears to be a promising approach to obtain nearly the best outcome from them. Even so, it should be noticed that 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. That 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 as good from these distribution policies.

Moreover, even for applications with a steady behavior, we have seen that the distribution policies present their worst results for higher values of the standard

deviation of the task processing time. It is because only the mean processing time of tasks is used for the estimation of the partition factor.

These were the reasons that motivated us to adapt the Dynamic Adjusting Factoring policy [BV02] (which is a partition policy that tries to adapt the partition factor immediately to the current conditions of the application) to the task distribution problem [MoC+05].

The original Factoring policy (intended for parallel loops 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 a mean execution 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 $M/(N*x_0)$, 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:

$$x_0 = \left(\mu + \sigma\sqrt{N/2}\right) \Big/ \mu \qquad \textbf{(1)}$$

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 whatever the state of the remaining workers may be.

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 for the currently remaining tasks (R_j) is proposed in order to compensate for the differences in the processing starting times of those chunks. The restriction over 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 $\mu F_j + \sigma F_j \sqrt{N/2} = \mu\left[\left(R_j / N\right) - F_j\right]$.

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

$$x_j = \left(2\mu + \sigma\sqrt{N/2}\right) \Big/ \mu \qquad \textbf{(2)}$$

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

1. With the goal of accumulating enough information to compute the adaptive factor, the first iteration will be executed using the DPF partition method with a fixed factor of 0.5. This initial factor is arbitrarily 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 as 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.

Before showing an example of the application of the DAF tasks distribution policy, we want to introduce some comments about the tasks per chunk lower limit that we mentioned above, as well as in the discussion of the DPF policy. Furthermore, some of these observations are also valid for the FSC partition policy.

At first, it seems pretty clear that a chunk should include a least one task and a batch should include as many chunks as there are workers, thus a batch will include

at least as many tasks as there are workers. There are some exceptions caused by rounding effects, for example, if we have 1123 tasks and we are using the FSC policy with a partition factor of 0.2, then we will have five batches with 224 tasks ($\lfloor 1123*0.2 \rfloor$) and 3 tasks left that can be used to build an extra 3 task chunk. The bottom line of what we are saying is that, mainly for DPF and DAF policies, the partition method will not be applied if the number of remaining tasks is about to fall below the number of workers.

However, in the end, we are going to face performance inefficiencies far before reaching this limit because the defined policies do not always directly consider some relevant application features, such as the communication cost and the Master's chunk managing capability.

The 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. Nevertheless, taking into consideration the Master's chunk managing capability will strongly minimize this problem.

We define the *Master's Chunk Managing Capability* as the maximum number of chunks that the Master can manage from the time it sends a chunk to a worker until the moment if 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, in the end, a communication capacity problem, which mainly depends on the number of workers, the chunk size, and the mean task processing time.

We will get back to this problem in much greater detail in the second part of this chapter, when we discuss the estimation of the number of workers. We can say now , however, that if the number of tasks per chunk causes the Master to fall below its chunk managing capability it will not be able to feed all workers and, as a result, there will be some idle workers.

The good news is that it is possible to estimate the number of tasks that will produce this inefficiency and, consequently, we can use this number as a lower bound for the number of tasks that will be included in each chunk.

Coming back to DAF policy, in figure 7, we show some experimental results of the application of that policy. The example is again the same one used to illustrate the previously defined policies, but this time we only show the execution with and without applying the distribution policy. We can see in the figure that we are getting

results not only very close to the theoretical optimal execution time, but also that they are little affected by the high values of the standard deviation.

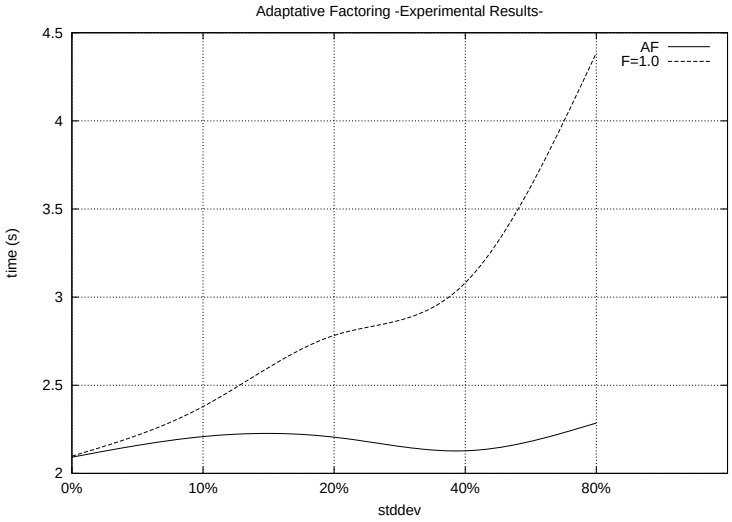


Figure 7. Execution times of a 10-worker application using the DAF distribution policy with different degrees of standard deviation.

Finally, in figure 8 we show the comparison of the execution times we have obtained with the application of the three policies described to the same 10-worker example, though for the DPF and FSC policies we are only displaying the adaptive version. It seems clear that DPF and DAF are overrunning the FSC results for any standard deviation, while presenting similar outcomes among them, except for high values of the standard deviation where DAF policy is clearly the best. We will try to confirm these results through a more extensive experimentation in the next section.

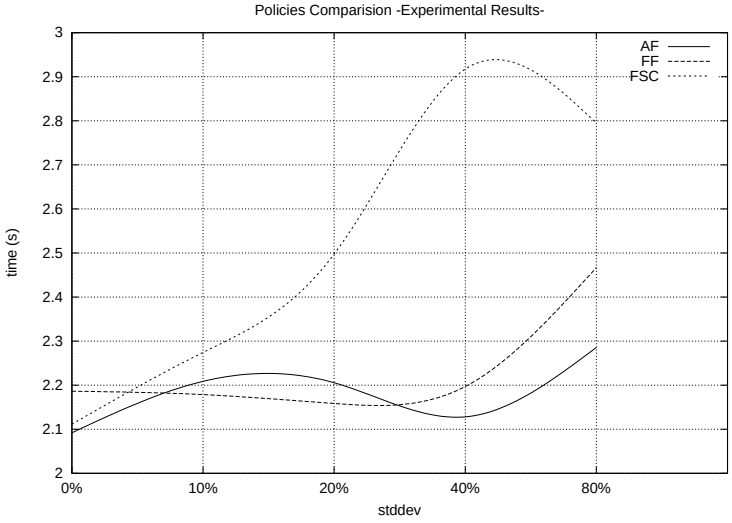


Figure 8. Comparison of results of the application of the three policies discussed on the same application.

2.4. 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. We discussed their theoretical pros and cons, and we illustrated their application using an example. Now, our goal is to provide stronger evidence about their performance through the use of extensive experimentation.

Before getting into the results discussion, we want to briefly describe the platform used to execute all these experiments, the tools developed to generate the appropriated synthetic programs, as well as the set of experiments we are presenting in this section.

		Mean Processing Time per Task and Standard deviation (%)														
		0,5 ms					2 ms					6ms				
# workers	**Data Volume**	0	10	20	40	80	0	10	20	40	80	0	10	20	40	80
25	240 Kb	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	1 Mb	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
50	240 Kb						x	x	x	x	x	x	x	x	x	x

Table 2. Summary of the configurations executed in order to test the task distribution policies.

We have executed our experiments on one of the clusters of the *Computer Science Department of the Wisconsin University at Madison*. It is a 150 dual 933MHz nodes connected to a 100Mbit switch, which has a gig-uplink to the core of the network (6 clusters). We have developed, on this platform, 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 to a given mean and standard deviation).

	Mean Processing Time per Task		
	0.5 ms (x10000 tasks ≈ 5 sec)	2 ms (x10000 tasks ≈ 20 sec)	6 ms (x10000 tasks ≈ 60 sec)
# workers	*Ideal Execution time* (Total execution time/number of workers)		
25	200 ms	800 ms	2.4 sec
50	100 ms	400 ms	1.2 sec

Table 3. Ideal execution time for 10000 tasks and 25, and 50 workers.

The set of experiments that we have designed for this section are summarized in table 2. We have executed each configuration using the three policies described and also distributing the whole set of tasks from the beginning, i.e., without any distribution policy. Each execution has been made with 10000 tasks, which results in the ideal execution time shown in table 3. In addition, each configuration has been executed for 15 iterations. The resulting times have been processed in order to eliminate statistical anomalies; finally, the mean execution time has been plotted in the figures displayed below (9 to 16). Each figure includes four graphs: three for the execution time of the application using each policy against the distribution without using any policy (a- FSC, b- DPF, and c- DAF), and the last one (d) for the comparison between the distribution policies.

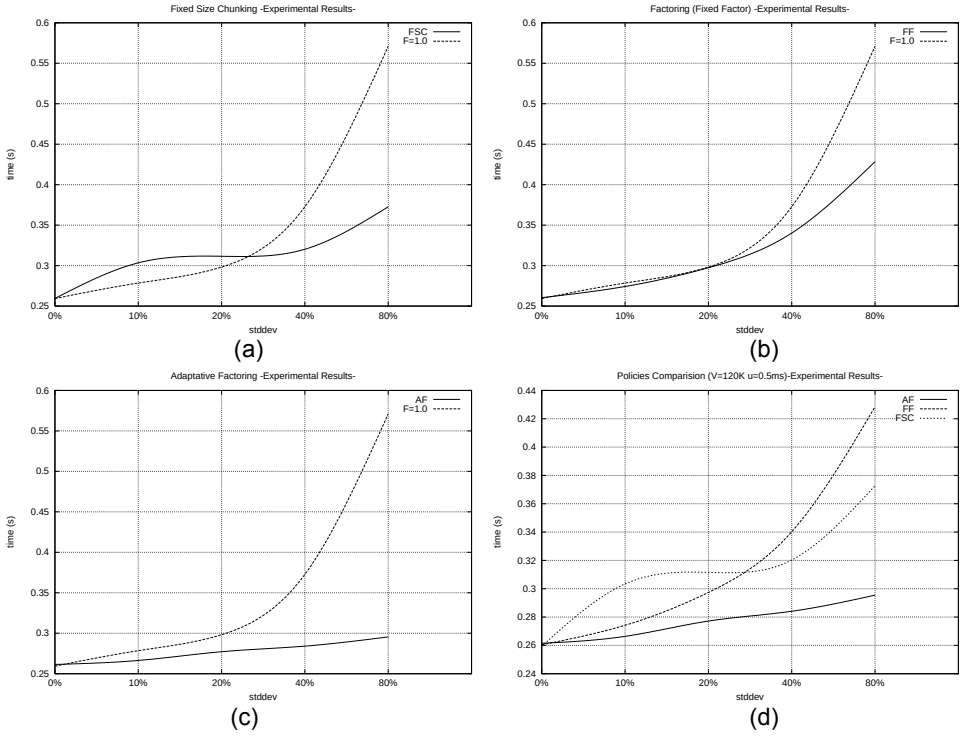


Figure 9. Execution of a 25-worker application with an associated mean processing time of 0.5 ms per task, a communication volume of 240 Kb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

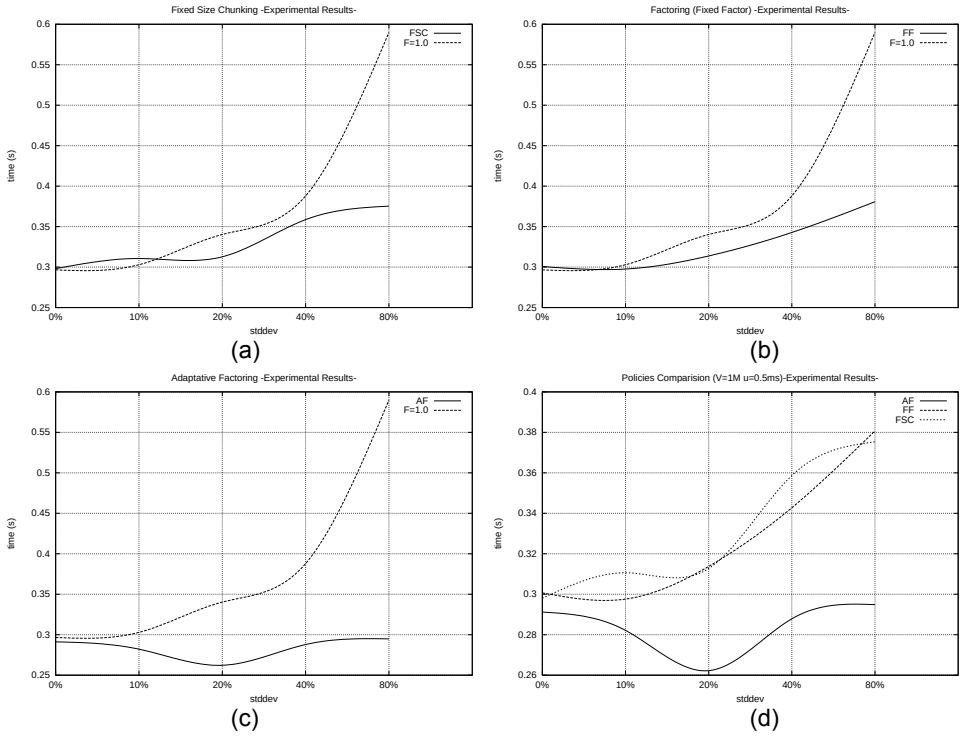


Figure 10. Execution of a 25-worker application with an associated mean processing time of 0.5 ms per task, a communication volume of 1 Mb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

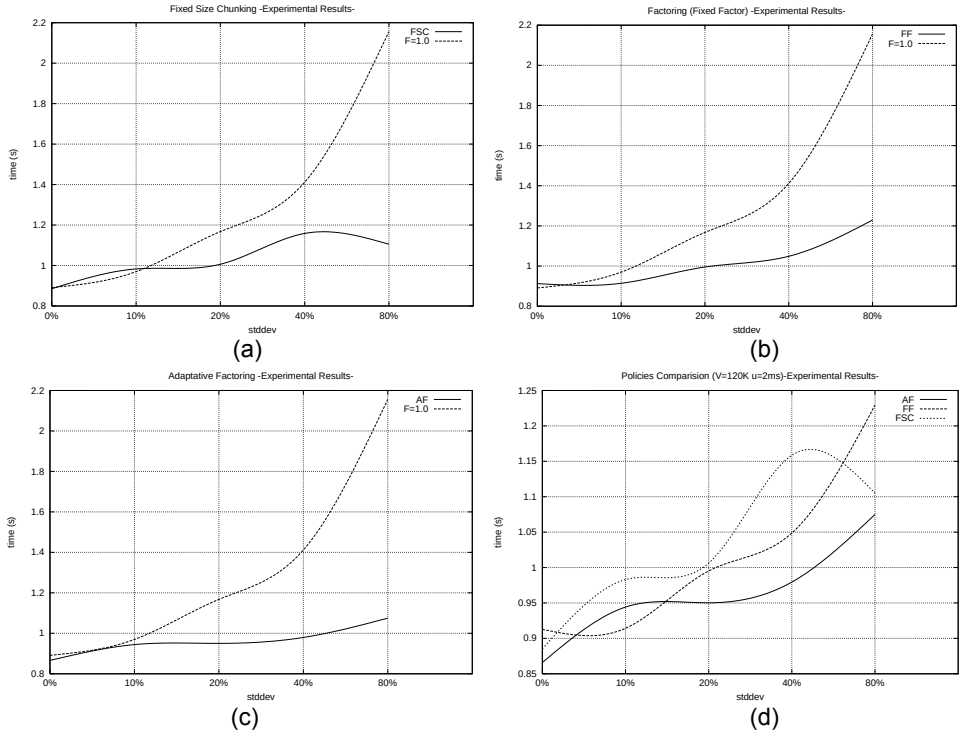


Figure 11. Execution of a 25-worker application with an associated mean processing time of 2 ms per task, a communication volume of 240 Kb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

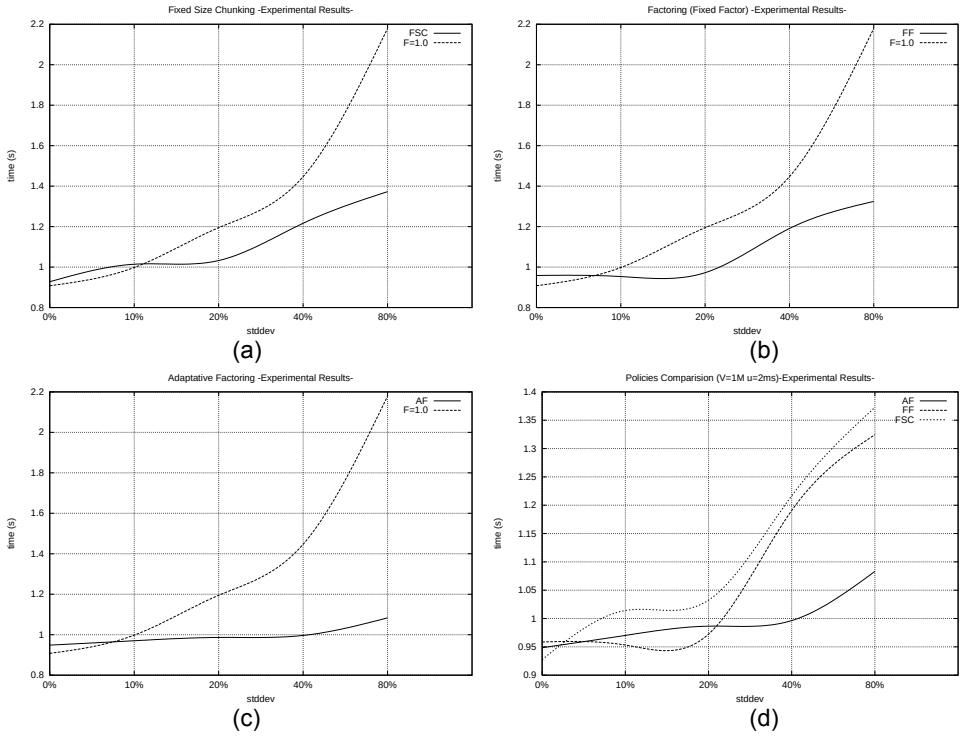


Figure 12. Execution of a 25-worker application with an associated mean processing time of 2 ms per task, a communication volume of 1 Mb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

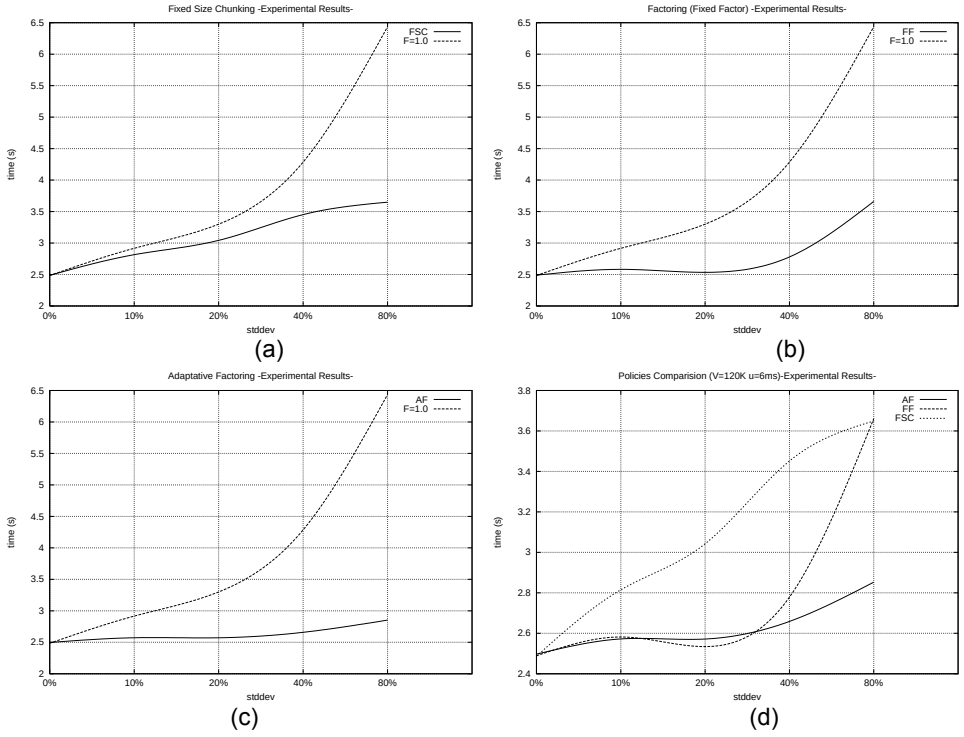


Figure 13. Execution of a 25-worker application with an associated mean processing time of 6 ms per task, a communication volume of 240 Kb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

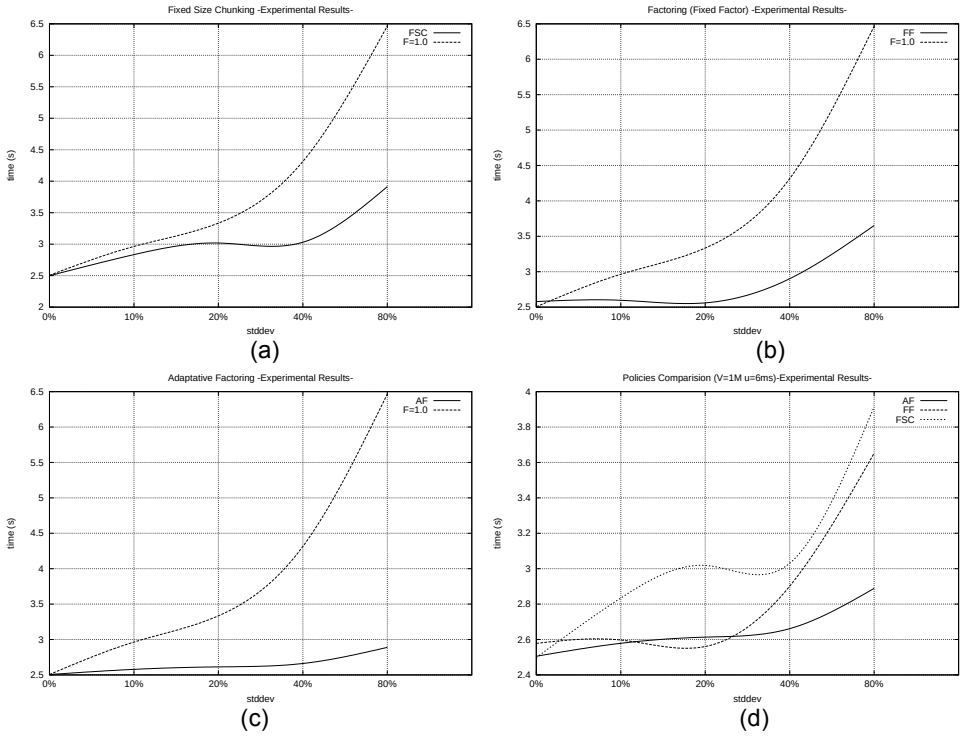


Figure 14. Execution of a 25-worker application with an associated mean processing time of 6 ms per task, a communication volume of 1 Mb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

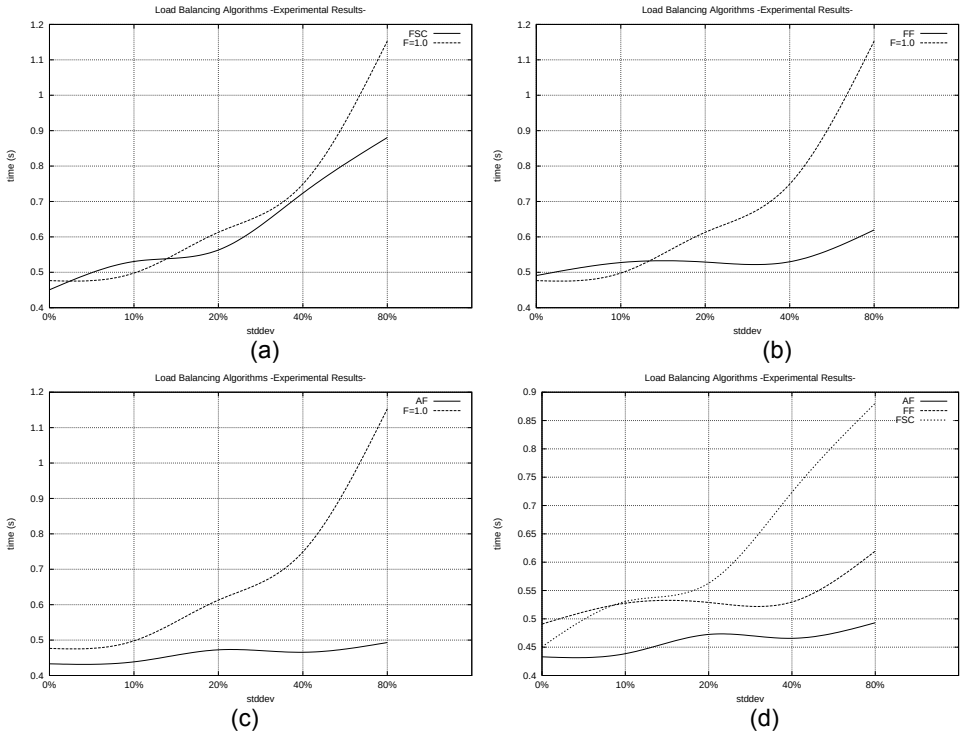


Figure 15. Execution of a 50-worker application with an associated mean processing time of 2 ms per task, a communication volume of 240 Kb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

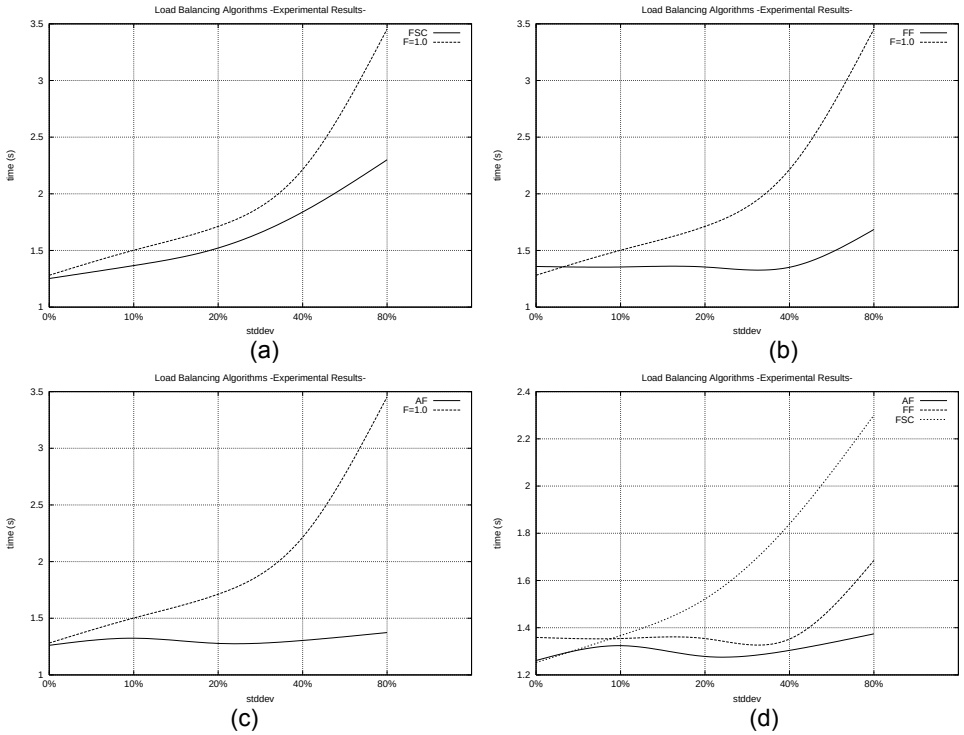


Figure 16. Execution of a 50-worker application with an associated mean processing time of 6 ms per task, a communication volume of 240 Kb, and for a standard deviation ranging from 0% to 80%. Distribution policy used: (a) FCS, (b) DPF, and (c) DAF. Policy comparison (d).

