STOCHASTIC MODELS FOR PERFORMANCE ANALYSES OF ITERATIVE ALGORITHMS IN DISTRIBUTED ENVIRONMENTS

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Henri Casanova

May 1998

To my parents and my sister.

Acknowledgments

The author expresses gratitude and appreciation to the members of his dissertation committee, Jack Dongarra, Jens Gregor, Samuel Jordan and Michael Thomason for their encouragement and participation throughout my doctoral experience.

Special appreciation is due to Professor Jack Dongarra, Chairman, and Professor Mike Thomason, Chairman, who provided guidance and support, appropriate and thorough commentaries as well as the opportunity to expand my perspectives in the ever changing computer science research community during the course of my graduate study.

I also would like to thank L. Susan Blackford who was always available to answer my questions, regardless of their nature.

Finally, I am grateful to the Department of Computer Science at the University of Tennessee for allowing me to use its computing facilities to do this doctoral work.

Abstract

This research aims at creating a framework to analyze the performance of iterative algorithms in distributed environments. The parallelization of certain iterative algorithms is indeed a crucial issue for the efficient solution of large or complex optimization problems. Diverse implementation techniques for such parallelizations have become popular. They are examined here with a view to understanding their impact on the algorithm behavior in a distributed environment. Several theoretical results concerning the sufficient conditions for, and speed of, convergence for parallel iterative algorithms are available. However, there is a gap between those results and what is relevant to the user at the application level. In particular, an estimate of the algorithm execution time is often desirable.

The performance characterization presented in this dissertation follows a stochastic approach partially based on a Markov process. It addresses different characteristics of the algorithmic execution time such as mean values, standard deviations and rare events. It is shown how this approach can fill the aforementioned gap thanks to stochastic models, which take into account the distributed environment used to run the algorithm. We concentrate on distributed-memory systems. The results of this research enable the end-user to make informed choices about what combinations of distributed environment and implementation style should lead to appropriate execution time distributions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Iterative algorithms are widely used to obtain solutions to a variety of problems that arise in different areas of science and engineering. Examples include solving systems of linear or non-linear equations that arise in modeling, simulation, and engineering design problems; and optimization problems, including linear programming, that arise in engineering design, economics modeling, and operations research applications.

A broad class of iterative algorithms aims at finding a fixed point of a given operator. Some of the well-know numerical methods in that class are Jacobi and Gauss-Seidel algorithms, overrelaxation methods (JOR and SOR), Gradient and Scaled Gradient algorithms, and Newton and Approximate Newton methods. The example problems studied in this dissertation are all non-linear methods, but our results are applicable to linear cases as well. We emphasize non-linear problems because they usually lead to larger amounts of numerical computation than linear problems of comparable size. In many iterative methods, if the operator is non-linear, the gradient and sometimes the Hessian of the operator, if not known analytically, must be approximated at each iteration usually by approximating partial derivatives in each dimension. Such approximations are usually costly, especially when no preliminary knowledge of the nature of the operator is available.

For problems with large numbers of dimensions or extensive numerical computation for each component at each iteration (e.g. Hessian computation), it is natural to consider a parallel implementation of the iterative algorithm. The parallel implementations that have been investigated in the past usually fall into two categories: *synchronous* implementations and *asynchronous* implementations. Several types of distributed environment can be used to run such implementations. One might consider shared-memory or distributed-memory super-computers. However, the use of networks of workstations as distributed-memory super-computers has become popular in the last years due to their increased versatility and reduced cost. In this research, we are primarily interested in distributed-memory environments as they seem to be most widely used. It becomes necessary to understand the behavior of parallel implementations in such environments and, ideally, predict their performance. This can only be done by taking into account the specifics of the distributed environment. Such a study should allow users to choose the most efficient implementation style for their purposes depending on the computational facilities they can access and may also provide some guidance in the design of distributed systems.

1.1 Motivation

Analyzing the behavior and thus the performance of a parallelized iterative algorithm running in a distributed environment is not an easy task. The environment imposes its own constraints on the execution. The users may not have frequent access to a dedicated system to run their applications. In such a case, the workloads of the processors and the network contention depend on the load of the system generated by other users in addition to the iterative algorithm execution itself. Therefore, the processor workloads are usually disparate and vary over time according to different patterns. Furthermore, the amount of computation performed by a processor to update a component of the solution vector may not be known a priori: generally, it depends on the *shape* of the operator around the current solution vector. As we have already mentioned, different versions of parallelization have been studied for such iterative algorithms. In some cases, convergence results indirectly support a quantitative assessment of the parallel algorithm convergence rate. However, almost all these results are purely theoretical and do not take into account the nature of the distributed environment itself.

For instance, a technique first introduced as *Chaotic Iterations* (an asynchronous implementation) partially addresses the distributed nature of the execution environment. Indeed, one of the purposes of chaotic iterations is to produce an implementation more adaptive to the fluctuations that can take place in the distributed environment. But even in cases that identify sufficient conditions for this method to converge, there has been little work concerned with measuring the improvement an asynchronous implementation offers over a synchronous parallel implementation, in a given distributed environment. The only commonly available result for a chaotic iteration implementation is a lower bound on its theoretical rate of convergence. It is difficult for the user to relate this measure of convergence to the actual performance he can expect for his implementation on his distributed environment in terms of execution time for instance.

Due to non-determinism (randomness) both in communication and in computation, stochastic methods appear to be a natural way to move towards more complex and relevant models. These new models should capture more details about the distributed environment itself in terms of computation and communication speeds and patterns, and their impact on the user's implementation. Few attempts at using stochastic models to analyze the performance and behavior of parallel iterative algorithms appear in the literature. Indeed, stochastic approaches seem to lead to complicated models which present difficulties in obtaining useful performance characterizations. Most difficulties come from trying to properly model the distributed environment in connection with the algorithm.

Hence, most stochastic approaches in the past used very stringent assumptions and are therefore limited in their domain of application. If some reasonable assumptions could lead to tractable models, then those models should provide insight into the performance analysis of parallel iterative algorithms in given distributed environments. Furthermore, a reasonable model of the distributed environment will provide performance results directly useful to the user for his application, rather than considerations on the algorithm's theoretical convergence rate that have no concrete relevance to that application.

1.2 Problem Statement

This dissertation examines different implementation techniques for parallelizations of certain iterative algorithms in a view to characterize their performance in a distributed environment. The performance characterization, in terms of execution times, is based on a stochastic approach and addresses different properties of this performance such as mean values, standard deviations and rare events.

1.3 Organization of this Dissertation

In Chapter 2, we precisely define the class of iterative algorithms that we consider in this research. We then describe in detail the two types of parallel implementation that we will study: (i) synchronous and (ii) asynchronous. Those implementations have been the object of extensive theoretical study in the past, especially concerning sufficient conditions for their convergence and their convergence rates. We review these results and select the ones that we will extend for this research. Finally, we present related works on the stochastic modeling of parallel iterative algorithms. We thereby not only motivate the use of a stochastic approach but also highlight the existing shortcomings that we plan to address.

Chapter 3 defines *application-level modeling* and outlines a model for the distributed environment and for the iterative algorithm. Once those preliminary foundations have been set, we give a more formal and rigorous definition of the complete model encompassing both the environment and the algorithm. Using these definitions as well as some reasonable assumptions, it is possible to extract a Markov chain as the underlying random process in our model. One can then define several random variables (RV) of interest and the distributions of those RVs can be derived from the Markov chain.

At this point, our stochastic model is complete and ready to be exploited. This is done in Chapter 4 which starts with examples of Markov chain computations. These examples are used to obtain more insight into the behavior of a parallel implementation of an iterative algorithm. We then propose three new estimates for the algorithm convergence rate. These estimates are stochastic extensions of existing results and take into account the distributed environment. The implementation speed in terms of number of iterations performed per time unit is also defined and characterized. We introduce *Large Deviation Theory* and show how it can be used to obtain in-depth details about the extreme behaviors of the implementations. We conclude the chapter with a description of our entire performance characterization in terms of execution time. It consists of three levels, each of which provides information about different features of the execution time distribution.

The last step is to validate our approach with experimental results. This is done in Chapter 5. First, we conduct a simulation and compare its results to our performance characterization. Second, we conduct experiments for a real implementation of an iterative algorithm on a real distributed environment. We obtain different experimental results for different time periods. Those results are compared to the simulation and to our performance characterization.

Chapter 6 concludes this dissertation by summarizing our results, explaining how this work contributes to the field of performance modeling for parallel iterative algorithms, and suggesting numerous future research directions.

Chapter 2

Parallel Iterative Algorithms

In this chapter, we give some background material about iterative algorithms and we discuss possible parallel implementations. We then describe convergence results available for those implementations. Finally, we explain our motivations for taking a stochastic approach for analyzing those algorithms and develop such an approach.

2.1 The Algorithm

This section contains some basic concepts of the algorithms we are considering, an example, and discussion of some implementation details.

2.1.1 Basics

In this research, we focus on iterative algorithms in which an operator is applied to a set of data (usually a vector) repetitively until some convergence criteria are met. If the set of data constructed by the algorithm is a sequence of vectors $x(t)$ in $\mathbb{R}^m$, then the algorithm can be written as

$$\begin{cases} x(0) \in \mathbb{R}^m \\ x(t+1) = Op(x(t)) \quad \text{for all } t \in \mathbb{N}. \end{cases} \tag{2.1.1}$$

If the algorithm converges, the sequence $x(t)$ converges to a fixed point of operator Op. Much work has been devoted in the past to finding useful operators for some specific problems or finding operators that provide the highest convergence speeds. Iterative algorithms are used in particular for two purposes:

- Solving linear systems of equations,
- Minimizing a continuously differentiable cost function.

In fact, solving a linear system of equations can be seen as a minimization problem, i.e., the solution to $Ax = b$ in R^m also minimizes the cost function $\mathcal{F} : \mathbb{R}^m \mapsto \mathbb{R}$ defined as $\mathcal{F}(x) = \frac{1}{2}x'Ax - x'b$. The algorithm converges for symmetric and positive definite matrices.

The names used in the literature for iterative methods vary from author to author, and usually depend on whether the operator is linear or non-linear. We use the terminology of [9] for iterative methods. There is a long history of iterative methods for solving linear systems [27](p. 326). The earliest methods are the Jacobi and Gauss-Seidel methods. The Gauss-Seidel method is a variation on the Jacobi method where the evaluation of the operator at each step uses components of the previously computed solution vector during that step. Variations of those methods are obtained by using a *relaxation parameter* and are respectively called the Jacobi Overrelaxation (JOR) and Successive Overrelaxation (SOR) methods. Those methods have immediate non-linear counterparts. Another method for solving linear systems of equations is sometimes called Richardson's method; its non-linear counterpart is called the Gradient algorithm or the Steepest Descent algorithm and can be generalized into the Scaled Gradient method. Other non-linear algorithms include Newton and Approximate Newton methods (the linear version of which converges in one step). It can be shown that for non-quadratic problems, the Newton methods converge much faster than the other methods we have introduced (see [42] for instance). All those methods can be used in a Jacobi or a Gauss-Seidel fashion.

2.1.2 An Example

In order to better understand what really happens when an iterative algorithm is implemented, we consider the following example. Say we want to find the minimum of a cost function $\mathcal{F} : \mathbb{R}^2 \mapsto \mathbb{R}$. To do so, we can use the Gradient algorithm which can be written as:

$$\begin{cases} x(0) \in \mathbb{R}^m \\ x(t+1) = x(t) - \gamma \nabla \mathcal{F}(x(t)) \quad \text{where } \gamma \in \mathbb{R}. \end{cases}$$

The convergence of this algorithm is insured under certain conditions. First, the gradient of the function to be minimized must be *Lipschitz continuous* in some subset D of $\mathbb{R}^m$. That is, $\|\nabla \mathcal{F}(x) - \nabla \mathcal{F}(y)\| \leq L\|x - y\|$ for all x and y in D where L is a real constant and $\|.\|$ denotes a norm of $\mathbb{R}^m$. Second, the step-size γ must be less than some constant that depends on the Lipschitz constant, L, of the gradient of the cost function. Details on those conditions can be found for instance in [9].

Figure 2.1 shows what happens during one step of the algorithm (this figure is inspired by the ones found in many reference manuals). On the figure, the curves represent sets of points where the value of $\mathcal{F}$ is constant. The minimum of $\mathcal{F}$ is attained for a point located inside the innermost curve. At the current iteration step, the solution vector lies on the outermost curve, meaning that $\mathcal{F}(x(t)) = 3$ according to the figure. To update this solution vector, the gradient of the function at point $x(t)$ is computed, scaled by the factor γ and subtracted from $x(t)$. This is shown on the figure with the thick arrow that goes from $x(t)$ to $x(t+1)$. This

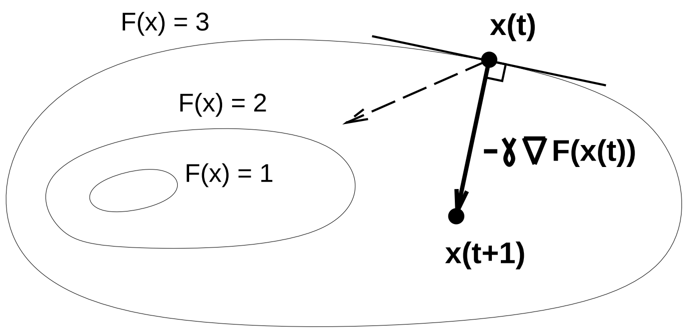

Figure 2.1: Gradient algorithm example

process is repeated until a convergence criterion is met. Usually such a criterion specifies that the distance between successive solution vectors is smaller than some ϵ. Incidentally, this figure shows that the descent direction of the Gradient algorithm (the steepest) is not the shortest path to achieve convergence in this case. The descent direction shown with the dotted arrow, for instance, would lead to a faster convergence. These situations have motivated other methods such as the Scaled Gradient algorithm.

For each iteration, the algorithm requires an evaluation of the gradient $\nabla\mathcal{F}(x(t))$. If this gradient is not known directly or analytically, then the algorithm must compute an approximation. Furthermore, some classic iterative methods require the knowledge of the Hessian matrix for each iteration. For instance, the Newton iteration can be written as:

$$x(t+1) = x(t) - \gamma[\nabla^2\mathcal{F}(x(t))]^{-1}\nabla\mathcal{F}(x(t)).$$

Not only is the Hessian needed at each iteration, but it must be inverted. The Approximate Newton method does not require the actual matrix inversion, but can be implemented instead thank to another iterative algorithm at each iteration. This second algorithm is used to solve a linear system; it is, however, executed only for a few steps.

The actual details of the numerical methods in use are not the central focus of this dissertation. The general equation 2.1.1 will be sufficient for our needs most of the time, and specific methods will be mentioned only for the sake of practical examples.

2.2 Parallel Implementations

2.2.1 Motivation

Iterative algorithms can have prohibitive execution times when implemented sequentially. First, the size of the problem influences the amount of computation to be performed because every component of the solution vector is updated at each iteration. Second, some particular iterative methods have inherently high time complexity at each step. For instance the Newton method for non-linear problems involves computation and inversion of a Hessian matrix at each iteration. Even though this method leads to fast convergence in terms of number of iterations, it might be too computationally intensive for medium or large size problems, when implemented sequentially. Many of today's challenging scientific applications lead to very large optimization problems. Examples include Positron Emission Tomography (PET) reconstruction for medical imaging [40], or non-linear network flow problems that occur in electrical networks, communication networks and financial models [7]. In order to satisfy such computational requirements, it seems natural to consider parallel implementations of iterative algorithms.

We recall (see Section 1.1) that we are mainly considering parallel implementations in distributed memory environments in this research. In the two following sections we describe the two main implementation strategies: *synchronism* and *asynchronism*.

2.2.2 Scheduling

When implementing a parallel version of an iterative algorithm, perhaps the first question concerns scheduling. The computation, in our case updating the components of the solution vector, has to be distributed among the processors contributing to the algorithm. One can usually distinguish two main scheduling strategies. In a *static scheduling* strategy, a processor always updates the same components of the solution vector and typically is the only one to update those components. In a *dynamic* scheduling strategy, processors can update any (or at least different) components of the solution vector at different iterations. In this research, we chose to consider only static scheduling. This choice is motivated by the scheduling analysis presented in [55] where it is shown that a static scheduling strategy is almost always a little more efficient than any dynamic ones for parallel iterative algorithms. Since dynamic scheduling is more difficult to implement than static scheduling, the choice seems clear. Furthermore, the main focus of [55] is the case of shared-memory environments which are usually the best ones for dynamic scheduling; hence we consider static scheduling since we are concentrating on distributed memory settings.

2.2.3 The Synchronous Case

Synchronous implementations of iterative algorithms are generalizations of sequential implementations. As such, their convergence properties are well known. Often, it is rather straightforward to convert a sequential implementation of a given algorithm into a synchronous parallel implementation.

The Principle

Let us assume that the distributed environment used to execute the algorithm consists of p processors and that each processor is the only one that can access its local memory. The processors, however, can communicate via a network. A rigorous description of such a distributed environment is given in Section 3.2. The algorithm operates on a vector $x(t)$ of $\mathbb{R}^m$ according to equations 2.1.1. Each processor is in charge of updating a *piece* of $x(t)$, that is some subset of the components of this vector. In general it is assumed that the components assigned to different processors are pairwise disjoint. For instance, if $x(t) \in \mathbb{R}^9$ and if $p = 3$, then each processor could be in charged of updating three components of $x(t)$.

Figure 2.2 depicts the execution of a synchronous implementation running on 3 processors where the solution vector is segmented in three pieces. Each processor starts each iteration with the entire current solution vector in its memory. Then, each processor updates its piece of $x(t)$ by applying part of the operator Op to the entire vector. This is symbolized by the thick arrows on the figure. To keep the figure readable, only the first piece of the solution vector, the one assigned to processor 1, is entirely described. It is shown in shaded grey on the figure and is labeled by the iteration number. After each processor has updated its piece of the solution vector, the processors have an inconsistent version of that vector in their memory. For instance, as shown in the figure, processor 1 possesses the first piece of $x(t+1)$ but processors 2 and 3 still possess the first piece of $x(t)$. To return to a coherent state, the processors perform an *all-to-all* communication and exchange their up-to-date pieces of the solution vector. This is shown on the figure by thin arrows. Once the all-to-all communication is completed, all the processor have the same vector $x(t+1)$ in memory and can perform the updates of the next iteration in a similar way.

More formally, if the components of the solution vector $x(t)$ are denoted by $x_i(t)$, $i = 1, ..., m$ and if the components of $Op(x(t))$ are denoted by $Op_i(x(t))$ or

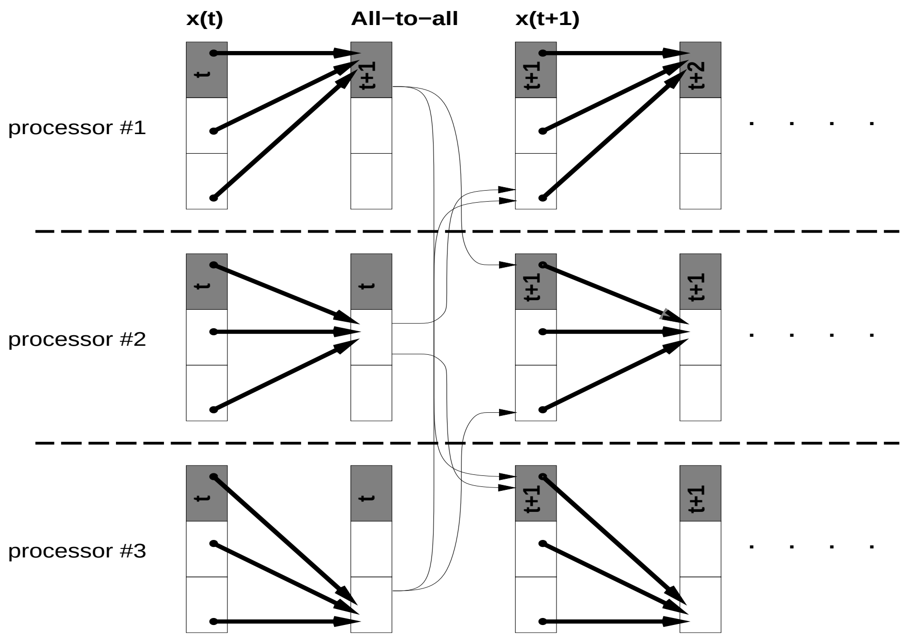

Figure 2.2: Synchronous algorithm

$Op_i(x_1(t), ..., x_m(t))$, the synchronous iteration can be written as:

$$\forall i, t \quad x_i(t+1) = Op_i(x_1(t), ..., x_m(t)). \tag{2.2.2}$$

Due to the all-to-all communication taking place at each algorithm iteration, the synchronous implementation is exactly equivalent to a sequential implementation in terms of numerical computations. It has therefore been a popular choice of implementation for many iterative algorithms since the convergence properties of sequential implementations have been studied extensively in the past. However, the performance of the algorithm in terms of execution time can suffer from several factors.

Performance Bottlenecks

The most obvious performance bottleneck in a synchronous implementation can be the network itself, during the all-to-all broadcast performed at each iteration. Indeed, if the time to send a message over the network is relatively long compared to the time necessary to perform an update of a piece of the solution vector, the processors spend a non-negligible amount of time being idle, waiting for the all-to-all communication to complete. This situation can occur if the network is inherently *slow* when compared to the processors, or if a large number of processors are participating in the computation and the network is flooded by messages at each iteration.

A second and somewhat less immediate performance bottleneck is created by a possible lack of synchronization among the processors [46, 35, 14, 25]. Let us suppose that a processor, say i, is slower than all other processors when computing its piece of the solution vector. Then, during the all-to-all communication phase of that iteration, all the other $p-1$ processors will be idle, waiting for processor i to finish its update and send its message. In other words, lack of synchronization among the processors can be detrimental to the execution time due again to the all-to-all communication.

The source of such lack of synchronization is twofold. First, the distributed environment can generate situations leading to losses of synchronization: the processors might not have the same computational speed. This can be due to actual difference in processor designs (*heterogeneous* set of processors) or to different processor workload distributions. Furthermore, a processor may not offer the same computational speed from one iteration to another, due to random workload fluctuations within that processor. The network behavior may also vary due to random fluctuations. Second, the iterative algorithm itself can lead to fluctuations in the computational times of the processors. Indeed, in many cases, applying the operator Op to the current solution vector can require more or less computation depending on the shape of the operator in a neighborhood of that vector. Typically, this happens in non-linear cases when, at each iteration, a gradient is approximated in each direction as the limit of a ratio involving actual cost function evaluations. In such cases a processor can exhibit various behaviors regarding the time necessary to perform an update.

Synchronous implementations are clearly a good choice when executed on perfectly homogeneous distributed environments, with a fast network, and for algorithms that exhibit uniform computational requirements for the solution vector updates. However, the use of Networks of Workstations (NoWs) as supercomputers has become popular in the last years. Such networks often contain workstations of different types, with varying workload distributions and networking performance depending on the current usage of those workstations. On the algorithm side, there are many large non-linear optimization problems that generate non-deterministic computation patterns as explained above. In order to provide efficient parallel versions of iterative algorithms under these conditions, another strategy has to be employed.

2.2.4 The Asynchronous Case

As early as 1969 [16], but only in some special cases, another possible implementation for parallel iterative algorithm has been under investigation. Even though the early motivations to examine such an alternative to the synchronous parallel implementation were not really the same ones that interest us today, that early work led the way for many other studies. Let us first describe the basic principle of this type of *asynchronous* implementation.

Principle

As for the synchronous case, we assume that there are p processors in the distributed environment and that the solution vector is segmented in pieces assigned to each processor. Figure 2.3 describes a possible execution of an asynchronous implementation of an iterative algorithm. As in Figure 2.2, we consider 3 processors and the first piece of the solution vector (the one assigned to processor 1) is the only one fully described. Unlike Figure 2.2, Figure 2.3 shows the beginning of the algorithm (for $t = 0$).

The execution starts as for a synchronous implementation: each processor up-

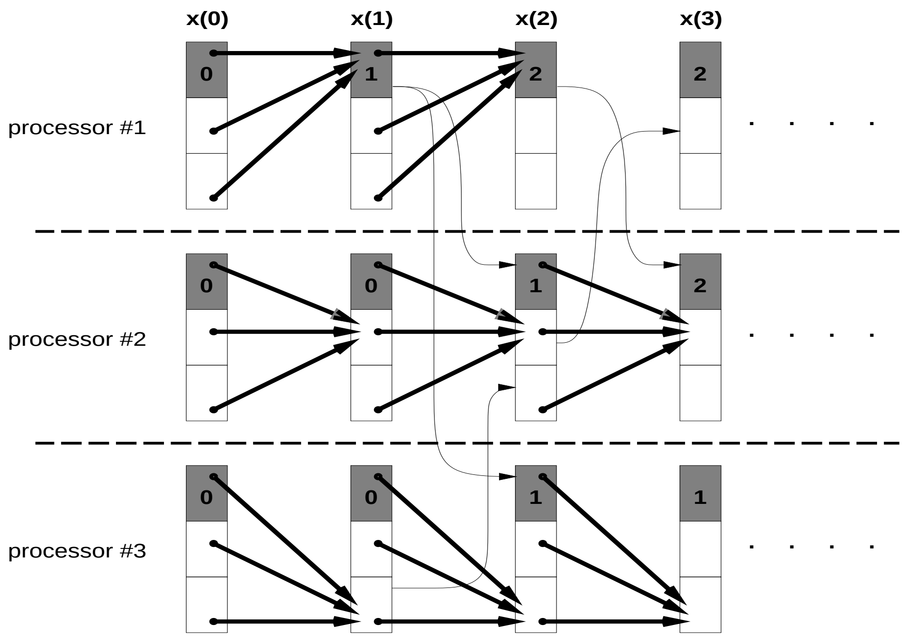

Figure 2.3: Asynchronous algorithm

dates its piece of the solution vector. However, there is no all-to-all communication phase following these updates to synchronize the processors. Instead, a processor is “free” to perform another update possibly using **out-of-date** data for the pieces of the other processors, or not to perform any update at all. In addition, a processor can decide at any time to send its most up-to-date piece to some of the other processors. For instance, on the figure, processor 2 computes the second piece of $x(2)$ with an out-of-date first piece of the solution vector. Processor 1 chooses not to perform any update from $x(2)$ to $x(3)$. Some communications are also performed (symbolized by thin arrows on the figure). For instance, processor 3, before computing $x(2)$, sends its most up-to-date piece of $x(1)$ to processor 2. On Figure 2.2 the numbering of the pieces of the solution vector was identical to the numbering of the solution vector sequence: $x(t+1)$ was equal to $Op^{t+1}(x(0))$. In the asynchronous case, this is not the case anymore. First, the processors do not have the same view of the current solution vector at each iteration phase. Second, the different pieces of $x(t+1)$ may not result from applying the operator $(t+1)$ times to the pieces of $x(0)$. For instance, the first piece of $x(3)$ on processor 1 results from updating the first piece of $x(0)$ only twice, and the first piece of $x(3)$ on processor 3 results from updating the first piece of $x(0)$ only once.

A good formal description of the asynchronous iteration is given in [5] and is inspired by the definition of chaotic relaxations in [16]. The definition we give here is very similar:

$$\forall i, t = 1, 2, ... \quad x_i(t) = \begin{cases} x_i(t-1) & \text{if } i \notin J_t \\ Op_i(x_1(s_1(t)), ..., x_m(s_m(t))) & \text{if } i \in J_t, \end{cases} \tag{2.2.3}$$

where J_t is a subset of $\{1, .., m\}$, $s_i(t)$ is an integer for all i, and $t = 1, 2, ...$ This definition is extremely general; it just states that a processor can sometimes update a component of the solution vector by applying the operator to some solution vector value. If $J_t = \{1, .., m\}$, then each processor updates its piece of the solution vector at each iteration. In order to make this definition more useful, Baudet in [5] proposes the three additional conditions:

Condition 2.2.1 *Conditions for asynchronous iterations:*

(i) $s_i(t) \leq t$ *for all* $t = 1, 2, \ldots$

(ii) $\lim_{t \to \infty}(s_i(t)) = \infty$.

(iii) *i occurs infinitely often in the sets J_t,* $t = 1, 2, \ldots$

Condition **(i)** states that when a processor updates a component of the solution vector it can only make use of *past* components. In other words, a processor cannot use components not yet computed. Condition **(ii)** states that the same value for a component cannot be used indefinitely when computing updates. This means that eventually, the value of a component will be replaced in favor of a more up-to-date value throughout the algorithm execution. Finally, condition **(iii)** says that a processor does not abandon a component for ever. If a processor updates component i of the solution vector, that processor will update component i an infinite number of times. Nothing is said on the convergence or termination of the algorithm and it is assumed to run forever.