We can see that, in general, we have obtained 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 and medium standard deviation, and the fixed size chunking policy (FSC) is, usually by far, the worst one.

Nevertheless, we think that the outcomes of the DPF policy for the configurations displayed in figures 9, 10, and 11 deserve some further discussion. For these cases, we can see that the DPF policy shows a behavior that is some times worse than the FSC one, and usually closer to FSC than to DAF.

We believe that the atypical values of these figures could be partially caused by statistical deviations, as a result of running only fifteen iterations of the application. For instance, we can see in figure 17 the mean execution time of each worker for the application of figure 9 and for an 80% standard deviation using the FSC policy (a) and that of the DPF (b); it is clear that there is a greater dispersion for the values in the case of DPF than for those of FSC, which is likely to make it more difficult to balance the DPF case. This is happening, regardless of using the same configuration for both cases, because the parameters passed to the application are

the mean execution time and the standard deviation and not the actual execution times.

However, this is not the main problem; actually, running more iterations in order to get more statistically sound results will not make the real problem disappear. Because the real problem is that both policies only take into consideration the mean execution time associated with each worker and neglect the standard deviation associated to this time. In consequence, in regard to the experiments, we use the same mean processing time for all tasks; thus, in the long run for all workers, the FSC and DPF policies will tend to be overoptimistic because on average the application seems more and more homogeneous for each new iteration, while the real situation is likely to be far from homogenous due to the high standard deviation. As an example, we can see in figure 17 the value of the partition factor in each iteration for the FSC policy (c) and the DPF policy (d) given the mean execution times per worker of graphs (a) and (b) respectively. There, we can see how, as the mean processing times of the workers become closer to each other, the partition factor increases because the policy assumes that the application is more and more homogeneous.

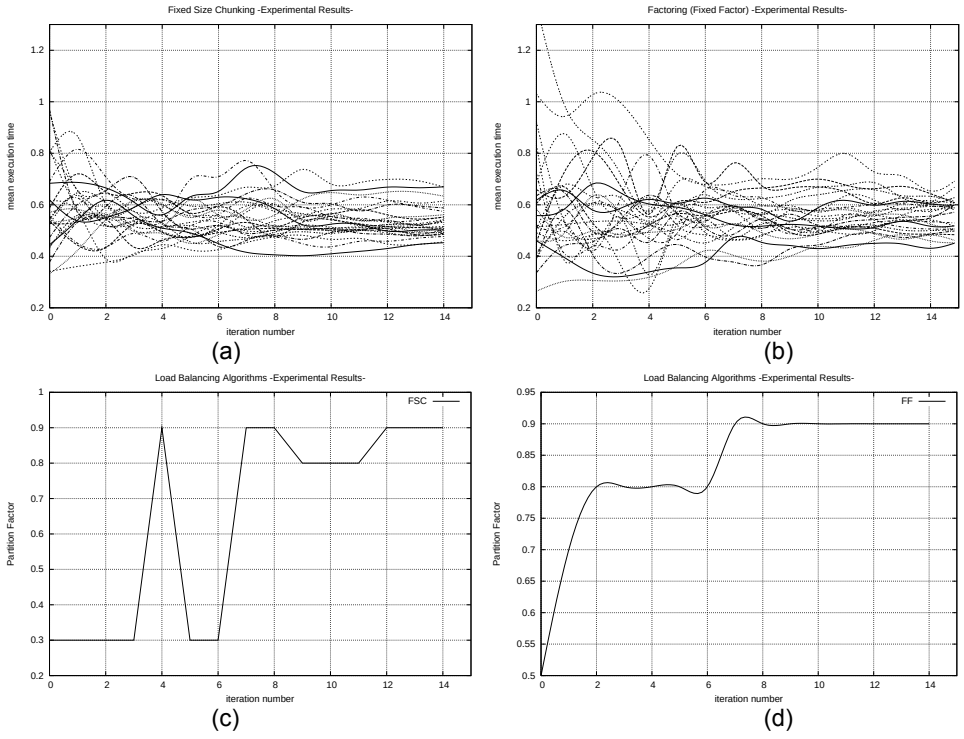


Figure 17. Mean execution time/processor in each iteration for a 25-worker application with a mean processing time of 0.5 ms/task, a communication volume of 240 Kb, and a standard deviation of 80%, using the FSC policy (a), and using DPF policy (b). Factor variation in each iteration for both policies, (a) FSC and (b) DPF.

We outlined this problem when we described the distribution policies and we said that estimating the partition factor for both FSC and DPF assumes a steady behavior of the application. It should be noticed that recalling less history could improve the results for a high standard deviation, just because the policy will expect more dispersion and, in consequence, will be more conservative; on the other hand, it could worsen the results for a low standard deviation by being more sensitive to those few values that are far from the mean.

Even though, empirically, this explains why the distribution policy does not always lead to the results expected, we still have to clarify why it seems to have a greater effect over the DPF policy than over the FSC one. To do so, it will be helpful to analyze what happens for different partition factor values in the FSC policy:

- *Partition factor below 0.4*. These values lead to several batches (no less than three); if the task processing time standard deviation is low then we will have an unnecessary increase of the communication overhead. On the other hand, if the task processing time standard deviation is high then many batches should help to balance the processor load.
- *Partition factor greater or equal to 0.5 and lesser or equal to 0.7*. These values lead to only two batches with the second batch including a significant number of tasks. In this situation, a worker that has received a regular chunk in the first round, which means that this worker neither needs too much time nor too little to process it, is likely to require a second chunk and if this second chunk is a tough one, which means that the worker will spend above average time to process it, then the overall performance of the application will be worsened because the policy is not flexible enough to recover from a situation like that.
- *Partition factor greater than 0.7*. These values also lead to only two batches, but this time the second batch includes few tasks. In this case, if the task processing time standard deviation is low, a set of chunks with a few tasks will help to minimize load unbalance.

As a result, it is very unlikely to get intermediate values for the partition factor from the estimation process for the FSC policy. This is a fundamental difference regarding DPF policy because, for this policy, any partition factor value (except 1 obviously) could lead to the creation of several batches and, consequently, it is more probable to have smooth variations of the partition factor for the DPF policy than for that of the FSC. This annoying behavior of the FSC policy will generally lead to

obtaining worse results with this policy than with the others, but for a high processing time standard deviation, which is only indirectly seen by the policy through the per processor mean, these steep variations on the partition factor could sometimes lead to better results than the gradual adaptation of the DPF one.

This analysis can be seen in graphs (c) and (d) of figure 17, but it is even clearer in graphs (a) and (b) of figure 18. In those graphs, we show the variation of the partition factor for the FSC (a) and DPF (b) policies during the execution of the 50-worker application of figure 15 for a standard deviation of 20% and 80%. We can see there how, for a low standard deviation (20%), both policies choose high partition factors, but for a high standard deviation the DPF policy only slightly decreases the chosen factor, while the FSC sharply jumps from very low factors to very higher ones and vice versa.

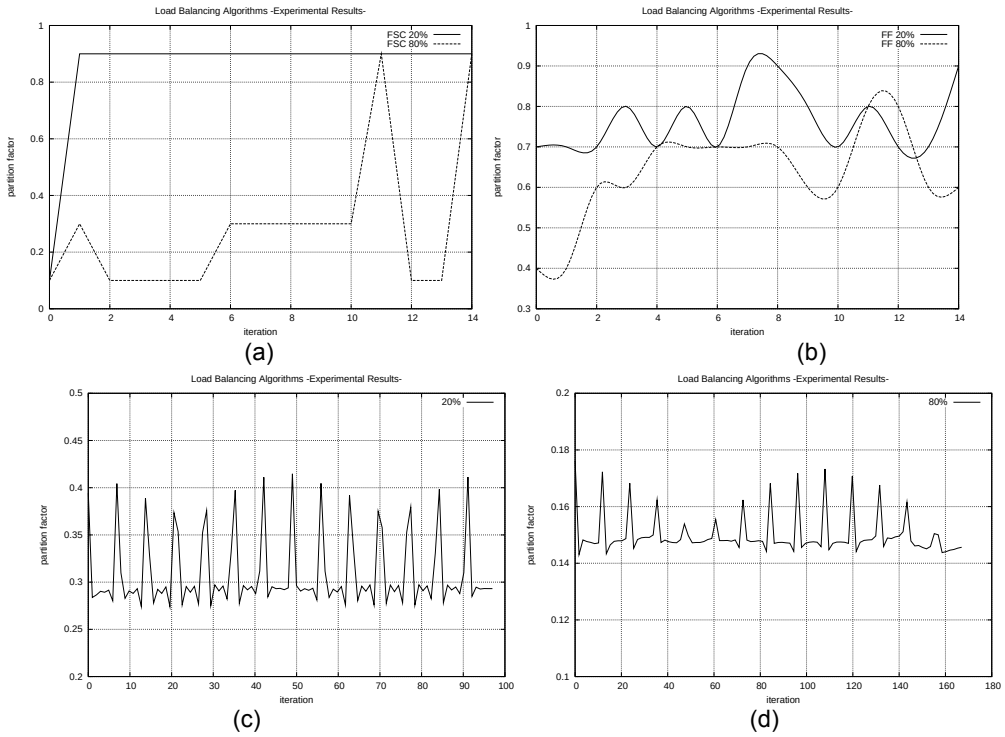


Figure 18. Factor variation during the execution of a 50-worker application with a mean execution time of 6 ms per task, 240 Kb of communication volume, and a standard deviation of 20% and 80%. FSC (a), DPF (b), DAF with a 20% standard deviation (c), and DAF with an 80% standard deviation.

Finally, we can also see in figure 18 (c) and (d) how the DAF distribution policy deals with load unbalancing. We can see that, independently of the standard deviation value, at the beginning of the iteration the partition factor is higher than at the end, assuming that there will be enough margin during the iteration to overcome any possible load unbalance. Moreover, we can see that the policy is aware of the

standard deviation of the processing time because it produces a consistent output with no sharp variations.

3. 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 performance.

However, one observes, in the real world 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 performance decreases because the costs introduced are greater than the advantages brought 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 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 [CMo+02], and successive extensions to it were presented in [CM+03] and [CM+04]. Finally, in [MCe+05] and [MC+05] we presented a couple of implementations of a tuning tool that uses part of the model to dynamically improve the performance of real applications. This model is employed to evaluate the behavior of the application when it is executing and decide if it will 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 taken 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 by efficiency reasons because having several workers

on 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 tasks 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 the results of the extensive experimentation we have done are shown and commented on the following subsections.

3.1. 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.

Firstly, we have characterized the interconnection network with the classical message start up time plus communication time formula, which is quite simple although [GL99] 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, or ignoring cache effects. Nevertheless, we use this expression because our model will be used in a dynamic tuning environment, so 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 computation, the time the master invests in building new sets of data, the amount of data sent and received to/from each worker, and the kind of communication protocol (blocking/synchronous or not).

Thirdly, we should be aware of some parameter dependence on the number of workers. For example, for some problem solutions a constant amount of data is distributed among the workers, while for other problem solutions the whole data set is sent to all workers; in the first case adding new workers not only decreases the number of tasks each worker will compute, but also the amount of data it will receive. By contrast, in the second case, the amount of data received by the worker

will be the same, although it will compute fewer tasks. The dependences that we have considered can be summarized as follows:

- Between the amount of data to be communicated and the variation of the number of workers.
- Between the computational load and the variation of the number of workers.

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, αV = $\sum_{i=0}^{n-1} v_i^m$ and (1- α)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.
- **Tc**: total processing time o workers ($\sum_{i=0}^{n-1} \mu_i$), in ms.

Now, it is time to develop the performance model expressions, which are based on the structure and functional behavior of the Master/Worker pattern. The idea is to define, in the first place, a general performance expression and then, considering different execution conditions and the parameter dependences, derivate more specific expression sets, the former must clearly reflect the pattern functional behavior, and the latter must be useful for improving the application performance.

In figure 19, 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. Each worker then goes on processing the set of tasks it has received; next, each worker sends back to the master the results of its calculations, implying another communication phase between each

worker and the master; finally, the master could spend some processing time to generate 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(**λ** , m_o, αV, n). The duration of the workers' computation phase depends on the total processing time and the number of workers; it can, therefore, be defined as a function C(Tc,n). Finally, the amount of time 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(**λ**,m_o,(1α)V,n).

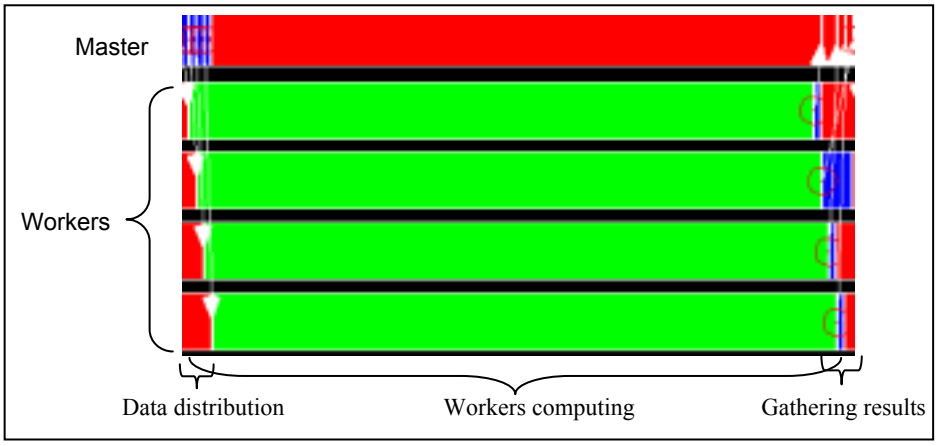


Figure 19. Typical execution trace of an iteration of a Master/Worker application.

It should be noticed that, when defining these functions, we must be aware of the existing overlapping times among them. In particular, we can graphically see in figure 19 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 (Tt) as:

$$Tt = S(\boldsymbol{\lambda}, m_o, \alpha V, n) + C(Tc, n) + R(\boldsymbol{\lambda}, m_o, (1-\alpha)V, n) + \mu_m \qquad \textbf{(3)}$$

As of now, we want to specify these functions for different execution conditions and, after that, we will consider the parameter dependences defined above. In the first place, we will develop the data distribution function (S) considering the communication protocol. As we previously mentioned, 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 necessary to perform the whole data distribution, which is not always the addition 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. As a consequence, 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:

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

- 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 could begin before the previous one has ended. Consequently, as sending operations are overlapped 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:

$$S(\lambda, m_o, \alpha V, n) = n m_o + \lambda v_{n-1}^m \quad \textbf{if} \quad m_o \geq \lambda v_i^m \left\{ \forall i \middle| 0 \leq i \leq n-1 \right\}; \textbf{ or}$$

$$S(\lambda, m_o, \alpha V, n) = m_o + \sum_{i=0}^{n-1} \lambda v_i^m \quad \textbf{if not}$$

In the second place, we will define the computation function (C) considering that there is a time overlap among the workers' 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 the execution is balanced, 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 the function C(Tc,n) as μ_{n-1}.

Finally, we will define the results gathering function (R) taking into consideration 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 some results back to the master (the event that marks the end of the computation phase) every other worker has already sent back its results. As a consequence, 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 overlapping is possible if a worker sends back its answer before the master has ended the data distribution. In that case, there will be a delay in the results reception by the master and the iteration will last longer. The problem is the same that was introduced when explaining the Dynamic Adjusting Factoring (DAF) data distribution policy, which led us to the definition of the *Master's Chunk Managing Capability* concept (see section 2.3). Summarizing, function R(**λ** , m_o, (1-α)V, n) can be defined as $m_o + \lambda v_{n-1}^w$, provided that the Master has not reached its *Master's Chunk Managing Capability*.

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

<table>
<tr><th></th><th colspan="2">Communication Protocol</th></tr>
<tr><th></th><th>Asynchronous</th><th>Synchronous</th></tr>
<tr><td>$m_o \geq \lambda v_i^m$</td><td>$Tt = (n+1)m_o + \lambda v_{n-1}^m + \mu_{n-1} + \lambda v_{n-1}^w + \mu_m$ (4)</td><td rowspan="2">$Tt = \sum_{i=0}^{n-1}\left(m_o + \lambda v_i^m\right) + \mu_{n-1} + m_o + \lambda v_{n-1}^w + \mu_m$ (6)</td></tr>
<tr><td>$m_o \leq \lambda v_i^m$</td><td>$Tt = 2m_o + \lambda \sum_{i=0}^{n-1} v_i^m + \mu_{n-1} + \lambda v_{n-1}^w + \mu_m$ (5)</td></tr>
</table>

Next, we will analyze the effect of the parameter dependences mentioned before in order to complement these performance expressions. These dependences should be considered with the objective of improving the prediction accuracy of the model. In particular, it is very important to be able to forecast which would be the communication volume (V) and processing time (Tc) for a certain number of workers because, as we have seen, the performance expressions depend on these values.

Firstly, we will analyze the dependence between the amount of data to be communicated (V) and the variation in the number of workers. Assuming that there is an abstraction of the problem being solved as a set of data structures, the issue we are considering now consists in knowing how the amount of data transferred back and forth between the Master and the Workers varies in relation to changes in the number of workers. Whereas it is fair to assume that the number of tasks assigned to each worker will decrease as the number of workers increases, the overall volume of data transferred could remain steady if each worker receives and sends back less data, or could increase if the decrease of the data received and sent by each worker is not proportional to the number of new workers.

More specifically, the new total volume of data to be communicated (V') will be in the range [V, (n'/n)V], where n' is the number of workers for which we are making the prediction, n is the current number of workers, and V is the current data volume.

We will get the lower bound of this range in those cases where the total communication volume is constant, no matter the number of workers used, and the upper bound when the communication volume increases in a one to one proportion with the number of workers. It should be noticed that for a decrease in the number of workers, which might be worthwhile to consider in some situations, there will be a swapping of roles between both bounds. Consequently we can define the new communication volume (V') as a function of the current communication volume (V), in the following way: $V' = V + \Delta_V$, where Δ_V will depend on the number of workers or will be 0.

Secondly, a similar analysis can be made on the dependence between the total workers' processing time (Tc) and variations in the number of workers. The variation degree of the workers' processing time (Tc) depends on the portion of processing time that each worker spends computing tasks and the portion that it spends executing task independent code (such as initialization). The greater the former, the littler the Tc variation when the number of workers changes, because increasing the number of workers decreases the number of tasks received by each worker and, as a result, only the task dependent portion of the processing time will proportionally decrease, while the independent portion will remain steady, thus increasing the overall processing time.

Therefore, we can say that the new overall processing time (Tc') will be Tc + (n'-n)(portion of processing time each worker spends executing task independent code), where n' is the number of workers for which we are making the prediction and n is the current number of workers. As a consequence, Tc' can be defined as a function of Tc in the following way: $Tc' = Tc + \Delta_{Tc}$. However, this time we may assume that task processing portion will be significantly greater than the task independent one because if the contrary happens the parallelization degree of the application will be low, which would be considered a design problem. Consequently, there should be a significant change in the number of workers in order to notice a meaningful variation of the overall processing time.

Summarizing, we can say that changing the number of workers could cause variations in the overall communication volume (V) and the overall workers processing time (Tc), in addition, the variation of V can be significant, while Tc variation should not. As a consequence, if we want to produce good performance predictions for our tuning tools we have to be able to measure the Δ_V and Δ_{Tc} variation factors.

Once we have decided how to deal with the dependences among some model parameters, it is time to make some final remarks that will lead to simpler expressions with the objective of simplifying the application of the model by a tuning tool. We have said before that both communication functions (S and R) depend on the number of workers (n) and the overall communication volume (V), and also that the processing function (C) depends on the number of workers and the overall processing time (Tc), but when writing the expressions none of these parameters explicitly appeared. That was because then it was clearer to describe the performance functions in terms of particular data transfers ($v_i^{m/w}$) and execution times (μ_i) than doing so in terms of V and Tc.

However, 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 Tc/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 (Tc), than the specific values of each worker. Moreover, it is necessary to define the performance expressions in terms of the number of workers to fulfill the objectives of being able to predict the performance of the application for a different number of workers. Applying these substitutions to expressions (4), (5), and (6) results in the following set of expressions to describe the performance of a Master/Worker application:

<table>
<tr><th></th><th colspan="2">Communication Protocol</th></tr>
<tr><th></th><th>Asynchronous</th><th>Synchronous</th></tr>
<tr><td>$m_o \geq \lambda v_i^m$</td><td>$Tt = (n+1)m_o + \frac{(Tc+\lambda V)}{n} + \mu_m$ (7)</td><td rowspan="2">$Tt = (n+1)m_o + \frac{[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m$ (9)</td></tr>
<tr><td>$m_o \leq \lambda v_i^m$</td><td>$Tt = 2m_o + \frac{[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m$ (8)</td></tr>
</table>

Finally, we would like to introduce a more formal definition of the *Master's Chunk Managing Capability* concept. We have said that it is the maximum number of data chunks the Master can manage from the time it sends a chunk to a worker until it receives the answer for the same chunk from that worker. If this capability is persistently exceeded then there will be workers answering more quickly than the Master's capability to send data to all of them. There can be two different consequences: if the Master prioritizes receives over sends then some workers will not receive data and will be idle, as can be seen in figure 20 (a); on the contrary, if

the Master prioritizes sends over receives then workers will have to wait for the Master to end the data distribution before their answers are received, as can be seen in figure 20 (b).

In order to calculate the *Master's Chunk Managing Capability* and, in consequence, 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 time it sends a chunk to a worker to the moment the answer of that worker, for the same chunk, is available.

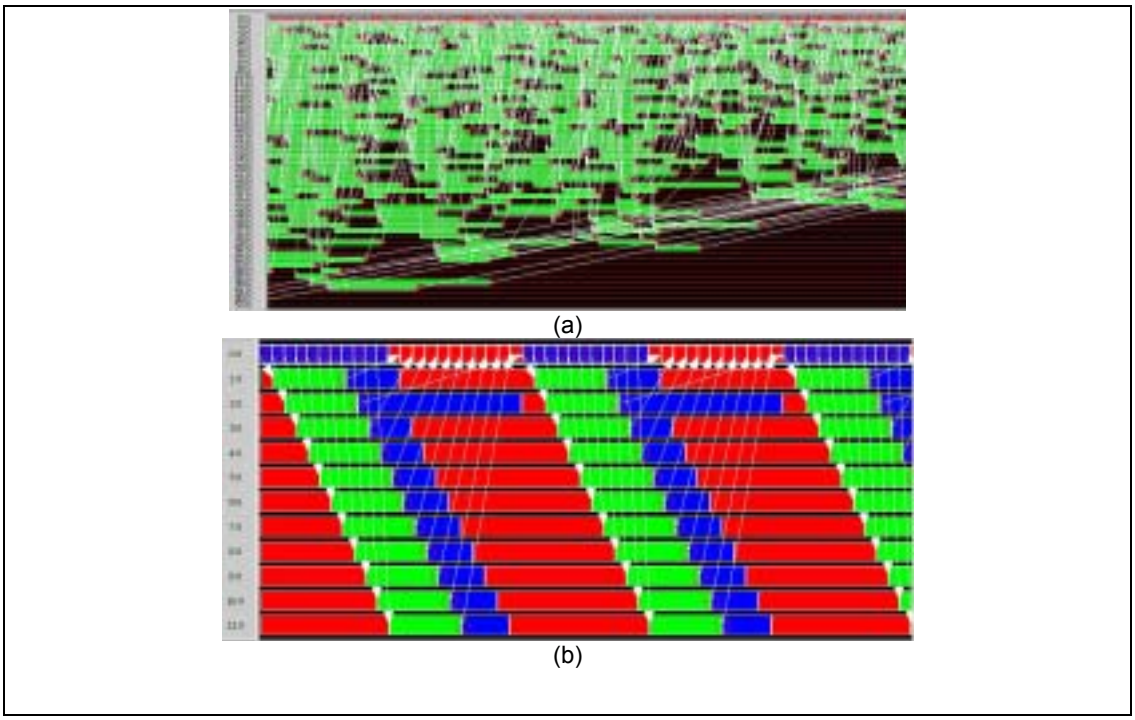


Figure 20. Illustration of the possible effects of exceeding the Master's Chunk Managing Capability. Workers left completely idle, because receives are being prioritized over sends (a). Workers delayed, because sends are being prioritized over receives (b).

The expression to calculate this value can be obtained from 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 back the results to the Master. Considering different communication conditions we get the following equations for the *Master's Chunk Managing Capability*:

	Communication Protocol	
	Asynchronous	**Synchronous**
$m_o \geq \lambda v_i^m$	$nm_o + \lambda v_{n-1}^m = 2m_o + \lambda v_0^m + \mu_0 + \lambda v_0^w$	$nm_o + \lambda \sum_{i=0}^{n-1} v_i^m = 2m_o + \lambda v_0^m + \mu_0 + \lambda v_0^w$
$m_o \leq \lambda v_i^m$	$m_o + \lambda \sum_{i=0}^{n-1} v_i^m = 2m_o + \lambda v_0^m + \mu_0 + \lambda v_0^w$	

Substituting the particular data volumes and processing times of these expressions by their simplifications on V and Tc, and solving them for the number of workers, we get that if the number of workers (n) is greater than the following expressions our application will be exceeding the Master's Chunk Managing Capability:

	Communication Protocol	
	Asynchronous	**Synchronous**
$m_o \geq \lambda v_i^m$	$\left\lfloor \frac{\sqrt{m_o^2 + m_o((1-\alpha)\lambda V + Tc)}}{m_o} + 1 \right\rfloor$ **(10)**	$\left\lfloor \frac{(2m_o - \lambda\alpha V) + \sqrt{(\lambda\alpha V - 2m_o)^2 + 4m_o(\lambda V + Tc)}}{2m_o} \right\rfloor$ **(12)**
$m_o \leq \lambda v_i^m$	$\left\lfloor \frac{\lambda V + Tc}{\lambda\alpha V - m_o} \right\rfloor$ **(11)**	

3.2. Analysis of the Master/Worker performance expressions

Before continuing with the definition of our performance model for Master/Worker applications, and discussing how the decision of changing the number of workers should be taken, we want to introduce some mathematical analyses of the expressions defined so far, with the aim of showing their interrelations, as well as their adaptability to real executing conditions.