Partial or Total Asynchronism

In the formal definition of asynchronous iterations that we have given so far, there is no limit on the amount by which a component used in an update can be out-of-date. The only statement concerning the use of out-of-date components is made by condition 2.2.1**(ii)** and it is fairly non-restrictive. For instance, if $s_i(t) = \lceil \frac{t}{2} \rceil$ for some i, then the components used in successive updates of $x_i(t)$

are more and more out-of-date and their "out-of-datedness" is unbounded. Such a case is referred to as a *totally asynchronous* implementation in [9]. If the amount by which a component used in an update is out-of-date is bounded, then the implementation satisfies the additional condition:

Condition 2.2.2 *Additional condition for asynchronous iterations*

(iv) *There exists a fixed integer s such that $t - s_i(t) \leq s$ for all i and $t = 1, 2, \ldots$.*

Such a case is referred to as *partially asynchronous.* In fact, the integer s can be seen as a measure of the asynchronism. If $s = 1$, then the algorithm behaves almost like a synchronous algorithm (a processor may still chose not to perform any update during an iteration). When s increases, the algorithm tends to behave like a totally synchronous algorithm. Actual implementations are often partially asynchronous since it is often practical to fix some kind of bound on the asynchronism for implementation purposes.

Discussion

One purpose of an asynchronous approach is to make a parallel iterative algorithm more adaptive in a distributed environment. The definition of the asynchronous iteration shows clearly that the algorithm can be as asynchronous as needed to take advantage of the very phenomena that were performance bottlenecks for a synchronous implementation. However, the definition also implies that the actual numerical computations performed in an asynchronous implementation are different from the ones performed in the synchronous case, and therefore different from the ones performed in the sequential case. The convergence of the algorithm is no longer implied by the same conditions as for the sequential case, and in fact, it is not even clear that an asynchronous parallel iterative algorithm ever converges. Furthermore, if convergence occurs, the rate of convergence is entirely unknown for the same reason. Fortunately, some convergence results are available and the most fundamental ones are presented in the next section.

2.3 Convergence Results

A fair amount of work has been devoted to proving and analyzing the convergence of asynchronous parallel iterative algorithms [9, 11, 16, 38, 37, 5, 10, 6, 51, 24, 53, 54]. Some of the earliest focused on specific iterative algorithms or on specific implementations. A sufficient condition for convergence for linear operators is available in [16], only for partially asynchronous implementations. In [38, 37], this sufficient condition is generalized to the case of certain non-linear operators, still in a partially asynchronous setting. In [5], Baudet gives a convergence theorem and a convergence analysis for those non-linear operators and for any asynchronous implementation (total or partial). Baudet's work is the object of the following section.

2.3.1 Convergence Theorem for Contracting Operators

In [5], Baudet establishes a sufficient condition of convergence for asynchronous iterative algorithms whose operator is *contracting*. He then proceeds with a convergence analysis that provides a lower bound for the convergence rate. Let us first define the contracting operator concept.

Contracting Operators

Definition 2.3.1 *An operator Op from $\mathbb{R}^m$ to $\mathbb{R}^m$ is a contracting operator on a subset D of $\mathbb{R}^m$ if there exists a nonnegative $m \times m$ matrix A such that*

$$\forall x, y \in D \quad |Op(x) - Op(y)| \leq A|x - y|, \quad \textit{component-wise}$$

and $\rho(A) \leq 1$ where $\rho(A)$ denotes the spectral radius of A.

The uniqueness of a fixed point for a contracting operator on a subset D is immediate. The existence of such a fixed point can be proved in different situations. For instance, if D is closed and if $Op(D) \subset D$, a proof of the existence of a fixed point is given in [42]. Many examples of contracting operators and references can be found in [5] and it is shown that many linear and non-linear problems give rise to contracting operators. Among other examples, Baudet describes how contracting operators arise in linear and non-linear elliptic differential equations and shows that virtually all iterative functions occurring in the classical super-linear methods are contracting operators.

The Theorem

Baudet's main theorem can be stated as follows:

Theorem 2.3.1 If Op is a contracting operator on a closed subset D of $\mathbb{R}^m$ and if $Op(D) \subset D$, then any asynchronous iteration corresponding to Op according to equation 2.2.3 and satisfying condition 2.2.1 converges to the unique fixed point of Op in D.

This theorem is impressive since it states that for contracting operators, *any* asynchronous algorithm will converge, no matter how asynchronous. Let us give here a short sketch of the proof since it impacts Baudet's convergence rate estimation and our work in Chapter 4. To prove this theorem, Baudet uses the following lemma:

Lemma 2.3.1 Let A be a nonnegative square matrix. Then $\rho(A) < 1$ if and only if there exists a positive scalar ω and a positive vector v such that $Av \leq \omega v$ and $w < 1$. Furthermore, $\omega \geq \rho(A)$ and in fact ω can be chosen arbitrarily close to $\rho(A)$.

In the proof of the theorem, Baudet constructs a sequence of indices $\{t_k\}$, $k = 0, 1, ...$, such that $|x(t) - \xi| \leq \alpha\omega^k v$ for $t \geq t_k$ where ξ is the unique fixed point of Op in D. Since $0 < \omega < 1$, this shows that $x(t) \to \xi$ as $t \to \infty$, meaning that the algorithm converges.

Convergence Rate

After this theorem, [5] contains an analysis of the convergence rate of asynchronous iterative algorithms. Baudet defines the rate of convergence of such an algorithm as:

Definition 2.3.2 *Rate of Convergence Definition*

$$\mathcal{R} \overset{\Delta}{=} \liminf_{t\to\infty} [(-\log \|x(t) - \xi\|)/t].$$

$\|.\|$ denotes a norm of $\mathbb{R}^n$. In all that follows, we will choose to use the norm $\|x\| = \max\{|x_i||i = 1, ..., n\}$, which corresponds to the worst possible case for the convergence of the components. The definition of the rate of convergence has an immediate interpretation. If the logarithm is in base 10, then $1/\mathcal{R}$ measures the asymptotic number of *iterations* required to divide the error by a factor of 10 where an iteration is the computation described by equation 2.2.3 for all i.

Again, we give here a sketch of the proof of the next theorem since that proof will be used in Chapter 4. Baudet constructs a sequence of integers $\{k_t\}$ for $t = 0, 1, ...$ as follows. The proof of theorem 2.3.1 constructed the sequence of integers $\{t_k\}$ for $k = 0, 1, ...,$ satisfying $\|x(t) - \xi\| \leq \alpha\omega^k$ where α can be chosen as $\|x(0) - \xi\|$, that is, the error with the initial guess of the solution vector. That sequence was constructed as:

$$\begin{cases} t_0 & = 0 \\ t_{k+1} & = t_k + a_k + b_k, \end{cases} \qquad (2.3.4)$$

with a_k and b_k defined as:

Definition 2.3.3 *The $\{a_k\}$ and $\{b_k\}$ sequences:*

(i) *starting with the $(t_k + a_k)$-th iteration, no solution vector update makes use of values of components corresponding to iterates with indices smaller than t_k.*

(ii) *all solution vector components are updated at least once between the $(t_k + a_k)$-th and the $(t_k + a_k + b_k)$-th iterations.*

Baudet then defines the sequence $\{k_t\}$ for $t = 0, 1, ...$ as:

$$k_t \triangleq \sup\{k \in \mathbb{N} | a_0 + b_0 + ... + a_{k-1} + b_{k-1} \leq t\}. \qquad (2.3.5)$$

The theorem follows as:

Theorem 2.3.2 Let Op be an operator satisfying the condition of theorem 2.3.1 and A the matrix of definition 2.3.1, then the rate of convergence $\mathcal{R}$ of any asynchronous iteration corresponding to Op according to equation 2.2.3 and satisfying condition 2.2.1 is such that:

$$\mathcal{R} \geq -[\liminf_{t\to\infty}(k_t/t)] \log \rho(A).$$

This theorem provides a lower bound on the convergence rate of the algorithm. The sequence $\{k_t\}$ depends on the implementation itself. This sequence is increasing and the more asynchronous the implementation, the less rapidly the sequence increases. Baudet concludes its study with experiments, and we discuss them in Section 2.4. Another study of the convergence of parallel asynchronous iterative algorithms can be found in [9] and is presented in the next section.

2.3.2 A More General Theorem ?

Bertsekas and Tsitsiklis in [9] also propose a convergence theorem stating a sufficient condition for the convergence of asynchronous implementations of iterative algorithms. This theorem is in a more general setting than Baudet's theorem since the conditions on the operator are less restrictive. They first state the two conditions on the operator *Op*:

Condition 2.3.1 *There is a sequence of non-empty sets* $\{X(t)\}$ *with*

$$... \subset X(k+1) \subset X(k) \subset ... \subset X(0) \subset \mathbb{R}^m,$$

such that:

(i) *(Synchronous Convergence condition) We have:*

$$Op(x) \in X(t+1), \quad \forall t \text{ and } x \in X(t).$$

Furthermore, if $\{y^t\}$ is a sequence such that $y^t \in X(t)$ for every k, then every limit point of $\{y^t\}$ is a fixed point of Op.

(ii) *(Box Condition) For every t, there exist sets $X_i(t) \subset \mathbb{R}$ for $i = 1, ..., m$ such that:*

$$X(t) = X_1(t) \times X_2(t) \times ... \times X_m(k).$$

The convergence theorem can then be stated as follows:

Theorem 2.3.3 (Asynchronous Convergence Theorem) If condition 2.3.1 holds, and if the solution vector initial guess $x(0)$ belongs to the set $X(0)$, then every limit point of $\{x(t)\}$ is a fixed point of Op, where $x(t)$ is defined by equation 2.2.3.

Bertsekas and Tsitsiklis note after the proof that the challenge in applying the theorem is to identify the sequence $X(t)$. They claim that it can be straightforward in some cases, but that in other cases, it requires "creative analysis". All the applications of the theorem that are described in [9] are in fact in the case of operators that are contractions or pseudo-contractions with respect to a *weighted maximum norm.* The key idea is that under these assumptions, the *boxes* of theorem 2.3.3 are spheres of $\mathbb{R}^m$. In fact, the condition on the operator is fairly similar to Baudet's definition of a contracting mapping (it is a "Lipschitz-like" condition).

As in [5], a study of the convergence is provided. However, it appears less useful than Baudet's. Indeed, some additional assumptions are required in order to be able to make any statement on the convergence rate, even in the case of contractions with respect to a weighted maximum norm. Those assumptions are not very easy to interpret, and even when they are satisfied, Bertsekas and Tsitsiklis only give a lower bound on the rate of convergence of the algorithm, which is what Baudet already provides. Due to these considerations, we will mainly use Baudet's theory in this research. The more general theorem in [9] seems to lead to less directly applicable results and it is not clear that those results would be in fact more powerful. Besides, Baudet's proof techniques prove to be fairly well suited to a stochastic approach as demonstrated in Chapter 4.

At the end of [9], a special study of Gradient-like optimization algorithms is presented. It is shown that for some cost functions, totally asynchronous implementations of the algorithms are never guaranteed to converge whereas there are some partially asynchronous implementations of the Gradient algorithm that always converge. This is an interesting result as it proves that total and partial asynchronism are not always equivalent for some operators. This can be seen as an extension of the result of theorem 2.3.1 to some other classes of operators. In fact, partially asynchronous implementations of a Gradient-like algorithm with a small enough step-size (see Section 2.1.2) always converge for some operators that are not contracting operators in Baudet's sense. This result seems to be rather attractive and will be certainly the object of future work. However, in this dissertation, we consider only the general case without consideration of the specifics of the numerical method being used for the iterative algorithm. This seems to be the best choice since this work is our first approach at stochastic modeling of such algorithms and should therefore not be tailored to fit some specific algorithm.

2.4 Towards a Stochastic Approach

In this section we first explain our motivation for taking a stochastic approach to a performance analysis of parallel iterative algorithms. We then examine previous work concerning stochastic models for such an analysis and finally conclude with the goals of our stochastic approach.

2.4.1 Motivation

An obvious weakness in all the convergence analyses presented in 2.3 is that they are one step removed from actual implementations. Indeed, those results are fairly difficult to exploit for an end-user, who wants to write a parallel iterative algorithm to solve some scientific computational problem. Such a user needs an estimate of the convergence rate of the algorithm in terms of execution time. The results available so far give lower bounds of the convergence rate only in terms of number of iterations. First, it may not be obvious to the user how to determine the amount of time required to perform an iteration, especially in heterogeneous environments or for large non-linear problems. Second, the distributed environment creates some lack of synchronization among the processors as explained in Section 2.2.3, and can therefore have a large impact on the actual computation performed in an actual run of the implementation. Therefore, it is difficult for a user to decide whether a synchronous or an asynchronous implementation is a good choice given the distributed environment and the numerical method to be used. And if an asynchronous implementation seems to be the right choice, then how can the user choose the degree of asynchronism that will yield the best average performance ?

Even if the estimates of the rate of convergence of the iterative algorithms were more in touch with what the end-user needs to make design decisions, they still provide only lower bounds on the rate of convergence. These bounds correspond to the worst-case scenario for a run of the implementation, the one for which asynchronism is entirely detrimental to convergence. This could still be useful if those bounds were tight. However, Baudet in [5] reports on several experiments with actual implementations running on a supercomputer. These experiments showed that the bounds on the rate of convergence are in fact fairly loose and that the actual implementations perform on average much better than the worst-case. This led Baudet to write

The bounds we have obtained (...) happen to be very conservative. It would certainly be very useful to obtain bounds (or estimates) corresponding to the average behavior of asynchronous iterations.

This is an early mention in the literature of the possibility of a stochastic approach to a performance analysis of parallel iterative algorithms. In the following section, we review research works since Baudet's observation.

2.4.2 Related Work

Baudet implies a set of stochastic models that would describe the behavior of parallel algorithms running on a given set of processors interconnected by some network. One of the objectives of our research is to provide such stochastic models. Few attempts at using stochastic models to analyze the performance and behavior of iterative algorithms appear in the literature. For instance, [9] (Section 6.3.5) provides comparisons between the synchronous and the asynchronous case, but the model is entirely non-random and therefore difficult to apply for non-deterministic *real-world* systems. In [52], it is shown that asynchronous algorithms have a "good" communication complexity as compared to synchronous ones, but here again, it is difficult to use these results to obtain quantitative measures of the actual performance of the algorithm in a given distributed environment.

A unique reference that proposes a real stochastic approach is Üresin and Dubois [55]. Its "probabilistic analysis" is based on the one in [33]. There are, however, several missing elements in this work that we address. This is understandable enough since it may be the first published work using a probabilistic approach to performance evaluation of distributed iterative algorithms. First, the authors agree that their model is interesting only for a shared memory implementation, of little relevance for a distributed memory system, and not suitable to model any heterogeneous system. By contrast, the research presented in this dissertation attempts to find performance results for distributed memory systems, including heterogeneous networks of workstations. Second, the scheduling strategy used in [55] is called *Age scheduling* - a reasonable choice as it is easy to implement on a shared memory machine. The entire stochastic model depends on this scheduling assumption. However, as mentioned in Section 2.2.2, the authors performed simulations to examine the accuracy of their stochastic model and took this opportunity to simulate other scheduling strategies. It appears that the best scheduling strategy is (expectedly) *static scheduling* whereas their probabilistic model is only valid for an *aging* strategy. Based on this analysis, our work will consider only static scheduling. Third, the stochastic model developed in [55] is valid only for certain distribution functions for the processors' execution times (Increasing Failure Rate (IFR) functions). This is a fairly rigid constraint, and the models of this research will be more flexible and, more importantly, tightly connected to the underlying distributed environment. Finally, the probabilistic model in [55] is in fact not a full-fledge stochastic model. It approximates expected values, and simulation is used to compute some of its parameters. It is unclear in the analysis, for instance, what influence the nature of the algorithm operator has on the performance of the asynchronous model. By contrast, our research will develop real models that capture the details of the distributed environment and will not include any simulation as part of the models.

2.4.3 Objectives

The main objective of our stochastic analysis is to provide the user with *useful* performance estimates for the implementation of a given iterative algorithm in a given distributed environment. These estimates should then help that user in making design decisions for the actual implementation of the algorithm. Two specific goals are (i) to describe a Markov chain model in detail, and (ii) to describe computations on the model that are relevant to algorithm performance (e.g., time to convergence) and that include contemporary development in probability calculations (the Theory of Large Deviations). The input to the model will be twofold. First, the model will take into account the nature of the distributed environment in terms of number of processors, computational speeds of these processors (including workload distributions) and network performance. Second, the model will also take into account the implementation strategy in terms of asynchronism. For each input, the model will provide performance measures and estimates, allowing comparisons of implementations in any distributed environment. The following chapter describes how such a model can be constructed.

Chapter 3

Stochastic Models

One thing late or early can disrupt everything,

and the disturbance runs outward in bands

like the waves from a dropped stone in a quiet pool.

John Steinbeck (1902-1968)

This chapter describes in detail the stochastic models that we have constructed for the analysis of parallel iterative algorithms. As it will be seen in Section 3.1, our modeling is at the *application-level.* It is designed to capture the behavior of these algorithms when implemented and executed on distributed environments. Corresponding models are introduced in Sections 3.2 and 3.3 and the following sections show how it is possible to formalize these models and extract relevant random variables (RVs). These RVs are used in Chapter 4 to obtain new estimations of several performance measures.

3.1 Application-level Modeling

Stochastic approaches are used in many fields of computer science, especially for a better understanding of the behavior of computer systems. Usually, the goals of such approaches are to provide one or more stochastic models that reflect the random behavior of the system under investigation. The models can then be used to make predictions about the system with different configurations. In this work, we are interested in providing the *end-user* with useful insight into the performance of different implementation strategies for parallel iterative algorithms in some user-defined distributed environment. As explained in Section 2.4.1, an end-user is one who is willing to write a parallel program to perform some scientific computation. This is why we characterize our approach as *application-level.* By contrast, one can find examples of stochastic models for computer systems that are at a much lower level; and even though their results might impact the end-user, they would be extremely difficult to interpret directly in terms of that user's application performance.

As an example of a non-application-level model one can cite the work described in [45]. In that work, Ren, Mark and Wong are concerned with analyzing the end-to-end performance of an ATM network, that is, a network of ATM multiplexer or ATM switches. They use queuing theory (more specifically, *tandem-queuing* [59]) as well as a *fluid flow approximation* in order to build two different models. Then, they compare the accuracy of the two models versus simulation results. In the conclusion of their article, they state that their work provides a better understanding of the end-to-end behavior of an ATM network, and as such, can be used to make "efficient resource allocation in call admission". It is then clear that the results from that work are very difficult to use directly by a end-user as we have defined him, even though the call admission scheme in the distributed environment, if any, is certainly going to have an impact on the user's application at some level.

In order to provide results that are directly useful to the end-user, some of the low-level elements of the computer system must be ignored or at least approximated to perform an application-level analysis. An example of such an analysis can be found in [2]. In that work, Adve and Vernon propose to study the influence of random delays on the behavior of a parallel program running in a shared-memory environment. After making stochastic assumptions about the system (mainly about independence and distributions), they perform some expectation and variance computations leading to the conclusion that process execution times in the presence of random delays asymptotically approach a normal distribution. Then, they conduct a series of experiments with actual parallel programs running on multiprocessors to validate their analysis. Their work has clear implications for parallel program performance prediction models, but also for actual programmers (end-users). Their results indicate that the effects of random communication delays can usually be ignored in making the choice between static and dynamic scheduling in the systems. This is a perfect example of application-level analysis since it has direct bearings on the implementation choices made by the end-user in our sense.

In this work, our goal is to carry out a stochastic analysis that provides directly useful results to the programmer of a parallel iterative algorithm in a distributed memory environment.

3.2 Modeling the Distributed Environment

In this dissertation, the distributed environment in which parallel iterative algorithms are executed is a computer network of p geographically distributed nodes connected by a communication facility. A node is composed of a processor, memory and a network interface. Each node has its own memory accessed only by its processor. In this distributed memory setting, nodes can exchange data via the communication facility, thanks to their network interfaces. We do not require that all the nodes be identical and are therefore supporting a *heterogeneous* environment.

The communication facility is seen as an abstract device that allows reliable point-to-point communication between any two nodes of the network. Figure 3.1 shows a schematic representation of such an organization. For the sake of simplicity, we will use the term *network* to describe the communication facility only.

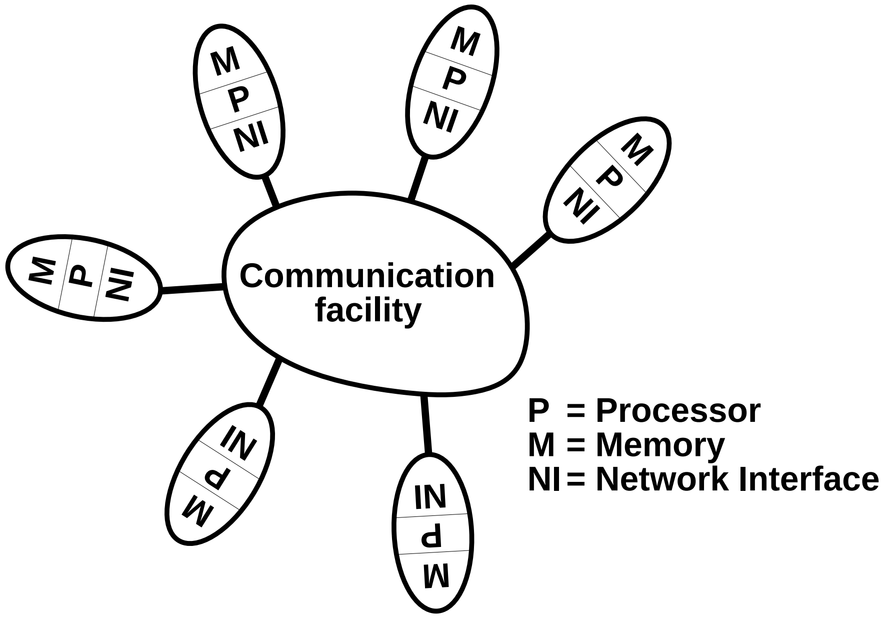

Figure 3.1: Distributed environment model

3.2.1 The Network

Our model does not make any assumption about the topology of the underlying physical network. This physical network delivers some service to the user that makes it appear as a reliable point-to-point communication facility. Our model will therefore be applicable for diverse computer networks, from a Massively Parallel System (MPP) to an Internet-wide collection of machines, and anything in between. In order to analyze the impact of this network on the behavior of any kind of parallel algorithm running in such a setting, it is necessary to somehow quantify the performance of the service that can be expected at the user end. Let us first define *network performance* in a way that is relevant to our purpose.

As seen in Section 2.2, parallel iterative algorithms exhibit some determinism in communications. Typically, the messages exchanged by the different nodes are always of the same lengths (or at least of very comparable lengths). This simplifies greatly the task of network modeling, for we do not expect real networks to deliver the same performance in terms of transmission rate for all message sizes. This is due not only to physical constraints (the number of workstations, for instance - see [50]), but also to system software overhead and communication protocol design [18], and traffic management mechanisms [41, 8]. A possible approach would then be to model the network service as a single number for each point to point communication, that is, the *time* required for each node to send a message to each other node. Since we are at the application level, this time would be measured from the instant when the user's program issues a *send* on the sending node, until the data has entirely arrived in the user space on the receiving node. This includes all the overheads aforementioned and allows the network to be seen as an abstract layer delivering constant service for each point-to-point communication. The entire network could then be modeled via a $p \times p$ real matrix, where p is the number of nodes in the system and element (i, j) of the matrix the time for node i to send a message to node j.

Even though attractive, that approach does not take into account one of the major factors that influence network performance: *network traffic*. Depending on the number of messages traveling on the network, depending on the sources and destinations of these messages and on the traffic patterns, one can expect varying network performance. Network performance can degrade rapidly with increasing traffic [8]. Measuring network performance with a single number seems inadequate because of network traffic fluctuations. A better way to estimate network performance is then to model each point-to-point communication time as an RV. The entire network can then be modeled with a $p \times p$ matrix of RVs. Different choices in the distribution of these RVs will reflect different network behaviors. Choosing a particular distribution of these Random Variables can be done by sampling network communication times and performing statistical inference (see [32] for instance).

3.2.2 The Nodes

As described in Section 2.2, a parallel iterative algorithm consists of a series of *updates* performed on parts of the solution vector. These updates take place on the nodes in the system. As for the network performance, one possibility would be to model a node's performance with a single number: the time it takes for that node to perform one update. The nodes could then be modeled with a vector of size p where the i^{th} component is the time for processor i to perform one update. This simplistic approach, however, is not satisfactory. First, as network traffic can cause fluctuations in network performance, processor load can also have an impact on execution speed. Depending on the load of its processor, a node will perform one update at different speeds, modifying the global execution time of the entire algorithm. Second, as explained in Section 2.2.3, the amount of computation required to perform an update is not constant: it depends of the shape of the operator Op around the current solution vector and can not be determined before the execution. We have thus identified two sources of randomness at the node level, justifying again the use of RVs in our model. Each node will be modeled by a single RV that describes the times the node requires to perform one update, and the entire set of nodes is modeled by a vector of RVs. The distributions of these RVs describe the behavior of the algorithm execution at the node level. As for the network, the distributions can be chosen thanks to sampling and statistical inference (see [32] for instance).

3.3 Modeling the Algorithm

The principle of the iterative algorithms we are studying has been already given in Section 2.1. In Section 2.2, we have seen that several parallel implementations are possible, mainly *synchronous* and *asynchronous* ones. In this section, we give the outline of our model for parallel implementations. By modifying some parameters of this model, the implementation can be either *synchronous* or *asynchronous*. This will allow us to compare different implementation schemes in different distributed environments as opposed to changing the entire model for each scheme.

First, we assume that if there are p nodes in the distributed environment used to execute the parallel algorithm, then there are p user processes (or threads of control) on each node's processor. From now on, we will use the terms *node*, *process* or *processor* indifferently, since there is always one single process running as part of the iterative algorithm on the single processor of each node. Second, we assume that each processor updates one *piece* of the current solution vector as stated in 2.2.2. The implementation of the algorithm is segmented in *phases*. At the beginning of a phase, the current solution vector is in the memory of the nodes. Each phase is composed of two *sub-phases*. During the first sub-phase, called the α *sub-phase*, each processor performs successive updates on its piece of the solution vector. If a processor performs more than one update during the α sub-phase, then it begins to use *out-of-date* data for the components of the solution vector that it is not updating. At the end of the α sub-phase, each processor broadcasts its piece of the current solution vector to all the other processors. Just after this broadcast, starts the β *sub-phase*. During the β sub-phase each processor is waiting to receive $p-1$ messages from the other processors. Each processor also has the possibility to perform additional updates on its piece of the solution vector, using more out-of-date data. A processor finishes its β sub-phase when it has received all the $p-1$ messages. This is also the end of the current phase, and the next phase is about to start.

Figure 3.2 depicts this algorithm. Three processors are numbered i, $i+1$ and

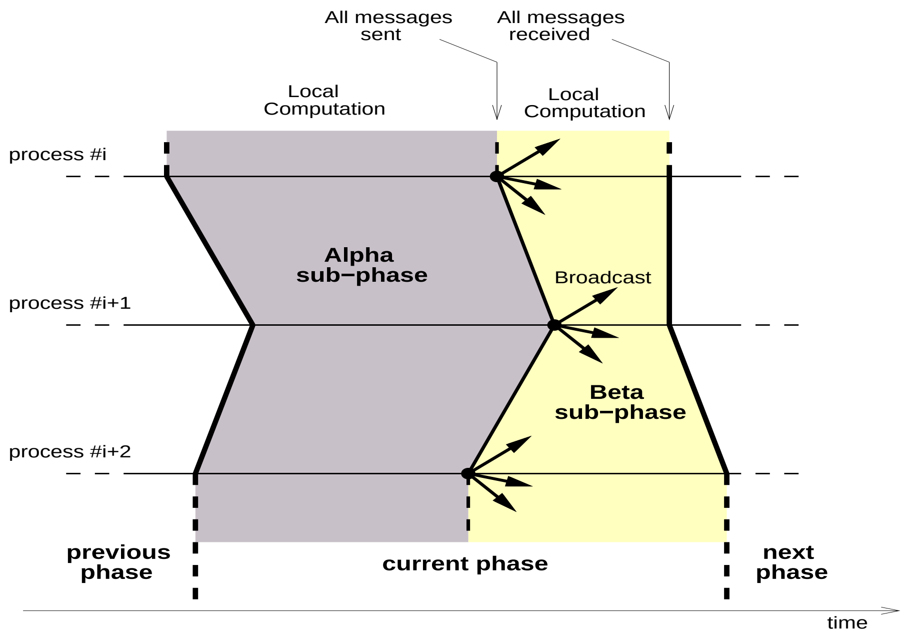

Figure 3.2: Decomposition of the algorithm in phases

$i+2$. On the left side of the figure, the three processors are starting a new phase of the algorithm (the *current* phase). One can see that processor $i+1$ is "late" to start the new phase as compared to processors i and $i+2$. Fluctuations in the network traffic and update computational times, as explained in Section 3.2, are responsible for this lack of synchronization among the processors (taking advantage of such lacks of synchronization is precisely one of the goals of asynchronous parallel iterative algorithms). After the α and the β sub-phases of the current phase, the processors start a new phase. In the figure, it is now processor $i+2$ which is late when compared to processors i and $i+1$.

The next section will establish definitions and notation that lead to a rigorous description of the algorithm execution in a given distributed environment.

3.4 The Complete Model

In this section, we introduce a set of formal definitions that will allow us to derive analytical results. We then state all the assumptions that we use in our study and conclude with some final remarks on the model.

3.4.1 Definitions and Notation

From now on, we assume that the distributed environment consists of p nodes, numbered $i = 1, 2, ..., p$ and that phases of the parallel iterative algorithm are numbered $k = 0, ...$ For mathematical simplicity, we also assume that each processor sends a message to **itself** as well as to every other processor after its α sub-phases. This means that each processor expects to receive p messages during each algorithm phase.

Definition 3.4.1 *Implementation dependent parameters*

(i) $A_i > 0$ *denotes the number of updates performed by processor* i *during the* α *sub-phase of each algorithm phase.*

(i) B_i *denotes the* **maximum** *number of updates that processor* i *is allowed to perform during the* β *sub-phase of each algorithm phase.*

These two definitions are fundamental, since they constrain entirely the way the algorithm iterates on the solution vector. For each process, the number of updates performed during the β sub-phase of each algorithm phase is not in general fixed ahead of time. The only thing that the user can do is to bound above the number of additional updates allowed during that sub-phase. For instance, if $A_i = 1$ for all i and $B_i = 0$ for all i, then the implementation is synchronous. If for some i, $B_i > 0$, then the implementation is asynchronous. By modifying the values of A_i and B_i for each i, the implementation can be made *more or less* asynchronous. The totally asynchronous case corresponds to $B_i = \infty$ for some i.

In order to quantify the algorithm execution time, we need to replace the continuous time by a discrete approximation. We therefore segment time on each processor into *CPU time units*. On each processor, this CPU time unit has a value in seconds, and every duration in the model can be expressed as a combination of the CPU time units of all the processors. For our model to fit reality exactly, this measure must be taken exactly equal to the *CPU cycle time* on each processor. Indeed, at the user level, processor time is discrete, and segmented in CPU cycles. Bigger values of the time unit lead to approximations. However, as it will be seen in what follows, taking an exact value leads to intractable models in terms of size. Furthermore, it will also be seen that such a level of precision, even if computable, would bring little or no improvement over less precise models. By increasing the CPU time unit value, one replaces the actual algorithm run by a coarser discrete approximation, but one creates more tractable models. We can now give the following definitions:

Definition 3.4.2 *Network- and node-specific variables*

(i) $\mu^i \in \mathbb{R}$ *denotes the duration in seconds of the CPU time unit on processor* i.

(ii) $\alpha^i(k,\theta) \in \mathbb{N}$ *for* $\theta \in \{1,..,A_i + B_i\}$ *denotes the duration of the* θ^{th} *update, if it is performed, of the solution vector during the* k^{th} *algorithm phase, on processor* i*, in CPU time units of processor* i.

(iii) $n_{i \to j}(k) \in \mathbb{N}$ *is the difference between the time when the user program on processor* i *posts a "message send" to processor* j*, and the time when the user program on processor* j *has entirely received the message in the user's space, during the* k^{th} *algorithm phase and in CPU time units of processor* i*. By convention,* $n_{i \to i}(k) = 0$ *for all* i *and* k.

Let us note that definition 3.4.2(**ii**) says that $\theta \in \{1,..,A_i + B_i\}$. Indeed, processor i performs A_i updates of the solution vector during its α phase, and performs **at most** B_i updates during its β phase (the actual number of updates performed in the β phase is random).

In our model, $\alpha^i(k,\theta)$ and $n_{i \to j}(k)$ are modeled as integer RVs for each i, j, k and θ. These RVs reflect all the randomness in the parallel algorithm execution in the distributed environment. Every other RV that will be introduced in future development will be in fact a function of these original RVs.

Finally, we define two variables that will allow us to describe the evolution of the algorithm throughout time:

Definition 3.4.3 *Time variables*

(i) $T^i(k) \in \mathbb{R}$ *denotes the* **time** *of the beginning of the* k^{th} *algorithm phase on processor* i*.*

(ii) $w^i(k) \in \mathbb{R}$ *is the* **duration** *in seconds of the* β *sub-phase of the* k^{th} *algorithm phase on processor* i*.*

In the next section, we derive the first equations from these definitions.

3.4.2 Main Time Equations

First, in order to simplify notations, let us define $\alpha^i(k)$ as:

$$\alpha^i(k) \stackrel{\Delta}{=} \sum_{\theta=1}^{A_i} \alpha^i(k,\theta)$$

for all i and k. Let us now consider the algorithm during its k^{th} phase on processor i. The phase started at time $T^i(k)$. One of the first questions one might ask is : "When does the $(k+1)^{th}$ phase start on processor i ?" Or in other words, what is the value of $T^i(k+1)$ knowing the value of $T^i(k)$? The duration of the k^{th} phase is the sum of the durations of its α and β sub-phases. By definition, the duration of the β sub-phase is $w^i(k)$. The duration of the α sub-phase is the sum of all durations of the A_i solution vector updates performed during that sub-phase. Therefore:

$$T^i(k+1) = T^i(k) + \mu^i \alpha^i(k) + w^i(k). \tag{3.4.1}$$

Let us now try to find the value of $w^i(k)$, or the duration of the β sub-phase on processor i during the k^{th} phase. Clearly, the β sub-phase starts at time $\beta^i_{start}(k)$, with

$$\beta^i_{start}(k) = T^i(k) + \mu^i \alpha^i(k).$$

On the other hand, the β sub-phase ends when processor i has received all expected p messages. Let us consider another processor, j. At the end of its α sub-phase, processor j sends a message to processor i. This message is sent at time $\beta^j_{start}(k)$, and received by processor i at time $\beta^j_{start}(k) + \mu^j n_{j\to i}(k)$, according to definition 3.4.2**(iii)**. Therefore, the last message received by processor i is received at time $\beta^i_{end}(k)$ with:

$$\beta^i_{end}(k) = \max_{j \in \{1,..,p\}} [\beta^j_{start}(k) + \mu^j n_{j\to i}(k)].$$

Now, $w^i(k)$ is given by

$$\begin{aligned}
w^i(k) &= \beta^i_{end}(k) - \beta^i_{start}(k) \\
&= \max_{j\in\{1,..,p\}} [\beta^j_{start}(k) + \mu^j n_{j\to i}(k)] - \beta^i_{start}(k) \\
&= \max_{j\in\{1,..,p\}} [\beta^j_{start}(k) + \mu^j n_{j\to i}(k) - \beta^i_{start}(k)] \\
&= \max_{j\in\{1,..,p\}} [T^j(k) + \mu^i \alpha^j(k) - \beta^j_{start}(k) + \mu^j n_{j\to i}(k)]
\end{aligned}$$

by replacing $\beta^j_{start}(k)$ with its expression. Since $n_{i\to i}(k) = 0$, one can rewrite the expression of $w^i(k)$ as

$$\begin{aligned}
w^i(k) = \max[0, \max_{j\in\{1,..,p\}-\{i\}} (T^j(k) + \mu^j \alpha^j(k) \\
- T^i(k) - \mu^i \alpha^i(k) + \mu^j n_{j\to i}(k))].
\end{aligned} \tag{3.4.2}$$

This last equation shows that $w^i(k) \geq 0$ for each i and k, which is of course natural, since $w^i(k)$ is a duration. Equations 3.4.1 and 3.4.2 are used as the first base of our model; they will be referenced frequently in future developments. In order to illustrate these notations, the next section shows a simple example of how an algorithm phase is modeled.

3.4.3 An Example

Let us consider a distributed environment that consists of 3 processors, and the algorithm during its k^{th} phase. The implementation of the parallel algorithm is such that $A^1 = A^2 = A^3 = 3$, $B^1 = B^2 = 2$ and $B^3 = 1$.

Figure 3.3 shows a diagram similar to Figure 3.1, but with all the details. The 3 processors enter their respective α sub-phase at different times. They each perform 3 updates during that sub-phase before entering their respective β sub-phase. Just before entering the β sub-phase, each processor broadcasts its current piece of the solution vector to every other processor. This is represented with thin dotted lines on the figure, and only for processor 2 for the sake of clarity. The time at which a processor receives a message is noted on that processor time axis with a big dot. There are 2 dots on each time axis since each processor receives 2 messages. Let us examine each processor's β sub-phase one by one.

Processor 3 can perform one solution vector update during its β sub-phase. Before it has finished that update, it has already received one message. The second expected message arrives after the update is completed, leading processor 3 to stay idle as shown on the figure by the thick dotted line. Processor 2 is in the same situation. It can perform at most 2 updates during its β sub-phase, and it has time to do so before the last message it expects arrives. As shown on the figure, processor 2 stays idle between the end of its second update and the arrival of the last message. Processor 1, however, is in a different situation. Indeed, it receives its last message while it is performing its second update. It then interrupts its execution, and puts an end to its β sub-phase.

As seen on the figure, the processors enter their $(k+1)^{th}$ phase in a different

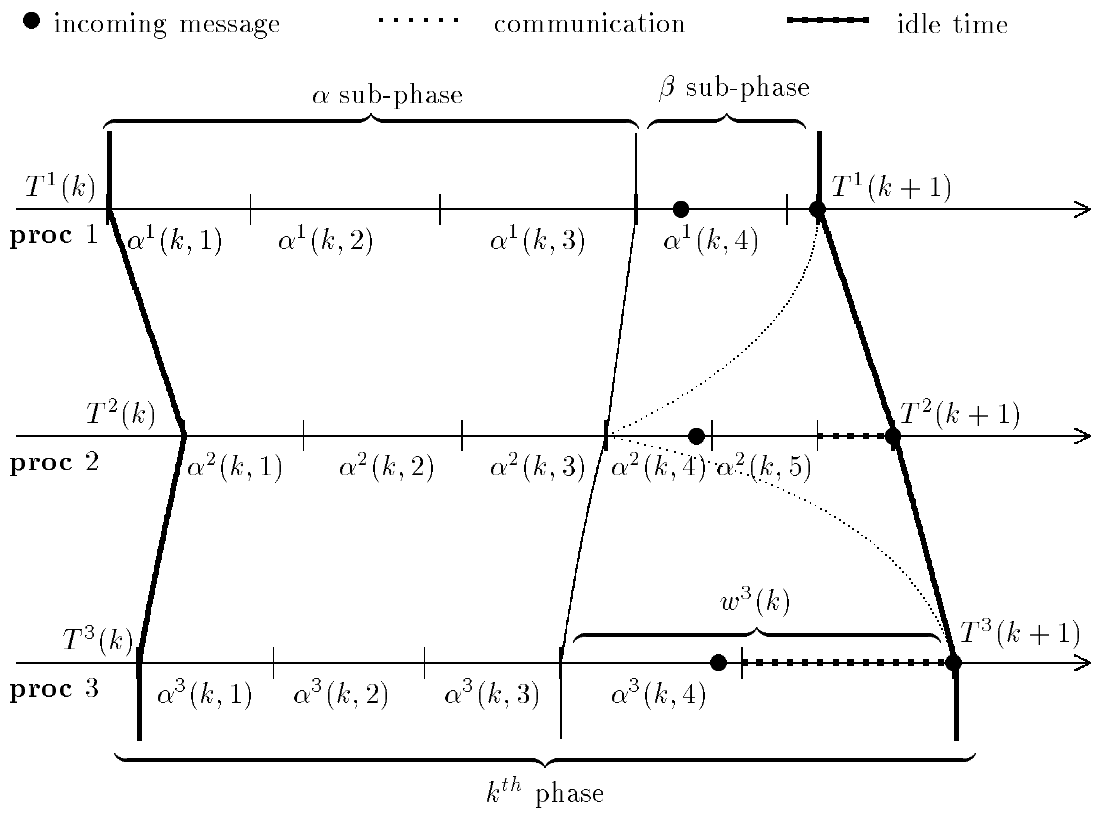

Figure 3.3: Example algorithm phase

order than the one with which they entered their k^{th} phase. For instance, processor 2 is the first one to enter the k^{th} phase but the last one to enter the $(k+1)^{th}$ phase. This example does not reflect all the possible scenarios. For instance, a processor can have an *empty* β sub-phase, or it is possible that no processor is able to perform any update of the solution vector during its β sub-phase.

3.4.4 Assumptions

In this section, we list all the assumptions made on the different components of our model.

Assumption 3.4.1 *The original RVs have a finite range, that is*

$$\forall i, k, \theta \quad \alpha^i(k,\theta) \in \{\underline{\alpha^i}, .., \overline{\alpha^i}\} \subset \mathbb{N},$$

$$\forall i, j, k \quad n_{i\to j}(k) \in \{\underline{n_{i\to j}}, .., \overline{n_{i\to j}}\} \subset \mathbb{N}.$$

From now on, the notations $\underline{x}$ and $\overline{x}$ design the smallest and largest value that can be taken by a RV x. Assumption 3.4.1 implies that solution vector updates and network communications are completed in a finite amount of time. In other words, there is no processor failure during the run of the algorithm, and no message is lost by the network.

Assumption 3.4.2

(i) *The RVs $\alpha^i(k,\theta)$ are independent and identically distributed.*

(ii) *The RVs $n_{i\to j}(k)$ are independent and identically distributed.*

This last assumption is in fact the entire basis of our stochastic model. Similar assumptions have been popular in the past for the purpose of generating tractable models [55, 2]. Some recent work [58] suggests that the network traffic in a distributed environment like the one that we are considering is such that it cannot be accurately modeled by independent and identically distributed (i.i.d.) RVs. However, in a model with no restrictions on the dependences and the distributions of the different RVs, it is usually very difficult or even impossible to obtain satisfactory and useful results. Without taking such an extreme approach, it would for instance be possible to model quantities such as the update time for a processor as a *Markov-modulated* random process [41] (e.g. Bernoulli or Poisson Markov Modulated process) to reflect processor workload fluctuation patterns. Or we could use *ON/OFF sources* [43, 22, 45, 31] or *Markov-modulated* arrivals [1] to model the network traffic and transmission rate. For now, we will consider that the i.i.d. case is a reasonably good approximation and we discuss this in more details in Chapter 6.

3.4.5 Model Discussion

Severals aspects of our model raise questions about the actual implementation of the parallel iterative algorithm. In this section, we answer the most crucial ones and state what assumptions we are making for the implementation, if any.

Broadcasting

We have already said that each processor, at the end of its α sub-phase *broadcasts* its piece of the solution vector to every other processor. In the model, we assume that when such a broadcast is performed:

- The sending processor spends 0 CPU time units initiating the broadcast.
- All the messages of a broadcast are sent at exactly the same time.

The first assumption is for the sake of simplicity. It would be very easy to add to the model a RV representing, for each processor and each phase, the number of CPU time units necessary to initiate the broadcast. However, in practice this RV should be small, when compared to the update time, or even the network time. We ignore it and set to 0 the number of CPU cycles spent by the sending processor to initiate a broadcast. Besides, from the point of view of the receiving processor, this additional time spent by the sender can easily be incorporated in the distributions of the $n_{i \to j}(k)$ RVs.

The second assumption is justified basically by the same considerations. Since the sender spends 0 CPU time units to initiate a broadcast, it is natural to assume that all the messages are put on the network at the same time. This is not technically true, since usually a broadcast is a succession of *sends* at the application level. If such a subtle distinction were to be taken into account by our model, one could always incorporate it again in the distributions of the $n_{i \to j}(k)$ RVs.

Receiving Messages

We have already mentioned that during its β sub-phase, a processor interrupts its execution when it receives the last message it was expecting. This may be difficult to implement exactly in the algorithm. A way to achieve this would be to use mechanisms like *active messages* [21, 20, 56]. When a message arrives at its destination, the receiving processor's execution is interrupted and a handler is called. This handler processes the message. If no more messages are to be received, then the ongoing β sub-phase is not resumed. However, active message facilities are not always available to the user. Another possibility is to have each process *periodically* check for messages received while it is in a β sub-phase. For instance, such a check can be performed each time the processor is about to update a single component of its piece of the solution vector. Or if necessary, this check can be performed even more often, several times during a single update. No matter how often this check is performed, a processor can not interrupt its β sub-phase *exactly* when the last message it was expecting arrives. We do not make any assumption on the implementation regarding this point. If the implementation uses active messaging, then the model fits exactly what happens during the β sub-phase. If the implementation performs periodical checks, then the model is an approximation, and depending on the frequency of these checks, this approximation is more or less accurate. For a "reasonable" implementation which performs "enough" checks, we expect our model to be a very good approximation.

Local Blocking

If the implementation of the iterative algorithm runs on p processors, the solution vector is generally of dimension greater than p. If the dimension of the solution vector is much greater than the number of available processors, then each processor is in charge of a relatively big piece of the solution vector. It may be interesting, convergence-wise, to perform more network communications than in our execution model. For instance, each processor could send to every other processor each half of its updated solution vector piece as soon as it is completed. More generally, each piece could be divided into *blocks*. Each time a processor updates a block, it is broadcasted to every other processor. This would minimize the number of out-of-date data used in the updates, and thereby improve the algorithm convergence in number of iterations. Choosing a block size smaller than the piece size is not a real issue in a shared-memory implementation of a parallel iterative algorithm, but can be a major issue in distributed memory settings. Of course, depending on the computational complexity of a component update, too small a block size can overflow the network and degrade the overall performance of the implementation.

In this work, we take the block size equal to the piece size, for the sake of simplicity. Supporting different block sizes would not require many modifications to the model.

3.5 Underlying Markov Chain

Now that the base of the stochastic model is established, we define a simple way to describe the evolution of the algorithm throughout time. This is done by associating a Markov chain to the algorithm; the current state of the Markov chain is related to the current *state* of the algorithm, as shown in the following sections.

3.5.1 Basic Definitions and Equations

The main idea is to represent the state of the algorithm in a phase k by the relative times at which every processor entered that phase. On Figure 3.2, those times are symbolized by points on the time axis. The processors enter the k^{th} phase of the algorithm at times $T^1(k)$, $T^2(k)$, .., $T^p(k)$, and they enter the next phase at times $T^1(k+1)$, $T^2(k+1)$, .., $T^p(k+1)$, according to definition 3.4.3**(i)**. On Figure 3.2, the start of a new phase is represented with a thick line joining the entry points of each processor in that phase. From now on, we call this line the *wavefront* and its *shape* for the current phase determines the current algorithm state. More formally:

Definition 3.5.1 *Wavefront definition*

$$X(k) = (0, T^2(k) - T^1(k), .., T^p(k) - T^1(k)) \in \mathbb{R}^p$$

is the wavefront for each algorithm phase k.

In this definition, the first processor is taken as a point of reference. The components of $X(k)$ are the times of the entrances of the other processors in the k^{th} phase, when $T^1(k)$ is taken as the time origin. Several questions about this wavefront are answered in the following sections.

3.5.2 Finite Space

In this section, we show that the wavefront vector, $X(t)$, can only take a finite number of values in $\mathbb{R}^p$. This is done by proving the following two lemmas.

Lemma 3.5.1 The vector $X(k)$ is bounded for $k \geq 1$:

$$\exists M, \forall k \geq 1 \quad \|X(k)\|_\infty \leq M \qquad \text{with } M = \max_{h,j \in \{1,..,p\}} (\mu^h \overline{n_{h\to j}}).$$

Proof. Recall that processor 1 is the reference processor. Consider a particular phase of the algorithm, say k, and a given processor, say i. We assume that $i \neq 1$, since $X_1(k) = 0$ where $X_1(k)$ denotes the first component of vector $X(k)$, and since the case where there is only one processor is of little interest. After performing their respective solution vector updates during their α sub-phases, both processors 1 and i are waiting for p incoming messages. When those messages are received, then both 1 and i can proceed to phase $k+1$.

The times at which processors 1 and i receive a message from some processor h are apart by $\mu^h|n_{h\to 1}(k) - n_{h\to i}(k)|$. Indeed, recall that we assume that when a processor broadcasts its message, all the messages are sent at the exact same time. Therefore, the times at which processors 1 and i receive the last messages they were waiting for are apart by at most $\max_{h\in\{1,..,p\}}(\mu^h|n_{h\to 1}(k) - n_{h\to i}(k)|)$.

Using assumption 3.4.1, one can write that:

$$\begin{aligned}\forall h \quad |n_{h\to 1}(k) - n_{h\to i}(k)| &\leq \max(\overline{n_{h\to 1}} - \underline{n_{h\to i}}, \overline{n_{h\to i}} - \underline{n_{h\to 1}}) \\ &\leq \max(\overline{n_{h\to 1}}, \overline{n_{h\to i}}) \\ &\leq \max_{j\in\{1,..,p\}}(\overline{n_{h\to j}}).\end{aligned}$$

Therefore, the times at which processors 1 and i receive their last message are apart by at most $\max_{h,j\in\{1,..,p\}}(\mu^h\overline{n_{h\to j}})$. By definition 3.4.3**(i)**, those times are apart by $|T^i(k+1) - T^1(k+1)| = |X_i(k)|$, implying that:

$$|X_i(k+1)| \leq \max_{h,j\in\{1,..,p\}}(\mu^h\overline{n_{h\to j}}) = M.$$

This is true for every processor i, and since $\|X(k+1)\|_\infty \equiv \max_{i\in\{1,..,p\}} |X_i(k+1)|$, the proof is completed. ∎

We now know that vector $X(k)$ is in a bounded sub-set of R^p, for each $k \geq 1$. The case $k = 0$ corresponds to the entrance of the processors in the first algorithm phase. We do not impose any constraint on how this entrance is done. The processors can enter the algorithm at any time the implementation and the operating system impose. Therefore, $X(0)$ may very well be such that $\|X(0)\|_\infty > M$. The preceding lemma implies that no matter how big $\|X(0)\|_\infty$, the following wavefronts will all be bounded by M.

We now prove that this bounded subset of $\mathbb{R}^p$ in which $X(k)$ resides is also finite. Let us first rewrite equation 3.4.1 in terms of $X(k)$. By subtracting from equation 3.4.1 the special case of $i = 1$, one obtains:

$$X_i(k+1) = X_i(k) + \mu^i \alpha^i(k) + w^i(k) - w^1(k) - \mu^1 \alpha^1(k).$$

Similarly, one can rewrite equation 3.4.2 as:

$$\begin{aligned} w^i(k) = &\max[0, \max_{j \in \{1,..,p\}-\{i\}} (X_j(k) + \mu^j \alpha^j(k) \\ &- X_i(k) - \mu^i \alpha^i(k) + \mu^j n_{j \to i}(k))]. \end{aligned}$$

These two equations imply that for all k and i, $X_i(k)$ can be written as a linear combination:

$$X_i(k) = \sum_{l=1}^{p} \lambda_{k,i,l} \mu^l + \sum_{l=1}^{p} \nu_{k,i,l} X_l(0) \quad \text{where} \quad \lambda_{k,i,l}, \nu_{k,i,l} \in \mathbb{Z}. \tag{3.5.3}$$

To prove that $X(t)$ is in a finite subset of $\mathbb{R}^p$, we need to following technical assumption:

Assumption 3.5.1 *The durations of the CPU time units and the components of* *the initial wavefront are rational numbers, that is:*

$$\forall i \in \{1, .., p\} \quad X_i(0) \in \mathbb{Q},$$

$$\forall i \in \{1, .., p\} \quad \mu^i \in \mathbb{Q}.$$

If assumption 3.5.1 is satisfied, then equation 3.5.3 implies that $X_i(k) \in \mathbb{Q}$ for all processors i and all phases k. This means that $X_i(k)$ can be written as $\mu_{i,k}/\delta$ where δ is the common denominator of the μ^l's and the $X_l(0)$'s. Using lemma 3.5.1, on obtains

$$\forall i, k \quad \mu_{i,k} \in \mathbb{N} \cap [-\delta M, +\delta M],$$

which is a finite set. Therefore, δ being fixed, $X_i(k)$ is in a finite set. Since this is true for all i, we have proved the following lemma:

Lemma 3.5.2 The vectors $X(k)$ for $k \geq 1$ reside in a **finite** subset of $\mathbb{R}^p$. This subset is at most of cardinality $(2M+1)^{p-1}$. We denote that subset by $\mathcal{X} = \mathcal{X}_1 \times ... \times X_p$ ($\mathcal{X}_1 = \{0\}$).

This upper bound on the cardinality of the subset is immediate, since the wavefront is determined by $p-1$ components, and each component can take at most $2M+1$ values. This bound is not expected to be tight, and in fact experiments in Chapter 4 we show that it can be fairly loose for some models. ∎

Assumption 3.5.1 is really purely technical. It is always valid in a *real-life* experiment, where all the data being manipulated is in $\mathbb{Q}$ since it is processed by computers with finite arithmetic.

3.5.3 The Wavefront as a Markov Chain

Let us recall that

$$X_i(k+1) = X_i(k) + \mu^i \alpha^i(k) + w^i(k) - w^1(k) - \mu^1 \alpha^1(k),$$

and

$$\begin{aligned} w^i(k) = & \max[0, \max_{j \in \{1,..,p\}-\{i\}} (X_j(k) + \mu^j \alpha^j(k) \\ & - X_i(k) - \mu^i \alpha^i(k) + \mu^j n_{j \to i}(k))]. \end{aligned}$$

Using the facts that $X_1(k) = 0$ and that

$$\forall x, y \in \mathbb{R} \qquad \max(0, x-y) + y = \max(x, y),$$

one can then write

$$\begin{aligned} X_i(k+1) = & \max[X_i(k) + \mu^i \alpha^i(k), \max_{j \in \{1,..,p\}-\{i\}} (X_j(k) + \mu^j \alpha^j(k) + \mu^j n_{j \to i}(k))] - \\ & \max[X_1(k) + \mu^1 \alpha^1(k), \max_{j \in \{1,..,p\}-\{1\}} (X_j(k) + \mu^j \alpha^j(k) + \mu^j n_{j \to 1}(k))]. \end{aligned}$$

Since $n_{i\rightarrow i}(k) = 0$ for all i, the equation above can be rewritten as

$$\boxed{\begin{aligned}\forall i \quad X_i(k+1) = &\max_{j\in\{1,..,p\}}[X_j(k) + \mu^j\alpha^j(k) + \mu^j n_{j\rightarrow i}(k)] - \\ &\max_{j\in\{1,..,p\}}[X_j(k) + \mu^j\alpha^j(k) + \mu^j n_{j\rightarrow 1}(k)].\end{aligned}} \tag{3.5.4}$$

This last equation is fundamental. We call it the *wavefront equation.* It shows that the wavefront at the entrance of the k^{th} phase depends only on the wavefront at the entrance of the previous phase and on the realizations of the RVs $\alpha^i(k,\theta)$ and $n_{i\rightarrow j}(k)$ for all i,j and θ. Thanks to assumption 3.4.2, we can now state that $X(k)$ is a **Markov variable**. Equation 3.5.4 can be used to compute the transition probabilities of the finite-state, time-homogeneous Markov chain associated to $X(k)$.

3.6 Random Variables of Interest

Now that we have extracted the Markov variable $X(k)$ as the driving RV for the algorithm execution, we can further define other RVs of interest that will help us quantify the overall performance of the implementation.