In the first place, we show in figure 21 the output of functions (7) and (8) (graphs (a) & (b) respectively), and (9) (graphs (c) & (d)), for fixed values of V (1Kbyte for (a) & (c) and 1Mbyte for (b) & (d)), and Tc (2 sec.), and different number of workers (from 5 to 120 (a) & (c) and from 1 to 150 (b) & (d)), assuming a message overhead of 1 ms. and a per byte transmission time of 1μs. The idea is to show the expected execution values for configurations that should be calculated with expressions (7) and (8) (graphs (a) and (b)), and compare them with the corresponding synchronous example (graphs (c) and (d)). Moreover, using expressions (10), (11), and (12) the number of workers that can be managed by the Master without exceeding the *Master's Chunk Managing Capability* has been calculated and it is indicated in each graph by an arrow and the label of the applied expression.

There are several observations that can be made in figure 21:

- First, we can see that expressions (7) and (9) (graphs (a), (c), and (d)) show that adding more workers can significantly improve the performance of the application, but also that beyond some point this action leads to performance loses. However, expression (8) (graph (b))

produces lower and lower values as we increase the number of workers. A simple analysis of the expressions is enough to explain these observations: It can be easily shown that the second term of expressions (7) ($\frac{(Tc+\lambda V)}{n}$), and (8) and (9) ($\frac{\left[\left((n-1)\alpha+1\right)\lambda V+Tc\right]}{n}$), decreases if n increases and V and Tc remain constant, getting closer and closer to 0. By contrast, whereas the first term of expressions (7) and (9) ($(n+1)m_o$) increases linearly with n, the first term of expression (8) ($2m_o$) is constant. As a result, while expressions (7) and (9) will have a decreasing value if and only if the reduction of the second term is greater than the growth of the first one, expression (8) will have a constantly decreasing value that will get closer and closer to $2m_o + \mu_m$. Nevertheless, if the overall data volume (V) does not increase linearly with the number of workers (V'<(n'/n)V), adding new workers to the application will lead to smaller messages and eventually to overrunning the $m_o \leq \lambda v_i^m$ condition and, consequently, from this point expression (7) should be used instead.

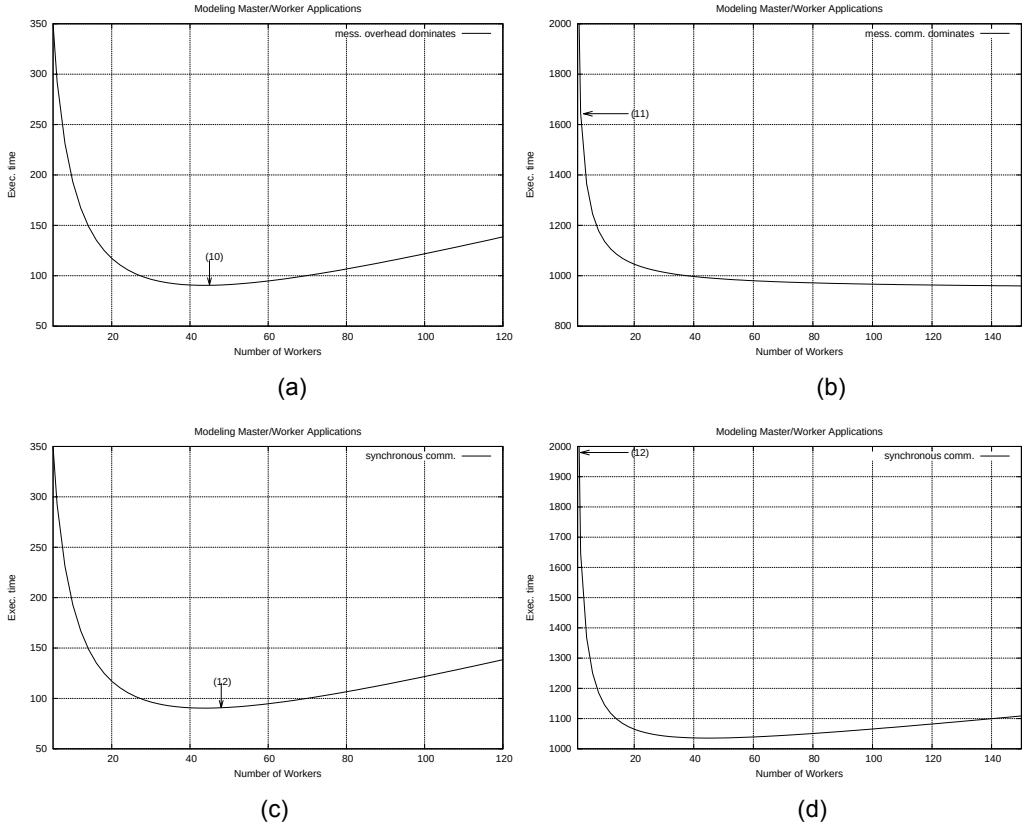


Figure 21. Expected execution times for an application with 1 Kbyte of communication volume (V) and 2 sec. of processing time (Tc), using asynchronous communication (a) and synchronous communication (c). The same for an application with 1 Mbyte of communication volume (V) (b and d).

- Next, it can also be seen that functions (7) and (9) have a minimum value, which tells us for what number of workers we obtain the lowest execution time. Clearly, this is the optimal number of workers in order to maximize the applications performance. Actually, it is possible to find this value analytically if the relationship between V and V' is clearly established beforehand (for badly designed applications, it may also be necessary to consider the relation between Tc and Tc'). We only have to solve the expression $\frac{\partial Tt}{\partial n} = 0$ for n in order to get the number of workers with the lowest associated execution time. We have solved this expression for the number of workers ([CM+04] and [MCe+05]) making V' = V and V'=(n'/n)V, and we obtained the following results:

	Communication Protocol	
	Asynchronous	**Synchronous**
V'=V	$n_{opt} = \left\lfloor \sqrt{(Tc + \lambda V)/m_o} \right\rfloor$ **(13)**	$n_{opt} = \left\lfloor \sqrt{(Tc + (1-\alpha)\lambda V)/m_o} \right\rfloor$ **(14)**
V'=(n'/n)V	$n_{opt} = \left\lfloor \sqrt{Tc/m_o} \right\rfloor$ **(15)**	$n_{opt} = \left\lfloor \sqrt{Tc/(m_o + \alpha\lambda V)} \right\rfloor$ **(16)**

In the examples shown above, these expressions lead to the following results: 44 workers for graphs (a) and (c) using expressions (13) and (14), and 45 workers for graph (d) using expression (14). In addition, if we had increased the communication volume (V) linearly with the number of workers we would have got the following results: 44 workers for graph (a) using expression (15), 32 workers for graph (c) using expression (16), and for graph (d) the models says that it is better to run the application sequentially.

We should introduce a commentary about applications modeled by expression (8) and the number of workers that maximizes performance. We mentioned before that condition $m_o \leq \lambda v_i^m$ will eventually be overrun if the data volume does not increase linearly, for example if the communication volume (V) stays constant this will happen for $n > \lambda\alpha V / m_o$ workers. From that number of workers, the application will be modeled by expression (7) and, as a consequence, either it reaches its minimum value at the switching point if function (7) is already increasing

for that number of workers, or it will reach its minimum value later following expression (13) if not.

Finally, finding the optimal number of workers that minimizes execution time could seem attractive, but, as we will explain in the next section, the efficiency in the use of the resources (processors) is likely to decrease sharply when the number of workers is around the optimal value, consequently, we must find a compromise between performance and efficient use of resources.

- Finally, we have included a mark in each graph that indicates the number of workers that can be managed by the Master without exceeding the *Master's Chunk Managing Capability* (MCMC), specifically by using expression (10) we got that the Master can manage up to 45 workers for the prediction shown in graph (a), by using expression (11) we got that it can manage up to 3 for the prediction shown in graph (b), and by using expression (12) we got that it can manage up to 48 and 2 for the predictions shown in graphs (c) and (d) respectively.

  It can be seen in graphs (a) and (c) that the value of the expression for this number of workers is closer to the minimum value of the expression, while for graphs (b) and (c) it is far from the minimum value. Actually, it can be demonstrated that for applications modeled by expression (7) the number of workers that make the Master exceed its MCMC will always be greater than the number of workers needed to get the lowest execution time, while it is possible for the Master to exceed its MCMC before or after arriving at the theoretical lowest execution time value for applications modeled by expressions (8) and (9).

  To demonstrate for an application modeled by expression (7) that the number of workers exceeding the MCMC is always greater than the number of workers for which the application gets its lowest execution time we assume that it is possible ($n_{opt} > n$ that causes the Master to exceed its MCMC) for a communication volume (V) that increases linearly with the number of workers (which is the less favorable situation for the application performance) and, as a result, we get a contradiction. Specifically the demonstration is:

  We know that the Master of an application modeled by expression (7) exceeds its MCMC if the number of workers goes beyond the value of

expression (10), i.e., if $n' > \left\lfloor \frac{\sqrt{m_o^2 + m_o((1-\alpha)\lambda V' + Tc)}}{m_o} + 1 \right\rfloor$ where V' = (n'/n)V and n is the current number of workers.

From this expression, we can get that $n'^2 > \frac{n'(1-\alpha)\lambda V}{nm_o} + \frac{Tc}{m_o} + 2n$, then if we have assumed that $n_{opt} > n'$ we know that $n_{opt}^2 > n'^2$, and substituting n_{opt} by expression (15) we get that $Tc/m_o > \frac{n'(1-\alpha)\lambda V}{nm_o} + \frac{Tc}{m_o} + 2n$ should be true. However this expression is equivalent to $0 > \frac{n'(1-\alpha)\lambda V}{nm_o} + 2n$, which is impossible because $\frac{n'(1-\alpha)\lambda V}{nm_o} > 0$ and 2n > 0. As a result n_{opt} cannot be greater than the number of workers that causes the Master to exceed its MCMC.

The same can be said for applications modeled by expression (8) that reach their lowest execution time modeled by expression (7).

All these observations and analyses can be summarized in the following way:

- An application modeled by expression (7) will reach its lowest execution time for some number of workers between the values of expressions (13) and (15) depending on the relationship between the communication volume (V) and the number of workers.
- The same will be true for an application modeled by expression (8) if the relationship $m_o \leq \lambda v_i^m$ does not hold for n_{opt}, which means that for that number of workers the application is modeled by expression (7). On the contrary, the application will reach its lowest execution time for min(expression (11), $n > \lambda\alpha V / m_o$), which is either the number of workers that causes the Master to exceed its MCMC or the number of workers that causes the application to be modeled by expression (7) because $m_o \leq \lambda v_i^m$ does not hold anymore.
- Finally, an application modeled by expression (9) will reach its lowest execution time for min(expression (12), between the values of expressions (14) and (15)), depending on the relation among the communication volume (V) and the number of workers.

Secondly, we want to discuss the sensitivity of the performance expressions to small variations of the parameters. We want to do that because for real executions, even for the more stable ones, we will get slightly variable values depending on the

current conditions of the network (such as network contention), the influence of the local cache memory, or even from the operating system (such as context switches). As a result, we will be evaluating the performance of an application based on the mean values of the observed execution and communication times, the mean values of the size of messages, and even the mean values of the network overhead and communication speed. Therefore, we must demonstrate that small variations of the parameters will not result in completely mistaken predictions of the application performance.

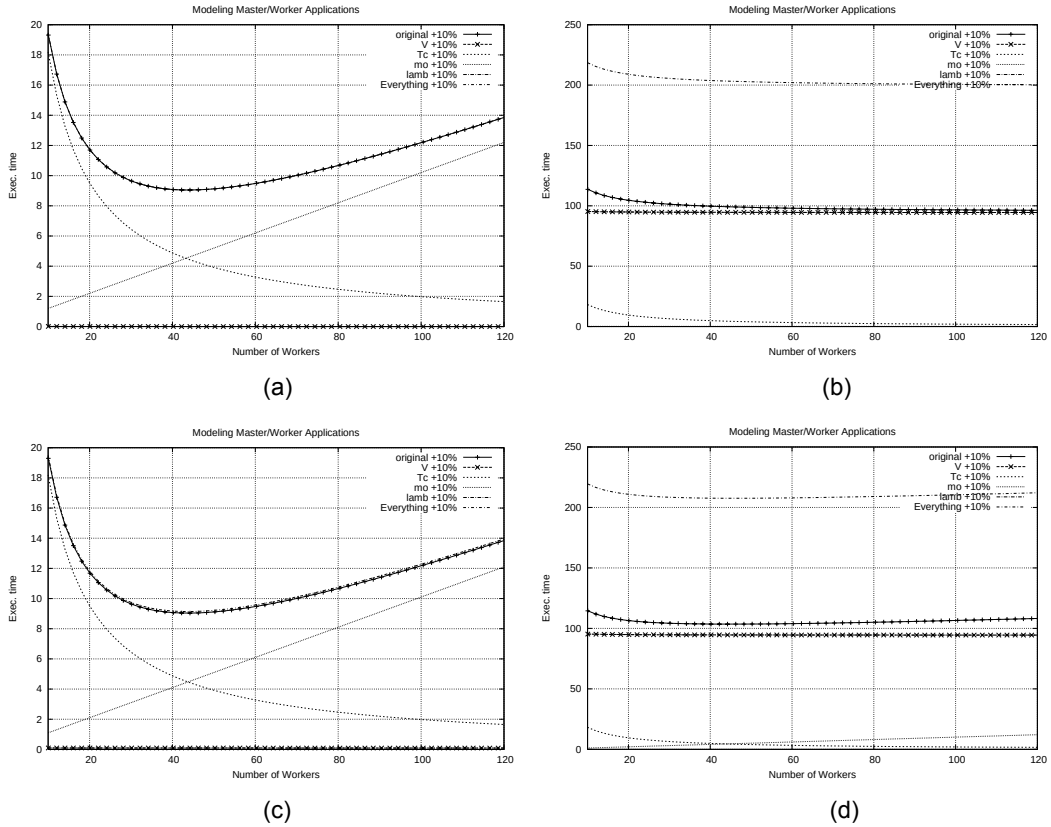


Figure 22. Differences of the execution times predicted by the performance expressions shown in figure 21 and the same example with an individual 10% variation of V, Tc, m_o, and λ, and a 10% variation of all parameters at once.

In order to fulfill this objective, we have analyzed what happens to the expression outcomes for a variability of ±X% on each parameter, looking first at the influence of each parameter individually and then looking at the influence of the combination of all parameters. We will illustrate this analysis with the graphs shown in figure 22. There,we can see the differences between the values obtained in figure 21 and the same example with a 10% individual variation of the communication volume (V), processing time (Tc), network overhead (m_o), and communication speed (λ), and a 10% variation of all parameters at once.

It can be easily seen that any single X% deviation of any parameter will produce a global deviation smaller than the X% of the total execution time. As an example let us analyze the effect of variations on the communication volume (V) and processing time (Tc); it is easy to see that a deviation of ±X% of any of these parameters, or of both of them at the same time, will produce in any case (expressions (7), (8), and (9)) a variation of less than X% on the whole prediction. Assuming the worst situation for a variation of ±X% of V and/or Tc the following condition for Tt will be held for expression (7):

$$(1-\frac{\mathrm{X}}{100})\mathrm{Tt} < (\mathrm{n}+1)\mathrm{m_o} + \frac{(1-\frac{\mathrm{X}}{100})(\mathrm{Tc}+\lambda\mathrm{V})}{\mathrm{n}} + \mu_\mathrm{m} \le \mathrm{Tt} \le (\mathrm{n}+1)\mathrm{m_o} + \frac{(1+\frac{\mathrm{X}}{100})(\mathrm{Tc}+\lambda\mathrm{V})}{\mathrm{n}} + \mu_\mathrm{m} < (1+\frac{\mathrm{X}}{100})\mathrm{T}t$$

The following one for expression (8):

$$(1-\frac{X}{100})Tt < 2m_o + \frac{(1-\frac{X}{100})[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m \le Tt \le 2m_o + \frac{(1+\frac{X}{100})[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m < (1+\frac{X}{100})Tt$$

And, finally, the following one for expression (9):

$$(1-\frac{X}{100})Tt < (n+1)m_o + \frac{(1-\frac{X}{100})[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m \le Tt \le (n+1)m_o + \frac{(1+\frac{X}{100})[((n-1)\alpha+1)\lambda V + Tc]}{n} + \mu_m < (1+\frac{X}{100})Tt$$

This relationship is clearly shown in the graphs of figure 22, where we can see how the differences between the originally predicted execution time and the one predicted with a single variation of each parameter, increased (or decreased) by 10%, are always below the differences between the originally predicted execution time and the predicted execution time increased (or decreased) by 10%.

Actually, only the combined deviation of the communication volume (V) and network speed (λ) could produce an overall deviation greater than the individual ones because these parameters appear in the same product. If the application is computation intensive, then this effect will occur almost unnoticed, as we can see in graphs (a) and (c) of figure 22. On the contrary, for communication intensive applications, the effect could lead to wrong predictions as can be seen in graphs (b) and (d) of figure 22. Specifically, an individual deviation of X% for V and λ could produce an overall deviation of: $(1+X/100)V(1+X/100)\lambda = (1+X/100)^2\lambda V \Rightarrow (1+2X/100+X^2/100^2)$, which more than doubles the individual ones (e.g. for individual deviations of 10% we could get a 21% overall deviation, or for individual deviations of 20% we could get a 44% overall deviation).

On the other hand, the same analysis for expressions (10), (11), and (12), which allow us to estimate the maximum number of workers that can be added without exceeding the *Master's Chunk Managing Capability* (MCMC), reveals that these expressions are more sensitive to some parameter deviations than expressions (7), (8), and (9). This happens because we have more products involving two or more parameters; thereby multiplying the effects of measurement errors.

In conclusion, the Master/Worker performance expressions ((7), (8), and (9)) are not too sensitive to small measurement mistakes, except for the accumulation of errors measuring the communication volume (V) and network speed (λ). In contrast, other expressions, such as (10), (11), and (12) are more sensitive to variations in a wide range of parameters. Consequently, we have to consider this fact when evaluating the performance of the application trying to give more credit to the results of the most reliable expressions.

3.3. Efficiency indexes

Thus far we have defined a set of expressions, which have been called performance expressions, that model the performance of a Master/Worker application under different circumstances ((7), (8), and (9)), and also others that can be useful to determine the limits to the number of workers that can be added to an application to improve its performance; moreover, some of these expressions ((10), (11), and (12)) complement the performance ones, while others ((13) to (16)) are directly derived from them.

Whereas this set of expressions can be useful for tuning the performance of a Master/Worker application, the efficiency in the use of the available resources is not taken into consideration in any of these expressions. Nonetheless, this is a relevant issue if these resources were to be shared among different applications, or even when a complex application, built as a composition of two or more frameworks, is analyzed and a decision taken about the best resource assignment. Moreover, it can be seen intuitively in the example shown in figure 23 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, very inefficient. In this figure, we show the expected execution times (using expression (7)) of an application with a communication volume (V) of 4Kbytes and an overall processing time (Tc) of 1.6 sec, assuming also a message overhead m_o of 1ms and a network speed of 1μs/byte, 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 (where the model predicts the lowest execution time -n_{opt}-) Workers means a gain of 3.99% for the same relative increment of the number of resources. Since we are increasing the number of resources in the same proportion, the decreasing gain of the execution time must be due to a drop in efficiency.

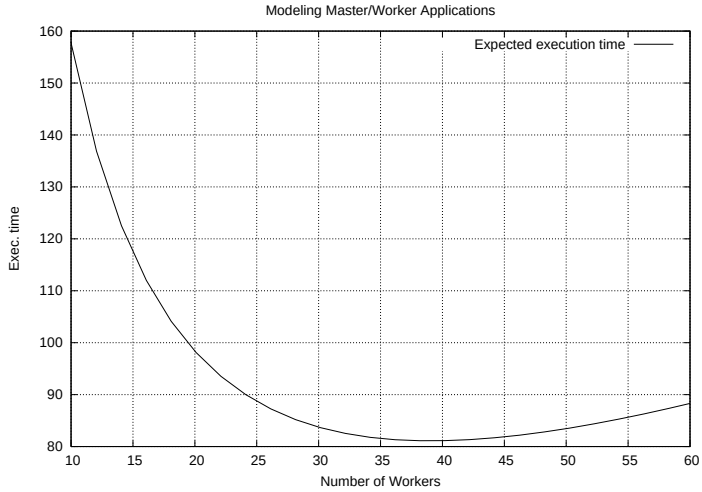


Figure 23. Expected execution time of an asynchronous Master/Worker application where V = 4096 bytes, and Tc=1600 ms.

Consequently, the definition of a performance index, which not only takes into consideration the performance gain but also the 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.

Several efficiency indexes could be defined for the purpose of taking the best decision when changing the number of workers. They may range from simple ones, such as fixing a lower bound for the relationship 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, which directly relates the performance with the efficiency in the use of resources, like the one defined in [HS+04]. The main advantage of such an index is that it can be automatically optimized because we can find the best possible relationship between efficiency and performance gain. Furthermore, we are going to discuss and illustrate in more detail some of the

indexes previously mentioned in order to emphasize the importance of having at hand a function that can be automatically optimized.

In the first place, we are going to take a look at a simple index, such as fixing a lower bound to the relationship between observed speedup (sequential execution time/observed execution time -Tt-) and ideal speedup (number of workers). That index indirectly takes into consideration the efficient use of the resources because if the observed speedup stays close to the ideal one then we are making good use of the assigned resources. However, the problem is to determine how close is close enough.

We show in figure 24 the expected and ideal evolution of the speedup for the example of figure 23; it can be seen that the distance between them becomes significant from 15 workers on. At that point, the relation among the expected and the ideal speedup is 0.87, it is 0.79 for 20 workers, it is 0.63 for 30 workers, and it is 0.49 for 40 workers. Supposing now that we have another application with the same parameters, except for the communication volume (which is 81Kbytes this time), recalculating the index we will get 0.42 for 29 workers, 0.36 for 38 workers, 0.28 for 56 workers, and 0.22 for 74 workers (where the model predicts the new lowest execution time -n_{opt}-). A careful look will reveal that we are indicating the value of the index in equivalent points: for n_{opt}, $(3/4)n_{opt}$, $(1/2)n_{opt}$, and $(3/8)n_{opt}$, but they are quite different from one configuration to the other. The problem is that what can be considered a good value for this index also depends on the communication volume. Consequently, for the user to be able to decide a good index threshold he must also be aware of the application communication volume.

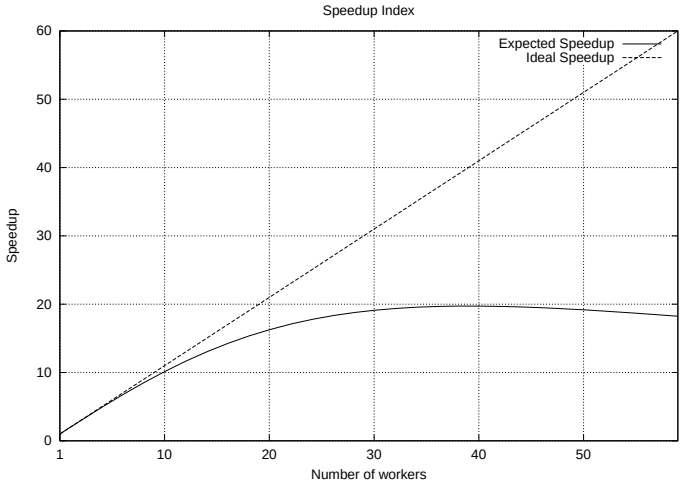


Figure 24. Expected speedup and ideal speedup for the application of figure 23.

In the second place, we are going to take a look at an index that is based on relating the variation rate of a performance function (like Tt or Speedup) to the amount of resources needed to achieve it. This index has the advantage of taking into consideration both the application performance improvement and the resources invested to obtain it, but again the user will be responsible for fixing a threshold for what will be regarded as a good value.

For instance, if we use the expressions for estimating the execution time (Tt) to define this index, we can define the variation rate as (Tt(y) – Tt(x))/Tt(y), which produces a value in the range (-1, 1). An absolute value of this function close to 1 indicates a significant variation of the execution time; on the contrary, a value of 0 indicates no variation at all. In addition, we define a resource variation rate as (x – y)/x, which also produces a value in the range (-1, 1), except 0 because it means that the number of resources does not change. An absolute value of this function close to 1 indicates a significant variation of the number of resources, while a value close to 0 indicates little variation on this number. Finally, we combine both expressions dividing the first one by the second one to get the following expression: $\frac{(Tt(y)-Tt(x))x}{(x-y)Tt(y)}$. This index will produce a value in the range [-1, 1], if this value is close to one then the application is expected to make good use of the additional resources. If the value is close to 0 then the application will not make efficient use of these resources, and if the value is below 0 then the performance of the application is expected to worsen iwith the additional resources.

In order to illustrate the usefulness of his index, we can use again the example of fig 23. For this example, if we are executing the application with 15 workers and want to know if it will be appropriate to go to 20 workers, we will get a value for the index of $\frac{(122.939733-101.2048)20}{(20-15)122.939733}=0.707$, and if we are executing the application with 30 workers and want to evaluate if it is worth going to 40 workers we will get $\frac{(84.469867-81.1024)40}{(40-30)84.469867}=0.16$. As it was expected, the index indicates that the first action is significantly better than the second (we mentioned before that going from 15 to 20 workers implies a performance improvement of more than 17%, while going from 30 to 40 workers implies an improvement of only 4%).

However, we wanted to know if we would get similar values for similar situations on other applications (we have just seen that it was not the case for the speedup

based index); thus, we used the same second example as before, which is the same application but with a data volume (V) of 81Kbytes instead of 4Kbytes. In order to get comparable results we evaluate equivalent transitions, which are going from 28 to 37 workers (from $(3/8)n_{opt}$ to $(1/2)n_{opt}$), and going from 55 to 74 workers (from $(3/4)n_{opt}$ to n_{opt}), getting an index value of 0.43 for the former and a value of 0.27 for the latter. These results are sound because the first transition leads to a performance improvement of 10.5%, which is worse than the one we got for the first example, while the second transition leads to an improvement of 6.9%, which this time is better than the one we got for the first example. Again, it is easy to see the difficulty for the user who has to decide a threshold for the index because what can be considered a good value for the index will depend on some application characteristics.

As a result, if using an index that asks the user for a threshold is quite difficult, we can see why it is very important to define an index that can be optimized without its intervention. We have adapted the index described in [HS+04] to our model, the basic idea behind this index, and its main difference with the ones described previously, is to define an efficiency index and relate it to the application performance. This efficiency index is defined as the portion of time that workers are doing 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 $\frac{Tc}{T_{avail}}$, where T_{avail} is $\sum_{i=0}^{x-1} tavail_i$, 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, where workers are not created or eliminated in the middle of an iteration, will be the whole iteration time (Tt). Consequently, the efficiency index will be defined as $\frac{Tc}{xTt(x)}$, and the performance index as:

$$Pi(x) = \frac{Tt(x)}{E(x)} = \frac{xTt(x)^2}{Tc} \quad \textbf{(17)}$$

We show in figure 25 the performance index value and execution time for the example of figure 23 for a number of workers ranging from 5 to 50. It can be seen that the performance index reaches its minimum value at the point from where adding more workers to the application is not expected to significantly improve its performance. Actually, there is only a 13.5% performance gain from 23 workers

Min (10 workers)

(93.7433 ms) to 40 workers (lowest expected execution time of 81.1024), increasing the amount of resources in by 74%.