3.6.1 The Waiting Time

This RV is $w^i(k)$. It measures the durations of the β sub-phases of processor i in seconds. We call it *waiting time* since it corresponds to the time a processor has to *wait* for all the messages coming from the other processors. We recall here the expression for the waiting time:

$$\boxed{\begin{aligned} w^i(k) =& \max[0, \max_{j\in\{1,..,p\}-\{i\}} (X_j(k) + \mu^j \alpha^j(k) \\ & - X_i(k) - \mu^i \alpha^i(k) + \mu^j n_{j\to i}(k))]. \end{aligned}} \tag{3.6.5}$$

Thanks to this equation, one can compute the conditional distribution of $w^i(k)$, conditioned on the current wavefront, $X(k)$. In this section, we compute bounds on the value that can be taken by the waiting time. From the equation above, clearly, $w^i(k) \geq 0$. Let us now find an upper bound. Let j be another processor, and let us compute the maximum time that processor i can wait for the message from processor j. Processor j can enter the k^{th} phase at most $2M$ seconds after processor i, where M is from lemma 3.5.1. Then, processor i can finish its α sub-phase in $A_i\mu^i\underline{\alpha^i}$ seconds at the least, whereas processor j can spend up to $A_j\mu^j\overline{\alpha^j}$ in its α sub-phase. Finally, the message from processor j can take at most $\mu^j\overline{n_{j\to i}}$ seconds. Therefore, the maximum amount of time that processor i in its β sub-phase can wait for a message from processor j, is given by $w_{j\to i}$ with:

$$w_{j\to i} = 2M - A_i\mu^i\underline{\alpha^i} + A_j\mu^j\overline{\alpha^j} + \mu^j\overline{n_{j\to i}}.$$

Therefore, $w^i(k)$ is bounded above as:

$$\begin{aligned} w^i(k) &\leq \max_{j\in\{1,..,p\}}(2M - A_i\mu^i\underline{\alpha^i} + A_j\mu^j\overline{\alpha^j} + \mu^j\overline{n_{j\to i}}) \\ &= 2M - A_i\mu^i\underline{\alpha^i} + \max_{j\in\{1,..,p\}}(A_j\mu^j\overline{\alpha^j} + \mu^j\overline{n_{j\to i}}) \\ &= 2\max_{h,j\in\{1,..,p\}}(\mu^h\overline{n_{h\to j}}) - A_i\mu^i\underline{\alpha^i} + \max_{j\in\{1,..,p\}}(A_j\mu^j\overline{\alpha^j} + \mu^j\overline{n_{j\to i}}). \end{aligned}$$

We have therefore proved the following lemma:

Lemma 3.6.1 Bounds on the waiting time:

$$\forall i, k \quad 0 \leq w^i(k) \leq \overline{w^i},$$

with

$$\overline{w^i} = 2\max_{h,j\in\{1,..,p\}}(\mu^h\overline{n_{h\to j}}) - A_i\mu^i\underline{\alpha^i} + \max_{j\in\{1,..,p\}}(A_j\mu^j\overline{\alpha^j} + \mu^j\overline{n_{j\to i}}).$$

∎

3.6.2 The Phase Time

Another interesting RV for the analysis of the performance of the algorithm is the *phase time*, the duration of an algorithm phase for a processor i in seconds. Let $\Phi^i(k)$ denote the duration of the k^{th} algorithm phase on processor i. $\Phi^i(k)$ is given by the following equation:

$$\boxed{\Phi^i(k) = \mu^i \alpha^i(k) + w^i(k).} \tag{3.6.6}$$

Thanks to equation 3.6.6 and 3.6.5, one can compute the conditional probability distribution of $\Phi^i(k)$, conditioned on the current wavefront, $X(k)$. The following lemma, giving lower and upper bounds on the phase time, is immediate from lemma 3.6.1:

Lemma 3.6.2 Bounds on the phase time:

$$\forall i, k \quad A_i \underline{\alpha^i} \leq \Phi^i(k) \leq A_i \overline{\alpha^i} + \overline{w^i}.$$

∎

3.6.3 The Number of Additional Updates

The last RV introduced here is the number of updates performed during a β sub-phase on a processor i. Let $N^i(k)$ denote the number of solution vector updates performed by processor i during the β sub-phase of the k^{th} algorithm phase. Given the conditional distribution of the waiting time, one can also compute the conditional distribution of $N^i(k)$ conditioned on the current wavefront.

We already know, by definition of B_i that:

$$\forall k, i \quad 0 \leq N^i(k) \leq B_i.$$

On the other hand, the maximum number of updates that can be performed during the β sub-phase of the k^{th} algorithm phase on processor i is equal to $w^i(k)/(\mu^i \underline{\alpha^i})$, since an update takes at least $\underline{\alpha^i}$ CPU time units of processor i. Therefore

$$N^i(k) \leq \min(B_i, \frac{w^i(k)}{\mu^i \underline{\alpha^i}}).$$

Using the definition of $\overline{w^i}$, we have thus proved the following lemma:

Lemma 3.6.3 Bounds on the number of additional updates:

$$\forall i, k \quad 0 \leq N^i(k) \leq \min(B_i, \frac{\overline{w^i}}{\mu^i \overline{\alpha^i}}).$$

∎

3.7 Conclusion

In this chapter, we have designed our stochastic model and we have described how we model the distributed environment and the algorithm implementation in that environment. After stating our assumptions, we were able to extract a Markov chain from the model: the wavefront. This Markov chain has and will be used to describe the evolution of the algorithm throughout time. We then introduced three other RVs, the *waiting time*, the *phase time* and the *additional number of updates*. The conditional distributions of these RVs, conditioned on the current wavefront, can be computed thanks to the equations we have given. The next chapter describes how to effectively compute these distributions and use them to obtain stochastic performance measures of the algorithm implementation.

Chapter 4

New Performance Estimates

This chapter explains how Baudet's theory described in Section 2.3.1 and the model defined in Chapter 3 can be combined to obtain performance results about the convergence of parallel iterative algorithms in distributed environments. First, we show how it is possible to compute the wavefront Markov chain defined in Section 3.5 and give a few illustrative examples. Second, we extend Baudet's results to obtain new estimates of the convergence rate of the algorithm (in terms of number of iterations to convergence). Third, we propose an estimate of the implementation speed (in terms of number of iterations performed per second). We then introduce the *Large Deviation Theory* and explain how it can be used to gain even more insight into the estimation of the implementation speed. Finally, we summarize the results of the chapter in describing the general method of performance characterization and conclude with an example.

4.1 Computing the Wavefront

As we have already stated in Chapter 3, the wavefront Markov chain is the driving random process behind the iterative algorithm execution. In this section, we compute the transition probability matrix of that chain, as well as its stationary distribution.

4.1.1 Transition Matrix and Stationary Distribution

To compute the transition matrix P_X of the wavefront, one needs to compute the following probabilities:

$$P_X(x,y) = \mathcal{P}\{X(k+1) = y|X(k) = x\} \quad \forall x, y \in \mathcal{X},$$

where $\mathcal{X}$ is the subset of $\mathbb{R}^p$ defined in lemma 3.5.2 and $\mathcal{P}$ is the probability measure. Thanks to equation 3.5.4, one can compute the probability:

$$\mathcal{P}\{X_i(k+1) = z|X(k) = x\} \quad \forall i \in \{1,..,p\}, z \in \mathcal{X}_i, x \in \mathcal{X},$$

where $\mathcal{X}_i$, $i = 1, ..., p$, is also defined in lemma 3.5.2. There are two ways to exploit equation 3.5.4 to compute the conditional probabilities. One can try to formally derive from that equation the conditional probabilities $P_X(x,y)$ by using standard results on independent and identically distributed RVs. Once these distributions are derived, a program can be written to compute actual numerical values. There are two major drawbacks to such an approach. First, the actual formal derivation of the distributions leads to very intricate discrete computations, and the resulting expressions are extremely complicated due to RV dependences. Second, an actual implementation of those equations is very inefficient and leads to prohibitive execution times as shown by some of our early attempts. Another possibility is to write a program that exhaustively enumerates all the possible observations of the original RVs in equation 3.5.4. Maybe surprisingly, such an implementation is reasonably efficient provided the distributions of the original RVs are of reasonable sizes.

In order to compute the matrix P_X we have written a parallel program in C using an implementation of the Message Passing Interface (MPI) [49]. The design of the program is relatively simple in terms of parallelism and involves very few inter-process communications. It is therefore extremely scalable and we have been able to distribute the computation of the transition matrix P_X on many workstations, when necessary, to obtain quick results. Once P_X is available, it is possible to compute the stationary distribution of the Markov chain. This distribution is represented by the left eigenvector π associated to eigenvalue 1 of matrix P_X: $\pi P_X = \pi$ (see [30, 12, 4] for instance). The stationary distribution of a Markov chain is interpreted as the long term state occupancy rates of that chain. In the experiments that we have conducted, the size of P_X is not extremely large.

In fact, we were able to use direct methods for dense matrices to compute the π vector. Our implementation used the LAPACK [3] package. For larger models leading to larger transition matrices, one might have to use parallel numerical libraries [13]. As we will see on a few examples in the next section, the structure and properties of P_X depend on the actual model.

4.1.2 Examples of the Wavefront Behavior

We give here three examples of wavefront Markov chains for three different distributed environments. In all these cases, $A_i = 1$ for all $i = 1, .., p$ (see definition 3.4.1).

Example 1: Homogeneous Case

We consider here a distributed environment consisting of 5 identical processors. Furthermore, we assume that the workload distributions on those processors are identical with small variance. This distribution is plotted on Figure 4.1. The CPU time unit of each processor is taken to be equal to 1. The network is assumed to deliver constant performance of 1 CPU time unit for each message from any processor to any other. This model is very idealistic but has the advantage of producing results that can be easily interpreted.

With the notation of lemma 3.5.1, $M = \max_{h,j\in\{1,..,p\}}(\mu^h \overline{n_{h\to j}}) = 1$, meaning that the wavefront $X(k)$ is in a subset of $\mathbb{R}^m$ of cardinality $(2 \times 1 + 1)^{5-1} =$

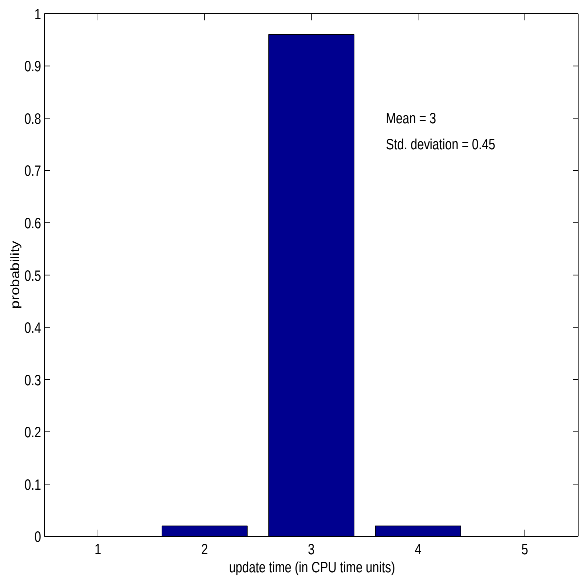

Figure 4.1: Workload distribution for example 1

81 (see lemma 3.5.2). In other words, there are at most 81 different shapes of the wavefront that can be observed. This does not mean that each of these 81 possibilities will occur with zero probability, especially in such a simplistic model. In our state numbering scheme, state #41 is called the *middle state* and it corresponds to a “flat” wavefront, the one for which $X(k) = (0, ..., 0)$.

Figure 4.2 shows the portrait of matrix P_X. The non-null entries of the matrix are represented by black dots. One can immediately see that P_X is extremely sparse: most of its columns are empty, meaning that most of the 81 states are unreachable. The Markov chain is reduced by removing those unreachable states to obtain a smaller recurrent chain with a much less sparse transition matrix P_X^*. There are only 6 non-empty columns in P_X, implying that P_X^* is a 6×6 matrix. In

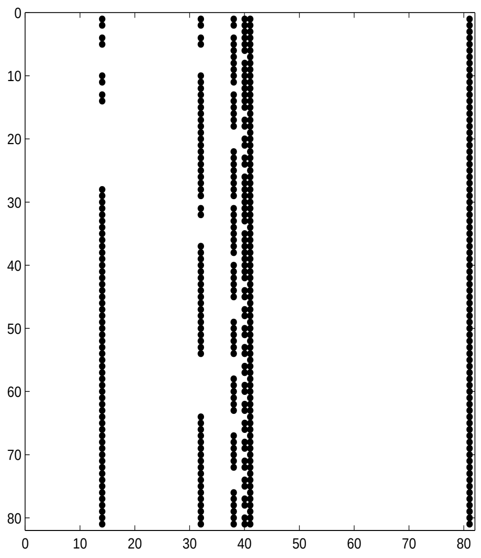

Figure 4.2: Portrait of the wavefront transition matrix P_X for the first example

this example, the states that are part of the reduced Markov chain are the states numbered 14, 32, 38, 40, 41 and 81.

The stationary distribution of the wavefront, computed from the matrix P_X^* thanks to dense eigensolvers, is depicted in Figure 4.3. The top part of the figure shows the actual distribution (or π vector). One can see that the middle state (#41) is the one with the highest π-*value*, with $\pi_{41} \approx 0.91$. The remaining 5 states have much lower π-values: all equal and ≈ 0.02. One can then conclude that in the long run, the wavefront is found with high probability to be flat. The bottom part of Figure 4.3 shows the actual shape of the wavefront corresponding to each of the 6 states. The line thickness indicates the magnitude of the corresponding π-value. On can see that, as explained above, state #41 corresponds to a flat

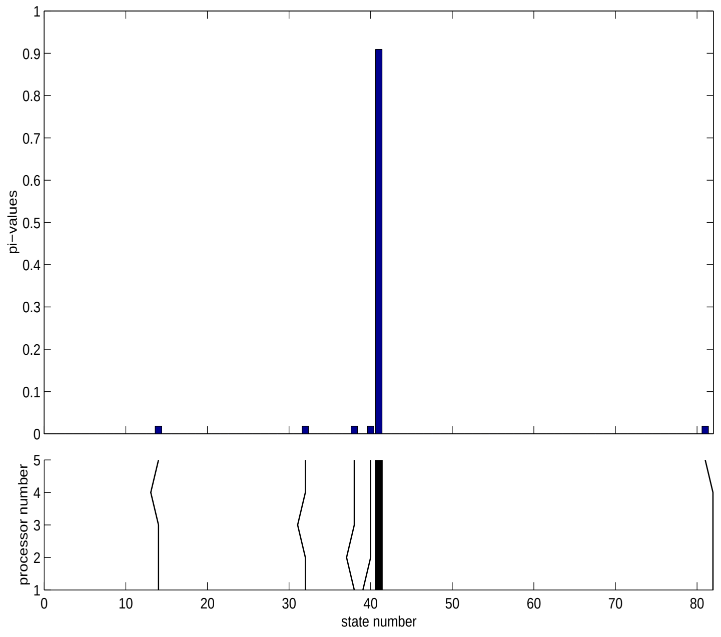

Figure 4.3: Stationary distribution and corresponding wavefront shapes for example 1

wavefront and it is shown with a very thick line since this state dominates all the others. The other five states are also represented on the bottom part of the figure (in thin lines).

There are many observations that can be made about Figure 4.3. First, the wavefront is "usually" in state #41 throughout the algorithm execution. This was expected in such an homogeneous environment where the processors are identical and with identical low-variance workload distributions. Second, besides state #41, every state corresponds to one processor entering an algorithm phase before all the others. The fact that these 5 states have the same π-values is easily understood because of the homogeneity of the system. The fact that these states correspond to basically the same situation can also be explained. Let us consider the 5 processors in the k^{th} algorithm phase. Let us assume that one processor, say i, finishes its α sub-phase after all the others. Because of the model assumptions, processor i has already received all the messages it was expecting (all the CPU time units are taken to be 1); it can start its $(k+1)^{th}$ phase immediately after the end of its α sub-phase and never enter a β sub-phase. All the other processors have to wait for 1 CPU time unit for the message from processor i. The five states numbered 14, 32, 38, 40 and 81 correspond to the cases where $i = 4, 3, 2, 1, 5$ as seen on Figure 4.3. Those states have small π-values because the workload distributions are identical and with small-variance so that a processor is "rarely" slower than all the others during a particular algorithm phase. One can compute the entropy, H, of the stationary distribution as an uncertainty measure (see [4]):

$$0 \leq H = - \sum_{l \in \{14,32,38,40,41,81\}} \pi_l \log_2(\pi_l) \sim 0.6494 \leq \log_2 6 \sim 2.5850.$$

Example 2: Mildly Heterogeneous Case

We now perturb the system of example 1 by making processor 1 slower on average than the other four. The workload distributions are represented on Figure 4.4. The matrix P_x has roughly the same structure as before, and the wavefront can be reduced to a 6×6 chain in the same way. Figure 4.5 is similar in intent to Figure 4.3 and shows the stationary distribution. Let us examine the differences between the two figures.

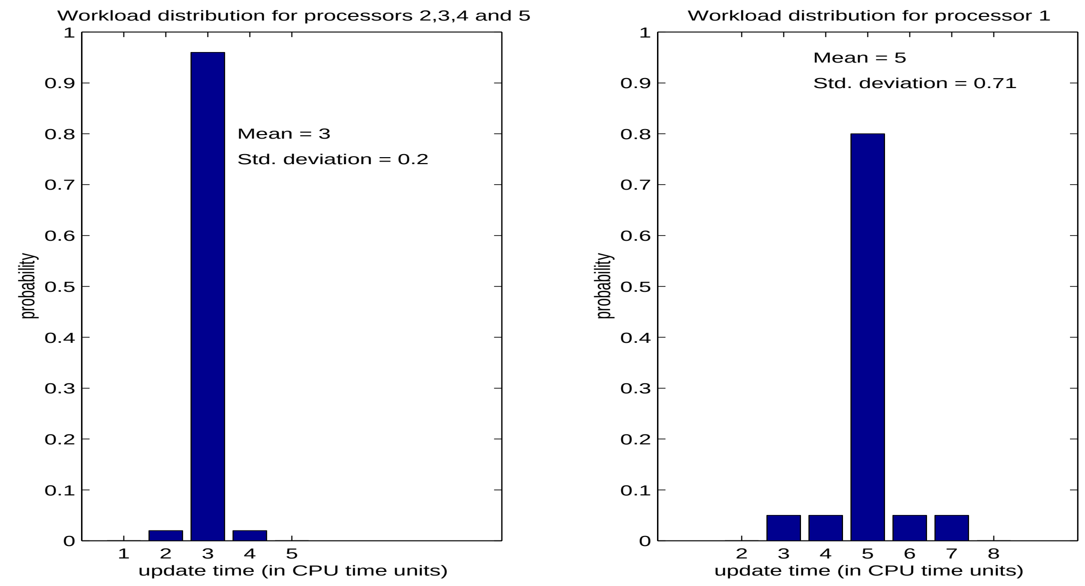

Figure 4.4: Workload distributions for example 2

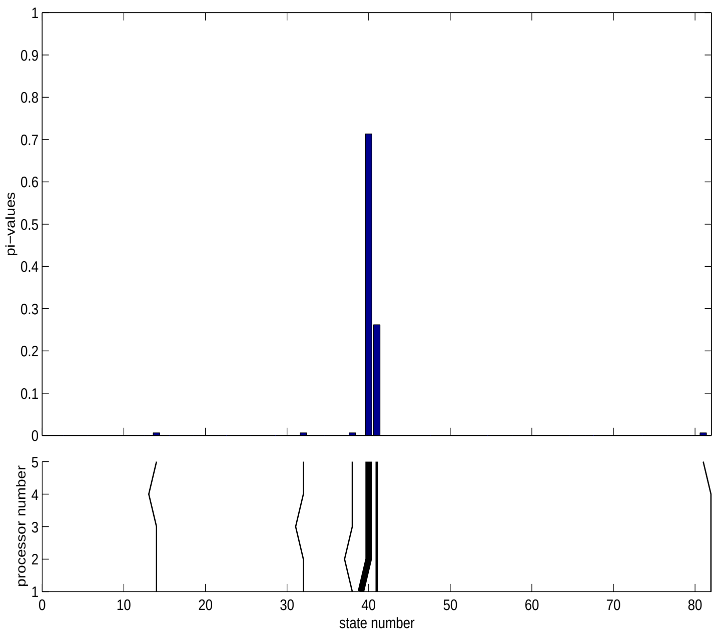

Figure 4.5: Stationary distribution and corresponding wavefront shapes for example 2

First, instead of one state dominating the other 5, there are two states that dominate the π distribution. The most likely state corresponds to processor 1 entering an algorithm phase first (state #40 with $\pi_{40} \approx 0.71$). This is consistent with the model since processor 1 is slower on average than every other one (cf. explanation in example 1). The other state that has a relatively large value is state #41 (with $\pi_{41} \approx 0.26$): this is the middle state. The other four states have very low π-values, all approximatively equal to 0.006.

The fact that state #41 still has a relatively high π-value is due to the workload distributions. Indeed, in this example, even though processor 1 is slower than all the others on average, there is still a fairly high probability that it is not slower on a particular observation, causing all the processors to be synchronized and the wavefront to be in state #41. If we had chosen the workload distribution of processor 1 such that it would have been always slower than the other, then Figure 4.5 would have been similar to Figure 4.3 with state #40 playing the role of state #41.

As mentioned earlier, our model generates the matrix P_X^*, allowing us to draw the diagram on Figure 4.6. On that diagram, one can observe how the wavefront behaves. The transitions between the states are represented with arrowed arcs and the thicker the arc, the larger the transition probability. One can see that all arcs leading to state #40 are corresponding to transition probabilities ≈ 0.94. This, of course, explains why π_{40} is large. Once the chain is in state #40, it stays in that

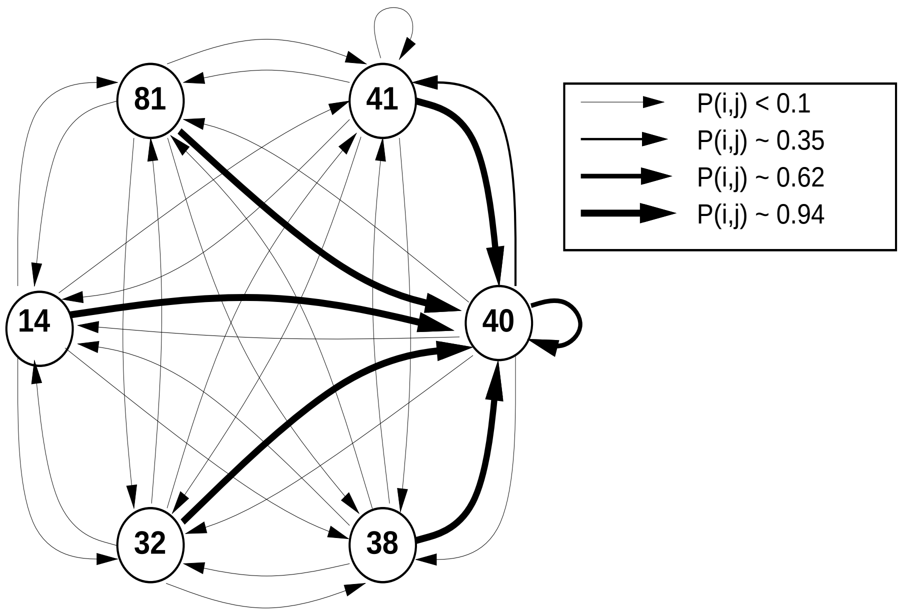

Figure 4.6: Wavefront Markov chain diagram for example 2

state with probability ≈ 0.62, goes to state #41 with probability ≈ 0.35 or goes to another state with probability ≈ 0.03. This causes π_{41} to have a relatively high value. Here again, one can compute the entropy, H, of the stationary distribution as an uncertainty measure:

$$0 \leq H = -\sum_{l \in \{14,32,38,40,41,81\}} \pi_l \log_2(\pi_l) \sim 1.0371 \leq \log_2 6 \sim 2.5850$$

This number reflects the fact that, in the long run, the state of the Markov chain is more *uncertain* than in example 1.

Example 3: Heavily Heterogeneous Case

Our third example is a heterogeneous system consisting of only 3 processors, to keep the figures simple. All the processors have different workload distributions. The network has the same distribution for every communication but, by contrast with the previous two examples, this distribution has a non-null variance. Figure 4.7 shows the different probability distributions.

Figure 4.8 shows the stationary distribution of the wavefront with the corresponding wavefront shapes. For this example, matrix P_X was 49×49, and matrix P_X^* is 34×34. By contrast with examples 1 and 2, no state strongly dominates the others in the stationary distribution. In fact, it is fairly difficult to interpret the figure in terms of a typical behavior of the wavefront.

4.1.3 Discussion

We give here some general directions in interpreting the structure and properties of P_X with reference to the experiments we have conducted.

The main observation is that the size of the wavefront Markov chain depends on the network model. We have already seen that the size of matrix P_X depends on the number of processors in the system and on the maximum time delay to send a message. But as stated in Chapter 3 and illustrated by the three previous examples, P_X can often be reduced to a smaller matrix P_X^* by eliminating all the unreachable states.

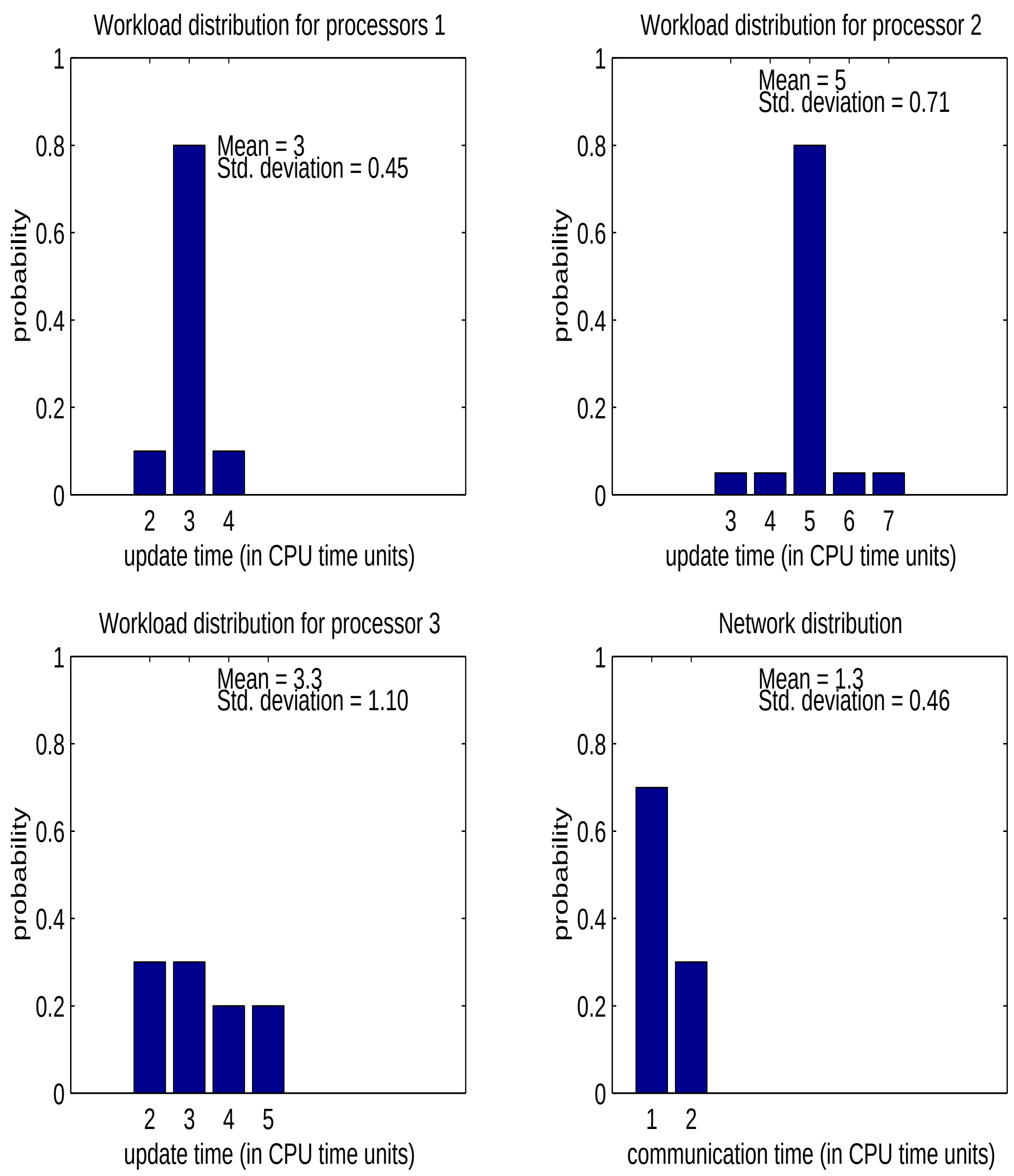

Figure 4.7: Workload and Network distributions for example 3

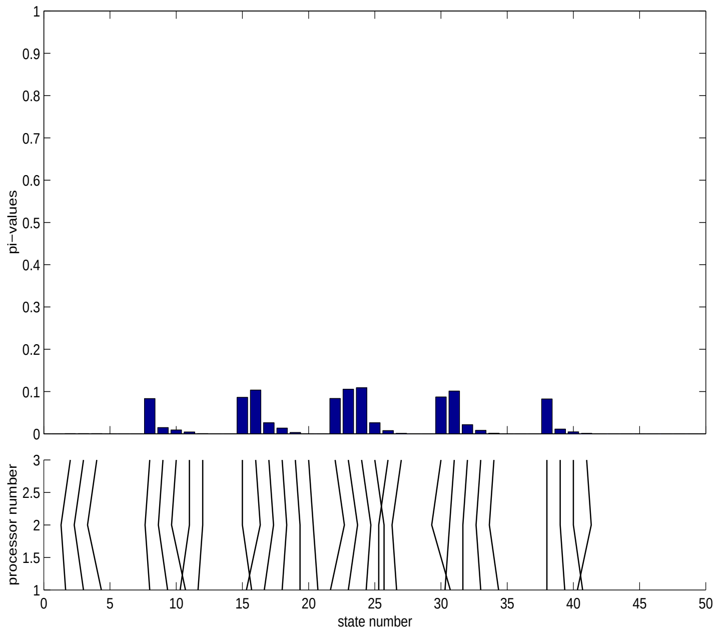

Figure 4.8: Stationary distribution and corresponding wavefront shapes for example 3

The relative sizes of those two matrices seem to depend on the variance of the network time distributions. This phenomenon can be seen already in the examples. In example 1, where the network times are distributed with zero variance, the sizes of the two matrices are such that $\dim(P_X^*)/\dim(P_X) = 0.0741$ where $\dim(A)$ is the number of lines (or columns) of a square matrix A. In example 3, however, $\dim(P_X^*)/\dim(P_X) = 0.6939$. Our other experiments seem to show that this ratio tends to 1 when the variance of the network times increases, meaning that P_X tends to being dense. Quantifying such a result could be an interesting thread to follow. The distribution of the actual shapes of the states in the reduced Markov chain depends on the workload distribution on the processors. This is seen in examples 1 and 2. The only difference between the models in those two examples is the workload distributions (explaining that the $\dim(P_X^*)/\dim(P_X)$ ratios are the same in the two examples). The impact of the differences in workload distributions has been illustrated on Figures 4.3 and 4.5 as differences in π-values, and similar observations have been possible in other experiments.