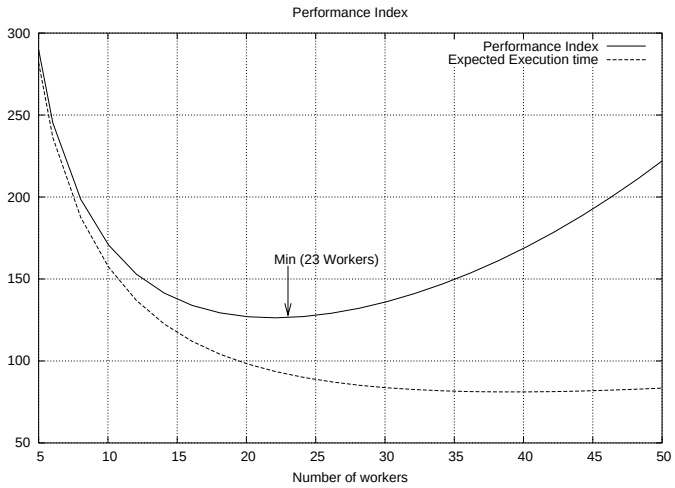


Figure 25. Performance Index (17) and expected execution time for the application of figure 23.

We introduce two more examples, shown in figures 26 and 27, in order to illustrate the index outcomes for applications modeled by expressions (8) and (9) respectively. The first example (figure 26) shows the expected execution time for a Master/Worker application with a communication volume (V) of 200Kbytes (90% distributed by the Master, 10% answered by workers), a processing time (Tc) of 2s, a network overhead of 1ms, and a network transference speed of 0.001 ms/byte. For the second example (figure 27), only the communication volume (V) has been changed from 200Kbytes to 20Kbytes.

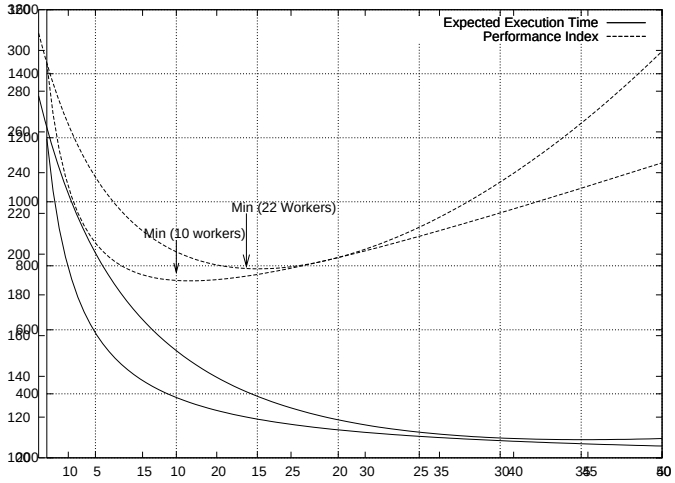


Figure 26. Performance Index (17) and expected execution time for an asynchronous Master/Worker application with V=200Kbytes and Tc=2s.

It can be seen in figure 26 that, although the expected execution time is expected to steadily decrease (the condition for the application of expression (8) is true), we are getting the performance index lowest value at 10 workers, which is just two workers before this application is expected to surpass the *Master's Chunk Managing Capability* (MCMC), which was calculated using expression (11). Actually, it can be demonstrated that for applications modeled by expression (8) the minimum value of this performance index will always be obtained before exceeding the MCMC. First, if we substitute the term Tt(x) of expression (17) by expression (8) the resulting expression is the particularized performance index for applications modeled by expression (8); next we calculate the first derivate of this expression ($\frac{\partial Pi(x)}{\partial x}$), and finally we equate the resulting expression to 0 (searching for a minimum) and solve for x. This process, which applied to expressions (7) and (9) leads to hard to solve quartic expressions, produces for expression (8) the following result:

$$x = \left\lfloor \frac{((1-\alpha)\lambda V + Tc)}{\lambda \alpha V + m_o} \right\rfloor .$$

Comparing this expression with expression (11) $\left\lfloor \frac{\lambda V + Tc}{\lambda \alpha V - m_o} \right\rfloor$, we can see that the dividend is smaller and the factor is greater than the ones of expression (11), consequently the value of expression (11) will be always greater than or equal to x.

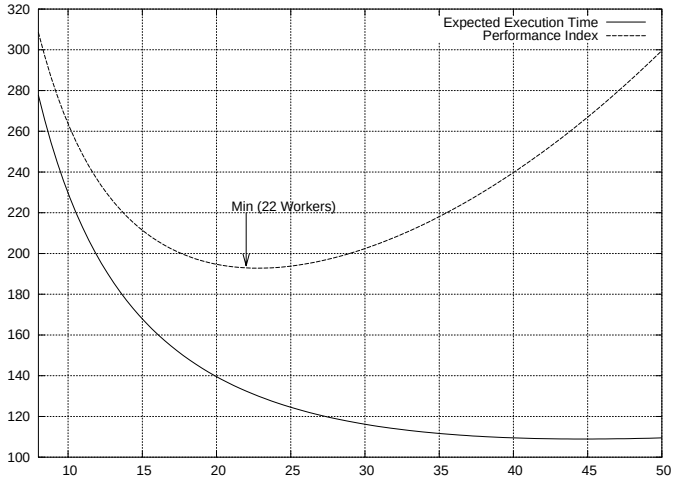


Figure 27. Performance Index (17) and expected execution time for a synchronous Master/Worker application with V=20Kbytes and Tc=2s**.**

In the last example of figure 27 it can be seen that the performance index reaches its lowest value at 22 workers, which is less than the number of workers that causes the Master to exceed its MCMC (37 workers accordingly to expression (12)), and

obviously less than the number of workers for the lowest expected execution time (44 workers accordingly to expression (9)).

Finally, there is the problem of determining for what number of workers we get the lowest index value. We have seen before that it is easy for applications modeled by expression (8), but when the application is modeled by expressions (7) or (9) and following the same steps described above (basically solving the $\frac{\partial Pi(x)}{\partial x} = 0$ for x) we find that the following quartic expressions must be solved for x:

$3m_o x^4 + 4m_o x^3 + (2(Tc + \lambda V) + m_o^2)x^2 - (Tc + \lambda V)^2 = 0$ (for expression (7) and making $\mu_m = 0$),

$3m_o x^4 + 4m_o(\alpha\lambda V + m_o)x^3 + (2m_o((1-\alpha)\lambda V + Tc) + (\alpha\lambda V + m_o)^2)x^2 - ((1-\alpha)\lambda V + Tc)^2 = 0$

(for expression (9) and making $\mu_m = 0$).

The roots of these expressions can be found using the Ferrari technique [Weiss] or, knowing that we are only interested on finding the floor of the first positive root of the expression, using a simpler bisection procedure.

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.

3.4. 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 2.4. The characteristics of the applications we have executed are summarized in table 4, where it can be seen that we have covered a wide range of possibilities, from low-compute low-communication to intensive compute and communication applications in order to reach this validation goal. Moreover, we have executed several configurations with a constant communication volume and an asynchronous communication protocol, then some configurations using a synchronous communication protocol, some configurations with a variable communication volume, and finally, some configurations using a synchronous communication protocol and a variable communication volume.

Comm. / Vol.(V) Proc Time (Tc)		10 Kbytes	100 Kbytes	512 Kbytes	2 Mbytes
1 sec	fig 28	From 2 to 40 workers. Constant com. volume & async. com.	From 2 to 40 workers. Constant com. volume & async. com.	From 2 to 35 workers. Constant com. volume & async. com.	From 2 to 20 workers. Constant com. volume & async. com.
5 sec	fig 29	From 4 to 55 workers. Constant com. vol & async. com.	From 4 to 55 workers. Constant com. vol & async. com.	From 4 to 40 workers. Constant com. vol & async. com.	From 2 to 20 workers. Constant com. vol & async. com.
	fig 31	From 2 to 55 workers. Constant com. vol & sync com.	From 2 to 55 workers. Constant com. vol & sync com.	From 2 to 55 workers. Constant com. vol & sync com.	From 2 to 55 workers. Constant com. vol & sync com.
	fig 32	From 2 to 40 workers. Async. com. & com. volume that grows a 10% of the original for each new worker.	From 2 to 40 workers. Async. com. & com. volume that grows a 10% of the original for each new worker.	From 2 to 40 workers. Async. com. & com. volume that grows a 10% of the original for each new worker.	From 2 to 30 workers. Async. com. & com. volume that grows a 10% of the original for each new worker.
	fig 33	From 2 to 40 workers. Async. com. & com. volume that grows a 100% of the original for each new worker.	From 2 to 35 workers. Async. com. & com. volume that grows a 100% of the original for each new worker.	From 2 to 20 workers. Async. com. & com. volume that grows a 100% of the original for each new worker.	
	fig 34	From 2 to 40 workers. Sync. com. & com. volume that grows a 30% of the original for each new worker.	From 2 to 40 workers. Sync. com. & com. volume that grows a 30% of the original for each new worker.	From 2 to 30 workers. Sync. com. & com. volume that grows a 30% of the original for each new worker.	From 2 to 30 workers. Sync. com. & com. volume that grows a 30% of the original for each new worker.
15 sec	fig 30	From 2 to 55 works. Constant volume of com, asynchronous communication.	From 2 to 55 works. Constant volume of com, asynchronous communication.	From 2 to 55 works. Constant volume of com, asynchronous communication.	From 2 to 55 works. Constant volume of com, asynchronous communication.

Table 4. Summary of the configurations executed in order to test the analytical performance model for Master/Worker applications.

The synthetic application algorithm can be summarized as follows:

1. Master & Workers calculate the size of messages that will be transferred between them, and the computation time associated with each worker, using the number of workers (x), communication volume (V), and processing time (Tc) arguments.
2. The Masters goes into a loop (number of iterations argument) for sending messages (using the indicated communication protocol) to every Worker and then waiting for answers from them all.
3. Each worker goes into a loop (number of iterations argument) waiting for a message from the Master, spending the associated processing time, and

answering back with a new message (using the indicated communication protocol).

The results obtained for each test configuration has been processed in order to eliminate statistical anomalies. Then, they have been plotted together the results predicted by the analytical performance model and the real and predicted values of the performance index for the same configuration. Finally, we have grouped four graphs in each figure (except in fig 33, which only includes three) corresponding to the configurations of a file of table 4, and each figure is supplemented by a table with some relevant magnitudes: the number of workers for the real and expected lowest execution time, the number of workers for the real and expected lowest performance index value, and the real and expected highest number of workers the Master can manage (real values are only shown if they are available).

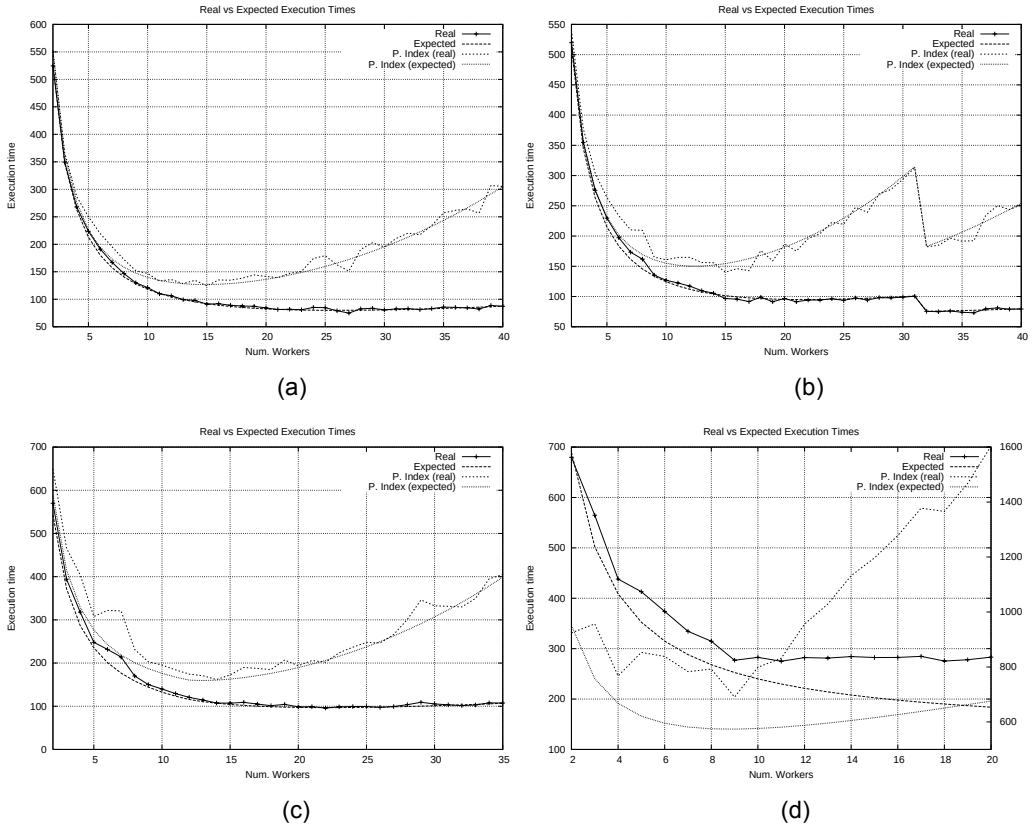


Figure 28. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 1 sec. and constant communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d). Ranging from 2 to 40 workers for cases (a), (b) and (c), and from 2 to 20 workers for case (d). Using asynchronous communication.

Using the experiments of figure 28, we show the results of executing an application with a low associated processing time of only one second and for different communication loads using the standard communication mode, which in

this case means asynchronous sends. Whereas, in general it can be seen that model matches the application behavior, there are some cases that deserve more detailed comments.

Firstly, it can be easily seen that small 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 ($Tt(x)^2$) introduces in the performance index. However, we can trust the expected value because, in general, the index tendency is preserved.

Secondly, a sudden discontinuity can be seen in figure 28 (b) for 32 workers, which is due to the communication library implementation. MPI library uses different communication protocols depending on the message size and buffers available [MPI95]. 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 removing some hand-shake messages; thereby 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; it 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 little messages and blocking or synchronous ones for long messages.

		Number of Workers for the Lowest Execution Time	MCMC Exceeding Point	Number of Workers for the Performance Index Lowest Value
Fig. 28 (a)	Real	30	—	15
	Expected	26	—	15
Fig. 28 (b)	Real	36	—	15
	Expected	32	—	12
Fig. 28 (c)	Real	22	—	14
	Expected	22	—	13
Fig. 28 (d)	Real	11	9	9
	Expected	64	10	9

Table 5. Relevant real and expected magnitudes associated with applications of figure 28.

Consequently, as far as we have used the standard communication mode in the synthetic applications, what is reflected in the graph is the change between *blocking* communication mode and the *ready* one. The possibility of having changes like this should be taken into consideration by the tuning tool (monitoring the message overhead) because they could cause significant mismatches between the observed results and the predicted ones.

Finally, the differences between the observed and the predicted values of case 28 (d) should be highlighted, in order to emphasize the relevance of calculating the Master's Chunk Managing Capability (MCMC) exceeding point (table 5). It can be seen that from 9 workers on, the divergence between the observed and expected values becomes bigger and bigger all the time. In addition, we do not include the MCMC exceeding point for the other cases because it is always beyond the point where the lowest execution time is reached.

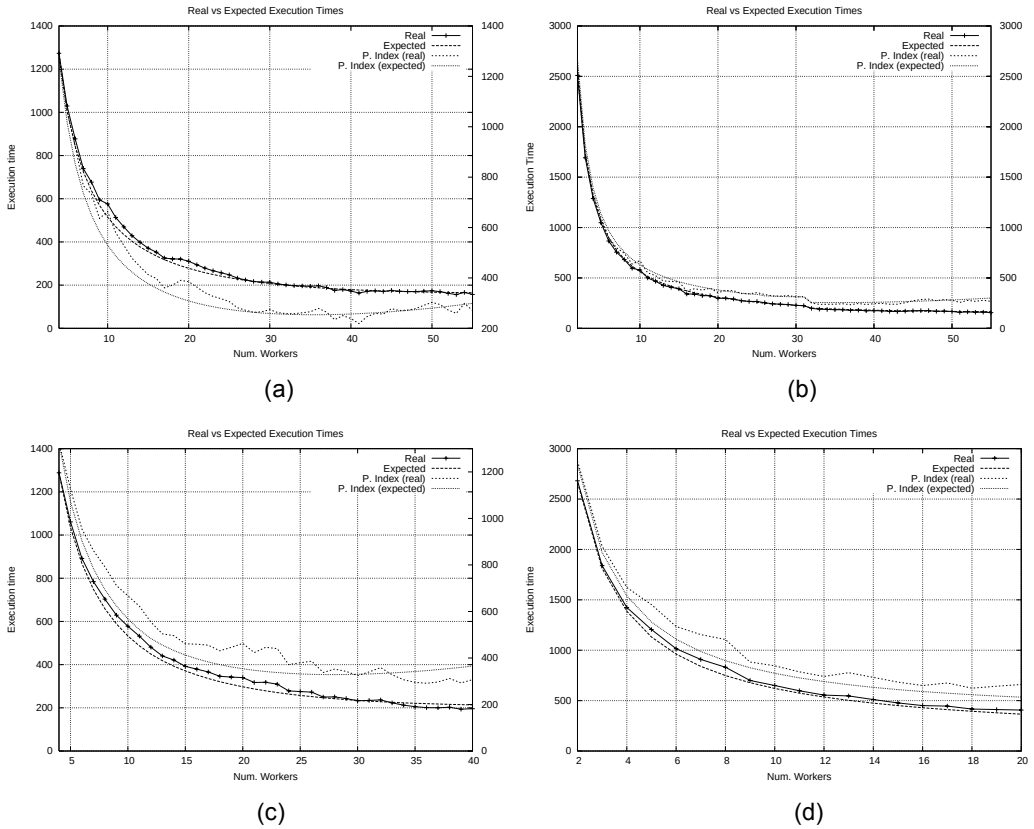


Figure 29. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 5 sec. and constant communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d). Ranging from 4 to 55 workers for cases (a) and (b), from 4 to 40 workers for case (c), and from 2 to 20 workers for case (d). Using asynchronous communication.

In the graphs of figure 29 we show the results of the execution of an application with an average 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 figure 28; nevertheless, it must be noticed that this increment is far from proportional to that of the processing time. Moreover, it seems that there are less mismatching points between the observed and the expected results than for the previous figure, which is because in the previous case the application executions were more sensitive to any external influence (such as a context switching) due to the low computing time associated with each worker, mainly when several workers were used.

A very good example of this fact can be seen in figure 29 (b), again we are reaching the point of communication mode switching of the communication library at 32 workers, but this time the effect is just barely reflected in the graph.

Finally, we can see in table 6 significant differences between the expected and the observed number of workers where lowest performance index is reached. We can partially blame the multiplicative effect of the total expected and observed execution time (Tt(x)) differences in the performance index expression for this fact. 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 too high. However, in the particular case of figure 29, we are getting significant differences of 21.47% for case (a) and of 20.9% for case (c) because in both cases 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 them we have gotten a few observed values below the expected ones.

		Number of Workers for the Lowest Execution Time	MCMC Exceeding Point	Number of Workers for the Performance Index Lowest Value
Fig. 29 (a)	Real	—	—	41
	Expected	62	—	35
Fig. 29 (b)	Real	—	—	37
	Expected	62	—	35
Fig. 29 (c)	Real	—	—	36
	Expected	49	—	28
Fig. 29 (d)	Real	—	—	—
	Expected	50	50	46

Table 6. Relevant real and expected magnitudes associated with applications of figure 29.

In the graphs of figure 30, we show the results of the execution of an application with a high associated processing time of fifteen seconds and for different communication loads. Again, a close matching 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 again that more workers can be added to the application (table 7) than for the previous case (table 6) before reaching the applications' lowest execution time and performance index lowest value, but also that this increment 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; it 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 ($Pi(x) = Tt(x)^2/Tc \approx Tt(x)$). Moreover, the low weight of communications also makes the protocol change of graph 30 (b) go unnoticed.

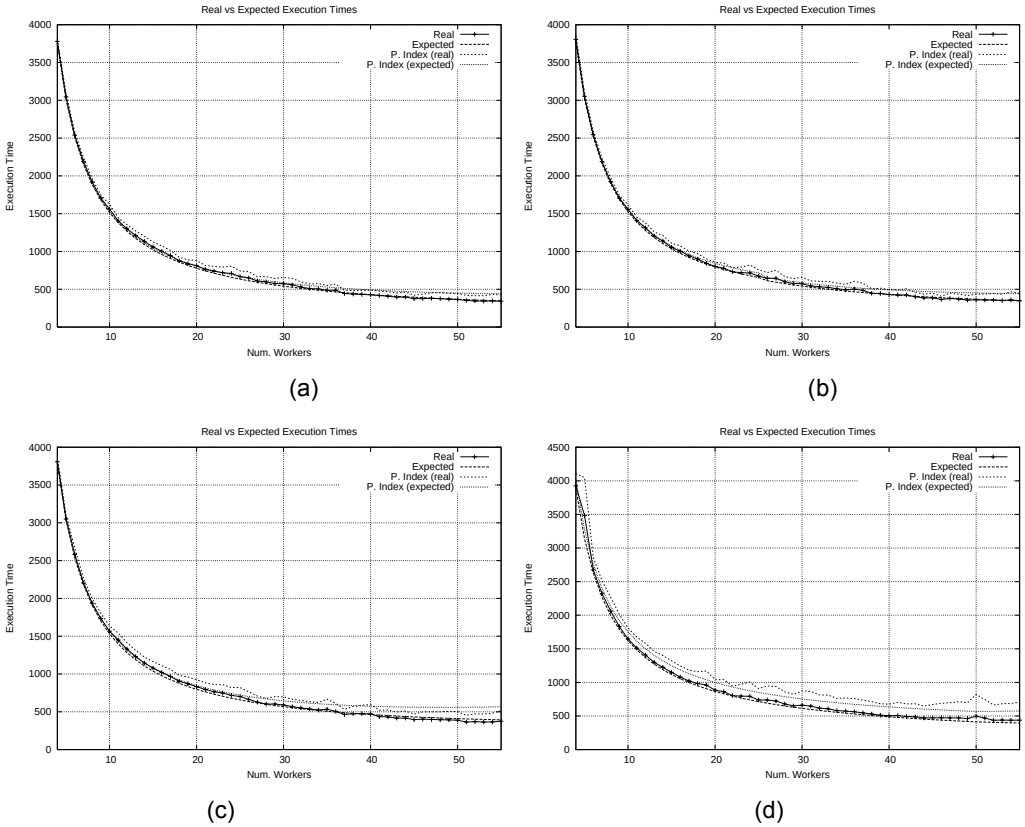


Figure 30. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 15 sec. and constant communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d). The range is from 4 to 55 workers in all cases and asynchronous communication is used.

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 figure 29. In particular, there are differences of only 1.3% and 1.9% between the observed and the expected lowest index value for cases (b) and (c) respectively.

		Number of Workers for the Lowest Execution Time	Number of Workers for the Performance Index Lowest Value
Fig. 30 (a)	Real	—	—
	Expected	107	61
Fig. 30 (b)	Real	—	—
	Expected	107	61
Fig. 30 (c)	Real	—	45
	Expected	84	48
Fig. 30 (d)	Real	—	44
	Expected	84	50

Table 7. Relevant real and expected magnitudes associated with applications of figure 30.

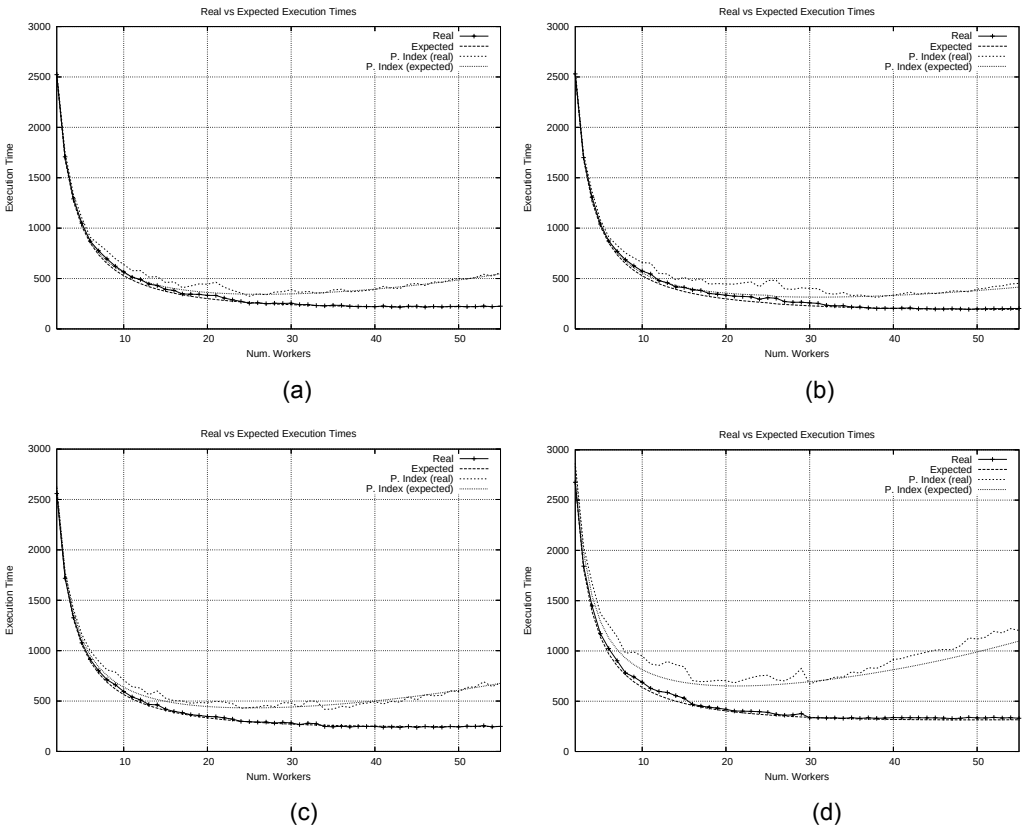


Figure 31. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 5 sec. and constant communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d). The range is from 2 to 40 workers for cases (a), (b) and (c), and from 2 to 30 workers for case (d). Synchronous communication is used.

In the graphs of figure 31, we show the results of the execution of an application with an average associated processing time of five seconds and for different communication loads using a synchronous communication protocol, which in a 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 closer match between the observed values and the expected ones, and we can say that, comparing these results with the ones shown in figure 29 (same application using standard communication protocol), using a synchronous communication protocol produces a significant decrease in the application's performance; as a consequence, the lowest execution time and lowest performance index values are reached with less workers. Actually, differences become greater when more workers (thus more communications) are added. For instance, with a communication volume of 100 Kbytes the differences ranges from 0.9% for 2 workers to 17% for 40 workers.

Moreover, there is a result that should be especially highlighted: it is the influence of the communication protocol on the MCMC for big messages (table 8), if compared to the results shown in figure 29. However, it should be noticed that, despite the performance index sensitivity to little variations of the total execution time, the index lowest value is reached before the MCMC exceeding point. This is happening because, for applications using this protocol, when the MCMC exceeding point is reached the possible performance gains are quite limited due to the dominating communication component of the total execution time.

		Number of Workers for the Lowest Execution Time	MCMC Exceeding Point	Number of Workers for the Performance Index Lowest Value
Fig. 31 (a)	Real	—	—	27
	Expected	55	—	27
Fig. 31 (b)	Real	—	—	38
	Expected	55	—	31
Fig. 31 (c)	Real	45	41	34
	Expected	49	43	25
Fig. 31 (d)	Real	48	30	21
	Expected	46	30	21

Table 8. Relevant real and expected magnitudes associated with applications of figure 31.

Using the experiments of figure 32 we show the results of executing an application with a medium associated processing time of five seconds and for different communication loads using the standard communication mode, which in this case

means asynchronous sends. The difference with the application of figure 29 is that the overall communication volume increases with each worker added to the application. In this case, with each new worker, starting from the second, the communication volume (V) increases by 10% of the original value, it is 1Kbyte/worker for the 10Kbytes case, 10Kbytes/worker for the 100Kbytes case, 51.2Kbytes/worker for the 512Kbytes case, and 204.8Kbytes/worker for the 2Mbytes case.

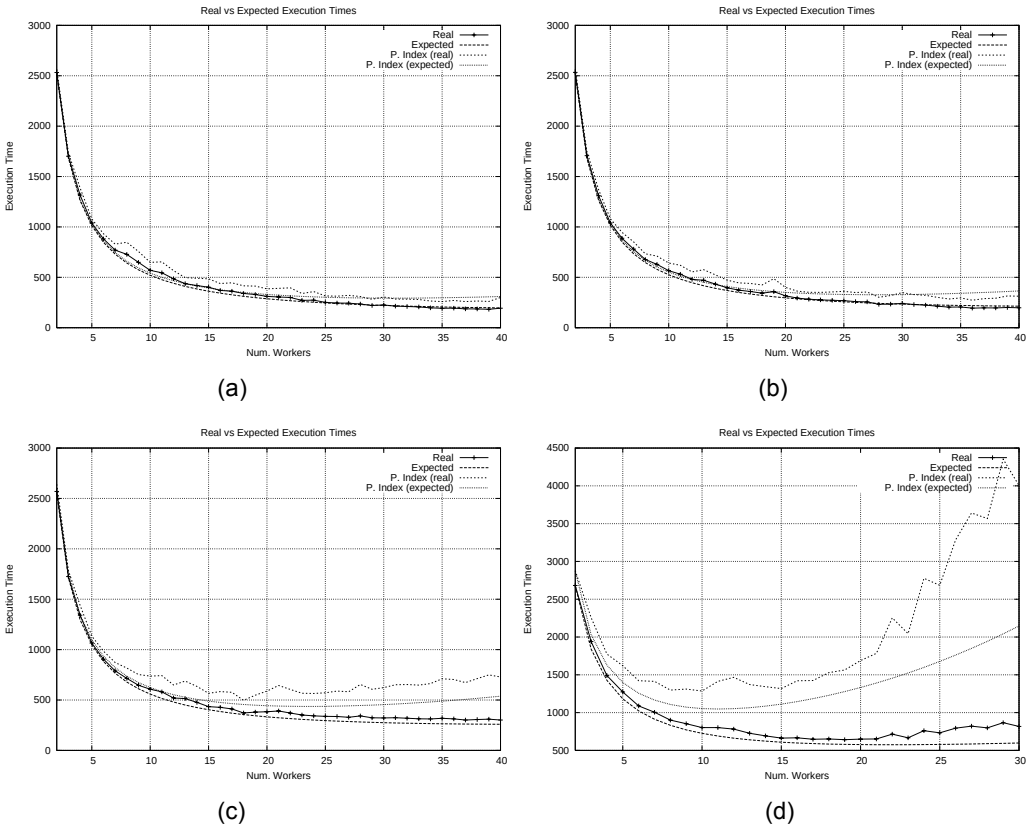


Figure 32. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 5 sec. and initial communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d), which are incremented in a 10% for each added worker. The range is from 2 to 40 workers for cases (a), (b) and (c), and from 2 to 30 workers for case (d). Asynchronous communication is used.

As expected, there is, in all cases, a negative effect on the application performance when compared to the results of figure 29. In addition, this negative effect is more significant when the communication volume is larger. For instance, we can see in this example that for an initial communication volume of 10 Kbytes (figure 32 (a)) the results are comparable to (slightly better than) those obtained for the synchronous communication protocol of figure 31 (a), while for a communication volume of 2 Mbytes (figure 32 (d)) the results are significantly worse than those of figure 31 (d).

We can also see that there are again significant differences between the real and expected number of workers for the performance index lowest value (table 9). However, the highest difference between the expected and observed index values is of only 13.5% for figure 32 (c), not far from the 9.6% of figure 32 (a), the 10.9% of figure 32 (b) and the 9.9% of figure 32 (d).

		Number of Workers for the Lowest Execution Time	MCMC Exceeding Point	Number of Workers for the Performance Index Lowest Value
Fig. 32 (a)	Real	—	—	37
	Expected	54	—	31
Fig. 32 (b)	Real	—	—	36
	Expected	48	—	28
Fig. 32 (c)	Real	—	—	18
	Expected	44	41	24
Fig. 32 (d)	Real	19	19	10
	Expected	22	19	11

Table 9. Relevant real and expected magnitudes associated with applications of figure 32.

The experiments of figure 33 show the results of executing an application with a medium associated processing time of five seconds and for different communication loads using the standard communication mode, which in this case means asynchronous sends. The difference with the application of figure 29 is that the overall communication volume increases with each worker added to the application. In this case, with each new worker, starting from the second, the communication volume (V) increases in a 100% of the original value, it is 10Kbyte/worker for the 10Kbytes case, 100Kbytes/worker for the 100Kbytes case, and 512Kbytes/worker for the 512Kbytes case. We have skipped the 2 Mb case because there won't be any benefit from parallelization for an application with the given characteristics.

It can be seen that for an application with little communication (10Kb case), like the one of figure 33 (a), it is possible to improve performance even if adding more workers implies a relatively significant growth of the communication volume. On the other hand, an application that from the beginning has an important communication component, like the one of figure 33 (c), will receive very little advantage from the addition of new resources.

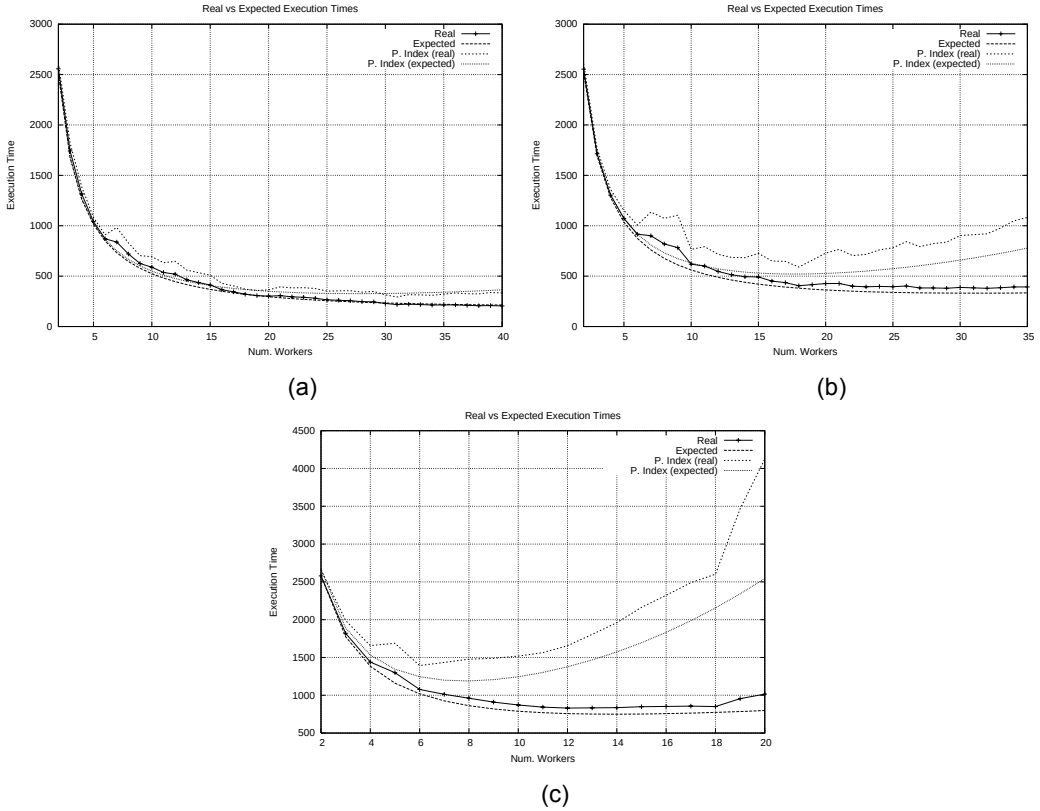


Figure 33. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 5 sec. and initial communication volumes of 10Kbytes (a), 100 Kbytes (b), and 512 Kbytes (c), which are incremented in a 100% for each added worker. The range is from 2 to 40 workers for case (a), from 2 to 35 workers for case (b), and from 2 to 20 workers for case (c). Asynchronous communication is used.

		Number of Workers for the Lowest Execution Time	Number of Workers for the Performance Index Lowest Value
Fig. 33 (a)	Real	—	31
	Expected	48	28
Fig. 33 (b)	Real	32	18
	Expected	31	18
Fig. 33 (c)	Real	12	6
	Expected	14	8

Table 10. Relevant real and expected magnitudes associated with applications of figure 33.

The experiments of figure 34 show the results of executing an application with a medium associated processing time of five seconds and for different communication loads using the synchronous communication protocol. The difference with the application of figure 31 is that, this time, the overall communication volume increases with each worker added to the application. In this case, with each new worker, starting from the second, the communication volume (V) increases by 30%

of the original value: it is 3Kbyte/worker for the 10Kbytes case, 30Kbytes/worker for the 100Kbytes case, 153.6Kbytes/worker for the 512Kbytes case, and 614.4Kbytes for the 2Mbytes case.

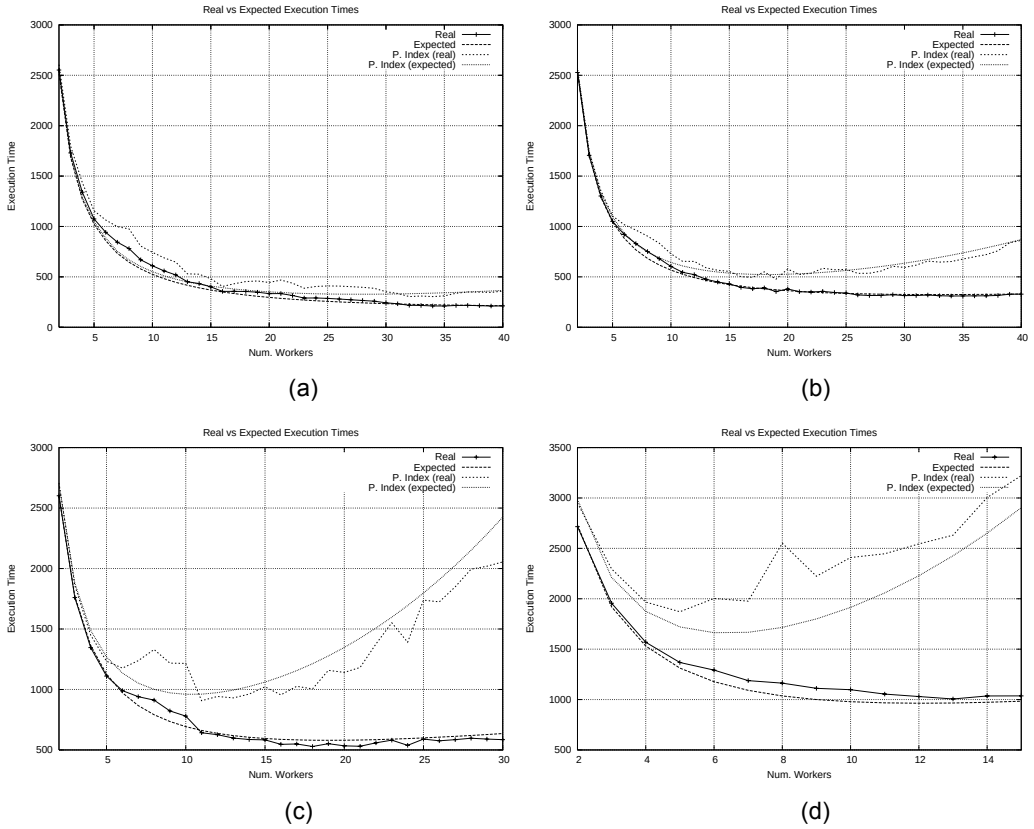


Figure 34. Execution times and Performance Index values of a Master/Worker application with an associated processing time (Tc) of 5 sec. and initial communication volumes of 10Kbytes (a), 100 Kbytes (b), 512 Kbytes (c) and 2Mbytes (d), which are incremented by 30% for each added worker. The range is from 2 to 40 workers for cases (a) and (b), from 2 to 30 workers for case (c), and from 2 to 15 workers for case (d). Synchronous communication is used.

		Number of Workers for the Lowest Execution Time	MCMC Exceeding Point	Number of Workers for the Performance Index Lowest Value
Fig. 34 (a)	Real	—	—	34
	Expected	48	—	28
Fig. 34 (b)	Real	34	—	19
	Expected	33	—	18
Fig. 34 (c)	Real	21	—	11
	Expected	19	—	10
Fig. 34 (d)	Real	13	13	5
	Expected	12	12	6

Table 11. Relevant real and expected magnitudes associated with applications of figure 34.

As expected, once we have separately seen the effects of synchronous communication and communication volume increase, the combination of both facts has a very significant worsening effect on the application's performance, which in some cases is as bad as the one caused by the 100% increment shown in figure 33 (see figure 34 (a) and (b)).

4. Global Master/Worker Model and Last Considerations

Our objective, in the last section of this chapter, is to sum up the performance model for Master/Worker applications according to the general model for dynamic performance tuning presented in Chapter II. Firstly, we summarize the set of expressions and strategies that must used by the dynamic tuning tool to evaluate the performance of the application and predict what will happen if some conditions change. Next, we indicate which are the application parameters that must be monitored at run time in order to be able to detect the performance bottlenecks. Finally, the parameters that can be changed at run time to improve the applications performance and when can those changes take place are indicated. In addition, given that the proposed performance model includes two phases (load balancing & adjusting the number of workers), some considerations are included in this summary about the conditions that must hold to guarantee that both phases can be safely combined.

In the first place, we have shown at the beginning of this chapter that, in order to improve the performance of a Master/Worker application, we should be able to balance the workers' load and then to 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 applications, 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 workers load, we have adopted a partial task distribution policy, which basically consists of dividing the set of tasks to be processed in sub-sets called batches and then distributing them among workers one by one in units called chunks. The number of tasks to be included in each batch is determined by a distribution strategy by calculating a partition factor. We have proposed a distribution balancing strategy that has been called: Dynamic Adjusting Factoring (section 2.3), which is based on making a self-adaptative partial distribution of the tasks to be processed. The main goal behind this strategy is to

statistically minimize the possibility that the time spent by a worker processing its current assigned chunk surpasses the optimal processing time of Tc/N (where Tc is the total processing time associated with the set of tasks and N the current number of workers). In order to reach this goal, the distribution strategy dynamically adapts the partition factor taking into consideration the mean processing time and standard deviation per task (expressions (1) & (2)).

A balanced application makes good use of its assigned resources and, as a result, achieves the best possible performance for those resources. However, it should be evaluated if it is possible to get further performance improvements 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 from 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. The model 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 expressions to determine the Master's workers management capacity, which is the highest number of workers that the master can send tasks to before receiving the first answer. It is called 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; once 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 (17)) 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, and in figure 35 we can see that the overall two phase model is also sound. In this figure, we are recovering the example of figure 2 adding the observed execution time of the balanced application for different numbers of workers, in order to show that the performance of an

application balanced through the use of a distribution strategy can be improved by adjusting the number of workers assigned to it.

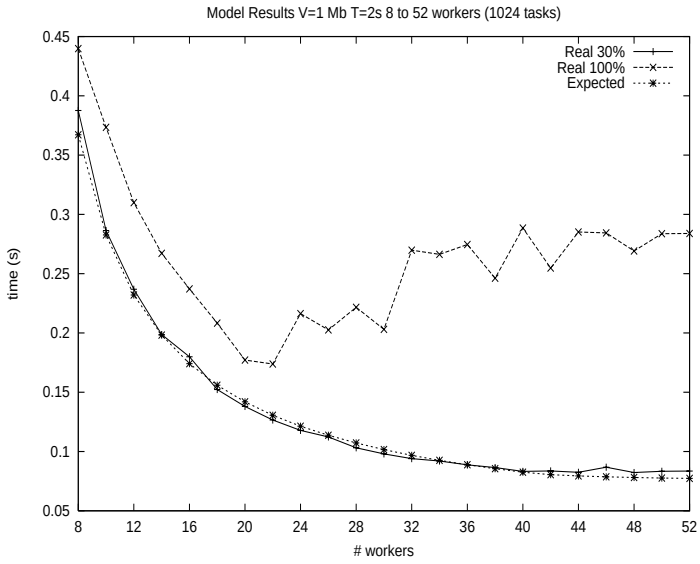


Figure 35. Real (with and without applying a load balancing strategy) and expected execution times of a Master/Worker application using from 8 to 52 processors, processing 1024 1Kb tasks each iteration, and task associated processing time distributed according to table 1 (section 1).

However, assuming that we start from a balanced execution, it is not known in advance if the distribution strategy will able to succeed in balancing the load for a different number of workers. There are some conditions that must hold in order to make it possible: one is that the number of tasks should be considerably greater than the number of workers, and the other that a major portion of processing time should not be concentrated on a small portion of tasks. Moreover, as 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 distribution strategy outcome and to predict if it is likely to succeed in balancing the load for a different number of workers.

One possible way for evaluating the quality of the distribution policy outcome is by relating the mean idle time per processor with the total execution time through an expression like: $1-(\frac{1}{x}\sum_{i=0}^{x-1}\mu_i \Big/ Tt(x))$ (where x is the number of workers, μ_i the processing time spent by worker i, and Tt(x) the total iteration time). For this expression, a 0 value indicates a perfectly balanced application (all workers have been busy the whole iteration), while a value closer to 1 indicates that there is a significant load unbalance in the application. Nevertheless, this expression cannot be used for predicting the quality of the distribution strategy outcome for a different

number of workers because we cannot apply expressions (7), (8), or (9) for predicting the total execution time in that they are only valid for balanced applications.

On the other hand, as we mentioned in section 2.3, 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: $Tt(x) \leq \mu(N/x) + \sigma N/\sqrt{2x}$, for a Master/Worker application with N tasks to process, a mean processing time of μ, and a standard deviation of σ. Then, substituting Tt(x) by its upper bound in the previously defined quality index leads to the following upper bound of it: $1/(1+\frac{\mu}{\sigma}\sqrt{\frac{2}{x}})$. As a result, this expression can be used as a guide to decide if it is worthwhile to change the number of workers depending on how easy it will be to balance the application load. However, the defined execution time upper bound is quite conservative (it can be much greater than the real value) and can lead to rejecting configurations that have a reasonable chance of improving the application performance.

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, which 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 are in accordance with the conditions required for balancing the load of an application that were stated before: a number of tasks significantly greater than the number of workers, and a major portion of the processing time should not be concentrated on a small number of tasks.

In the second place, 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 allowing the adaptation of the system to the network load conditions.
- *Message sizes ($v_i^{m/w}$)* have to be captured when 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), but also to track the relation between the communication volume and the number of workers (Δ_V).

- *Workers' 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 (Tc). 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 workers load or the expression for adapting the number of workers, some changes might be introduced in the application at run time. It is very important to know exactly what parameters should be changed and when can these changes 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 moment.
- *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, and only if the added workers (if the number of workers has been increased) have already been set up and are ready to receive tasks.

Chapter IV: Pipeline Framework Performance Model

Abstract

In this chapter, we present the performance model we have defined to dynamically improve the performance of applications developed with the Pipeline framework. The objective of this model is to improve the application's throughput by devising the best stage replication pattern on the available resources. We also present a set of experiments that validate the model.

1. Introduction

In this chapter we will introduce our proposal for a performance model associated with the Pipeline framework. Whereas, we have not defined a performance model as comprehensive and detailed for Pipeline applications as the one presented in Chapter III for Master/Worker ones (mainly because this is already a work in progress [CM+05]), we include this proposal with the aim of demonstrating that the general dynamic performance tuning methodology based on associating a performance model with high-level programming structures can be applied to programming structures other than the Master/Worker. To fulfill this objective, we will first recall the framework associated performance problems presented in Chapter II. Next, we will define, according to the general performance model introduced in the same chapter, the strategies and expressions that are part of our performance model and the experimentation that has been carried out to validate them.

In Chapter II, we have described the Pipeline framework that will be analyzed and the possible inefficiencies of pipelined applications. We saw that, on the one hand, the concurrency is limited at the beginning of the computation as the pipe is filled (also called ramp-in time), and at the end of the computation as the pipe is drained (also known as ramp-out time); this is a transient inefficiency that should be dealt with at the design phase of the application because the way to avoid it is to ensure that the number of calculations the application will perform is substantially higher than the number of stages of the pipe. On the other hand, it is important for there not to be any significant differences between the computational efforts of the pipe stages because the application throughput of a pipe is determined by its slowest stage. This is the most important inefficiency of this structure, and the most difficult to overcome because it does not depend exclusively on the application design, but also on run-time conditions. Consequently, this drawback is suitable for being solved dynamically and there are different approaches for doing so depending on the target index to be optimized and availability of resources.

Therefore, we may want to improve the efficiency in the use of resources, or even try to free some underused resources to increase their availability, in this case dynamic mapping of stages could be used to group fast stages; thereby improving the use of resources. On the other hand, we may want to improve the application throughput, in which case, if there are available processors, replicating slower

stages will increase throughput and reduce the application execution time. Furthermore, we may want to increase the application throughput but also make an adequate use of resources. Consequently, a mixed approach could be defined, as a compromise between optimizing throughput and efficient resource management.

Our aim is to implement a mixed strategy, with the main objective of optimizing the application throughput but also of making a reasonable use of resources. However, as a first step towards this objective we have concentrated on optimizing application throughput and, as a consequence, the model presented does not include considerations about the efficiency of resource management. Hence, we are going to discuss only the strategies and expressions that will be applied to decide, given a certain number of available resources, which pipe stages should be replicated in order to improve the application throughput.

2. Stage Modeling

The general strategy for increasing the throughput of the slower stages, in order to improve the global application performance, will consist of calculating the best replication pattern for the current application's characteristics and available number of processors.

Consequently, if we want to increase the application throughput, we must minimize the time needed by each stage to process its inputs, including the time required to deliver the results to the next stage. We call this quantity *the production time*; hence, we need expressions to find the production time each stage can reach (its *independent production time*), and also expressions that explain its observed production time due to the influence of other stages (*its dependent production time*). Moreover, we should find different expressions to make these calculations for single, and replicated stages.

Nevertheless, before introducing these expressions, we have to discuss the general model of a replicated stage. The idea is that the original application has been written using a linear pipeline framework, which is a simplification that does not affect the applicability of the model but reduces the complexity of the replication strategy. Then the tuning tool, after taking into consideration the performance analysis and available resources, could decide to replicate some stages. In addition, this replication must be transparent to the rest of the pipe stages, particularly to those stages immediately preceding and succeeding the replicated one. Therefore, a replicated stage consists of several copies of the original linear stage plus a data

management and distribution process that bridges these copies with the rest of the pipe.

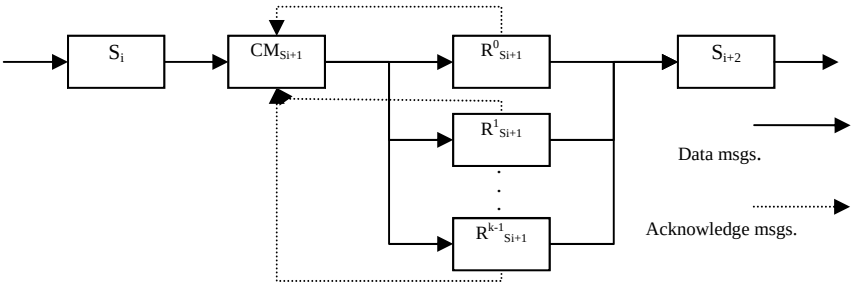


Figure 1. Schematic representation of a replicated stage. Stage i+1 has been replicated k times and a communication manager (CM) has been added to control the replicas' state and distribute incoming tasks.

This replicated stage scheme is shown in figure 1. It can be seen there that a new process, called Communication Manager (CM), is responsible for receiving messages from the preceding stage and distributing them among the stage replicas. The CM functionality consists basically of monitoring the replicas' state (which replicas are busy and which are free), receiving messages from the preceding stage (provided that there is a least one free replica in order to be consistent with the linear case where a stage only receives a message when it is free), and distributing these messages to free replicas. To be aware of the state of the replicas the CM receives an acknowledgement message each time one of them ends its processing.

In adition, we should decide whether the CM should run in a separate processor or should share one with a replica. The first approach is simpler to model but could lead to a poorer use of resources. The second, in contrast, seems to lead to a better use of resources, but is more difficult to implement with some communication libraries, and is also difficult to model because the CM affects, and is affected by, the activity of the replica that shares a processor with it. We have modeled both options, but we have only validated experimentally the first one; therefore, we will put a greater emphasis on this model.

Before defining the expressions that describe the behavior of a Pipeline application, we have also to indicate the parameters that we have taken into consideration and the terminology that will be used from now on. In the first place and for the same reasons explained in section 3.1 of Chapter III, we have characterized the interconnection network with the classical message start up time plus communication time formula. In the second place, in order to be able to evaluate the model expressions, we need to know the time each stage is making

useful computation, the amount of data sent and received to/from each stage, and the kind of communication protocol (synchronous or not).