A reasonable hypothesis is that the structure of the Markov chain depends mostly on the network model, whereas the actual stationary distribution is mostly driven by the processor model. There are however strong interactions between the two models and the hypothesis may be difficult to confirm analytically. In our work, we are focused on designing new performance characterization techniques for parallel iterative algorithms in distributed environments. Even though the wavefront is the driving random process behind our stochastic model, it is too early here to try to completely analyze its behavior. In the rest of this research, we will use our implementation of the model to obtain the wavefront matrix P_X, and use that matrix to compute stochastic performance measures. Rather than an extremely precise analysis of the wavefront behavior, measuring the performance of the algorithm requires estimations of the time (in seconds) between two observations of the wavefront. Obtaining such estimations is the goal of the following sections.

4.2 Estimating the Convergence Rate

In this section, we estimate the convergence rate of the algorithm in terms of number of iterations to convergence by extending Baudet's work [5]. Then, we estimate the speed of the implementation of that algorithm in a given distributed environment in terms of number of iterations performed per time unit.

4.2.1 Preliminary Remark

Before using and extending Baudet's work, we need to give a strict definition of an *algorithm iteration*. We recall that in the formal definition of asynchronous implementation given by equation 2.2.3, a processor can choose not to perform an update during an iteration. In our model, a processor i performs $A_i + N^i(k)$ updates during the k^{th} algorithm phase. It is therefore possible (and most likely) that the p processors do not perform the same number of updates of the solution vector during a phase. According to our definition of asynchronism, the p processors all perform $N(k) = max_{i \in \{1,..,p\}}(A_i + N^i(k))$ iterations during a phase of the algorithm. Some processors might not perform any update during some of these iterations and this can be formalized by removing elements from the J_t sets of equation 2.2.3.

4.2.2 Three Estimates of the Convergence Rate

To estimate the convergence rate of the iterative algorithm, we use the part of Baudet's work that has been described in detail in 2.3.1. We recall that one of Baudet's main theorems states that:

$$\mathcal{R} \geq -[\liminf_{t\to\infty}(k_t/t)] \log \rho(A),$$

where $\rho(A)$ is the spectral radius of the matrix associated with the contracting operator Op, and $\{k_t\}$ is the sequence defined by equations 2.3.4 and 2.3.5. Equation 2.3.4 is a recursive definition of another integer sequence, $\{t_k\}$, in terms of a_k and b_k from definition 2.3.3. Let us compute the $\{t_k\}$ sequence in connection with our stochastic model.

In the previous section, we have seen that the processors perform $N(k)$ iterations per algorithm phase. According to our execution model, the components of the solution vector used in the updates corresponding to these iterations can not be out-of-date by more than $N(k)$ iterations. The first iteration following those $N(k)$ iterations updates all the components of the solution vector since $A_i \neq 0$ for all $i = 1, .., p$. Therefore, in terms of definition 2.3.3,

$$\forall k = 0, 1, ... \begin{cases} a_k & = N(k) \\ b_k & = 0. \end{cases}$$

One can now write the expression of k_t as:

$$k_t = \sup\{k | a_0 + b_0 + ... + a_{k-1} + b_{k-1} \leq t\}$$

$$= \sup\{k | N(0) + ... + N(k-1) \leq t\},$$

leading to:

$$k_t = \sup\{k | \sum_{l=0}^{k-1} max_{i \in \{1,..,p\}} (A_i + N^i(l)) \leq t\}, \tag{4.2.1}$$

which is a RV. The problem is now to compute a new lower bound on the asymptotic rate of convergence, $\mathcal{R}^*$, given by Baudet as:

$$\mathcal{R}^* \overset{\Delta}{=} -[\lim_{t \to \infty} \inf(k_t/t)] \log \rho(A).$$

The term k_t/t in the expression for $\mathcal{R}^*$ is a RV and, as such, makes it difficult to compute $\mathcal{R}^*$ directly. However, the implementation of our stochastic model can be used to compute the conditional probability $\mathcal{P}\{N(k) = x | X(k) = y)$ where x is an integer and y is in $\mathbb{R}^p$ (see Section 3.6.3); additionally, Section 4.1.1 explained how the stationary distribution of the wavefront, $X(k)$, can be computed. We are here interested in the *asymptotic* rate of convergence of the algorithm, and can therefore use the stationary distribution of the wavefront to compute a reasonable approximation of the unconditional probability $\mathcal{P}\{N(k) = x\}$ as:

$$\mathcal{P}\{N(k) = x\} = \sum_{l=1}^{\dim(P_X^*)} \pi_l \mathcal{P}\{N(k) = x | X(k) = X_l\}, \tag{4.2.2}$$

where:

- P_X^* is the transition matrix of the reduced wavefront as defined in Section 4.1.2.
- π_l is the π-value of the l^{th} state of the reduced wavefront.
- $X_l \in \mathbb{R}^p$ is the actual wavefront shape corresponding to the l^{th} state of the reduced wavefront.

Equation 4.2.2 is only an approximation since it uses the stationary distribution of the wavefront to convert a conditional probability into an unconditional one. Possibly better ways to estimate this rate of convergence are described in Chapter 6. At this point, we have access to an approximation of the actual distribution of the RV $N(k)$, and it does not depend on k. It is now possible to compute the distribution of k_t since:

$$k_t = \sup\{k | N(0) + ... + N(k-1) \leq t\}.$$

Our implementation of the model performs this computation and can generate the distribution for all $t = 0, 1, ...$

We have now reduced the problem to somehow computing

$$\liminf_{t\to\infty}(k_t/t),$$

where the distribution of each k_t is known. One might argue that this limit has no meaning in the strict sense of the definition of convergence for a real sequence, so we look here at asymptotic averages and bounds in order to compute $\mathcal{R}^*$. For each value of t, one can compute the minimum value that can be taken by k_t. Let $\underline{k_t}$ denote that value. One can also compute the maximum value that can be taken by k_t and we denote it by $\overline{k_t}$. Finally, one can compute the expectation of k_t and we denote it by $\widehat{k_t}$. It is then possible to compute three different estimates of $\mathcal{R}^*$:

$$\begin{cases} \underline{\mathcal{R}^*} & \triangleq -\left[\liminf_{t\to\infty}(\underline{k_t}/t)\right]\log\rho(A) \\ \widehat{\mathcal{R}^*} & \triangleq -\left[\liminf_{t\to\infty}(\widehat{k_t}/t)\right]\log\rho(A) \\ \overline{\mathcal{R}^*} & \triangleq -\left[\liminf_{t\to\infty}(\overline{k_t}/t)\right]\log\rho(A). \end{cases} \tag{4.2.3}$$

Those three estimates have different interpretations. $\underline{\mathcal{R}^*}$ is a worst case estimate and it is close, in concept, to Baudet's estimate. In fact, Baudet's estimate corresponds to an even worse case than our model because he does not account for any improvement due to long-run "averaging" over random events. In general, Baudet's worst case is not realistic, and our model shows that it happens with probability 0. In other words, Baudet gives a theoretical worst convergence rate given the A_i and B_i sequences whereas our worst case estimate takes also into account the distributed environment's behavior. $\overline{\mathcal{R}^*}$ is obviously a best case estimate. We call $\widehat{\mathcal{R}^*}$ an average estimate. At this point, the usefulness of such estimates is still unclear. It is even unclear that the limits in their expressions exist. In fact, the existence of $\underline{\mathcal{R}^*}$ is proven by Baudet's work. Proving the existence of the limits in $\overline{\mathcal{R}^*}$ and $\widehat{\mathcal{R}^*}$ is much more difficult and we leave it for future research. In all the experiments we have conducted however, the three estimates converge and we assume this convergence in all that follows.

4.2.3 An Example

Figure 4.9 shows the computation of our three estimates for a given distributed environment, a given algorithm and for different implementations. The distributed environment is the one of example 3 in Section 4.1.2. The algorithm corresponds to a contracting operator whose contracting matrix has a spectral radius $\rho = 0.9$. The different implementations are for different values of A_i and B_i for $i = 1, ..., p$. To make the figure simpler, we assume that those values are the same for all three processors. The figure shows six different graphs ((a) through (f)), corresponding to all the possible cases where $A_i = 1, 2$ and $B_i = 0, 1, 2$ for all i. Each graph contains four curves and these curves show different convergence rate estimates for increasing values of t (as in equation 4.2.3).

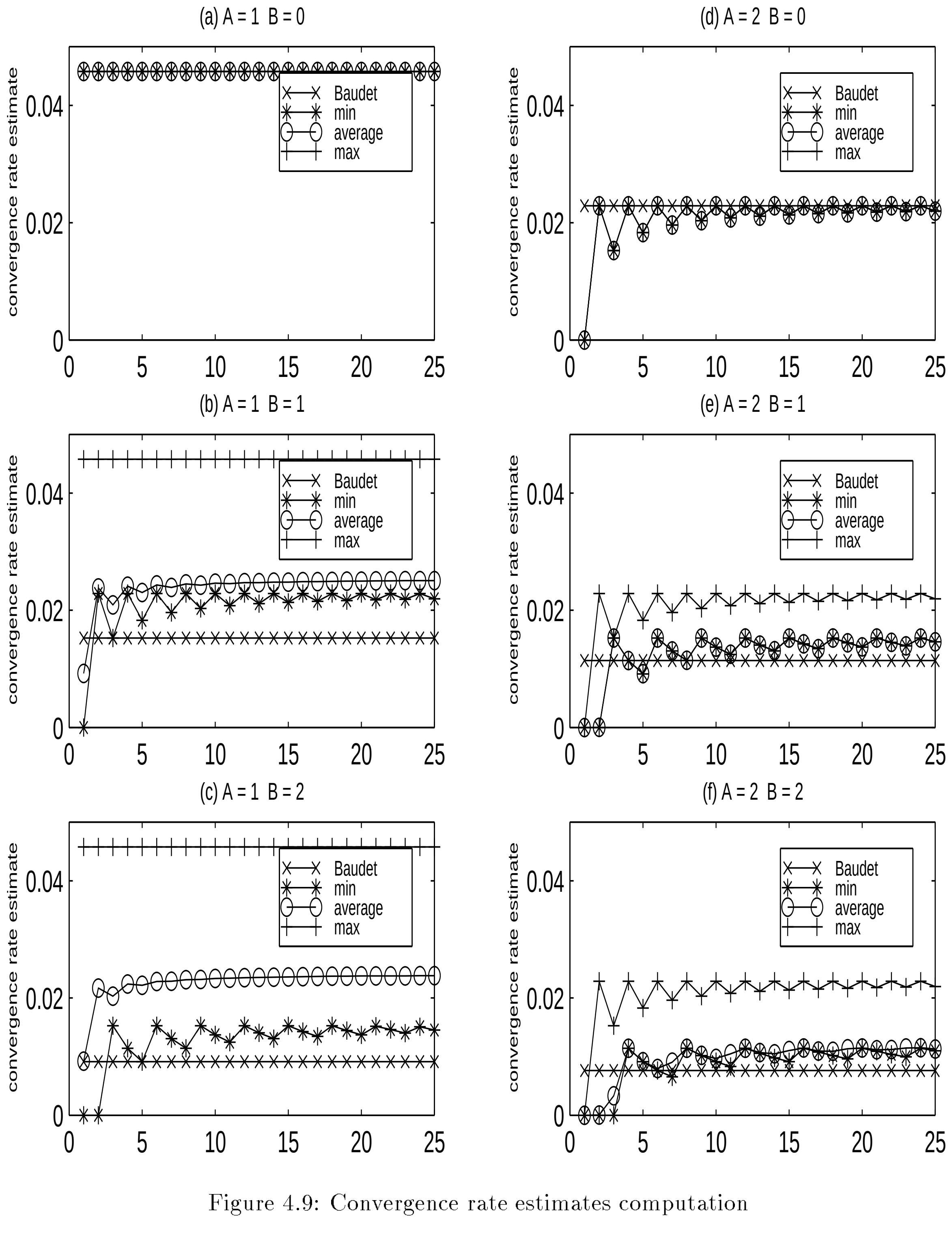

Figure 4.9: Convergence rate estimates computation

The "Baudet" curve show the estimated convergence rate as computed with Baudet's corollary of theorem 3 in [5]. This curve is flat, since Baudet's estimate is not computed as the limit of a sequence, but rather as a fixed lower bound on that sequence. The "min" curve corresponds to the computation of $\underline{\mathcal{R}^*}$ for increasing values of t. The "max" curve corresponds to $\overline{\mathcal{R}^*}$ and the "average" curve to $\widehat{\mathcal{R}^*}$. The values of these three estimates come directly from equation 4.2.3. Let us briefly comment on these six graphs.

Graph (a) corresponds to a synchronous implementation of the parallel iterative algorithm. All the estimates in that case are identical since the computation is deterministic (the number of iterations performed during an algorithm phase is not random). Graph (b) is for $A_i = 1$ and $B_i = 1$ for all i. The implementation is therefore asynchronous. The first immediate observation is that Baudet's estimate is below our three estimates. The second immediate observation is that $\underline{\mathcal{R}^*}$, $\widehat{\mathcal{R}^*}$ and $\overline{\mathcal{R}^*}$ all three seem to converge to finite values. And according to the graph, $\underline{\mathcal{R}^*} \leq \widehat{\mathcal{R}^*} \leq \overline{\mathcal{R}^*}$, which of course is consistent with equation 4.2.3. This observation can be made on all the graphs of Figure 4.9 that correspond to non-deterministic implementations (ones with random number of iterations performed at each phase). The fact that our worst-case estimate ($\underline{\mathcal{R}^*}$) is still higher than Baudet's is easily explained. Baudet's estimate ignores the underlying distributed environment, and therefore takes into account situations that might in fact not be possible. $\underline{\mathcal{R}^*}$, however, is a *refinement* of Baudet's estimate and considers only situations that occur with a non-null probability.

Graph (c) is for $A_i = 1$ and $B_i = 2$ for all i. In other words, graph (c) corresponds to a more asynchronous implementation than graph (b). The same observations can be made here, however, one can notice that the gaps between our estimates are bigger than for graph (b). This can be explained as follows. The implementation described by graph (c) has the possibility to perform more additional updates during the processors' β sub-phases than the one in graph (b), leading to more possible observations of the RVs $N^i(k)$ for all i. Therefore, according to equation 4.2.1, the RV k_t can take more values: it has a larger variance. This leads to bigger gaps between "min", "max" and "average".

The implementation described in graph (d) is asynchronous but with deterministic computations ($A_i = 2$ and $B_i = 0$ for all i). Here again, our estimates all equal Baudet's, even though their convergence is not as fast as in graph (a). This is due to the fact that this implementation is asynchronous and allows use of out-of-date data. Graph (e) introduces more asynchronism in the implementation of graph (d) and one can now observe that our estimates are all above Baudet's. The gaps between our three estimates are much smaller than the ones on graph (c) for instance. This shows that the RV k_t takes fewer values. Graph (f) if similar to graph (e) but shows wider gaps between our estimates than graph (d). This can be explained by the same argument as for graphs (b) and (c).

4.2.4 Conclusion

The main conclusion to draw from Figure 4.2.1 is that our three estimates appear to converge to finite values. $\underline{\mathcal{R}}^*$, our worst-case estimate, is always equal or larger than Baudet's because we take into account the distributed environment in which we run the application. The gaps between our estimates depend on the degree of asynchronism of the implementation. For implementations that have deterministic computations, our three estimates are all equal to Baudet's. Again, a formal proof of those results seems to be rather difficult, and will be reserved for future work.

Finally, one can notice that for this algorithm in this distributed environment, a synchronous implementation seems preferable, in terms of number of iterations to convergence, to any asynchronous implementation. This is true, no matter which estimate is used. This result was of course expected, since asynchronism slows down the convergence of the algorithm because of the use of out-of-date data. However, this does not mean that an asynchronous implementation cannot yield better performance to the end-user, in terms of time to convergence: an asynchronous implementation may perform more iterations in total, but its expected number of iterations per time unit may be larger - the net result being that convergence is achieved in less wall-clock time. In the next section, we are estimating the actual speed of a given implementation of the parallel iterative algorithm.

4.3 Estimating the Implementation Execution Speed

4.3.1 Defining a Speed Measure

In the previous section, we have computed different estimates for the rate of convergence of an implementation of a parallel iterative algorithm in a specific distributed environment. The rate of convergence is immediately connected to the number of algorithm iterations, as defined in Section 4.2.1, to divide the initial error on the solution by some factor (see Section 2.3.1). In order to provide something directly useful to the user, one now needs to estimate the speed of the implementation in terms of number of iterations performed per time unit. This is the purpose of this section.

We have already seen that the processors perform $N(k) = max_{i \in \{1,..,p\}}(A_i + N^i(k))$ algorithm iterations during the k^{th} algorithm phase. The speed of the algorithm during the k^{th} phase in terms of number of iterations per time unit is then given by $N(k)$ divided by the duration of that phase. The duration of the k^{th} phase might be slightly different on each processor and is denoted by $\Phi^i(k)$ on processor i (see Section 3.6.2). Since the p processors are partially resynchronized at the beginning of each algorithm phase, a reasonable and applicable definition is that the speed of the implementation during the k^{th} phase as $s(k) \stackrel{\Delta}{=} N(k)/\Phi^1(k)$. Let us see how our model can help us estimate this speed.

Our implementation of the stochastic model computes the probability:

$$\mathcal{P}\{ \begin{pmatrix} N(k) \\ \Phi^1(k) \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} | X(k) = z \} \quad \text{for } x \in \mathbb{N}, y \in \mathbb{R}, z \in \mathbb{R}^p,$$

where $\binom{a}{b}$ denotes the vector of $\mathbb{R}^2$ with components a and b. Similar to Section 4.2.2, one can replace this conditional probability by an unconditional one. This is done thanks to the π-values of the wavefront Markov chain. Here again, we are interested in long-run behavior of the algorithm and use the following approximation:

$$\mathcal{P}\{ \begin{pmatrix} N(k) \\ \Phi^1(k) \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \} = \sum_{l=1}^{\dim(P_X^*)} \pi_l \mathcal{P}\{ \begin{pmatrix} N(k) \\ \Phi^1(k) \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} | X(k) = X_l \}, \tag{4.3.4}$$

with the notation of equation 4.2.2. This approximation is performed by our implementation of the model. Note that with this approximation, the probability $\mathcal{P}\{ \binom{N(k)}{\Phi^1(k)} = \binom{x}{y} \}$ does not depend on k any longer. Therefore, this approximation implies that the $S(k)$ RVs are i.i.d., which is consistent with a long-run observation of the Markov chain.

From now on, $S(k)$ will denote the vector $\binom{N(k)}{\Phi^1(k)}$ in $\mathbb{N} \times \mathbb{R}$. Since the distribution of $S(k)$ is known, one can easily compute $\mathbb{E}[S(k)]$ where $\mathbb{E}$ denotes the expectation of a RV. One can write $\mathbb{E}[S(k)] = S$ that does not depend on k.

4.3.2 An Example

Figure 4.10 shows the probability distributions of $s(k) \stackrel{\Delta}{=} N(k)/\Phi^1(k)$ for a given distributed environment, a given algorithm and for different implementations. As we have seen earlier, $N(k)/\Phi^1(k)$ can be interpreted as the implementation speed during an algorithm phase. The distributed environment is the one of example 3 in Section 4.1.2. The algorithm corresponds to a contracting operator whose contracting matrix has a spectral radius $\rho = 0.9$. The different implementations are for different values of A_i and B_i for $i = 1, ..., p$. As in Section 4.2.2, we assume that those values are the same for all three processors. The figure shows six different graphs ((a) through (f)), corresponding to all the possible cases where $A_i = 1, 2$ and $B_i = 0, 1, 2$ for all i as in Figure 4.9. On each graph, the mean value of $N(k)/\Phi^1(k)$ is indicated by a vertical line, and the standard deviation is represented on each side of the mean value as a horizontal solid line segment. On graph (a), one can see that the distribution of $s(k)$ has a relatively smooth shape and a mean value of 0.1655. This means that, on average, a synchronous implementation performs 0.1655 algorithm iterations per second. Graph (b) corresponds to an asynchronous implementation, and the distribution of $s(k)$ is much less regular. This is due to the fact that number of iterations performed during an algorithm phase is random.

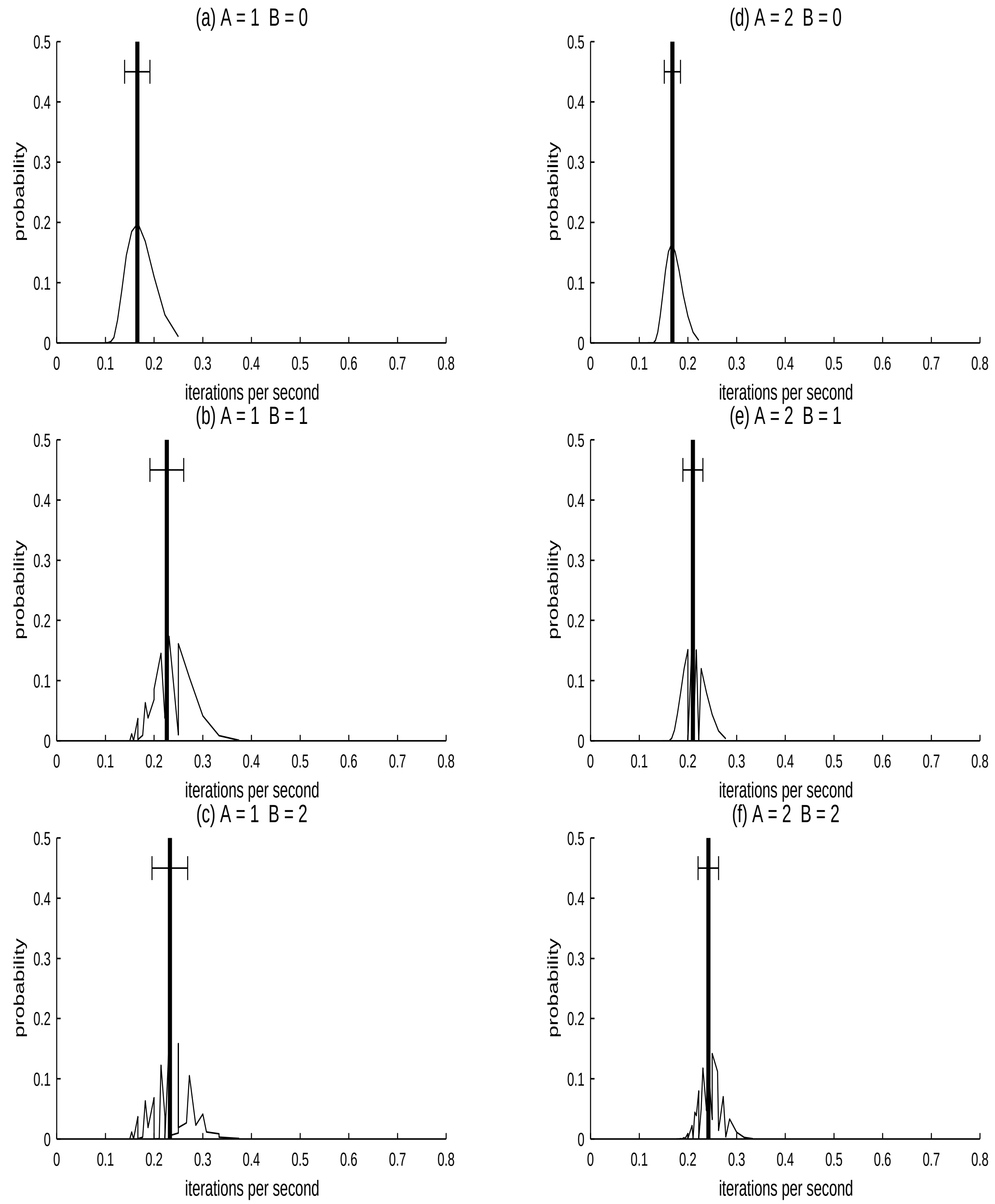

Figure 4.10: Implementation Speed

The mean value here is of 0.2262, making the implementation approximatively 37% faster than for the synchronous case. This is due to the use of otherwise wasted CPU cycles by performing additional updates during the β sub-phases. Such an increase in implementation speed can make an asynchronous implementation worthwhile. Even though it might require more iterations than a synchronous implementation in order to converge, it performs them faster. The mean value on graph (c) is fairly similar to the one on graph (b), because there are few cases in which a processor has the time to perform two additional updates during its β sub-phases. The same observations can be made on graphs (e), (d) and (f).

A fundamental question that the end-user wants to answer is : "How long before convergence (in seconds) ?" Let us assume that the asymptotic convergence rate of the algorithm is $\mathcal{R}$ as defined in definition 2.3.2. Let us also assume that the algorithm runs for enough iterations so that we can use $\mathcal{R}$ as a good approximation. Additionally, let $\omega \in \mathbb{N}$ be the user's convergence requirement: convergence is reached when the initial error is divided by a factor of 10^{ω}. This is a standard definition of user-defined convergence and is used by Baudet in [5] for instance. Let Θ be the answer to the user's question, that is, a time in seconds. We are going to see how Θ can be estimated.