Finally, as we did for the Master/Worker framework, we assume in our analysis that there is just one process per processor, although this time it is not a matter of effiecincy, as it was in the Master/Worker case, but a way of simplifying the analysis process. As previously mentioned, grouping quick stages in the same processor can be useful for balancing the application computational load while freeing resources. In addition, we use the following terminology:

- $\mathbf{m_o}$ = per message start up time, in ms.
- $\boldsymbol{\lambda}$ = per byte communication cost (inverse bandwidth), in ms/byte.
- $\mathbf{v_i}$ = data volume sent by stage i, in bytes.
- $\mathbf{tc_i}$ = computation time stage i needs to process an input, in ms.
- $\mathbf{Tr^k_i}$ = production time of k replica of stage i, in ms.
- $\mathbf{Tr_i}$ = independent production time of stage i, in ms.
- $\mathbf{rTr_i}$ = dependent production time of stage i, in ms.
- **P** = communication protocol (synchronous or asynchronous sends).

2.1. Single stage modeling

A single pipe stage is one that receives messages with data, except for the first one, makes its portion of calculation of this data, and sends the results to the next stage, except for the last one. It is clear that the independent production time (Tr_i) of such a stage will depend on its position in the pipe, its associated computation time (tc_i), and the current communication characteristics -$C(P,v_i)$- (communication protocol -P- and message size -v_i-).

This way, we can define the independent production time of a single stage as:

$$Tr_i = tc_i + C(P,v_i) \qquad \mathbf{(1)}$$

Where $C(P,v_i)$ is defined as:

$$0 \quad \text{if } (i == n-1) \text{ (n = total number of pipe stages)}$$

Because the last stage will be able to process its next message just after it finishes the computation of the latest one.

$$m_o \quad \text{if } (i < n-1) \text{ and (P is not synchronous)}$$

If the communication protocol in use does not force synchronous sends, the stage will just have to wait to deliver the message to the library interface before being ready to accept a new one.

$m_o + \lambda v_i$ if (i < n-1) and (P is synchronous)

Because when using synchronous sends, the stage will have to wait for the whole communication to take place before going to the next receive operation.

On the other hand, the dependent production time of the stage depends on its independent prodution time (Tr_i), if the application computational load is balanced or the current stage is among the slowest stages of the application, or it depends on the dependent production times of the following and previous stages. As these dependent production times could also depend on those of its neitghbouring stages, it can be seen that there is a propagation of the times of the slowest stages through the pipe until all stages are synchronized with them.

Consequently, it can be said that:

rTri = Tri if the application computational load is balanced or this stage is among the slowest ones.

Or,

rTri = rTr_i–1 if the preceeding stage is slower than the succeeding one and both are slower than the current stage, or the preceeding stage is slower than the current one and it is the last stage.

Or,

rTri =rTri+1 if the succeeding stage is slower than the preceeding one and both are slower than the current stage, or the succeeding stage is slower than the current one and it is the first stage.

While it is easy to visualize that the current stage has to wait for a preceeding slower stage, having to wait for a slower succeeding stage deserves more detailed comment. If the communication is synchronous the current stage will have to wait in the send call until the succeeding one issues a matching receive, but if the communication is not synchronous, it is supposed that the message will be stored in a library/system buffer and the sender will continue its execution. However, if the processing time difference between both stages is big enough, these buffers will eventually get full and the faster stage will be forced to wait until some of them are freed, in practice, this is like changing to a synchronous protocol.

2.2. Replicated stage modeling

A replicated pipe stage is one where data messages are received by a special process called Communication Manager (CM), which is responsible for deciding which stage replica will process the data. In addition, the CM will only issue a receive operation for the previous stage if there is at least a free replica. Then the

chosen replica makes the stage portion of calculation of this data and sends the results to the next stage, unless it is the last one.

To calculate the independent production time of such a stage, we must decide first if the CM is executed in an independent processor or it shares a processor with one of the replicas. As we mentioned previously, the first option could lead to a poorer use of resources, but is easier to model and implement, while the second option could lead to a more efficient use of resources, but it is more difficult to model and could be difficult to implement for some communication libraries or execution environments. As a consequence, we will discuss in detail the model for the first option, which will be experimentaly validated later, and we also will include the definition of the model for the second one.

Thus, to calculate the independent production time of a replicated stage, considering that the CM is executed in an independent processor, we must take into account the managing time associated with the CM (tg_i) and the waiting time for one free replica (wc_i). The term tg_i depends on the communication protocol and possibly on the message size. Basically, the CM looks at the communication channel and waits for messages that could come from the previous stage or from one of the stage replicas (acknowledgments indicating that the replica is free). As there could be many message sources it should look at the channel without blocking. Consequently, the managing time will be the time needed to make 1 or 2 probes of the channel with its corresponding receives plus the time needed to send the requirements to the free replica.

Therefore, if the communication protocol is synchronous, the CM should wait $2*(m_o+\lambda v_i)$ to be ready to process the next requirement message. It has to spend twice the communication time because it has to synchronously receive the message from the previous stage ($m_o + \lambda v_i$) and then synchronously send it to a free replica ($m_o + \lambda v_i$). On the other hand, if the communication protocol is asynchronous then the CM will only have to wait for some network overhead before seeing if there is a new requirement message, because, in this case, library buffers allow for overlapping communications.

The term wc_i depends on the processing capacity of the replicas and the managing capacity of the CM. Given m replicas, if the CM spends more time managing m input messages than the time spent by the set of replicas processing the same number of messages then there will always be free replicas ($wc_i = 0$), which could be an undesirable situation because it means that there is at least some

idle time and, in consequence, the application is wasting resources. Actually, this is, in the long run, the same problem of exceeding the Master's Chunk Managing Capacity (MCMC) discussed in section 3.1 of Chapter III, with the CM acting as the Master and the replicas as the Workers.

Moreover, if the CM has the capacity to feed the m replicas, then the term wc_i will be less than or equal to the production time of the set of replicas, plus the time needed to send the message to a replica, unless the protocol is synchronous, because, in such a case, the communication time is included in the tg_i. Furthermore, the production time of a given set of replicas depends on the independent production time of each replica Tr^k_i, which in turn is calculated in the same way as the independent production time of a single stage plus the time needed to send the acknowledgement message to the CM.

In particular, in order to calculate the production time of the set of replicas we know that if there is a set of m processes and each one is able to process a requirement in a certain time period, which we have called Tr^k_i (0=<i<m), then we can say that in time units $\prod_{j=0}^{m-1} Tr_i^j$ the set of processes shall have produced an integer number of requirements and none will be currently processing a new one. In this period of time, the replica k (0≤i<m) has processed $\prod_{j=0}^{m-1} Tr_i^j \Big/ Tr_i^k$ requirements; in consequence, the set of m $\sum_{l=0}^{m-1}(\prod_{j=0}^{m-1} Tr_i^j \Big/ Tr_i^l)$ replicas have processed requirements. If we divide this expression by the elapsed time period then we have the number of $\sum_{l=0}^{m-1}(\prod_{j=0}^{m-1} Tr_i^j \Big/ Tr_i^l) \Big/ \prod_{j=0}^{m-1} Tr_i^j$ requirements processed in one time unit, i.e. , which could be simplified to $\sum_{l=0}^{m-1}(1/Tr_i^l)$. Finally, the inverse of this expression ($1/\sum_{l=0}^{m-1}(1/Tr_i^l)$) will tell us the time needed by the set of processes to process a single requirement.

Summarizing, the definition of independent production time of replicated stages is:

$$Tri = tg_i + wci \qquad \textbf{(2)}$$

Where, $tg_i = \begin{cases} m_o + c & \text{if (protocol is asynchronous)} \\ 2*(m_o + \lambda v_i) + c & \text{if not} \end{cases}$

and $wc_i = \begin{cases} 0 & \text{if } 1/\sum_{k=0}^{m-1}(1/Tr_i^k) <= tg_i \\ 1/\sum_{k=0}^{m-1}(1/Tr_i^k) + \lambda v_i & \text{if not and protocol is asynchronous} \\ 1/\sum_{k=0}^{m-1}(1/Tr_i^k) & \text{if not and protocol is synchronous} \end{cases}$

The constant c that appears in these expressions represents the overhead introduced by the probes made by the CM on the communication channel in order to monitor all the different potential message sources, and it is included for completion reasons, but it will be discarded later because it is difficult to evaluate and its values are too small to affect the results.

Before introducing the analysis of the CM sharing processor with a replica, we want to include more detailed comments about how to detect whether the replica mangement capability of the CM has been exceeded, a problem that, as we mentioned above, is equivalent to the prediction of the exceeding point of the Master's Chunk Management Capacity (MCMC) for the Master/Worker framework.

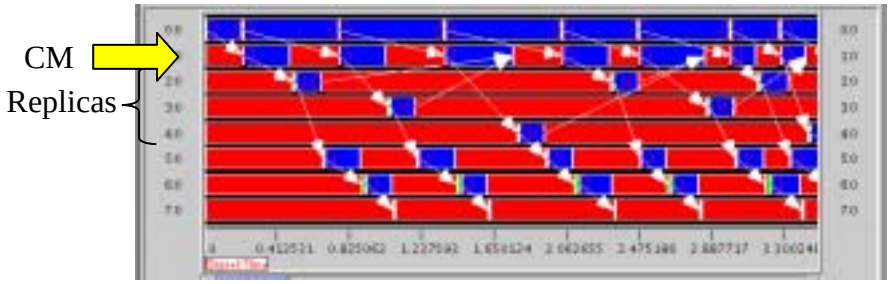


Figure 2. Example of a replicated stage where the CM has exceeded its replica managing capability. In this case, the 2nd stage has three replicas (processes 2, 3, and 4) and, as it can be seen, the notification messages are received by its CM (process 1) only when it needs a free replica.

Suppose, that there are m replicas of a stage and that when the CM has managed k input messages (k < m) there is always some replica which has finished its processing and has already sent back the corresponding acknowledging message.

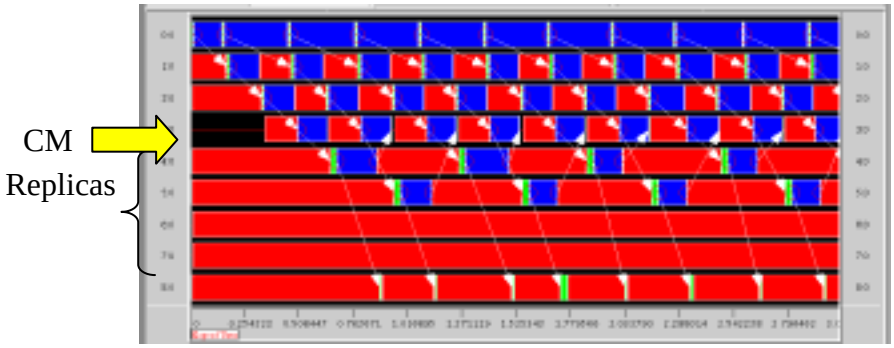


Figure 3. The 4th stage has four replicas (processes 4, 5, 6, and 7) and, as can be seen, process 4 and process 5 are always able to deliver their notification messages before the CM (process 3) is able to send requirements to processes 6 and 7.

So, if the previous stage is producing requirements quickly enough, and the CM gives a higher priority to input messages than to notification ones wherever there are free replicas (a fair situation), then the freed replica will be idle till the moment the CM needs a free replica. This situation is unlikely to happen if the protocol is

asynchronous because the tg_i is quite short, in this case, and it is probable that the CM will check for notification messages frequently, not only when it needs a free replica but when there are no input messages. On the other hand, when the communication protocol is synchronous, this situation can arise if Tr_{i-1} is smaller than tg_i, which happens if the previous stage is the first one and tc_0 is smaller than the communication time or in the unlikely situation when tc_{i-1} is smaller than the constant c (channel probes overhead). This case is illustrated in figure 2, where it can be seen that all replicas are used, but also that they are idle most of the time.

Otherwise, if the CM frequently checks for notification messages because there are periods when there are no input messages from the previous stage, then there will be some replicas that will not be used at all as we can see in figure 3. This is the most common situation (even for synchronous communications).

Going back to the model, if the CM shares the processor with one of the replicas, we will decrease the cost in resources because we are not using extra processors for the CMs, but it is harder to analyze, its implementation depends on the communication library or execution environment capabilities, and can be less efficient than the CM in an independent processor option. Moreover, analyzing this option must take into consideration the possibility of implementing the CM as an independent process or as a thread. For the first possibility, we will get an easier implementation, because all replicas will be managed the same way, but the efficiency is worsened in any case. The worse case is when the communication library implementation is itself not efficient. The reason is that, as we will keep using communication primitives to communicate the CM with the local replica, the efficiency will depend on whether the local communications are done on shared memory, or not. If threads are used then a different management has to be done for the local replica because local communications should be explicitly made through shared memory.

Therefore, to calculate the independent production time of a replicated stage with the CM sharing processor with a replica, we must now take into consideration the waiting time for the CM (wg_i), which is the time we could be forced to wait for its activation, the managing time associated with the CM (tg_i), and the waiting time for one free replica (wc_i). Then, the independent production time of a replicated stage with the CM sharing processor with a replica can be defined as:

$$Tr_i = wg_i + tg_i + wc_i \quad \textbf{(3)}$$

The term wg_i will depend on the CM implementation. When it tests the shared variables, and hears the communication channels once, and there are no free

replicas and/or no new input messages (requirements) then it could leave the processor to the sharing replica (which in turn could be doing nothing if it is free), or it could be busy waiting until something happens. In any case, if there are not more processes in the same processor, the CM activation will have to wait between 0 and less than 2 times the quantum assigned by the system to each thread/process, plus the context switching time. On the other hand, the term tg_i is like the one defined for expression (2) if the received message is sent to a replica running in a different processor. However, if the replica sharing processor with the CM is free then a copy in memory is done instead of a full communication and the management time is reduced, especially if the communication protocol is synchronous. Finally, the term wc_i is like the one defined for expression (2) because the way the waiting time is calculated does not change. However, there is a especial case when calculating the replicas independent production time (Tr_i^k), which is the case of the replica sharing the processor with the CM. The production time of this replica has to include a processor sharing overhead that will be in the (0, tc_i) interval assuming that there are no other processes in the same processor, 0 if the replica associated processing time is less than the system quantum, and tc_i or (tci – quantum) if tc_i is greater than the quantum.

Finally, knowing that the dependent production time of a stage just defines the effect of its neighbors on the stage, we can say that the dependent production time of a replicated stage is defined in exactly the same way as for a single stage.

2.3. Calculating the best replication pattern

If there were always enough available processors, a straightforward strategy to optimize the application throughput would be to replicate each stage until it matches the throughput of the fastest stage. However, as it is not the regular case, a strategy has to be defined to find the best replication pattern for a limited number of available processors.

In the first place, we must be able to calculate the number of processors needed for equating the Tr_i of an stage to the Tr_k of another given that $Tr_k < Tr_i$. As stage i must be replicated in order to achieve that matching, we can use expression (3) and say that the following expression must hold: $Tr_k = tg_i + wc_i$, which substituting tg_i and wc_i by the corresponding expressions leads to: $Tr_k = m_o + \lambda v_i + (1 \Big/ \sum_{l=0}^{m-1} 1/Tr_i^l)$ if the

communication is asynchronous and to $Tr_k = 2(m_o + \lambda v_i) + (1 \Big/ \sum_{l=0}^{m-1} 1/Tr_i^l)$ if not. Furthermore, assuming a homogenous hardware (processors and network), the term $\sum_{l=0}^{m-1} 1/Tr_i^l$ can be rewritten as m / Tr_i^l where m is the number of replicas that must be included in order to achieve Tr_k=Tr_i. Consequently, solving the previous expressions for m will let us to know the number of processors that will be needed for the replicas of stage i. Then this number, plus the processor for the CM, if it is executed in an independent processor, is the number of processors that should be dedicated to stage i in order to match the throughput of stage k. Summing up, the expressions for knowing the number of processors needed for matching the throughput of stages Tr_i and Tr_k (Tr_k < Tr_i) are:

$$p_i = \frac{Tr_i^r}{Tr_k - (m_o + \lambda v_i)} + 1 \quad \text{if the communication protocol is asynchronous} \quad \textbf{(4)}$$

$$p_i = \frac{Tr_i^r}{Tr_k - 2(m_o + \lambda v_i)} + 1 \quad \text{if not} \quad \textbf{(5)}$$

In the second place, we use these expressions to define an algorithm to calculate the best replication pattern for a n-stage pipeline. The goal is to find which is the best stage throughput that can be matched with the available number of processors. The steps to achieve it are:

1. Search stage k where Tr_k = min(Tr_i) ((0 ≤ i < n) and if stage k has not been previously probed). Mark stage k as probed.
2. Search stage g where Tr_g = max(Tri) ((0 ≤ i < n) and if stage g has not been previously considered). Mark stage g as considered.
3. Depending on the communication protocol, use expressions (4) or (5) to calculate the number of processors (p_i) needed to equate Tr_g with Tr_k.
4. If the current ∑p_i > m then
    a. Unmark all considered stages.
    b. Go to 1.
5. If there are no considered stages go to 2

Finally, it is worth noting that this algoritm can be used, slightly modified, even if there are stages that are already replicated because expressions (4) and (5) are based on the independent production time (Tr_i^r) of the replicas; in consequence, if a stage is already replicated, these expressions can be used to determine the number of extra replicas that should be added to improve the stage throughput. In contrast,

this strategy should be supplemented with an idle replica detection method, based on monitoring those replicas that are idle most of the time, in order to determine if a replicated stage has too many replicas.

3. Experimental Validation of the Model

As for the Master/Worker framework, we have developed a set of configurable programs to test our model. We have executed our experiments on one of the clusters of the Computer Science Department of the Wisconsin University at Madison. It is a 150 dual 933MHz nodes connected to a 100Mbit switch, which has a gig-uplink to the core of the network (6 clusters). These programs have been developed in C plus MPI, and the ones that are used specifically to test the model for Pipeline applications accept the following parameters: message size for all messages (this decision simplifies the program and does not affect results), time needed for each stage to process its input data and number of replicas of each stage, and if a stage is replicated then the processing time associated with each replica must be specified.

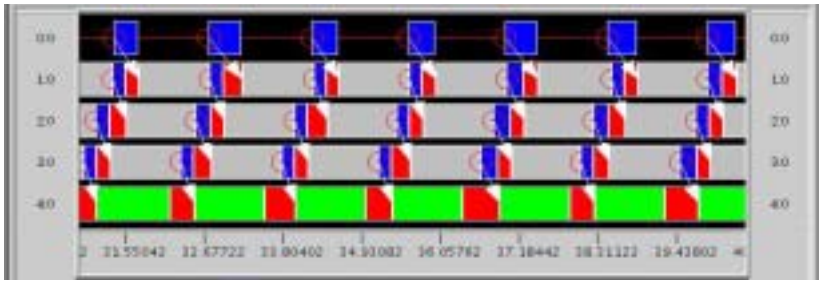
Single Stage		Replicated Stage	
Message size	Processing time	Message size	Processing time
2 Mbytes	1 sec/stage	512 bytes	stages 0,2, &4: 1 sec stage 1: 1.5 sec stage 3: 3 sec
50 Kbytes	100 ms/stage	1.5 Mbytes	stages 0,2, &4: 1 sec stage 1: 1.5 sec stage 3: 3 sec
50 Kbytes	10 ms/stage	512 bytes	stages 0,2, &4: 10 ms stage 1: 15 ms stage 3: 30 ms
200 Kbytes	stages 0,2, &4: 1 sec stage 1: 1.2 sec stage 3: 2 sec		
1 Kbyte	stages 0,2, &4: 100 ms stage 1: 120 ms stage 3: 200 ms		

Table 1. Summary of the configurations executed in order to test the pipeline model.

The set of experiments that have been executed is summarized in table 1. We have included a set of experiments for validating the single stage model and another for validating the replicated stage with CM running in an independent processor model. It can be seen that it is a comprehensive set that covers all the cases described in section 2 (except for replicated stages), with the CM sharing a processor with one replica because the MPI library that has been used to implement

the test applications supports neither threads nor customized process mapping. Once the results of these experiments have been presented and discussed we will introduce a final example for discussing the application of the strategy presented in section 2.3.

In figure 4, a portion of the Gantt trace of a pipeline application execution with a significant processing time associated with each stage and large messages is shown. It is easy to see that the communication protocol is synchronous forced by the message size because the sends are blocked until the matching receive is issued. It can also be seen in table 2 that, by applying expression (1), the defined model is matching the application's behavior.

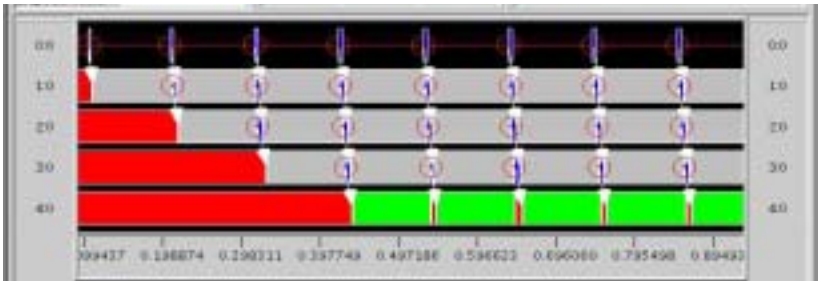


Send Receive Inner stage, last stage, and first stage processing

Figure 4. Five-stage pipeline with an associated processing time of 1 sec/stage and a communication volume of 2 Mbytes/msg.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.197 sec	1.197 sec	1.197 sec	1.197 sec	1 sec
Expected rTri	"	"	"	"	1.197 sec
Observed T	1.22 sec	1.22 sec	1.22 sec	1.22 sec	1.22 sec

Table 2. Expected and observed times for the example of figure 4.



Send Receive Inner stage, last stage, and first stage processing

Figure 5. Five-stage pipeline with an associated processing time of 100 ms/stage and a communication volume of 50 Kbytes/msg.

In figure 5, a portion of the Gantt trace of the execution of a pipeline application with medium size messages and small processing time associated with each stage is shown. This time, the communication protocol is asynchronous (non-

blocking/buffered sends) and it can still be shown that, applying expression (1), the defined model is matching the application behavior (table 3).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	106 ms	106 ms	106 ms	106 ms	100 ms
Expected rTri	"	"	"	"	106 ms
Observed T	107 ms	107 ms	107 ms	107 ms	107 ms

Table 3. Expected and observed times for the example in figure 5.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	17.3 ms	17.3 ms	17.3 ms	17.3 ms	10 ms
Expected rTri	"	"	"	"	17.3 ms
Observed T	17.5 ms	17.5 ms	17.5 ms	17.5 ms	17.5 ms

Table 4. Expected and observed times for the example in figure 6.

Figure 6 shows the portion of a Gantt trace of the execution of a pipeline application with a very small associated processing time. When using low processing times the application is more sensitive to the influences of the environment, such as other processes executing in the same cluster, or even in the same machine, network traffic, and so on. For cases like this we have statistically filtered the data in order to eliminate anomalous measurements. The obtained results are shown in table 4.

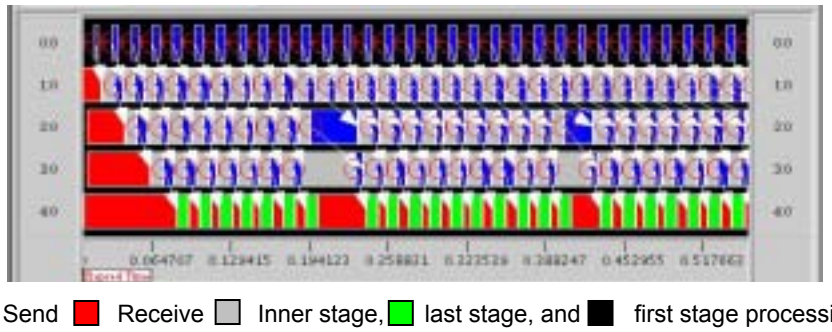
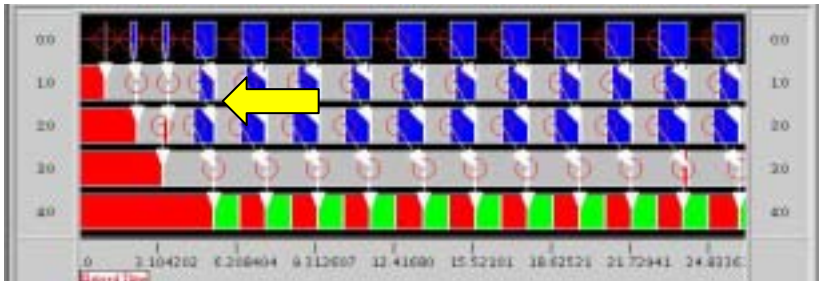


Figure 6. Five-stage pipeline with an associated processing time of 10 ms/stage and a communication volume of 50 Kbytes/msg.

Figure 7 shows the portion of a Gantt trace of the execution of a pipeline application with a different, but always significant, processing time associated with each stage and a message size of 200 Kbytes, this is significant but not large enough to immediately force a synchronous communication protocol. It can be seen that, at the point marked by the arrow, stages 0, 1 and 2 are affected by a change in the communication protocol produced by the low pace of stage 3 and the message

size. The model says that, when the communication protocol is synchronous and the processing time of the next stage is greater than ours and than that of our previous stages, the stage has to wait and its effective processing time will be the same as the next stage. This statement is confirmed by results shown in table 5.



Send Receive Inner stage, last stage, and first stage processing

Figure 7. Five-stage pipeline with an associated processing time of 1 sec for stages 0, 2, and 4; 1.2 sec for stage 1; and 2 sec for stage 3. Communication volume of 200 Kbytes/msg.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.002 sec	1.202 sec	1.002 sec	2.002 sec	1 sec
Expected rTri	2.002 sec	2.002 sec	2.002 sec	"	2.002 sec
Observed T	2.015 sec	2.03 sec	2.03 sec	2.03 sec	2.03 sec

Table 5. Expected and observed times for the example in figure 7.

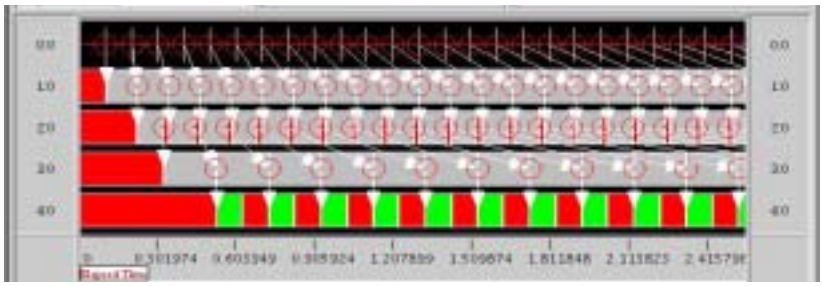
	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	101.54 ms	121.54 ms	101.54 ms	201.54 ms	100 ms
Expected rTri	"	"	121.54 ms	"	201.54 ms
Observed T	100.05 ms	120.2 ms	120.2 ms	200.09 ms	200.09 ms

Table 6. Expected and observed times for the example in figure 8.