4.3.3 A Mean Estimate

The idea here is to find the asymptotic speed of the implementation. Let us consider the algorithm after it has completed n phases. At that time, the implementation has been running for $\sum_{k=1}^{n} \Phi^1(k)$ seconds and has performed $\sum_{k=1}^{n} N(k)$ iterations. Let us consider the vector S_n defined as:

$$S_n \stackrel{\Delta}{=} \begin{pmatrix} \sum_{k=1}^{n} N(k) \\ \sum_{k=1}^{n} \Phi^1(k) \end{pmatrix} = \sum_{k=1}^{n} S(k).$$

The Strong Law of Large Numbers states that:

$$\mathcal{P}\{\lim_{n\to\infty} \frac{1}{n} S_n = S\} = 1,$$

where $S = \mathbb{E}[S(k)]$. Let us write S as $\binom{N}{\Phi}$. Then:

$$S_n \sim \begin{pmatrix} n \times N \\ n \times \Phi \end{pmatrix} \quad \text{with probability 1.}$$

The speed achieved so far by the implementation after phase n can clearly be seen as the ratio of the first and second components of S_n. Let s_n denote that speed; the equation above then shows that:

$$s_n \underset{\infty}{\sim} \frac{n \times N}{n \times \Phi} \sim \frac{N}{\Phi} \quad \text{with probability 1.}$$

This gives us our asymptotic implementation speed. Indeed, our model implementation, among other things, computes the vector S. It is then easy to find an estimate of Θ. Assuming that the implementation performs N/Φ iterations per second and that the algorithm must run for $\omega/\mathcal{R}$ iterations to meet the user's convergence criterion, we have the following estimate:

$$\Theta = \frac{\Phi \times \omega}{N \times \mathcal{R}}.$$

4.3.4 Refining this Estimate

After finding this asymptotic measure of the implementation speed, it is natural to compute some certainty measure on that estimate; namely a variance. Since we are considering here RVs in $\mathbb{R}^2$, we can compute a covariance matrix because the entire distribution of $S(k)$ is known. Using that covariance matrix, it is natural to construct the covariance matrix of S_n for any value of n. As it will be seen in what follows, this covariance matrix can in turn be used to compute an approximation of the standard deviation of the algorithm execution time. One can also try to use Chebyshev's inequality to estimate the deviations of the sample average from the mean. Such a deviation analysis can provide answers to meaningful questions like: "What is the probability that a run of the implementation is particularly slow or particular fast?". In fact, the analysis of the deviations of the sample average of i.i.d. observations of a RV from its mean can be answered extremely precisely for large deviations, much more precisely than by applying Chebyshev's inequality. The estimation of the probability of rare events such as deviations from the mean by a significant amount can be performed thanks to the *Large Deviation Theory.*

4.4 Large Deviations Results

In this section, we present a brief outline of the Large Deviation Theory (LDT), identify how it can be used in our analysis, and give an example.

4.4.1 Large Deviations Theory

LDT is a branch of probability concerned with quantifying and explaining the behavior of rare events. It is a very active field of research at the moment, and several reference books are available, as well as introductory papers. The material we present in this section comes from [15, 48, 57]. Let x_i for $i = 0, 1, \ldots$ be i.i.d. real RVs of expectations $\mathbb{E}[x_i] = \mathbb{E}[x_1]$. The simplest large deviation question is: what is

$$\mathcal{P}\{\frac{x_1 + \ldots + x_n}{n} \geq a\} \quad \text{where } a > \mathbb{E}[x_1] \quad ?$$

The Strong Law of Large Numbers tells us that the sample average converges to $\mathbb{E}[x_1]$ with probability 1. The event where the sample average deviates from the mean is a rare event as n goes to ∞. Estimating the probability of such a rare event is answered by Chernoff's theorem for i.i.d. RVs (also called Cramer's theorem) [17, 19]. Let us define:

$$\begin{aligned} M(\theta) &\overset{\Delta}{=} \mathbb{E}[e^{\theta x_1}] \quad \forall \theta \in \mathbb{R}, \\ l(a) &\overset{\Delta}{=} \sup_{\theta} (\theta a - \log M(\theta)) \quad \forall a \in \mathbb{R}. \end{aligned} \tag{4.4.5}$$

$M(\theta)$ is called the *moment generating function* of x_1. Note that the function l is non-negative ($M(0) = 0$) and convex (as the supremum of a family of convex functions). It is called the *rate function* of x_1. Chernoff's theorem can be written as:

Theorem 4.4.1 Consider the sequence $x_1, x_2, ...$ of i.i.d. RVs. For every $a > \mathbb{E}[x_1]$ and positive integer n,

$$\mathcal{P}\{x_1 + ... + x_n \geq na\} \leq e^{-nl(a)}. \tag{4.4.6}$$

Assume that $M(\theta) < \infty$ for θ in some neighborhood of 0 and that the supremum in equation 4.4.5 is attained in that neighborhood. Then for every $\epsilon > 0$ there exists an integer n_0 such that whenever $n > n_0$,

$$\mathcal{P}\{x_1 + ... + x_n \geq na\} \geq e^{-n(l(a)+\epsilon)}. \tag{4.4.7}$$

Equations 4.4.6 and 4.4.7 imply that

$$\mathcal{P}\{x_1 + ... + x_n \geq na\} = e^{-nl(a)+o(n)}. \tag{4.4.8}$$

A proof of this theorem can be found in [48]. It is valid for discrete, continuous, or mixed RVs. This result can be stated in a more general theorem:

Theorem 4.4.2 Let $x_1, x_2, ...$ be i.i.d. RVs. The function l defined in equation 4.4.5 is convex and lower semi-continuous. For any closed set F,

$$\limsup_{n\to\infty} \frac{1}{n} \log \mathcal{P}\{\frac{x_1 + ... + x_n}{n} \in F\} \leq -\inf_{a\in F} l(a), \tag{4.4.9}$$

and for any open set G,

$$\liminf_{n\to\infty} \frac{1}{n} \log \mathcal{P}\{\frac{x_1 + ... + x_n}{n} \in G\} \geq -\inf_{a\in G} l(a). \tag{4.4.10}$$

This theorem just states that the sample average of i.i.d. RVs satisfies a Large Deviation Principle (LDP) according to Definition 2.2 in [48]. In [48] one can also find the following definition:

Definition 4.4.1 *A set S is called an l-continuity set for a rate function l if*

$$\inf_{x\in S^o} l(x) = \inf_{x\in \overline{S}} l(x),$$

where S^o denotes the interior of S and $\overline{S}$ its closure.

The following theorem is given as an exercise in [48], but its proof is immediate:

Theorem 4.4.3 If S is an l-continuity set, then

$$\lim_{n\to\infty} \frac{1}{n} \log \mathcal{P}\{\frac{x_1 + ... + x_n}{n} \in S\} = -\inf_{a\in S} l(a).$$

These results can be extended to the case where the RVs x_i are vectors of $\mathbb{R}^d$. The definition of the rate function in such a case is:

$$\begin{cases} M(\theta) & \triangleq \mathbb{E}[e^{\langle\theta,x_1\rangle}] \quad \forall\theta \in \mathbb{R}^d \\ l(a) & \triangleq \sup_\theta(\langle\theta, a\rangle - \log M(\theta)) \quad \forall a \in \mathbb{R}^d, \end{cases} \tag{4.4.11}$$

where $\langle ., . \rangle$ denotes the Euclidean inner product in $\mathbb{R}^d$. Under the same assumptions as in the one-dimensional case, the average mean of i.i.d. random vectors satisfies a LDP with that rate function.

All these versions of Chernoff's Theorem are useful in telling us how often specific rare events occur. Another theorem by Sanov [47] tells how these events occur when they do. Sanov's theorem indicates that rare events, with overwhelming probability, happen only one way: by a "conspiracy". This means that when a rare event occurs, the observations of the RVs behave as if they were samples from a different distribution, often referred to as the "tilted" distribution. This kind of consideration is part of what is called "level 2 large deviations" in [48]. LDT contains many more fascinating results, but is noted for being mathematically very demanding, involving solving difficult problems in the calculus of variations for instance. The fundamental results we have already stated will be sufficient for our purpose in this dissertation.

4.4.2 Rate Function in our Model

The rare events we consider here concern the speed of the implementation. That speed has already been quantified as the sample average of observations of the vector $S(k)$ throughout time. The distribution function of that vector is provided by our implementation of the model. We know from LDT that this sample average, denoted $\frac{1}{n}S_n$, satisfies a LDP. The problem is to compute its rate function. Let us proceed step by step.

Let $\theta = \binom{\theta_1}{\theta_2}$ be a vector in $\mathbb{R}^2$. We recall that the moment generating function of $S(k)$ is defined as:

$$M(\theta) = \mathbb{E}[e^{\langle \theta, S(k) \rangle}].$$

At this point, we need to describe the distribution of $S(k)$ formally (it is computed by our implementation of the model). Let us call D the subset of $\mathbb{R}^2$ such that:

$$D \stackrel{\Delta}{=} \{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2 | \mathcal{P}\{(S(k) = \begin{pmatrix} x \\ y \end{pmatrix}\} \neq 0\}.$$

In other words, D is the set of possibles values for $S(k)$ and is finite. Let also $p_{x,y}$ denote $\mathcal{P}\{(S(k) = \binom{x}{y})\}$ for $\binom{x}{y}$ in D. The moment generating function of $S(k)$ can then be written as:

$$M(\begin{pmatrix}\theta_1\\ \theta_2\end{pmatrix}) = \sum_{\binom{x}{y}\in D} p_{x,y}e^{\theta_1 x+\theta_2 y}.$$

This makes it clear that $M(\theta) < \infty$ for all $\theta < \infty$, and therefore in a neighborhood of 0. The rate function for the sequence $S(k)$ can now be written as:

$$\forall a = \begin{pmatrix}a_1\\ a_2\end{pmatrix} \in \mathbb{R}^2 \quad l(a) = \sup_{\theta_1,\theta_2\in\mathbb{R}} [\theta_1 a_1 + \theta_2 a_2 - \log(\sum_{\binom{x}{y}\in D} p_{x,y}e^{\theta_1 x+\theta_2 y})]. \tag{4.4.12}$$

The rate function is therefore, at each point, the supremum over $\mathbb{R}^2$ of an infinitely continuously differentiable real function. Let us call this function f so that

$$l(a) = \sup_{\theta_1,\theta_2\in\mathbb{R}} f(\theta_1,\theta_2).$$

One can rewrite $f(\theta_1,\theta_2)$ as:

$$f(\theta_1,\theta_2) = -\log(\sum_{\binom{x}{y}\in D} p_{x,y}e^{\theta_1(x-a_1)+\theta_2(y-a_2)}).$$

Let us see under which conditions the supremum in 4.4.12 is attained. If this supremum is attained for $\theta_1 = \theta_1^*$ and $\theta_2 = \theta_2^*$, then:

$$\begin{cases} \frac{\partial f}{\partial \theta_1}(\theta_1^*, \theta_2^*) = 0 \\ \frac{\partial f}{\partial \theta_2}(\theta_2^*, \theta_2^*) = 0, \end{cases}$$

which can be rewritten as:

$$\begin{cases} -\dfrac{\displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}(x-a_1)e^{\theta_1(x-a_1)+\theta_2(y-a_2)}}{\displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}e^{\theta_1(x-a_1)+\theta_2(y-a_2)}} = 0 \\ -\dfrac{\displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}(y-a_2)e^{\theta_1(x-a_1)+\theta_2(y-a_2)}}{\displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}e^{\theta_1(x-a_1)+\theta_2(y-a_2)}} = 0 \end{cases}$$

$$\iff \begin{cases} \displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}(x-a_1)e^{\theta_1(x-a_1)+\theta_2(y-a_2)} = 0 \\ \displaystyle\sum_{\binom{x}{y}\in D} p_{x,y}(y-a_2)e^{\theta_1(x-a_1)+\theta_2(y-a_2)} = 0. \end{cases}$$

Let us define $\underline{x}$, $\overline{x}$, $\underline{y}$ and $\overline{y}$ as:

$$\begin{cases} \underline{x} \stackrel{\Delta}{=} \min\{x | \binom{x}{y} \in D\} \\ \overline{x} \stackrel{\Delta}{=} \max\{x | \binom{x}{y} \in D\} \\ \underline{y} \stackrel{\Delta}{=} \min\{y | \binom{x}{y} \in D\} \\ \overline{y} \stackrel{\Delta}{=} \max\{y | \binom{x}{y} \in D\}. \end{cases}$$

We are going to prove that the rate function, l, can take finite values only inside a rectangle of the 2-D plane. This rectangle is denoted by Ψ and defined by:

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \in \Psi \Longleftrightarrow \begin{cases} \underline{x} \leq a_1 \leq \overline{x} \\ \underline{y} \leq a_2 \leq \overline{y}. \end{cases}$$

The rectangle is shown on Figure 4.11. Let $a = \binom{a_1}{a_2}$ be an arbitrary vector of $\mathbb{R}^2$ outside of the rectangle Ψ. To prove that $l(a) = +\infty$, we are going to construct a sequence of vectors in $\mathbb{R}^2$, say $\{\theta(n) = \binom{\theta_1(n)}{\theta_2(n)}\}$ for $n \in \mathbb{N}$, such that $\lim_{n\to\infty} f(\theta_1(n), \theta_2(n)) = +\infty$. Let us assume for instance that $a_1 < \underline{x}$. The cases $a_1 > \overline{x}$, $a_2 < \underline{y}$ and $a_2 > \overline{y}$ can be treated similarly. One can write that:

$$\forall \begin{pmatrix} x \\ y \end{pmatrix} \in D, \qquad x - a_1 > 0.$$

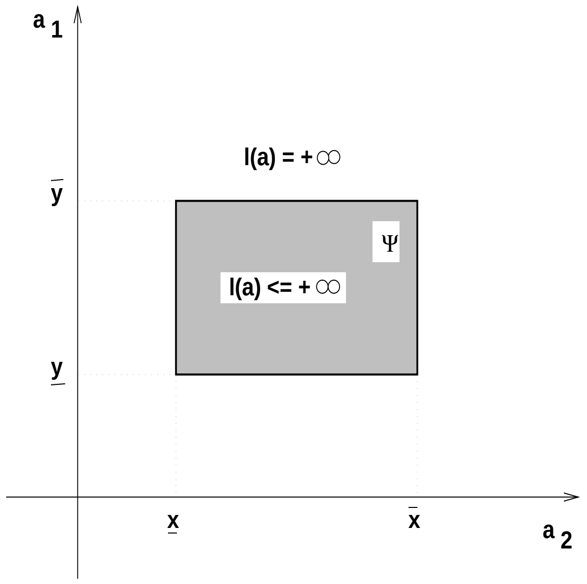

Figure 4.11: Rectangle Ψ

If we define

$$\begin{cases} \theta_1(n) \overset{\Delta}{=} -n \\ \theta_2(n) \overset{\Delta}{=} 0, \end{cases}$$

then

$$f(\theta_1(n), \theta_2(n)) = -\log\Big(\sum_{\binom{x}{y} \in D} p_{x,y} e^{-n(x-a_1)} \Big) \xrightarrow[n\to\infty]{} +\infty,$$

since $x - a_1 > 0$ for all $\binom{x}{y}$ in D.

This proves that $l(a) = +\infty$ outside of Ψ. This result is fairly intuitive. Indeed, the sample average S_n of observations of $S(k)$ is outside of Ψ with probability 0, since

$$\frac{1}{n}\begin{pmatrix} n \times \underline{x} \\ n \times \underline{y} \end{pmatrix} \leq \frac{1}{n} S_n \leq \frac{1}{n}\begin{pmatrix} n \times \overline{x} \\ n \times \overline{y} \end{pmatrix}.$$

The fact that the rate function outside of Ψ is infinite can be interpreted as follows: The probability that the sample average stays outside Ψ decays with an infinite exponential rate.

However, depending on the probability distributions generated by our model, the rate function might also take infinite values inside Ψ. It seems difficult to find a general analytical expression for the rate function inside Ψ, but it is possible to use numerical software to compute the supremum in equation 4.4.12. For this research, we have used Matlab [28] because it is very straightforward, provides visualization capabilities and seems to give reasonably accurate numerical results. The next section gives examples of rate functions for a given iterative algorithm in a given distributed environment, and shows how they can be used to estimate probabilities of deviation of the implementation speed from its mean.

4.4.3 An Example of Large Deviation Computation

We first show three examples of rate functions for different models. We then explain how the rate function can be used in practice to compute deviation probabilities. Finally, we perform such computations on one of the examples.

Sample Rate Functions

The rate function is computed be means of a series of Matlab scripts. Those scripts input the distribution of the RV $S(k)$ and produce a discretized version of the rate function. Once again, we use the distributed environment of example 3 in Section 4.1.2. The algorithm corresponds to a contracting operator whose contracting matrix has a spectral radius $\rho = 0.9$. The different implementations are for the cases where $A_i = 1$ and $B_i = 0, 1, 2$ for all $i = 1, 2, 3$. Let us describe the three rate functions generated by our stochastic model.

Figure 4.12 shows the rate function l for $A_i = 1$ and $B_i = 0$. This is a synchronous implementation and exactly 1 iteration is performed at each algorithm phase. On the figure, the mean vector S is represented as a vertical line and the rectangle Ψ is represented on the horizontal plane with a thick line. First, let us identify Ψ. One recalls that Ψ is defined by $\underline{x}$, $\overline{x}$, $\underline{y}$ and $\overline{y}$ (see Section 4.4.2). In this case, our implementation of the model gives us $\underline{x} = \overline{x} = 1$, $\underline{y} = 7$ and $\overline{y} = 21$. The rectangle Ψ is therefore reduced to a segment. One can also observe that the rate function attains its minimum of 0 at S. This is a property of the rate function (see [48]). It is very intuitive; since the Strong Law of Large Numbers states that the probability that the sample average converges to the mean is 1, it should therefore decay with an exponential rate equal to 0. As a general rule, every point where the function is not represented is a point where l takes an infinite value.

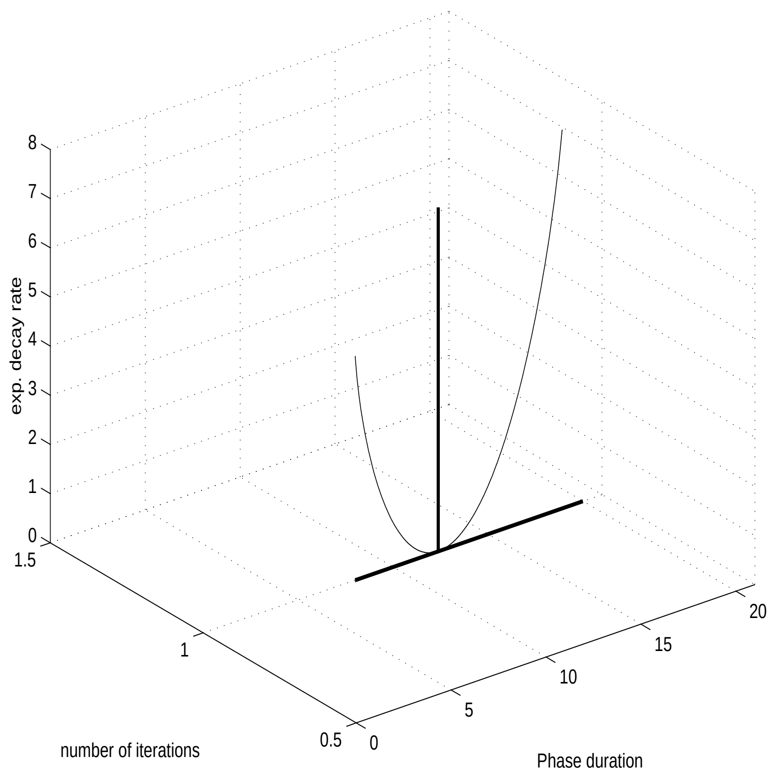

Figure 4.12: Rate function for $A_i = 1$ and $B_i = 0$

Figure 4.13 shows the rate function l for the asynchronous implementation corresponding to $A_i = 1$ and $B_i = 1$. Here, Ψ is not reduced to a segment and it is represented on the horizontal plane. The mean vector S is represented as a vertical line and one again has $l(S) = 0$. The ragged aspect of the edge of the curve is due to the Matlab discretization. Figure 4.14 shows another view of the rate function, from "above". On this figure, one can see more clearly on what subset of Ψ the rate function takes finite values. Again, the staircase shape of the edges is due to the Matlab discretization. The mean vector is shown with a white cross.

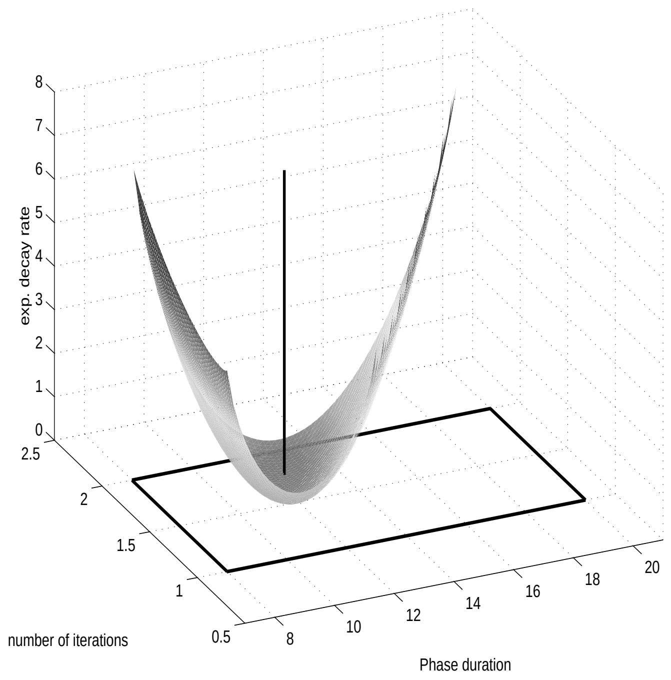

Figure 4.13: Rate function for $A_i = 1$ and $B_i = 1$

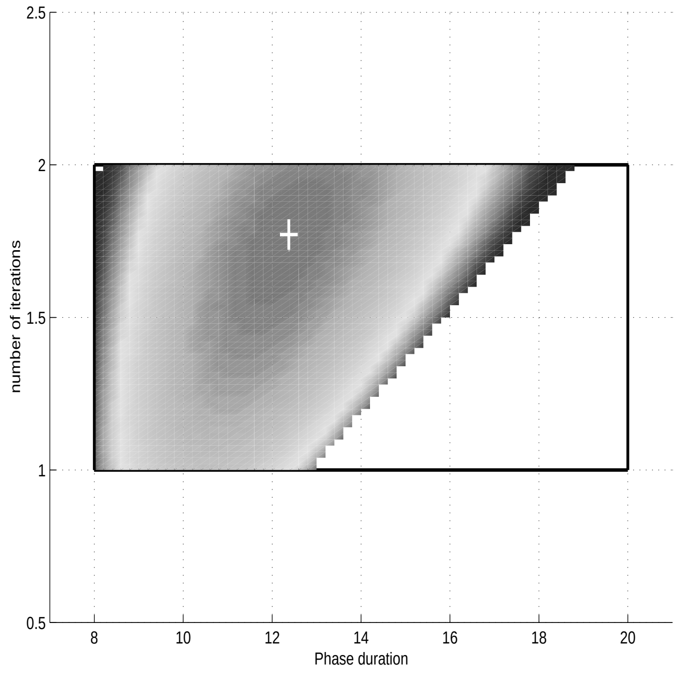

Figure 4.14: Overview of the rate function for $A_i = 1$ and $B_i = 1$

Figure 4.15 and 4.16 are similar to the two previous ones, but are for $A_i = 1$ and $B_i = 2$. The rectangle Ψ is wider since the implementation of the iterative algorithm can now perform up to three iterations per phase.

Example Use of the Rate Function

Let us see how the rate function can be used to compute a useful result concerning the implementation speed. The mean algorithm speed in terms of number of iterations performed per second is

$$s = \frac{N}{\Phi},$$

where N and Φ are the components of the mean vector S.

Let us compute, for instance, the asymptotic probability that the observed implementation speed after n phases is lower than the mean, meaning that:

$$s_n < s - \epsilon \quad \text{where} \quad \epsilon > 0.$$

This can only happen if the sample average vector S_n lies in R_ϵ^-, the subset of $\mathbb{R}^2$ defined as:

$$R_\epsilon^- \overset{\Delta}{=} \{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2 | \frac{x}{y} \leq s - \epsilon \}.$$

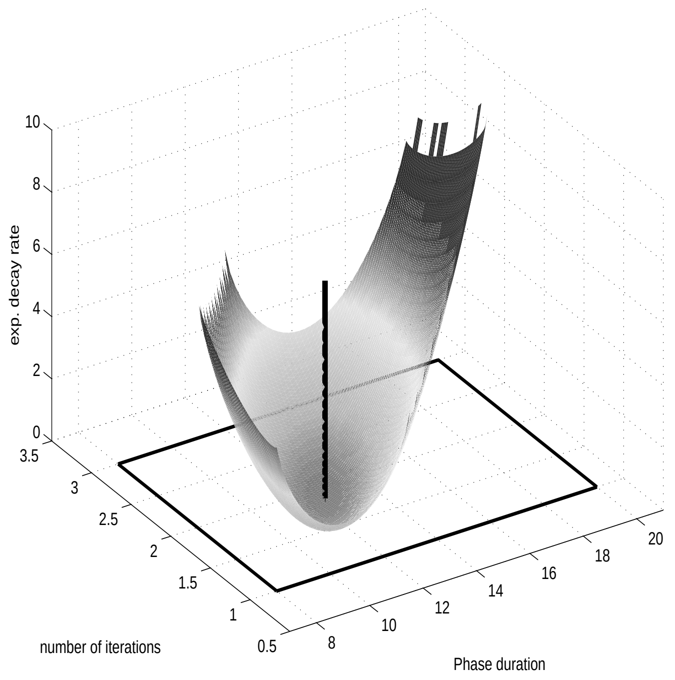

Figure 4.15: Rate function for $A_i = 1$ and $B_i = 2$

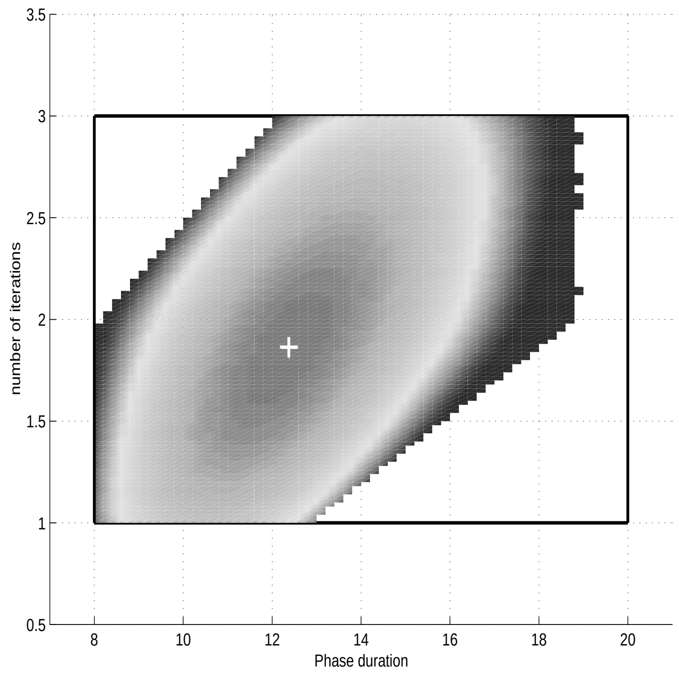

Figure 4.16: Overview of the rate function for $A_i = 1$ and $B_i = 2$

R_ϵ^- is an open 2-D half plane *above* the line D_ϵ of equation

$$D_\epsilon^- : y = \frac{1}{s-\epsilon}x.$$

Let Ψ^* denote a closed subset of Ψ where the rate function takes finite values and is continuous. The existence of Ψ^* is far from being obvious and depends on the rate function properties. We do not perform here an in-depth analysis of the rate function as it depends on the probability distributions in the stochastic model. Experience shows that the rate function takes finite values in closed subsets of Ψ and appears to be continuous on those subsets. Furthermore, those subsets seem to be convex, and one can chose Ψ^* to be the "largest" one, meaning that it contains all subsets where the rate function is continuous and finite. On the previous rate function examples, we have seen that the set Ψ^* can be clearly identified. In all that follows, we will see assume that such a Ψ^* always exists, but we do not provide a formal proof of its existence. Such a proof seems to be very involved and we leave it for future work. It is now easy to see that the set $\Psi^* \cap R_\epsilon^-$ is an *l-continuity* set according to definition 4.4.1. One can now use theorem 4.4.3 and state that:

$$\lim_{n\to\infty} \frac{1}{n} \log \mathcal{P}\{s_n \in \Psi^* \cap R_\epsilon^-\} = - \inf_{a \in \Psi^* \cap R_\epsilon^-} l(a).$$

The same kind of computation can also be performed to compute the asymptotic probability that the implementation speed observed after n algorithm phases is larger than the mean. The only difference is in the definition of the half-plane. In this case, we call it R_ϵ^+ and it is the half-plane *below* the line D_ϵ^+ defined as:

$$D_\epsilon^+ : y = \frac{1}{s+\epsilon} x.$$

Figure 4.17 shows the intersection of the two half planes R_ϵ^- and R_ϵ^+ with Ψ^*.

In order to perform actual numerical computations, we are going to use the third example rate function that we have considered at the beginning of this section, that is, corresponding to the case where $A_i = 1$ and $B_i = 2$ for $i = 1, 2, 3$. The components of the mean vector S are respectively $N = 1.8631$ and $\Phi = 12.3754$. The asymptotic algorithm speed is therefore $s = N/\Phi = 0.1505$ in iterations per second. Let us compute the asymptotic probabilities that the speed observed after n algorithm phases, s_n, deviates from the mean s by at least $\epsilon = 0.025$ or $\epsilon = -0.025$. LDT tells us that those probabilities decay exponentially, and that they respectively decay with rates r_ϵ^+ and r_ϵ^- with:

$$\begin{cases} r_\epsilon^+ & \triangleq \inf_{a \in \Psi^* \cap R_\epsilon^+} l(a), \\ r_\epsilon^- & \triangleq \inf_{a \in \Psi^* \cap R_\epsilon^-} l(a). \end{cases} \tag{4.4.13}$$

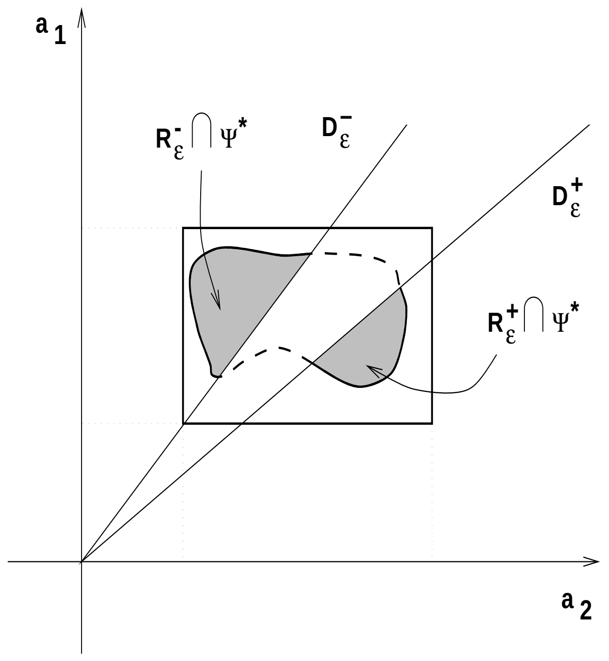

Figure 4.17: $R_\epsilon^- \cap \Psi$ and $R_\epsilon^+ \cap \Psi$

Figure 4.18 shows the sets $\Psi \cap R_\epsilon^-$ and $\Psi \cap R_\epsilon^+$ as “triangles” on the horizontal plane. The problem here is to find the minimum of the rate function on those two triangles. The rate function is represented only on the triangles in the figure and the ragged shape of the edges is here again due to Matlab's discretization. One can use Matlab to perform the computation of the minima, and the numerical results obtained in this example are:

$$\begin{cases} r_{0.025}^+ &= 0.2917 \\ r_{0.025}^- &= 0.2508, \end{cases}$$

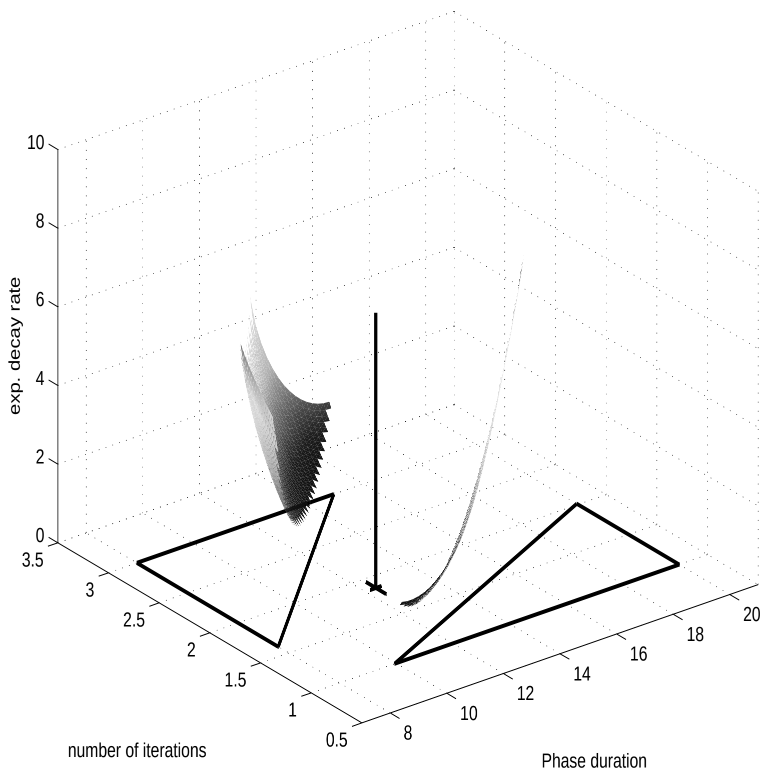

Figure 4.18: Example of Large Deviation Computation

so that:

$$\begin{cases} \mathcal{P}\{s_n \geq 0.1755\} = e^{-0.2917 \times n + o(n)} \\ \mathcal{P}\{s_n \leq 0.1255\} = e^{-0.2508 \times n + o(n)}. \end{cases}$$

Such a computation can be performed for any value of ϵ. Figure 4.19 shows a graph of r^{+}_{ϵ} for positive values of ϵ and of $r^{-}_{-\epsilon}$ for negative values of ϵ. Therefore, the part of the graph for negative values of ϵ corresponds to implementations of the iterative algorithm that are slower than the mean, and the part of the graph

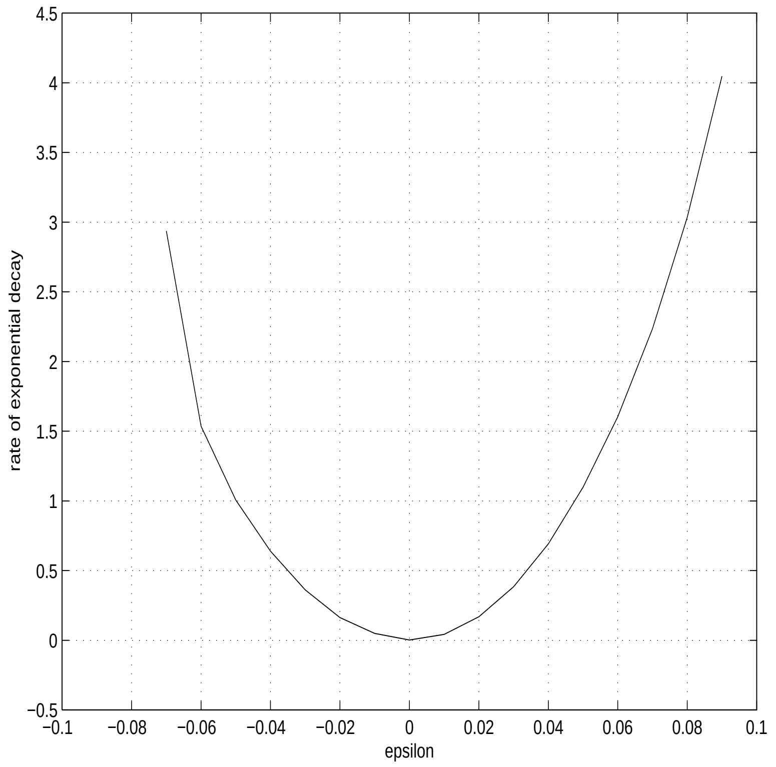

Figure 4.19: Exponential decay rate for various ϵ values

for positive values of ϵ is for implementations that are faster than the mean. The rate of exponential decay is infinite wherever it is not represented on the graph. With such a graph, it is now possible to compute all the rates of exponential decay that we are need for quantifying the behavior of the observed average speed of an implementation of an iterative algorithm.

4.5 Performance Characterization

In this section, we summarize the results presented in this chapter as well as the steps that are followed by our implementation of the stochastic model. We then explain how these results can be used to obtain different levels of performance characterization. Finally, we present a complete example.

4.5.1 Putting Things Together

Figure 4.20 shows a diagram that summarizes our performance characterization process. Let us describe this diagram step by step. As symbolized by the topmost box in the diagram, the input to our performance characterization mechanisms consists of:

- The number of processors in the distributed environment (p),
- The probability distributions of the processor update times ($\alpha^i(k, \theta)$),
- The probability distributions of the network communications ($n_{i \to j}(k)$),
- The spectral radius of the matrix associated with the contracting operator (Op),
- The A_i and B_i values for $i = 1, ..., p$.

The diagram contains three boxes drawn with dashed lines. The middle box corresponds to Section 4.1, the right box to Section 4.2, and the left box to

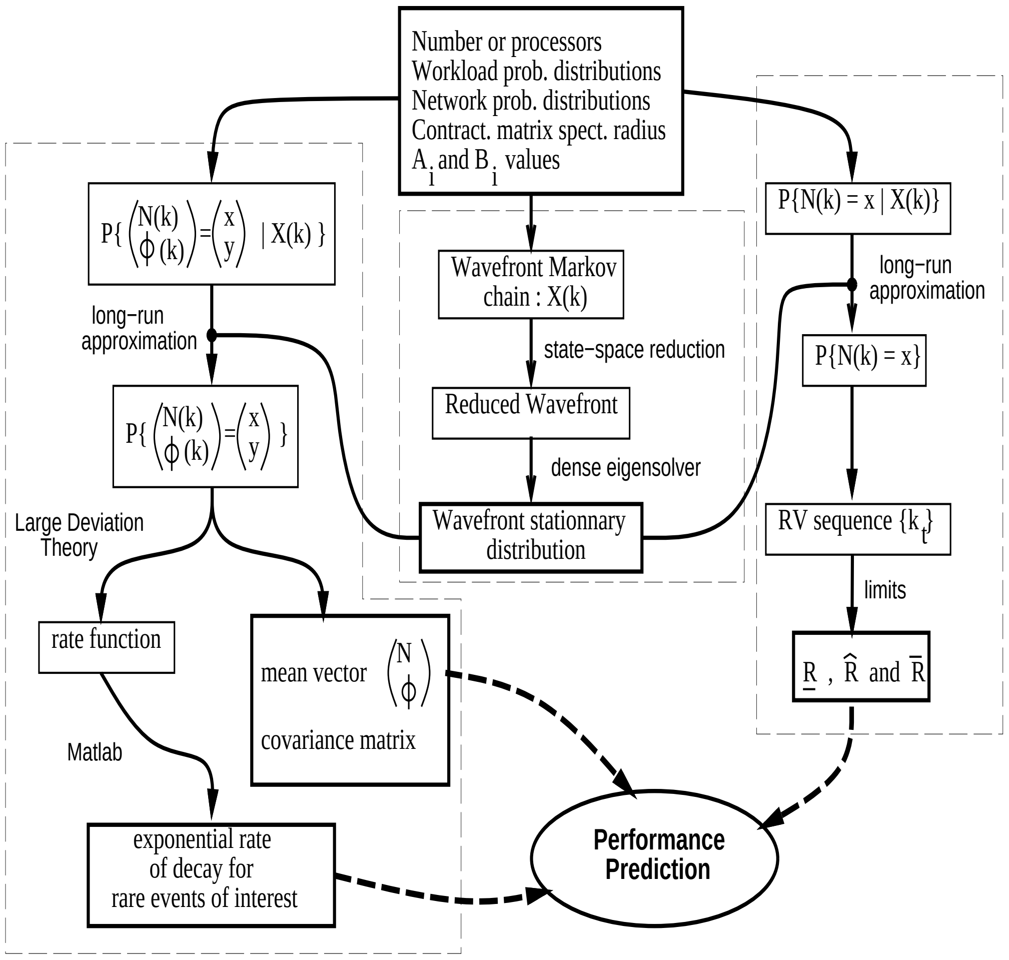

Figure 4.20: Performance Characterization method

Section 4.3. We are going to describe each box step by step.

We start with the middle box since it impacts the other two. The first thing that our model computes is the wavefront transition matrix. As seen in Section 4.1.1, this is done with a parallel program in C using MPI. Once the transition matrix is available, the state-space of the Markov chain is reduced to eliminate unreachable states. The stationary distribution of the wavefront can then be computed with a classic eigensolver (from the LAPACK numerical library). Examples of such computations were given in Section 4.1.2.

Let us now have a look at the right box. It corresponds to the computation of convergence speed estimates (in terms of number of iterations to convergence). Our implementation of the stochastic model first computes the conditional probabilities $\mathcal{P}\{N(k) = x|X(k)\}$ where $N(k)$ is the number of iterations performed by the implementation of the iterative algorithm during the kth algorithm phase and $X(k)$ is the wavefront state at the beginning of that phase. Using the wavefront stationary distribution, an approximation of the unconditional probability $\mathcal{P}\{N(k) = x\}$ is computed. This approximation is justified in Section 4.2.2. The sequence $\{k_t\}$ for $t = 0, 1, ...$ can then be constructed as a sequence of RVs of known probability distributions. Giving three different senses to the limit of that sequence yields the three estimates $\underline{\mathcal{R}^*}$, $\widehat{\mathcal{R}^*}$ and $\overline{\mathcal{R}^*}$ (those estimates are actually computed as limits of the sequence). We have at this point computed three estimates of algorithm speed -worse, average and best cases- in terms of number of iterations to convergence.

The third box corresponds to the computation of the algorithm speed in terms of number of iterations performed per second. From its input, our implementation of the stochastic model computes the conditional probability $\mathcal{P}\{S(k) = \binom{x}{y}|X(k)\}$ where $S(k)$ is the vector of $\mathbb{N}\times\mathbb{R}$ whose first component if the number of iterations performed during algorithm phase k, and whose second component is the duration in seconds of that phase. Similarly to the computation of the convergence speed in the right box, the stationary distribution of the wavefront can be used to obtain an approximation of the unconditional probability $\mathcal{P}\{S(k) = \binom{x}{y}\}$. Once this distribution is known, one can easily compute the mean vector $S = \binom{N}{\Phi}$. This vector can be used to compute the asymptotic algorithm speed in terms of number of iterations performed per second. This speed is denoted by s and is $s = N/\Phi$ (see Section 4.3). As shown in the next section, one can compute the covariance matrix of $S(k)$.

Detailed information about probabilities that the observed algorithm speed deviates from this asymptotic speed (leading to particularly short or long execution times) can be obtained with the Large Deviation Theory (see Section 4.4), which is much more precise for rare events than a computation based on variance alone. We can define a rate function for the sequence $S(k)$ as in Section 4.4.2. Computed numerically, this rate function gives precise values for the exponential rate of decay for the probabilities of specific rare events. The rare events that we are considering, since we want to analyze the algorithm performance, correspond to the deviations aforementioned. Example of such computations were given in Section 4.4.3.

One can distinguish two parts in the performance characterization scheme: estimating the convergence rate of the algorithm and estimating the speed of the implementation. For the latter, we additionally have access to Large Deviation results. It is clear that the most difficult part is the first one. Indeed, precisely estimating the convergence rate of the algorithm requires knowledge of the shape of the operator Op, and therefore of the shape of the cost function. The three estimates for the rate of convergence are guidelines, and only experience will show which ones are most useful in practice (see Chapter 5). In what follows, we emphasize characterizing the implementation speed, assuming that the convergence rate is known precisely enough, hopefully thanks to one of our three estimates.

4.5.2 Characterizing the Execution Time

As already mentioned in Section 4.2.3, the end-user is interested in knowing how long the parallel algorithm will run until convergence is attained. Convergence is attained when the initial error has been divided by a factor of 10^{ω} where $\omega \in \mathbb{N}$ is specified by the user. In Section 4.2, we proposed three estimates of the asymptotic rate of convergence for the algorithm. In what follows, each of those estimates can be used. Let $\mathcal{R}$ be the chosen estimate of the asymptotic convergence rate and let $\mathcal{N}$ be an estimate of the number of iterations that are to be performed by the algorithm before convergence. We will assume that:

$$\mathcal{N} = \frac{\omega}{\mathcal{R}}.$$

This estimate of the required number of iterations is asymptotic and is valid only when the number of iterations is large. Let Θ be the time to convergence for a run of the parallel iterative algorithm. In the next four sections, we describe how our performance characterization scheme can be used to obtain several levels of estimations for Θ. These levels are called 1, 2 and 3 and offer different information on the probability distribution of Θ.

4.5.3 Level 1 Performance Characterization

Level 1 performance characterization has been done in Section 4.3.3. It uses the mean vector $S = \binom{N}{\Phi}$ shown in the left dashed box in Figure 4.20. It says that Θ can be estimated by Θ_1

$$\boxed{\Theta_1 \triangleq \frac{\mathcal{N}}{s},} \qquad (4.5.14)$$

where again $s = \frac{N}{\Phi}$. In this case, our performance characterization consists of one single expected value.

4.5.4 Level 2 Performance Characterization

Level 1 provides an estimate of the mean of Θ. Level 2 is concerned with approximating its standard deviation. Our implementation of the model gives us the probability distribution of the RV $S(k)$. Let S^{cov} denote the 2×2 covariance matrix of $S(k)$. S^{cov} is of course easy to compute since the entire distribution function is known. Let us recall that we are assuming the observations of $S(k)$ to be i.i.d. The covariance matrix of S_n, denoted by S_n^{cov}, is given for each n by:

$$S_n^{cov} = n \times S^{cov}.$$

One can now compute the standard deviation of the execution time as follows. Convergence is attained when some number of iterations, $\mathcal{N}$, have been performed. We are therefore interested in computing the variance of the execution time knowing that $\mathcal{N}$ iterations have been performed. The execution time after n algorithm phases is given by the first component of S_n, whereas its second component gives the number of iterations performed so far. If we assume that on average and in the long run N iterations are performed at each algorithm phase, then convergence is attained after $\mathcal{N}/N$ algorithm phases. Our goal is then to compute the standard deviation of the conditional distribution of the first component of $S_{\mathcal{N}/N}$ knowing its second component. Due to the "large number" and i.i.d. assumptions, we approximate the distribution of $S_{\mathcal{N}/N}$ by a Normal (Gaussian) multivariate distribution with covariance matrix $S_{\mathcal{N}/N}^{cov}$. This is of course inspired by the Central Limit Theorem [30]. It is well known that the conditional distribution of the second component of $S_{\mathcal{N}/N}$ conditioned on its first component is Normal. Furthermore, its standard deviation does not depend on the value of the second component. If the covariance matrix is written as:

$$S^{cov}_{\mathcal{N}/N} = \begin{bmatrix} \sigma_X^2 & \sigma_{XY} \\ \sigma_{XY} & \sigma_Y^2 \end{bmatrix},$$

then the standard deviation that we need to compute is given by:

$$\sigma = \sigma_Y \sqrt{1 - (\frac{\sigma_{XY}}{\sigma_X \sigma_Y})^2}. \qquad (4.5.15)$$

This result is available in [30] for instance. σ is our estimate for the standard deviation of Θ.

4.5.5 Level 3 Performance Characterization

Now that we have estimates for the mean and the standard deviation of Θ, one may wonder about the tails of Θ's distribution. Similarly to Section 4.4.3, we use LDT to obtain estimates of rare events corresponding to extreme behaviors of the algorithm (particularly fast or particularly slow). If we assume that on average and in the long run N iterations are performed at each algorithm phase, then convergence is attained after $\frac{\mathcal{N}}{N}$ algorithm phases. The computation in 4.4.3 gives us an asymptotic estimate of the following probabilities as:

$$\mathcal{P}\{s_n \geq s + \epsilon\} = e^{-n \times r_\epsilon^+ + o(n)}$$

$$\mathcal{P}\{s_n \leq s - \epsilon\} = e^{-n \times r_\epsilon^- + o(n)},$$

where s_n is the algorithm speed observed after n algorithm phases, and r_ϵ^+ and r_ϵ^- are defined in equation 4.4.13. But Θ can be approximated as:

$$\Theta = \frac{\mathcal{N}}{s_{\frac{\mathcal{N}}{N}}},$$

where $\mathcal{N}$ is the number of iterations to be performed to achieve convergence. One can then write that:

$$\begin{cases} \mathcal{P}\{s_{\frac{\mathcal{N}}{N}} \geq s + \epsilon\} & = e^{-r_\epsilon^+ \frac{\mathcal{N}}{N} + o(\mathcal{N})} \\ \mathcal{P}\{s_{\frac{\mathcal{N}}{N}} \leq s - \epsilon\} & = e^{-r_\epsilon^- \frac{\mathcal{N}}{N} + o(\mathcal{N})}, \end{cases}$$

leading to:

$$\begin{cases} \mathcal{P}\{\frac{\mathcal{N}}{\Theta} \geq s + \epsilon\} & = e^{-r_\epsilon^+ \frac{\mathcal{N}}{N} + o(\mathcal{N})} \\ \mathcal{P}\{\frac{\mathcal{N}}{\Theta} \leq s - \epsilon\} & = e^{-r_\epsilon^- \frac{\mathcal{N}}{N} + o(\mathcal{N})}, \end{cases}$$

and finally:

$$\boxed{\begin{cases} \mathcal{P}\{\Theta \leq \frac{\mathcal{N}}{s+\epsilon}\} & = e^{-r_\epsilon^+ \frac{\mathcal{N}}{N} + o(\mathcal{N})} \\ \\ \mathcal{P}\{\Theta \geq \frac{\mathcal{N}}{s-\epsilon}\} & = e^{-r_\epsilon^- \frac{\mathcal{N}}{N} + o(\mathcal{N})}. \end{cases}} \qquad (4.5.16)$$

Equation 4.5.16 is an easy way to compute the tails of the probability distribution of Θ.

Note that the 3 level of characterizations are related to each other as they all depend on $\mathcal{N}$. In the next section, we give an example and characterize the performance of some implementations of an iterative algorithm in a given distributed environment.

4.5.6 A Complete Example

Once again, we consider the distributed environment of example 3 in Section 4.1.2. We assume that the matrix associated to the contracting operator Op has a spectral radius $\rho = 0.9$ and we consider the performance of implementations of this algorithm for $A_1 = A_2 = A_3 = 1$ and $B_i = 0, 1, 2$ for $i = 1, 2, 3$. We assume that the user defines convergence with a single integer ω: convergence is obtained when the initial error has been divided by a factor of 10^ω. In this example, we chose $\omega = 4$.

Convergence Rate

The characterization for the convergence rate comes from the computation depicted in Figure 4.9. The values of our three estimates as well as Baudet's for the three different implementations are shown in Table 4.1. We recall that $\underline{\mathcal{R}^*}$ is a worst case estimate, $\widehat{\mathcal{R}^*}$ can be interpreted as a mean estimate and $\overline{\mathcal{R}^*}$ is an ideal estimate.

Level 1 Characterization

Level 1 characterization is very easy to perform. It provides an estimate of Θ, the mean time in seconds to convergence. This estimate is denoted Θ_1 and computed according to equation 4.5.14. Table 4.2 shows the numerical value of Θ_1 for each convergence rate estimate and each implementation. If the mean estimate $\widehat{\mathcal{R}^*}$ is taken to be the most informative one, then it seems that a synchronous implementation will be more efficient than an asynchronous one.

Table 4.1: Convergence rate estimates for each implementation

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$	Baudet
1	0	0.045757	0.045757	0.045757	0.045757
1	1	0.021964	0.025079	0.045757	0.015252
1	2	0.014532	0.023804	0.045757	0.009151

Table 4.2: Level 1 characterization : Mean of Θ (in seconds) for various convergence rates

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$	Baudet
1	0	1081	1081	1081	1081
1	1	1272	1141	610	1832
1	2	1828	1116	580	2903

Level 2 Characterization

Table 4.3 shows the numerical results provided by Level 2 characterization.

Level 3 Characterization

Let us compute, for instance, the probabilities that the observed execution time deviates from Θ_1 by 2, 5, 10 or 20 percent. One can use equation 4.5.16 to this end, with $\epsilon = s/99, s/9, 2 \times s/8, 3 \times s/7$ for the cases where Θ is 2%,5%, 10% or 20% bigger than Θ_1. Similarly, the values $\epsilon = s/101, s/11, 2 \times s/12, 3 \times s/13$ are for the cases when Θ is 2%,5%, 10% or 20% smaller than Θ_1. This is due to some algebra and the fact that $\Theta_1 = \mathcal{N}/s$. Notice that a deviation of 2% is not strictly speaking a *rare event*. Equation 4.5.16 gives the content of Table 4.4, with r_ϵ^+ and

Table 4.3: Level 2 characterization : standard deviation of Θ

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$
1	0	17.54	17.54	17.54
1	1	15.97	14.95	11.06
1	2	17.67	13.80	9.96

Table 4.4: Asymptotic deviation probabilities.

$\mathcal{P}\{\Theta \leq 0.98 \times \Theta_1\}$	$\sim$	$e^{-r^{+}_{2 \times s/98} \frac{\mathcal{N}}{N}}$	$\mathcal{P}\{\Theta \geq 1.02 \times \Theta_1\}$	$\sim$	$e^{-r^{-}_{2 \times s/102} \frac{\mathcal{N}}{N}}$
$\mathcal{P}\{\Theta \leq 0.95 \times \Theta_1\}$	$\sim$	$e^{-r^{+}_{5 \times s/95} \frac{\mathcal{N}}{N}}$	$\mathcal{P}\{\Theta \geq 1.05 \times \Theta_1\}$	$\sim$	$e^{-r^{-}_{5 \times s/105} \frac{\mathcal{N}}{N}}$
$\mathcal{P}\{\Theta \leq 0.9 \times \Theta_1\}$	$\sim$	$e^{-r^{+}_{s/9} \frac{\mathcal{N}}{N}}$	$\mathcal{P}\{\Theta \geq 1.1 \times \Theta_1\}$	$\sim$	$e^{-r^{-}_{s/11} \frac{\mathcal{N}}{N}}$
$\mathcal{P}\{\Theta \leq 0.8 \times \Theta_1\}$	$\sim$	$e^{-r^{+}_{2 \times s/8} \frac{\mathcal{N}}{N}}$	$\mathcal{P}\{\Theta \geq 1.2 \times \Theta_1\}$	$\sim$	$e^{-r^{-}_{2 \times s/12} \frac{\mathcal{N}}{N}}$

r^{-}_{ϵ} defined in equation 4.4.13.

Computing the right-hand sides in Table 4.5 is possible thanks to our knowledge of the rate function. Table 4.5 shows the values (computed with Matlab) of these asymptotic probabilities for each implementation and each convergence rate estimate. The table shows that the probabilities are obviously over-estimated for small deviations. Let us consider the values in the table for deviations of 2% and -2% from the mean. It is easy to see that, according to the table:

$$\mathcal{P}\{\Theta \leq 0.98 \times \Theta_1\} + \mathcal{P}\{\Theta \geq 1.02 \times \Theta_1\} > 1!$$

This is due to the fact that a deviation from the mean by 2% is not a rare event for a number of sample as small as $\mathcal{N}/N$. More formally, when ϵ is *small*, then r^{+}_{ϵ} and r^{-}_{ϵ} are also small. In our approximation, we fixed the number of algorithm phases (to $\mathcal{N}/N$). Therefore, the term $o(n)$ in the Large Deviation asymptotic estimate is not negligible and causes Table 4.5 to give incoherent probability values for small deviations. In fact, LDT tells us that $e^{-r^{-}_{\epsilon} \times n}$, for instance, is an upper bound on

Table 4.5: Level 2 deviations from Θ_1 by $\pm 2\%$,$\pm 5\%$, $\pm 10\%$ and $\pm 20\%$.

Impl.		$\underline{\mathcal{R}^*}$		$\widehat{\mathcal{R}^*}$		$\overline{\mathcal{R}^*}$	
A_i	B_i	-2%	+2%	-2%	+2%	-2%	+2%
1	0	0.4652	0.4690	0.4652	0.4690	0.4652	0.4690
1	1	0.5650	0.6133	0.6066	0.6517	0.7603	0.7908
1	2	0.5606	0.5585	0.7023	0.7008	0.8321	0.8311
Impl.		$\underline{\mathcal{R}^*}$		$\widehat{\mathcal{R}^*}$		$\overline{\mathcal{R}^*}$	
A_i	B_i	-5%	+5%	-5%	+5%	-5%	+5%
1	0	0.0079	0.0090	0.0079	0.0090	0.0079	0.0090
1	1	0.0256	0.0588	0.0404	0.0837	0.1723	0.2567
1	2	0.0207	0.0444	0.0936	0.1493	0.2916	0.3718
Impl.		$\underline{\mathcal{R}^*}$		$\widehat{\mathcal{R}^*}$		$\overline{\mathcal{R}^*}$	
A_i	B_i	-10%	+10%	-10%	+10%	-10%	+10%
1	0	2.3808e-09	7.0465e-09	2.3808e-09	7.0465e-09	2.3808e-09	7.0465e-09
1	1	6.3040e-08	4.2623e-05	4.9426e-07	1.4875e-04	3.4972e-04	0.0080
1	2	2.7317e-08	1.2820e-05	2.4130e-05	0.0010	0.0040	0.0279
Impl.		$\underline{\mathcal{R}^*}$		$\widehat{\mathcal{R}^*}$		$\overline{\mathcal{R}^*}$	
A_i	B_i	-20%	+20%	-20%	+20%	-20%	+30%
1	0	2.8013e-38	1.0457e-33	2.8013e-38	1.0457e-33	2.8013e-38	1.0457e-33
1	1	6.2358e-38	8.2600e-15	2.6064e-33	4.6366e-13	1.3837e-18	1.7376e-07
1	2	1.0324e-39	6.2265e-17	1.5831e-24	1.2783e-10	4.1529e-13	7.1319e-06

the probability of deviation from the mean after n observations. The values in the table are therefore upper bounds on the deviation probabilities and we expect them to be fairly tight for large deviations. According to the table, the tail of Θ's distribution on the left side (corresponding to particularly slow executions) is *heavier* than on the right side (particularly fast executions). This implies that the distribution function of Θ is not symmetric. Chapter 5 will present some tentative results that aim at quantifying this dissymmetry.