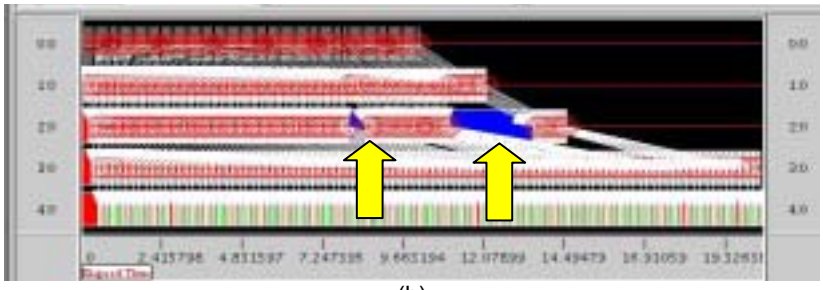
Figure 8 shows the portion of a Gantt trace of the execution of a pipeline application with a different, but always small, processing time associated with each stage and a message size of 1Kbyte. It can be seen in figure 8 (b) that computation time differences between stages 0 and 1 have no effect on the communication protocol, while the differences between stages 2 and 3 have a limited effect at the signaled points. In both cases it is due to the tiny message size.

Actually, the communication library (MPI) uses a more efficient communication protocol (see the explanation about *ready* sends in section 3.4 of Chapter III) when sending small messages to processes that have already issued the corresponding receive. In the example, this is happening between stages 1 and 2, and 3 and 4; however, it cannot take place between stages 0 and 1, and 2 and 3 because the

matching receives have not been issued yet. Consequently, small messages make it difficult to get communications blocked not only because buffers are difficult to get full but also because the slow stages send their messages faster. In addition, using expression (1) we get the results shown in table 6.



(a)



(b)

Send Receive Inner stage, last stage, and first stage processing

Figure 8. Five-stage pipeline with an associated processing time of 100 ms for stages 0, 2, and 4; 120 ms for stage 1; and 200 ms for stage 3. Communication volume of 1 Kbyte/msg. Detail (a) and broad view (b).

Figure 9 shows the portion of a Gantt trace of the execution of a pipeline application with a different, but always significant, processing time associated with each stage, a message size of 512 bytes, and for different replication patterns: no replication (a), replication of stage 1 (b) and (c), replication of stage 3 (d), and replication of stages 1 and 3 (e). It can be seen again in figure 9 (a) that, despite the small messages, slow stages affect the performance of the succeeding ones, but also the one of its previous stages (at the signaled points). For this case (figure 9 (a)), the expected and observed Tr_i and rTr_i of each stage are shown in table 7.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.0021 sec	1.5021 sec	1.0021 sec	3.0021 sec	1 sec
Expected rTri	“	“	1.5021 sec	“	3.0021 sec
Observed T	1.0025 sec	1.5001 sec	1.5001 sec	3.0023 sec	3.0023 sec

Table 7. Expected and observed times for the example of figure 9 (a).

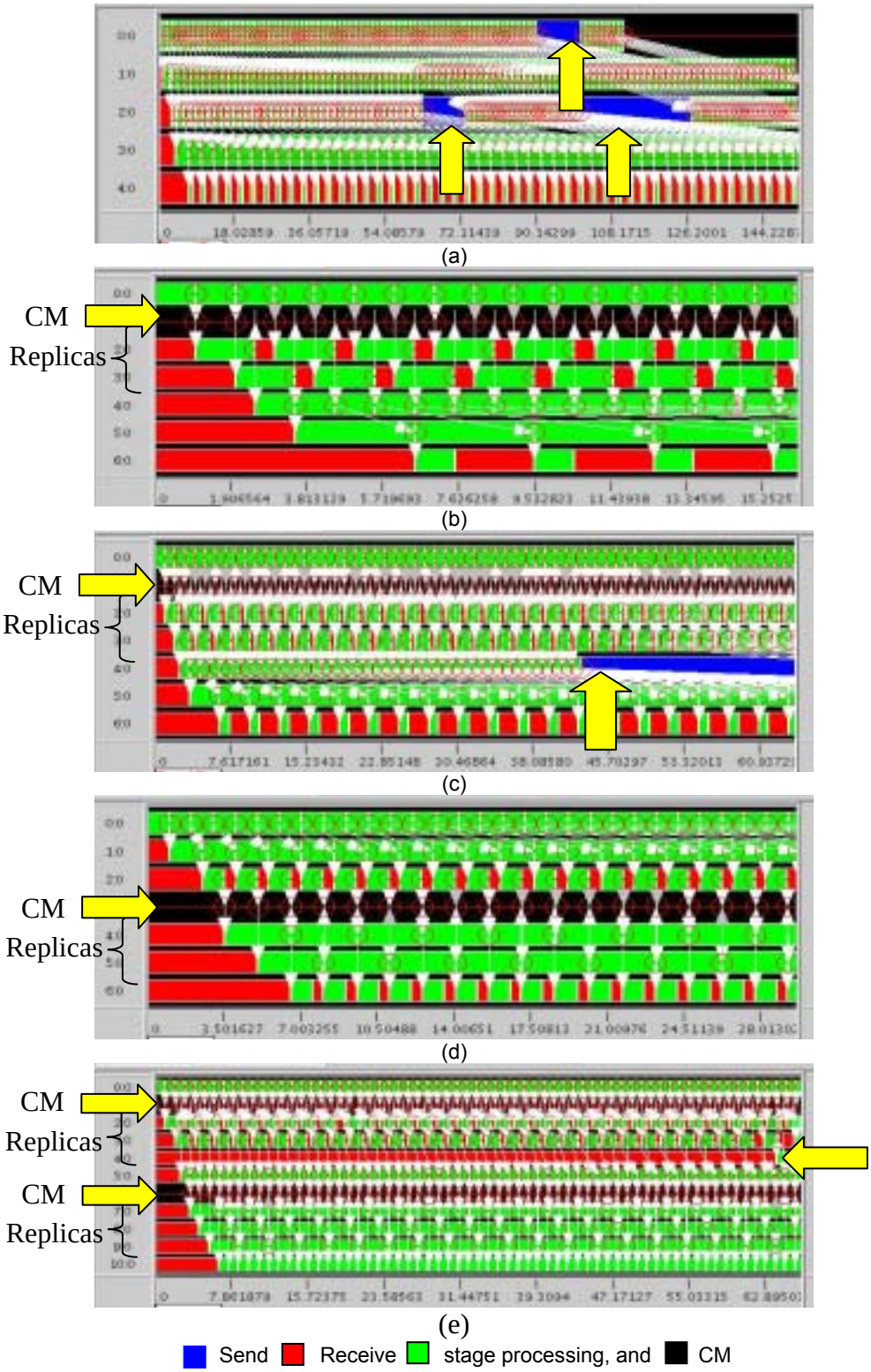


Figure 9. Five-stage pipeline with an associated processing time of 1 sec for stage s 0, 2, and 4; 1.5 sec for stage 1, and 3 sec for stage 3. Communication volume of 512 bytes/msg. Without replicated stages (a), replicating stage 1 (b) & (c), replicating stage 3 (d), and replicating stages 1 and 3 (e).

In figure 9 (b) and (c), it can be seen that stage 0 never becomes blocked because the replicated stage 1 is now processing even faster than stage 0; moreover, as

stage 1 has increased its throughput, stage 2 (process 4) has also increased its own and it is becoming blocked earlier (signaled point in figure 9 (b)) than in the non-replicated case because stage 3 is as slow as it was. For this case (figure 9 (b) and (c)), the expected and observed Tr_i and rTr_i of each stage, and the Tr_i^k of the stage 1 replicas, are shown in table 8.

In figure 9 (d), it can be seen that replicating stage 3 is improving the overall application throughput, which is obvious since it was the slowest pipe stage; in addition, stage 2 does not get blocked anymore. However, as stage 1 is not replicated, it will happen to stage 0. For this case (figure 9 (d)), the expected and observed Tr_i and rTr_i of each stage, and the Tr_i^k of the stage 3 replicas, are shown in table 9.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.0021sec	0.75496 sec $Tr_1^{0,1} = 1.51$	1.0021 sec	3.0021 sec	1 sec
Expected rTri	"	1.0021sec	"	"	3.0021 sec
Observed T	1.0015 sec	1.0017 sec	1.002 sec	3.001 sec	3.001 sec

Table 8. Expected and observed times for the example of figure 9 (b) and (c).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.0021 sec	1.5021 sec	1.021 sec	1.502625 sec $Tr_3^{0,1} = 3.012$	1 sec
Expected rTri	"	"	1.5021 sec	"	1.502625 sec
Observed T	1.003 sec	1.51 sec	1.509 sec	1.504616 sec	1.504616 sec

Table 9. Expected and observed times for the example of figure 9 (d).

Finally, in figure 9 (e), it can be seen that, as expected, adding an extra replica to stage 1 is completely useless because the communication manager (CM) is not receiving enough inputs to keep three replicas busy. Actually, we have already seen that with two replicas we are not using all of the stage capabilities. However, introducing a third replica of stage 3 improves the application throughput to its best (one output every second). It is clear that the best replication pattern for this application is to have two replicas of stage 1 and three of stage 3. In this case (figure 9 (e)), the expected Tr_1 is 0.5022 sec, but the observed one is 1.0013126 sec, which is the Tr_0 that limits the throughput of stage 1. In addition, the expected and observed Tr_3 are 1.0021879 sec and 1.00073 sec respectively (with an observed $Tr_3^0 = Tr_3^2 = 3.0032$ sec and $Tr_3^1 = 3.00013$ sec).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.0021 sec	0.5022 sec $Tr_1^{0,1,2}$ = 1.515 sec	1.0021 sec	1.002188 sec $Tr_3^{0,1}$ = 3.022 sec	1 sec
Expected rTri	"	1.0021 sec	"	"	1.002188 sec
Observed T	1.001313 sec	1.00132 sec	1.002 sec	1.012 sec	1.0123 sec

Table 10. Expected and observed times for the example of figure 9 (e).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.1486 sec	1.6486 sec	1.1486 sec	3.1486 sec	1 sec
Expected rTri	3.1486 sec	3.1486 sec	3.1486 sec	"	3.1486 sec
Observed T	3.129 sec	3.132 sec	3.133 sec	3.137 sec	3.136 sec

Table 11. Expected and observed times for the example of figure 10 (a).

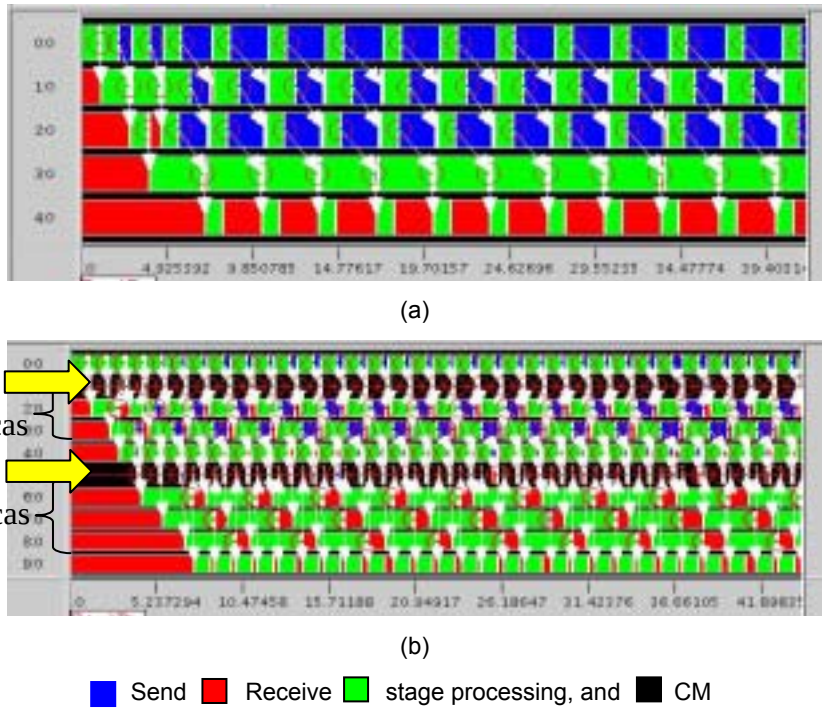


Figure 10. Five-stage pipeline with an associated processing time of 1 sec for stages 0, 2, and 4; 1.5 sec for stage 1; and 3 sec for stage 3. Communication volume of 1.5 Mbytes/msg. Without replicated stages (a) and replicating stages 1 and 3 (b).

Figure 10 shows the portion of a Gantt trace of the execution of a pipeline application with a different, but always significant, processing time associated with each stage, a message size of 1.5 Mbytes, and for different replication patterns: no replication (a), replication of stage 1 and 3 (b). In figure 10 (a), supplemented by table 11, it can be seen that, as expected, all pipe stages are quickly synchronized with the slowest one (stage 3) due to the message size that forces a synchronous

communication protocol. In addition, it can be seen in figure 10 (b) that having two replicas of stage 2 and 3 of stage 3 is the best replication pattern for this application, in spite of the fact that stage 1 replicas are not being made the most of, as can be seen in table 12.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	1.1486 sec	0.92 sec $Tr_1^{0,1}$ = 1.515 sec	1.1486 sec	1.151 sec $Tr_3^{0,1,2}$ = 3.002 sec	1 sec
Expected rTri	1.151 sec	1.151 sec	1.151 sec	"	1.151 sec
Observed T	1.15 sec	1.1503 sec	1.1512 sec	1.1513 sec	1.1513 sec

Table 12. Expected and observed times for the example of figure 10 (b).

Figure 11 shows the portion of a Gantt trace of the execution of a pipeline application with a different, but always very low, processing time associated with each stage, a message size of 512 bytes, and for different replication patterns: no replication (a), replication of stage 1 (b), and replication of stage 3 (c). It can be seen that, although the application is more sensitive to environmental influences, and in consequence the measurements obtained are less accurate, the model is able to catch the application's behavior. Thus, it can be seen in figure 11 (b) and table 14 that it predicts that 3 replicas for stage 1 are too many, but it can also be seen in figure 11 (c) and table 15 that the same measure for stage 3 is profitable, even though not all the processing capacity of the replicated stage is being used.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	12.131 ms	16.427ms	12.131 ms	31.427 ms	10 ms
Expected rTri	"	"	16.427 ms	"	31.427 ms
Observed T	12.01 ms	15.05 ms	15.34 ms	30.64 ms	30.67 ms

Table 13. Expected and observed times for the example of figure 11 (a).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	11.427 ms	6.8638 ms $Tr_1^{0,1,2}$ = 15.437 ms	12.131 ms	31.427 ms	10 ms
Expected rTri	"	11.427 ms	"	"	31.427 ms
Observed T	11.13 ms	11.34 ms	12.39 ms	30.32 ms	31.24 ms

Table 14. Expected and observed times for the example of figure 11 (b).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	11.427 ms	16.427ms	11.427 ms	12.187 ms $Tr_3^{0,1,2}$ = 30.6 ms	10 ms
Expected rTri	“	“	16.427 ms	16.427	16.427
Observed T	10.93 ms	15.8 ms	15.83 ms	15.9 ms	15.87 ms

Table 15. Expected and observed times for the example of figure 11 (c).

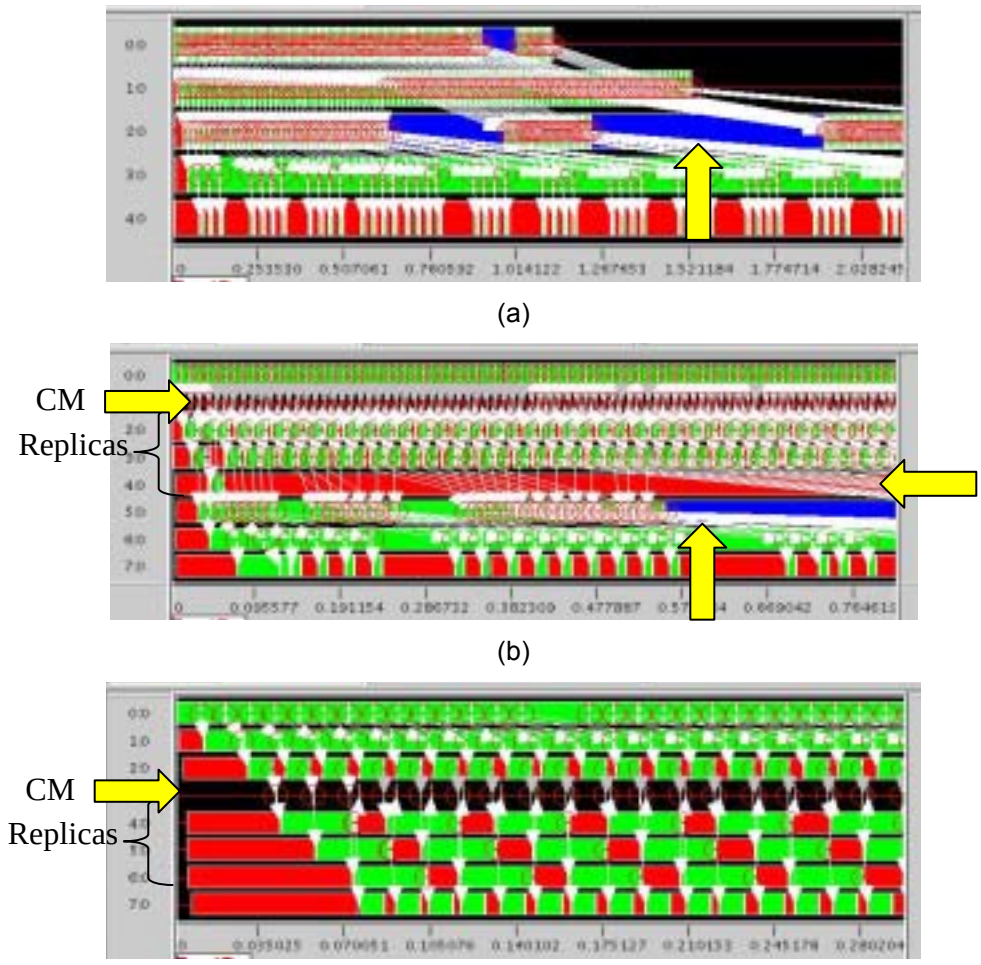


Figure 11. Five-stage pipeline with an associated processing time of 10 ms for stages 0, 2, and 4; 15 ms for stage 1; and 30 ms for stage 3. Communication volume of 512 bytes/msg. Without replicated stages (a), replicating stages 1 (b) and replicating stage 3 (b).

Finally, once we have presented and discussed the results of this comprehensive set of experiments that were mainly intended to validate the stage models introduced in sections 2.1 and 2.2, we want to include a final example with the objective of illustrating the application of the performance improving strategy defined in section 2.3. The application under consideration is again a five-stage pipe with 10Kb messages and the following processing times associated with each stage: 100

ms for stages 0 and 4, 400 ms for stage 1, 300 ms for stage 2, and 200 ms for stage 3. It can be seen that it has been designed to have an intuitive best replication pattern of 4 replicas for stage 1, 3 for stage 2, and 2 for stage3.

Nevertheless, we want to describe in more detail the application of the strategy assuming that there are more than 9 available processors. In the first place, stage 1 or stage 4 is chosen as the min(Tr_i) and marked as probed (step 1). Next, stage 1 is chosen as the one with max(Tr_i) and marked as considered (step 2). Then, expression (4) is applied to calculate that 4 processors are needed to match Tr_0 (step 3). Finally, as the number of available processors has not been exceeded (step 4) and there are already more stages to be considered (step 5), we return to step 2. It is clear that in the second iteration of this algorithm it will be determined that stage 2 has to be replicated three times, and in the third and last iteration of the algorithm it will be determined that stage 3 must be replicated twice.

On the contrary, supposing that the number of available processors is between 6 and 8, then in the second or third iteration the number of processors needed will be exceeded, and then the algorithm will have to be executed from step 1, then stage 3, which is the stage with the next min(Tr_i), will be selected. Next, it will be determined that stage 1 will have to be replicated twice, as well as stage 2, which is the best replication pattern that can be obtained for that number of processors.

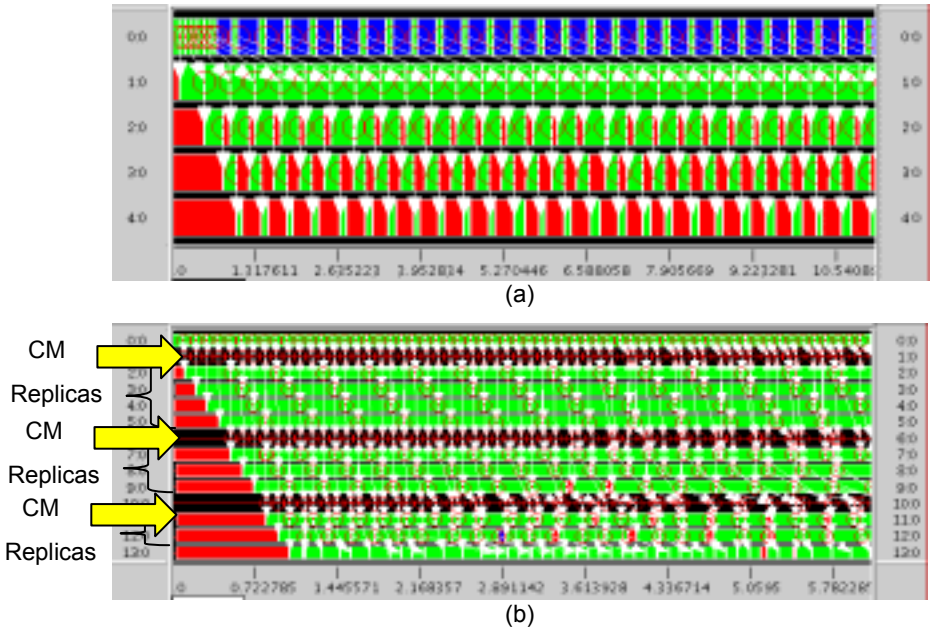


Figure 12. Five-stage pipeline with an associated processing time of 100 ms for stages 0 and 4; 400 ms for stage 1; 300 ms for stage 2; and 200 for stage 3. Communication volume of 10 Kbytes/msg. Without replicated stages (a), replicating stages 1, 2, and 3 (b).

Finally, this discussion is illustrated in figure 12, where the portion of a Gantt trace of the execution of this application without replicas (a) is shown. There, the problems described before (all stages synchronized with the slowest one) can be easily identified. There also can be seen the results of applying the strategy with enough processors to develop the best pattern (b). Moreover, in tables 16 for (a) and 17 for (b) the results that supplement these traces are included.

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	102.131 ms	402.131 ms	302.131 ms	202.131 ms	100 ms
Expected rTri	402.131 ms	"	402.131 ms	402.131 ms	402.131 ms
Observed T	407.34 ms	409.045 ms	409.7 ms	410.1 ms	409.9 ms

Table 16. Expected and observed times for the example of figure 12 (a).

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Expected Tri	102.131 ms	101.067 ms	101.421 ms	102.131 ms	100 ms
Expected rTri	"	"	"	"	102.131 ms
Observed T	101.45 ms	101.502 ms	101.283 ms	110.2 ms	110.43 ms

Table 17. Expected and observed times for the example of figure 12 (b).

4. Global Pipeline Performance Model

Our objective, in the last section of this chapter, is to sum up the performance model for Pipeline applications according to the general model for dynamic performance tuning presented in Chapter II. We summarize, in the first place, 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. We then indicate which are the application parameters that must be monitored at run time in order to be able to detect the performance bottlenecks. Finally, we will indicate the parameters that can be changed at run time to improve the applications' performance and when can these changes take place.

In the first place, we want to recall once more that the proposed model is not completely defined because a comprehensive model has to include, in addition to the replication strategy presented, mechanisms to decide when and why fast stages should be grouped or under what conditions they could be separated. Moreover, our main objective has been to demonstrate that our hypothesis, about the utility of relating a performance model to the most common distributed application structures

in order to be used in a dynamic performance tuning environment, can be realistically implemented for frameworks other than the Master/Worker.

Nevertheless, we have designed a strategy to optimize the throughput of a pipeline application by looking for the best slow stages replication pattern for a given number of processors. In order to fulfill this objective, we have defined a set of expressions for estimating the capabilities of each stage (expressions (1), (2), and (3)), as well as a rationale to identify the bottleneck stage(s) (the slowest one(s)) and to decide how many replicas of each stage should be included.

In the second place, in order to be able to apply the strategies and calculate the performance expressions several application parameters must be monitored. Said 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 allowing the adaptation of the system to the network load conditions.
- *Message sizes* (v_i) have to be captured for each stage when it sends (or receives) data to (or from) next (or previous) stage.
- *Stages processing times* (tc_i) *plus the time spent sending the message to the next stage* have to be measured in order to calculate the independent production time Tr_i of each stage.

Finally, when applying the stage replication strategy, some changes might be introduced in 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 the following:

- *Stage type*: a non-replicated stage is a single process responsible for performing the computation associated with the stage. If the tuning application decides that the stage should be replicated then a set of new processes (the replicas) have to be created and the original process becomes the communication manager (CM). As a result, there are three different stage types: single, replica, and CM, as well as two transformations: from single to CM and vice versa. Finally, a stage may become a CM only before receiving a data message from its previous stage and if the replicas have already been set up and are ready to receive data.

- *Number of replicas*: The number of replicas of a replicated stage can be changed if the application conditions change (for example, some stages start to do more work, or simply more processors become available). This parameter can be changed before the CM goes into checking the communication channel for new incoming data messages or replica acknowledgements, and only if the new replicas have already been set up and are ready to receive data.

Chapter V:
Conclusions and Future Work

Abstract

This chapter contains the main conclusions obtained from the work included in this thesis, as well as the main work lines that are currently being undertaken and the future work plan aimed at continuing the research on framework-related performance models for dynamic automatic application performance tuning.

1. Conclusions

In this work, we have proposed and developed structure-related performance models for dynamic performance tuning of parallel/distributed applications, based on the study of the characteristics of the parallel programming support tools. In this last chapter, we shall review the main objectives stated in this work and see how they have been attained.

The main motivation of our work is that developing parallel/distributed applications is, for many reasons, harder than developing sequential ones but, on the other hand, these kinds of applications are the most promising way of coping, within reasonable time limits, with many complex problems.

This means that developing parallel/distributed applications is worth the effort because of the huge potential performance gains over sequential applications. Moreover, supportive tools can be developed in order to facilitate the design and development of these kinds of applications. In that regard, one of the most successful classes of supportive tools is the one that is based on exploiting the degree of commonality that many solving strategies share by providing a set of pre-defined structures called patterns (at design level), and frameworks or skeletons (at the implementation level). However, the price for enjoying the advantages of development tools is usually the loss of some degree of flexibility for performance tuning, as well as the introduction of some execution time overhead attributable to the tool.

In the end however, the result achieved by the first versions of a parallel/distributed application in terms of performance is usually disappointing, and forces programmers to engage in a difficult process of performance analysis and tuning of the application. Again, supportive tools are likely to be helpful in this process. The most sophisticated performance-related supportive tools are those including some degree of automatic performance tuning, and especially the ones that are able to do dynamic tuning. However, dynamic automatic tuning tools must fulfill two requirements: keep instrumentation low and make quick and accurate decisions. Therefore, in order to fulfill these conditions, the tuning tool must have a clear improvement target (library level, program level, application level), and it must include as much previous knowledge about the application as possible.

The core idea behind our work, extensively described in Chapter I, was to take advantage of the intrinsic knowledge that the use of framework or skeletons

provides about the application's structure and functionality for developing performance models intended to be used in a dynamic tuning environment.

In addition, the general structure of these performance models is determined by the steering loop architecture of the dynamic performance environment and consists of a set of parameters that should be monitored (measure points), a set of performance analysis and tuning strategies and/or expressions, and a set of tuning parameters and actions. This steering loop architecture, as well as the structure of the frameworks that have been treated in this work, were described in Chapter II. The discussion about this approach can be found in:

[MCe+01] A. Morajko, E. Cesar, T. Margalef, J. Sorribes, E. Luque, "Dynamic Performance Tuning Environment", LNCS, Vol. 2150 (Euro-Par 2001), pp. 36-45, Springer-Verlag. 2001.

[CMo+02] E. Cesar, A. Morajko, T. Margalef, J. Sorribes, A. Espinosa, E. Luque: Dynamic Performance Tuning Supported by Program Specification. Scientific Programming, Vol. 10, pp. 35-44. IOS Press. 2002.

The first step towards developing this idea consisted of defining a performance model for homogeneous Master/Worker applications. A homogenous Master/Worker application is one in which every worker receives roughly the same number of tasks and spends roughly the same amount of time processing them. Under these conditions, the main tuning action is to adapt the number of workers of the application. Consequently, a set of expressions for modeling the application's execution time were developed depending on communication parameters (network latency and bandwidth, and communication volume) and computation time. From these expressions, which modeled the behavior of the application, we derived a target function for calculating for what number of workers the application would run in the minimum time. This study can be found in:

[CM+03] E. Cesar, J. G. Mesa, J. Sorribes, and E. Luque. "POETRIES: Performance Oriented Environment for Transparent Resource-management, Implementing End-user parallel/distributed applications", LNCS, Vol. 2790, pp. 141-146. Springer-Verlag. 2003

This model was further developed, incorporating expressions for determining the number of workers the Master can deal with for the current communication conditions. In addition, a simple strategy for dynamically minimizing load unbalancing was defined, thereby giving us the ability to apply a two-phase model to improve the performance of any Master/Worker application. Finally, a more complex

and powerful balancing strategy, based on the factoring algorithm for scheduling parallel loops, was defined and incorporated into the model, thereby taking advantage of the dynamic nature of our model to define a self-adjusting strategy. The resulting Master/Worker performance model, as well as its application on real applications, can be found at:

[CM+04] E. Cesar, J. G. Mesa, J. Sorribes, E. Luque. “Modeling Master-Worker Applications in POETRIES”. Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004). pp. 22 - 30. IEEE Computer Society. Santa Fe, New Mexico. April 2004.

[MCe+05] Anna Morajko, Eduardo César, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, and Emilio Luque. “Automatic Tuning of Master/Worker Applications”, LNCS, Vol. 3648, pp. 95-103. Springer-Verlag. 2005.

[MC+05] Anna Morajko, Eduardo César, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, and Emilio Luque. “Development and Tuning Framework of Master/Worker Applications”. Invited paper. Journal of Computer Science & Technology (JCS&T). Vol. 5, num. 3, pp 115-120. 2005.

[MoC+05] Andreu Moreno Vendrell, Eduardo César, Joan Sorribes, Tomàs Margalef, Emilio Luque. “Balanceo de carga en sistemas distribuidos Master-Worker”, XVI Jornadas de Paralelismo 2005, pp. 443-450, Granada, 2005.

Finally, the performance model for Master/Worker applications described in Chapter III has been supplemented by taking into consideration the possibility of variations in the communication volume related to changes in the number of workers, as well as an efficiency index to determine for what number of workers the best performance/resource efficiency relationship is obtained. We also have defined a second index focused on calculating how likely it would be to get a balanced execution, applying our adaptative strategy, for a certain number of workers, which allows us to evaluate the convenience of adding more workers to the application. All this work has led to the definition of a very complete and detailed performance model for Master/Worker applications that can be summarized as:

1. **A load balancing phase**: applying the Dynamic Adjusting Factoring (DAF) strategy, which is based on making a partial distribution of tasks, and that leads to highly balanced executions, even for applications with big differences among task associated computation times (high variance). In

order to determine the portion of tasks to be distributed in any given moment the following expressions are used:

$$x_0 = \left(\mu + \sigma\sqrt{N/2}\right) \Big/ \mu \quad \textbf{(1)} \text{ and } x_j = \left(2\mu + \sigma\sqrt{N/2}\right) \Big/ \mu \quad \textbf{(2)}$$

Where μ is the worker's mean execution time per task, σ is its standard deviation, N is the number of workers (processors), x_0 is the factor to be applied at the beginning of each iteration, and x_i represents the dynamically adjusting factor for the rest of the iteration.

2. **A number of workers adjusting phase**: once we have a balanced execution that has very likely improved the application's performance by efficiently using the available resources, it is possible to evaluate if adding more workers can lead to further execution time reductions. To do so, we have to determine firstly the number of workers that would lead to the best performance/efficiency relationship, using the expression:

$$Pi(x) = \frac{xTt(x)^2}{Tc} \quad \textbf{(17)}$$

Where x is the number of workers being considered, Tc is the overall computation time, and Tt is the expected execution time for x workers calculated by using one of the following expressions:

	Communication Protocol	
	Asynchronous	**Synchronous**
$m_o \geq \lambda v_i^m$	$Tt = (n+1)m_o + \frac{(Tc + \lambda V)}{n} + \mu_m$ **(7)**	$Tt = (n+1)m_o + \frac{[((n-1)\alpha + 1)\lambda V + Tc]}{n} + \mu_m$ **(9)**
$m_o \leq \lambda v_i^m$	$Tt = 2m_o + \frac{[((n-1)\alpha + 1)\lambda V + Tc]}{n} + \mu_m$ **(8)**	

Where, m_o and λ are the network parameters, V is the overall (predicted) communication volume, and n the number of workers. However, some of these expressions can only be applied if the Master has the capacity of dealing with the number of workers under consideration for the predicted communication conditions. We have called this restriction the Master's Chunk Managing Capability (MCMC), which can be estimated with the following set of expressions:

	Communication Protocol	
	Asynchronous	**Synchronous**
$m_o \geq \lambda v_i^m$	$\left\lfloor \frac{\sqrt{m_o^2 + m_o((1-\alpha)\lambda V + Tc)}}{m_o} + 1 \right\rfloor$ **(10)**	$\left\lfloor \frac{(2m_o - \lambda\alpha V) + \sqrt{(\lambda\alpha V - 2m_o)^2 + 4m_o(\lambda V + Tc)}}{2m_o} \right\rfloor$ **(12)**
$m_o \leq \lambda v_i^m$	$\left\lfloor \frac{\lambda V + Tc}{\lambda\alpha V - m_o} \right\rfloor$ **(11)**	

Finally, the tuning system should only change the number of workers if it is likely to get a balanced execution for that number of workers. We have defined an index ($1 \Big/ (1 + \frac{\mu}{\sigma}\sqrt{\frac{2}{x}})$), based on the number of workers and the task-related execution time variance, in order to get a hint about how difficult it would be to get a balanced application for the new number of workers.

The next step was focused on demonstrating that the idea of developing framework-related performance models for dynamic automatic tuning is extensible to frameworks other than Master/Worker. Consequently, we have developed a performance model related to the structure of the Pipeline framework, described in Chapter IV, though it is not as comprehensive as the one for the Master/Worker.

In this case, the general strategy for improving the application's performance consists of determining the best replication pattern for the pipe stages over the available resources. To achieve this objective, we have designed expressions that model the behavior of single and replicated stages, differentiating the observed behavior, which can be influenced by previous or succeeding stages, from the potential one. Based on these expressions, we have defined an algorithm for determining the best replication pattern for the application. This algorithm can be summarized as follows:

1. Search stage k where $Tr_k = \min(Tr_i)$ (($0 \leq i < n$) and if stage k has not been previously probed). Mark stage k as probed.
2. Search stage g where $Tr_g = \max(Tri)$ (($0 \leq i < n$) and if stage g has not been previously considered). Mark stage g as considered.
3. Depending on the communication protocol, use expresions (4) or (5) to calculate the number of processors (p_i) needed to equate Tr_g with Tr_k.
4. If the current $\sum p_i > m$ then
    a. Unmark all considered stages.
    b. Go to 1.
5. If there are no considered stages go to 2

Where Tr_i is the potential execution time of stage i, and expressions (4) and (5) are:

$$p_i = \frac{Tr_i^r}{Tr_k - (m_o + \lambda v_i)} + 1 \quad \text{if the communication protocol is asynchronous} \quad (4)$$

$$p_i = \frac{Tr_i^r}{Tr_k - 2(m_o + \lambda v_i)} + 1 \quad \text{if not} \quad (5)$$

Where m_o and λ are the network parameters, and v_i the communication volume of stage i to stage i+1.

This performance model for Pipeline applications can be found in:

[CM+05] E. Cesar, J. Sorribes, E. Luque, “Modeling Pipeline Applications in POETRIES”. LNCS, Vol. 3648, pp. 83-92. Springer-Verlag. 2005

Finally, both models are supported by a real, though synthetically generated, experimentation specifically designed to demonstrate the validity of each defined strategy and set of expressions.

In conclusion, we have achieved the objective of demonstrating that we can take advantage of the knowledge about the application contained on the frameworks or skeletons, provided as supportive tools for their development, in order to define performance models intended to be used in a dynamic automatic performance tuning environment.

Moreover, we have found that a common generic framework analysis methodology can be used to define the performance model associated with any framework. This methodology consists, in the first place, of finding the performance bottlenecks related to the framework that are suited to be solved dynamically and, in the second place, of defining the parameters (inputs) that should be monitored to detect those problems and the parameters (outputs) that can be changed to overcome them. The methodology also consists of defining the analysis expressions and strategies that, using the monitored input parameters, detect the problems and correct them by changing the appropriate output parameters.

2. Current and Future Work

There are three clearly defined lines of work that are being currently developed in different degrees. The first one is the completion of current performance models and development of new ones, the second, is the implementation of a robust dynamic tuning tool with MATE for the framework-related dynamic automatic performance tuning of parallel applications. The third is the generalization of the idea of

developing models based on the applications structure for dynamic tuning to Grid environments.

Regarding the completion of current performance models and the development of new ones, we are, firstly, focused on closing the last minor issues of the Master/Worker model; secondly, working on new performance models for other frameworks, especially the Pipeline, but also on a performance model for mixed applications.

The Master/Worker model issues pending are, in the first place, the definition of a better indicator of how likely it would be to achieve a balanced execution for a different number of workers because the current one uses an upper bound that is not close enough to the real values, and, in the second place, the study of how much history should be recorded in order to calculate the mean processing time and variance for the Dynamic Adjusting Factoring (DAF) policy.

In reference to completing the Pipeline model and defining new ones for other frameworks, we are working on a model for Pipeline applications that includes the possibility of grouping several stages in the same processor in order to liberate resources for replicating other stages, and also with the objective of not wasting resources dedicated to stages with a too low associated computation time. In addition, we have started the study of the performance characteristics of Divide & Conquer applications.

The main challenge concerning the definition of performance models is to find the way of mixing models for complex applications, with the objective of deciding what tuning actions will be more advantageous. The idea is that a very complex application can be implemented with a composition of frameworks, for example: a Master/Worker in which each worker is parallelized as a Pipeline. Thus, in a case like this, the tuning application will need a mechanism to decide whether to adapt the number of workers before improving the pipeline throughput or, on the contrary, start with the pipelines and then adapt the number of workers.

As for the implementation of a robust and more general framework-related performance tuning tool on MATE, the main problems are the difficulty of implementing complex *tunlets* and, even more, to coordinate them when there are many *tunlets* responsible for different aspects (or phases) of the performance analysis. Currently, other members of our research group are focused on the definition of a high level *tunlet* specification language aimed at solving these problems.

Finally, in relation to the application of the idea of dynamically and automatically tuning parallel applications on the Grid, there are some previous related studies in our research group [Hey01], and currently there are other members of the group working on the problem of dynamically inserting instrumentation on a parallel application running on the Grid. It is clear that this is a major challenge, as is the general idea that it can be applied to the Grid, which is a suitable place for long execution time applications. However, it is also clear that performance models must take into consideration many extra parameters, such as the possibility of machines' losses, the cost of check-pointing, the highly heterogeneous nature of communication channels and processors (including the possibility of processes sharing a processor with others non-related to the application processes), and the middleware overhead.

References

[AG+02] F. Almeida, D. González, L.M. Moreno, C. Rodríguez. “An Analytical Model for Pipeline Algorithms on Heterogeneous Clusters”. Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting, pp. 441 – 449. 2002.

[AG+03] Almeida, F. Gonzalez, D. Moreno, L.M. Rodriguez, C. Toledo, J. “On the prediction of master-slave algorithms over heterogeneous clusters”. Proceedings. Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 433- 437. 2003.

[Amd67] Amdahl, G. M. "Validity of the single-processor approach to achieving large scale computing capabilities". In the proceedings of the AFIPS Conference. AFIPS Press: 483-485, 1967

[AI98 “Wind: The Production Flow Solver of the NPARC Alliance”, AIAA 98-0935, available from: http://www.grc.nasa.gov/www/winddocs/aiaa98/aiaa-98-0935.html.

[BR+03] Rosa M. Badia, G. Rodriguez, Jesús Labarta. "Deriving Analytical Models from a Limited Number of Runs". Minisymposium on Performance Analysis, ParCo 2003.

[BV01] Ioana Banicescu and Vijay Velusamy. “Performance of scheduling scientific applications with adaptative weighted factoring”. Proceedings of the 15th International Parallel & Distributed Processing Symposium, pp. 84. IEEE Computer Society, 2001.

[BV02] Ioana Banicescu and Vijay Velusany. “Load balancing highly irregular computations with the adaptative factoring”. Proceedings of the 16th International Parallel & Distributed Processing Symposium, pp. 195. IEEE Computer Society, 2002.

[BC+04] A. Benoit, M. Cole, S. Gilmore and J. Hillston. “Evaluating the performance of skeleton-based high level parallel programs”. The International Conference on Computational Science (ICCS 2004), Part III, LNCS, pp. 299-306. Springer Verlag, 2004.

[Can98] E. Cantu-Paz, “Designing efficient master-slave parallel genetic algorithms”, in J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. E. Goldberg, H. Iba and R. Riolo, editors, Genetic

Programming: Proceeding of the Third Annual Conference, San Francisco, Morgan Kaufmann, 1998.

[CMo+02] E. Cesar, A. Morajko, T. Margalef, J. Sorribes, A. Espinosa, E. Luque: Dynamic Performance Tuning Supported by Program Specification. Scientific Programming, Vol. 10, pp. 35-44. IOS Press. 2002.

[CM+03] E. Cesar, J. G. Mesa, J. Sorribes, and E. Luque. "POETRIES: Performance Oriented Environment for Transparent Resource-management, Implementing End-user parallel/distributed applications", LNCS, Vol. 2790, pp. 141-146. Springer-Verlag. 2003

[CM+04] E. Cesar, J. G. Mesa, J. Sorribes, E. Luque. "Modeling Master-Worker Applications in POETRIES". Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004). pp. 22 - 30. IEEE Computer Society. Santa Fe, New Mexico. April 2004.

[CM+05] E. Cesar, J. Sorribes, E. Luque, "Modeling Pipeline Applications in POETRIES". LNCS, Vol. 3648, pp. 83-92. Springer-Verlag. 2005

[CK+01] M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. "Nepal, Nested data-parallelism in Haskell". LNCS, Vol. 2150, pp. 524-534. Springer-Verlag. 2001.

[Col04] Murray Cole. "Bringing Skeletons out of the Closet**:** A Pragmatic Manifesto for Skeletal Parallel Programming". Parallel Computing, Vol 30, no. 3, pp. 389-406. 2004.

[DB+99] Deelman, Bagrodia, Dube, Browne, Hoisie, Luo, Lubeck, Wasserman, Oliver, Teller, Sundram-Stukel, Vernon, Adve, Houstis, and Rice. "POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems". http://www.cs.utexas.edu/users/poems/Papers/Wosp/wosp_dave.html

[DG+03] A.J. Dorta, J.A. González, C. Rodríguez, and F. Sande. "Llc: A parallel skeletal language". Parallel Processing Letters, 13 (3), pp. 437-448. 2003.

[ES98] Greg Eisenhauer and Karsten Schwan. "An Object-Based Infrastructure for Program Monitoring and Steering". Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98). pp. 10-20. 1998.

[EM+00] A. Espinosa, T. Margalef, E. Luque, “Integrating Automatic Techniques in a Performance Analysis Session”, LNCS, Vol. 1900 (Euro-Par 2000), pp. 173-177, Springer-Verlag, 2000.

[FP00] T. Fahringer, A. Požgaj, “P^{3}T+: A Performance Estimator for Distributed and Parallel Programs”, Scientific Programming, IOS Press, Vol. 8, no. 2, The Netherlands, 2000.

[FSF92] Susan Flynn Hummel, Edith Schonberg, and Laurence E. Flynn. “Factoring: a method for scheduling parallel loops”. Communications of the ACM, vol. 35, num. 8, pp. 90 – 101, 1992.

[GB+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. “PVM: Parallel Virtual Machine”. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1994.

[GL99] William Gropp and Ewing Lusk. “Reproducible Measurements of MPI Performance Characteristics”. Technical Report ANL/MCS-P755-0699. Mathematics and Computer Science Division, Argonne National Laboratory, 1999.

[Gus88] J.L. Gustafson. “Reevaluating Amdahl's Law". Chapter for book, *Supercomputers and Artificial Intelligence,* Edited by Kai Hwang, 1988.

http://www.scl.ameslab.gov/Gus/Publications/Gus/AmdahlsLaw/Amdahls.pdf

[Hae03] N. V. Haenel. “User Guide for the Java Edition of the PEPA Workbench”. LFCS, University of Edinburgh. 2003.

[HF03] Michael T. Heath and Jennifer E. Finger. “ParaGraph: A Performance Visualization Tool for MPI”. http://www.csar.uiuc.edu/software/paragraph/userguide.pdf 2003

[Her00] C. Herrmann. “The Skeleton-based Parallelization of Divide-and-Conquer Recursions”. Ph.D. thesis. University of Passau. 2000.

[Hey01] Elisa Heymann. “Effective Resource Management for Master/Worker Applications in Opportunistic Environments”. PhD. Thesis. Autonomous University of Barcelona. 2001.

[HS+04] E. Heymann, M.A. Senar, E. Luque, and M. Livny. “Efficient resource management applied to master-worker applications”. Journal of Parallel and Distributed Computing, 64 (2004), pp. 767-773. 2004.

[Hill96] J. Hillston. “A Compositional Approach to Performance Modeling”. Cambridge University Press. 1996.

[HK99] Jeffrey K. Hollingsworth and Peter J. Keleher. “Prediction and adaptation in Active Harmony”. Cluster Computing, vol. 2, pp. 195-205. 1999.

[HB02] Hollingsworth, J. K., Buck, B. “DyninstAPI Programmer’s Guide. Release 3.0”. University of Maryland, 2002.

[HH92] P. Hudak, J. H. Fasel, “A Gentle Introduction to Haskell”, Technical Report, Computer Science Department, Yale University, 1992.

[HN02] F. Huch, U. Norbisrath, “Distributed Programming in Haskell with Ports”, Proceedings of the 12th International Workshop on Implementation of Functional Languages, 2002.

[JHS02] “The Java HotSpot Virtual Machine, v1.4.1”. White Paper. 2002. http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html

[JM+98] Jorba, J., Margalef, T., Luque, E., Andre, J, Viegas, D.X. "Application of Parallel Computing to the Simulation of Forest Fire Propagation", Proc. 3rd International Conference in Forest Fire Propagation, Vol. 1, pp. 891-900. Portugal, November 1998.

[KC+97] P. Kacsuk, J.C. Cunha, G. Dózsa, J. Lourenco, T. Antao and T. Fadgyas. “GRADE: A Graphical Development and Debugging Environment for Parallel Programs”. Parallel Computing Journal, Vol. 22, No. 13, pp. 1747-1770, Elsevier. 1997.

[KT99] P. Kelly, F. Taylor. “Coordination Languages”. Research Directions in Parallel Functional Programming. pp. 305-321. Springer-Verlag. 1999.

[KK96] S. Krishnan, L. V. Kale. “Automating Parallel Runtime Optimizations Using Post-Mortem Analysis”. International Conference on Supercomputing, pp. 221-228. 1996.

[KW85] Clyde P. Kruskal and Alan Weiss. “Allocating independent subtasks on parallel processors”. IEEE transactions in Software Engineering, vol. 11, num. 10, pp. 1001–1016, 1985.

[LR+03] H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á.J. Rebón, P. W. Trinder.

"Comparing Parallel Functional Languages: Programming and Performance". Higher-Order and Symbolic Computation, 16, pp. 203-251. Kluwer Academic Publishers. 2003.

[LOP05] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. "Parallel Functional Programming in EDEN". Journal of Functional Programming, No. 15 (2005), 3, pp. 431-475.

[LS+05] Róbert Lovas, Raül Sirvent, Gergely Sipos, Josep M. Pérez, Rosa M. Badia, Péter Kacsuk. "GRID superscalar enabled P-GRADE portal". Preliminary proceedings, CoreGrid Workshop on Integrated Research in Grid Computing. Pisa, Italy, pp. 467-476. 2005.

[MA+02] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafrom, K. Tan. "From Patterns to Frameworks to Parallel Programs". Parallel Computing, Vol. 28, no. 12, pp. 1663-1683. 2002.

[Ma95] Eric Maillet. "Tape/Pvm an efficient performance monitor for Pvm applications. User guide". LMC-IMAG, Grenoble, France, 1995.

[MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. "Patterns for Parallel Programming". Addison-Wesley, 2004.

[Mes04] Jose G. Mesa. "Desarrollo de un Framework para Aplicaciones Master/Worker". MD. Research work. Universitat Autònoma de Barcelona, February, 2004.

[MC+95] 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, pp. 37-46, November 1995.

[MCa+05] Anna Morajko, Paola Caymes, Tomàs Margalef, Emilio Luque. "Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications", LNCS, Vol. 3515/2005, pp. 132 -139, Springer-Verlag. 2005

[MCe+01] A. Morajko, E. Cesar, T. Margalef, J. Sorribes, E. Luque, "Dynamic Performance Tuning Environment", LNCS, Vol. 2150 (Euro-Par 2001), pp. 36-45, Springer-Verlag. 2001.

[MCe+05] Anna Morajko, Eduardo César, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, and Emilio Luque. "Automatic Tuning of Master/Worker Applications", LNCS, Vol. 3648, pp. 95-103. Springer-Verlag. 2005.

[MC+05] Anna Morajko, Eduardo César, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, and Emilio Luque. “Development and Tuning Framework of Master/Worker Applications”. Invited paper. Journal of Computer Science & Technology (JCS&T). Vol. 5, num. 3, pp 115-120. 2005.

[Mor03] Anna Morajko. “Dynamic Tuning of Parallel Applications”, PhD. Thesis, Universitat Autònoma de Barcelona, December, 2003.

[MM+03] A. Morajko, O. Morajko, J. Jorba, T. Margalef and E. Luque. “Automatic Performance Analysis and Dynamic Tuning of Distributed Applications”. Parallel Processing Letters. Vol. 13, number 2. 2003.

[MA+01] L. M. Moreno, F. Almeida, D. González, and C. Rodríguez. “The Tuning Problem on Pipelines”. LNCS, Vol. 2150, pp. 117-121. Springer-Verlag. 2001.

[MoC+05] Andreu Moreno Vendrell, Eduardo César, Joan Sorribes, Tomàs Margalef, Emilio Luque. “Balanceo de carga en sistemas distribuidos Master-Worker”, XVI Jornadas de Paralelismo 2005, pp. 443-450, Granada, 2005.

[MPI95] “MPI: A Message-Passing Interface Standard”. Message Passing Interface Forum. June, 1995.

[NA+96] W. E. Nagel and A. Arnold and M. Weber and H. C. Hoppe and K. Solchenbach. “VAMPIR: Visualization and Analysis of {MPI} Resources”, Supercomputer, Vol. 12, no. 1, pp. 69-80. 1996.

[Paraver] “Paraver Reference Manual”. http:// //www.cepba.upc.edu/paraver/

[PF02] S. Pllana and T. Fahringer. “On Customizing the UML for Modeling Performance-Oriented Applications”. In proceedings of <<UML>> 2002, "Model Engineering, Concepts and Tools", LNCS 2460, Dresden, Germany. Springer-Verlag 2002.

[PF05] S. Pllana and T. Fahringer. “Performance Prophet: a performance modeling and prediction tool for parallel and distributed programs”. ICPP 2005 Workshops. International Conference Workshops. p.p. 509 – 516. 2005.

[RS+01] Randy L. Ribler, Huseyin Simitci, Daniel A. Reed. “The Autopilot performance-directed adaptive control system”. Future Generation Computer Systems. Vol. 18. pp. 175-187. 2001.

[RR+93] Reed, D.A. Roth, P.C. Aydt, R.A. Shields, K.A. Tavera, L.F. Noe, R.J. Schwartz, B.W. "Scalable performance analysis: the Pablo performance analysis environment". Proceedings of the Scalable Parallel Libraries Conference, pp. 104-113. USA. 1993.

[RR+98] J. Roda, C. Rodríguez, F. Almeida, and 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, pp. 57-63. 1998.

[TC+02] Cristian Ţăpuş, I-Hsin Chung, and Jeffry Hollingsworth. "Active Harmony: Towards Automated Performance Tuning". Proceedings of the 2002 ACM/IEEE conference on Supercomputing. Pp 1-11. Baltimore, Maryland, 2002.

[TLP02] P.W. Trinder, H-W. Loidl, R.F. Pointon. "Parallel and Distributed Haskells". Journal of Functional Programming. Volume 12 , Issue 5 (July 2002). pp 469 – 510.

[Weiss] Eric W. Weisstein. MathWorld™, Online mathematics encyclopedia. http://mathworld.wolfram.com

[VR99] J. S. Vetter, D. A. Reed, "Managing Performance Analysis with Dynamic Statistical Projection Pursuit", Proceedings of SC'99, Portland, 1999.

[WK+98] Randolph Wang, Arvind Krishnamurthy, Richard P. Martin, Thomas E. Anderson, and David E. Culler. "Modeling Communication Pipeline Latency". Measurement and Modeling of Computer Systems, pp. 22-32. 1998.

[WW95] K. Williams, S. Williams, "Implementation of an Efficienct and Powerful Parallel Pseudo-random Number Generator", available from: http://www.cs.reading.ac.uk/cs/CCL/rand.epvm95.html.

[WM00] Felix Wolf and Bernd Mohr. "Automatic Performance Analysis of MPI Applications Based on Event Traces". LNCS, Vol. 1900 / 2000, p.123. Springer-Verlag. 2000.

[Yan94] Jerry Yan. "Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers". Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences. pp. 625-633. 1994.