Figure 4.21 summarizes what elements of the probability distribution of Θ can be approximated by our different characterization levels.

4.6 Conclusion

In this chapter, we have isolated the components of our model that are used to perform a performance analysis of parallel iterative algorithms. The analysis of the wavefront Markov chain (see Section 4.1) is the basis of the performance analysis. Once the wavefront behavior is quantified, it is possible to obtain estimates for the algorithm rate of convergence (see Section 4.2) and the implementation speed (see Section 4.3). The analysis of the implementation speed can benefit from the use of Large Deviation Theory, as explained in Section 4.4. Finally, we have summarized our characterization techniques in Section 4.5 and given an example. There are many ways in which the material presented in this chapter can be improved or extended. Such developments are left for future work and are described in detail in Chapter 6.

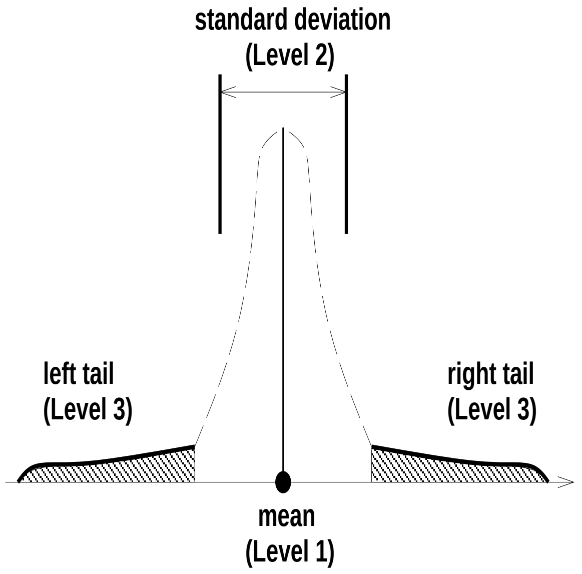

Figure 4.21: Probability distribution of Θ

Chapter 5

Model Validation

In the preceding chapter, we have seen how the model described in Chapter 3 can be used to obtain performance characterizations for parallel iterative algorithms in distributed environments. In this chapter, we examine these characterizations and draw conclusions about their validity. First, we present simulation results for a given distributed environment and iterative algorithm. Second, we present experimental results for a real distributed environment and real runs of an iterative algorithm.

5.1 Simulation Results

The first step when trying to validate our performance characterizations is to perform simulations. We can make those simulations satisfy our basic assumptions exactly and thereby verify that our models yield coherent results.

5.1.1 The Experiment

We implemented a simulation program that simulates any iterative algorithm in any distributed environment. The program can be used to easily simulate a large number of the algorithm runs. For each of these runs, the program produces the execution time, the number of iterations to convergence and the error reduction of the solution vector. Let us give more details about the specifics of the simulation that we performed to obtain numerical results.

The Simulated Distributed Environment

The simulated distributed environment consists of three processors. Those processors are identical but with different workload distributions. Figure 5.1 shows the three distributions of the solution vector update times. The processors are interconnected with a network that delivers constant performance of 0.001 seconds per message. This environment is fairly simple, in order to make the results easier to interpret.

The Iterative Algorithm

We chose to simulate a Gradient algorithm (see Section 2.1.2). The cost function is multi-polynomial, from $\mathbb{R}^{30}$ to $\mathbb{R}$, and the step-size, γ, is equal to 0.005. The operator of the iterative algorithm is then a contracting operator according to definition 2.3.1. The matrix A associated to this contracting operator has a spectral

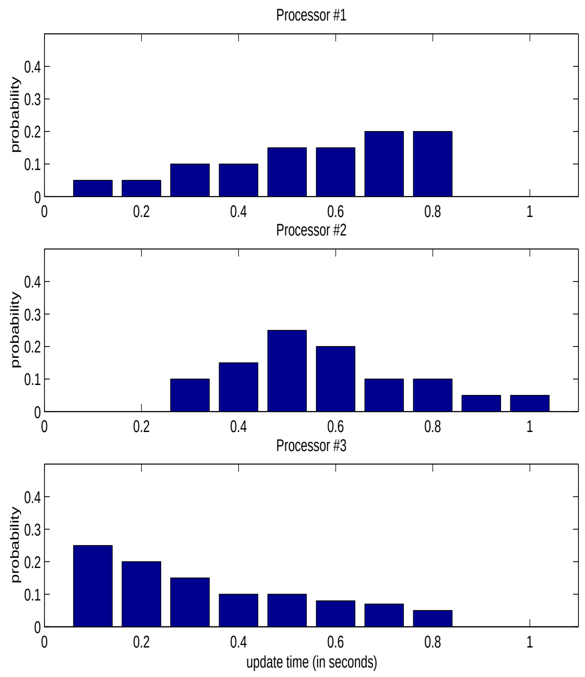

Figure 5.1: Update time distributions for the three processors

radius of 0.7. The initial guess of the solution vector is the same for each run of the algorithm.

In what follows, we present the results of the simulation for different implementations. We then discuss all three characterization levels accordingly.

5.1.2 Synchronous Implementation

Figure 5.2 shows the simulation results for a synchronous implementation of the iterative algorithm as well as the performance characterizations. The empirical distribution of the algorithm execution time is shown on the figure as a bar diagram. The empirical mean is shown as a vertical solid line and the empirical

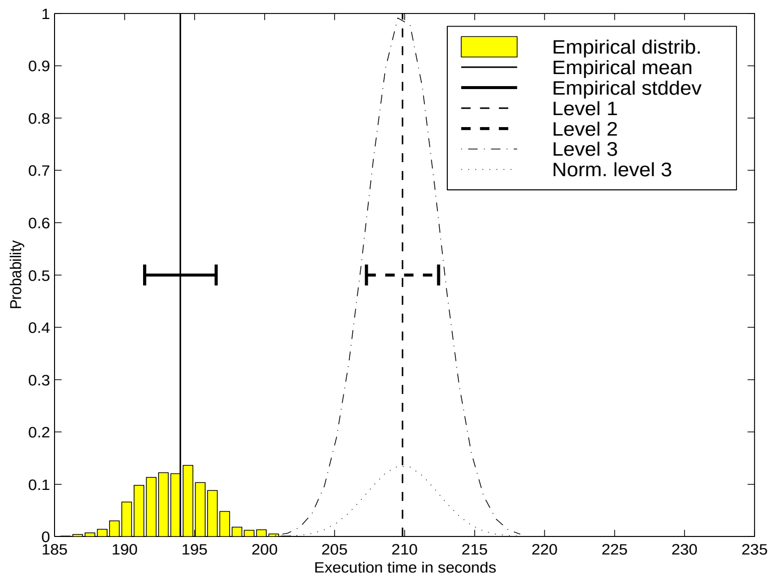

Figure 5.2: Simulation vs. characterization for a synchronous implementation

standard deviation is represented as a horizontal line segment on each side of the empirical mean. We recall from Section 4.2.3 that our three estimates for the rate of convergence of the algorithm are all equal to Baudet's estimate, hence, there is only one set of characterizations on the figure. Level 1 characterization is shown as a dashed vertical line. Level 2 characterization is shown as two horizontal dashed line segments on each side of level 1. Level 3 characterization is represented by a dash-dot curve. A dotted curve shows the normalized level 3. As we have seen in Section 4.5.6, level 3 is not a probability density function as the area under its curve is greater than 1. However, it is possible to normalize the curve so that it becomes a probability density function. Note that we have not given any formal justification that this normalized curve should fit the simulation data. Such a justification is left for future investigation. Most graphs in this chapter will be similar to Figure 5.2.

Figure 5.3 shows the same data as Figure 5.2, but the level 1 characterization has been made equal to the empirical mean, so that it is easy to observe the level 2 and level 3 characterization versus the shape of the empirical distribution. Thorough comments on these curves will be given in Sections 5.1.4, 5.1.5, and 5.1.6.

5.1.3 Asynchronous Implementations

Let us now turn to the simulation results for asynchronous implementations. We consider two such implementations as in the examples of Chapter 4: first, an implementation for which $A_i = 1$ and $B_i = 1$ for $i = 1, 2, 3$ in terms of definition 3.4.1; second, an implementation for which $A_i = 1$ and $B_i = 2$ for $i = 1, 2, 3$. This second implementation can be seen as "more asynchronous" than the first one.

Figure 5.4 shows the empirical distribution, mean and standard deviation of the algorithm execution time (it is similar to Figure 5.2). However, four set of performance characterizations are shown. Indeed, the three estimates of the rate of convergence defined by equation 4.2.3 have here three different values. These values have been computed as in Section 4.2.3 (see Figure 4.9). On Figure 5.4, we also show the performance characterization based on Baudet's convergence rate estimate.

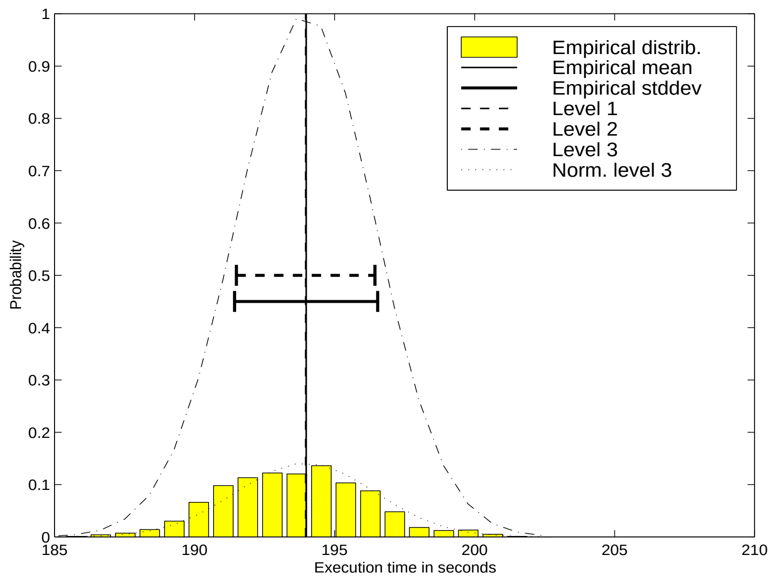

Figure 5.3: Adjusted curve for a synchronous implementation

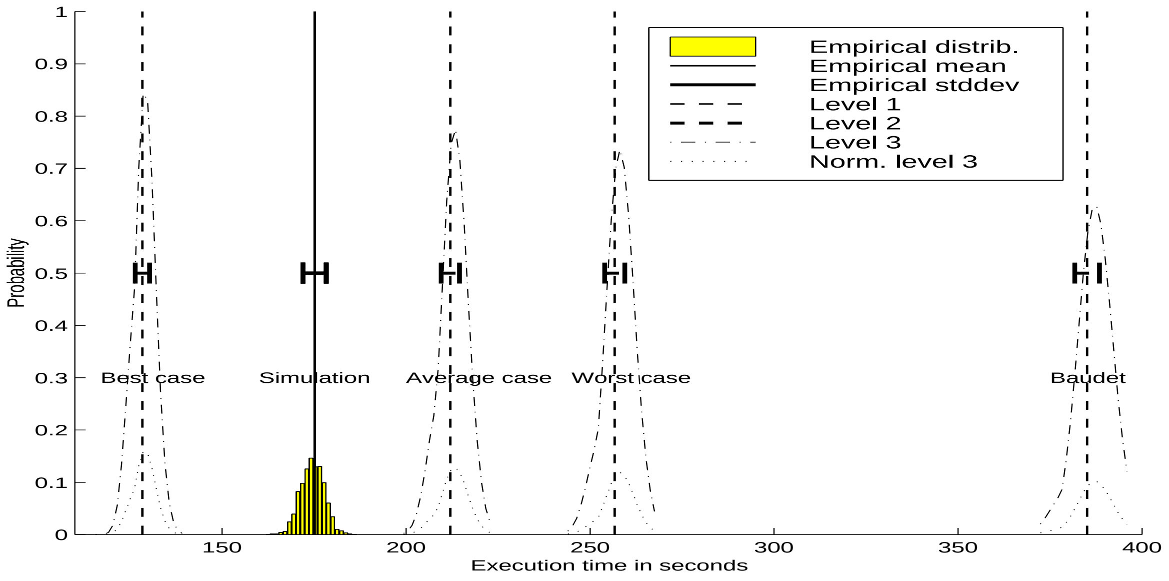

Figure 5.4: Simulation vs. characterizations for the first asynchronous implementation

Figure 5.5 is similar to Figure 5.3: level 1 characterization has been made equal to the empirical mean so that it is easier to observe the level 2 and level 3 characterizations. This adjusted curve can be seen as the characterization for a perfectly accurate convergence rate estimate. In the following sections, we will compare the empirical distribution to characterizations for our three convergence rate estimates, Baudet's estimate and for the perfectly accurate estimate aforementioned.

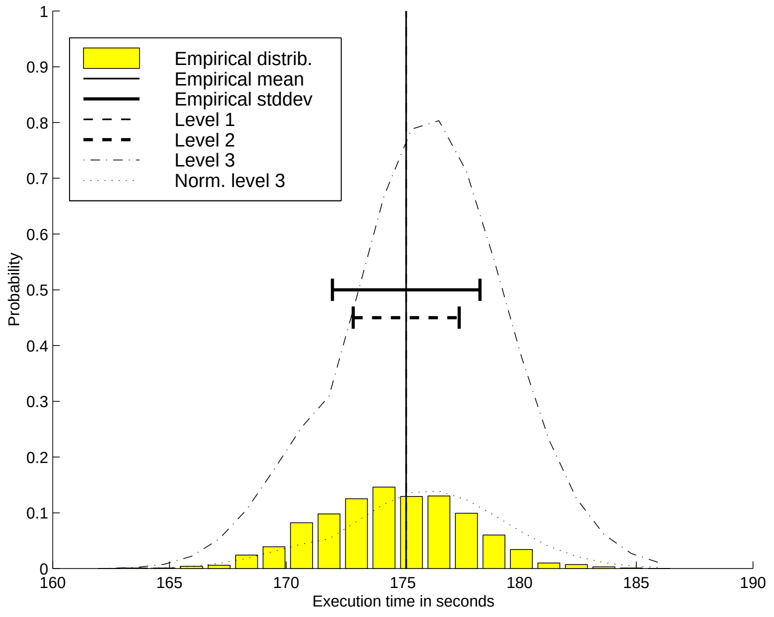

Figure 5.5: Adjusted curve for the first asynchronous implementation

Figure 5.6 shows the same results as Figure 5.4, but for the second asynchronous implementation. Figure 5.7, also for the second implementation, is similar to Figure 5.5 with the adjusted level 1 characterization. In the following three sections, we discuss each characterization level in detail using the graphs for the synchronous and asynchronous implementations.

5.1.4 Discussion of the Level 1 Characterization

Level 1 characterization is an estimate of the mean execution time for the iterative algorithm in a given distributed environment. In the case of the synchronous implementation, our three estimates of the algorithm convergence rate are all equal to Baudet's estimate. As seen on Figure 5.2, level 1 characterization predicts a mean execution time of approximatively 210 seconds whereas the observed mean execution time is around 194 seconds. This means that there is around 8% error between observed and predicted mean. This difference can be explained rather easily. Indeed, our estimate of the rate of convergence, in terms of number of iterations to convergence, depends only on the spectral radius of the matrix associated to the contracting operator for a synchronous implementation. This spectral radius does not describe the entire *shape* of the operator, which depends itself on the shape of the cost function. In fact, two contracting operators with matrices of identical spectral radius can lead to different (but hopefully close) rates of convergence.

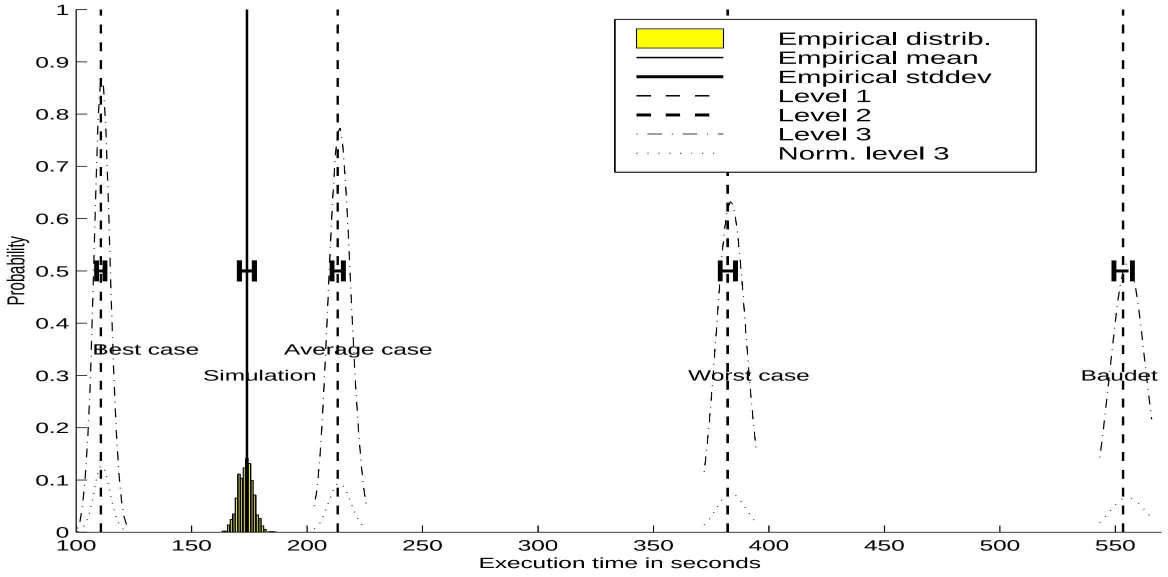

Figure 5.6: Simulation vs. characterizations for the second asynchronous implementation

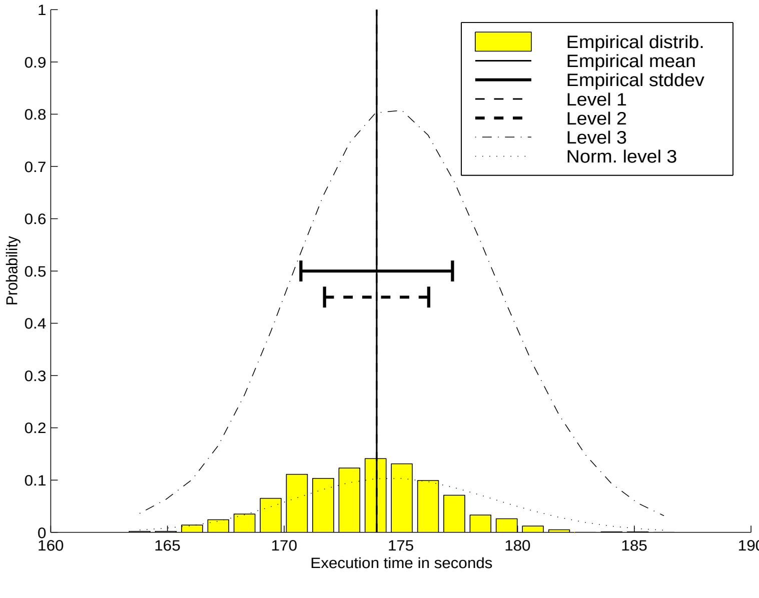

Figure 5.7: Adjusted curve for the second asynchronous implementation

Our estimate of the rate of convergence is a lower-bound on the actual convergence rate for all contracting operators with matrices of spectral radius 0.7. This explains why the level 1 mean execution time is larger than the observed one. This phenomenon occurs for any implementation, but it is especially easy to observe for a synchronous implementation since all the convergence rate estimates are equal.

For the asynchronous implementations, we already explained that we have obtained a set of four different characterizations. Each characterization yields a different level 1, depending on the convergence rate estimates. The values of these estimates are shown in Table 5.1 for all three implementations. As noted in Section 4.2.3,

$$\underline{\mathcal{R}^*} \leq \widehat{\mathcal{R}^*} \leq \overline{\mathcal{R}^*} \leq \mathcal{R}_{Baudet}.$$

Let us recall from Section 4.5.3 that for a convergence rate estimate $\mathcal{R}$, the level 1 characterization is computed as:

$$\Theta_1 = \frac{\omega}{\mathcal{R}s},$$

Table 5.1: Convergence rate estimates and observed convergence rate

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$	Baudet	Observed
1	0	0.0132	0.0132	0.0132	0.0132	0.0143
1	1	0.0.0066	0.0080	0.0132	0.0044	0.0097
1	2	0.0038	0.0069	0.0132	0.0026	0.0084

where ω is the user's convergence criterion and s the average speed of the implementation in terms of number of iterations per seconds, as computed thanks to our stochastic model. As explained in Section 4.2.3, the gaps between the four convergence rate estimates depend on the variances of the RVs N^i for $i = 1, 2, 3$. This variance depends itself on the asynchronicity of the implementation and on the distributed environment. The gaps increase when the asynchronicity of the implementation increases. This comes directly from the definition of N^i. For a synchronous implementation, we have seen that the gaps are reduced to zero. For an asynchronous implementation, if the probability distribution of the solution vector update times on the different processors are small, then the gaps between the different level 1 characterizations are small. Conversely, if those variances are large then the gaps are also large (as for the synchronous implementations in our simulation). More intuitively, the gaps between the different level 1 characterizations increase with the number of additional updates that can be performed during the β sub-phases of the algorithm run.

The absolute positions of the level 1 characterizations depend on the operator of the iterative algorithm, the initial guess on the solution vector, the shape of the cost function, the distributed environment and the end-user's convergence criterion. The implementation of our model produces those four estimates for any implementation.

In this simulation, it appears that the observed execution times lie between the characterizations for our "best case" and "average case" estimates. As mentioned in Section 4.2.2, it is fairly difficult to generalize this result to any iterative algorithm in any distributed environment. Indeed, the influence of the shape of the cost function is difficult to quantify. It is clearly possible to contrive different cases where the cost functions, even though still leading to contracting operators, can have different influences on the execution time empirical distribution. This issue is part of the theoretical study of iterative methods and is outside the scope of this work.

However, for *classes* of cost functions, it is likely that the observed execution times will behave similarly for each function in a class. If several runs of an iterative algorithm are to be performed for functions in the same class, in the same distributed environment, then one can assume that the empirical mean of the execution times will be located at some fixed position, relatively to our four level 1 characterizations. For example, according to this simulation, using our "average case" estimate, $\widehat{\mathcal{R}^*}$, leads to 18% error on the mean execution time.

Let us make a last observation on Figures 5.4 and 5.6. One can observe that our level 1 characterizations for the "average" convergence rate estimate for the two asynchronous implementations are very similar: 212.03 seconds and 213.24 seconds respectively. This can be easily interpreted. In fact, due to the characteristics of the distributed environment, the second asynchronous implementation *rarely* has the opportunity to perform more than one additional update of the solution vector during the β sub-phases. It is in general fairly equivalent to the first asynchronous implementation, and this can be seen easily on the empirical distributions. Our stochastic model reflects this behavior when generating the level 1 characterizations. However, one can see on the two figures that the level 1 characterizations for the "best" and "worst" case convergence rate estimates are different for the two implementations. The second implementation exhibits much larger gaps between the different level 1 characterizations. This can be easily explained. Those two convergence rate estimates correspond to extreme cases. With non-null probability, but rarely in this simulation, the second asynchronous implementation can perform many more iterations on out-of-date data than the first implementation, depending on the distributed environment. Therefore, the second implementation can exhibit more extreme behaviors than the first implementation, explaining the larger gaps in Figure 5.6. We had already stated this fact by saying that the gaps increase with asynchronicity.

5.1.5 Discussion of the Level 2 Characterization

Level 2 characterization provides an estimate of the standard deviation of the algorithm execution time. Table 5.2 shows the four level 2 characterizations, the level 2 characterization for the adjusted level 1, and the empirical standard deviations, all for our three implementations.

The first obvious observation to make on Table 5.2 is that the level 2 characteriza-

Table 5.2: Level 2 characterizations and observed standard deviation

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$	Baudet	Adjusted	Observed
1	0	2.57	2.57	2.57	2.57	2.47	2.55
1	1	2.75	2.50	1.94	3.37	2.27	3.16
1	2	3.31	2.47	1.94	3.37	2.23	3.25

tion increases for decreasing values of the convergence rate estimate. This is easy to explain from the definition of the level 2 characterization given in Section 4.5.4. Level 2 is in fact the standard deviation of a bi-variate Normal distribution whose covariance matrix is computed as $n \times S^{cov}$, where n is the number of algorithm phases to convergence and S^{cov} a fixed matrix. Therefore, according to equation 4.5.15, level 2 characterization grows linearly with the predicted number of algorithm phases. The number of phases of course decreases when the convergence rate estimate increases, which agrees with our observation.

Table 5.3 shows the error percentages between the level 2 characterizations and the observed standard deviations for all three implementations. Those errors are caused by two factors. First, since the convergence rate estimates are not equal to the actual convergence rate, the number of algorithm phases predicted and used to compute level 2 characterization is not equal to the actual number of algorithm phases. This is closely related to our earlier comment about the linear growth in equation 4.5.15. Furthermore, the level 2 characterization uses the "average" number of iterations performed per algorithm phase to compute

Table 5.3: Error between level 2 characterization and observed standard deviation

A_i	B_i	$\underline{\mathcal{R}^*}$	$\widehat{\mathcal{R}^*}$	$\overline{\mathcal{R}^*}$	Baudet	Adjusted
1	0	0.8%	0.8%	0.8%	0.8%	3.1%
1	1	12.97%	20.89%	38.61%	6.65%	28.16%
1	2	1.85%	24.00%	40.31%	3.69%	31.38%

the number of phases to convergence. This approximation certainly contributes to the errors reported in Table 5.3. But, as we can see in that table, there are still errors for the adjusted characterization. As we said in Section 5.1.4, the adjusted characterization can be seen as one for the actual convergence rate. It does not seem reasonable to attribute all the errors only to the use of the "average" number of iterations performed per algorithm phases. Therefore, there has to be another phenomenon other than the error in predicting the number of algorithm phases.

As we have already seen, the level 2 characterization is computed from a bi-variate Normal approximation. This approximation is motivated by the introduction of the sample average of i.i.d. observation of a RV, $S(k)$, of known distribution (see Section 4.5.4). If the number of samples is not large enough, then the distribution of the sample average might be far from the Normal approximation. Furthermore, the more asynchronous the implementation, the more values can be taken by $S(k)$. In the case of a synchronous implementation, this RV is actually mono-variate and this explains why there is a small error for the corresponding level 2 characterization in Table 5.3. Table 5.4 shows the level 2 characterization error for our first asynchronous implementation ($A_i = 1$ and $B_i = 1$ for $i = 1, 2, 3$) and for decreasing values of ϵ, the achieved error between the final solution vector and the real solution. It appears clearly that the level 2 characterization error decreases with ϵ. Decreasing ϵ corresponds to increasing the number of algorithm phases performed, and therefore getting closer to the bi-variate Normal distribution (as the number of samples of $S(k)$ increases). It seems difficult to precisely quantify the different factor contributions to the errors in Table 5.4.

5.1.6 Discussion of the Level 3 Characterization

Level 3 characterization is shown on the graphs as a dash-dot line. First, let us note that the graphical representation of level 3 in the figures in the beginning on this chapter is not complete. We have not plotted the level 3 curve for all points at which it is non-zero. This curve however is only non-zero on a bounded subset of $\mathbb{R}$. This subset is easily determined by our stochastic model to obtain values for the maximum and minimum observable implementation speeds in terms of

Table 5.4: Level 2 error for increasing values of ϵ

ϵ	Level 2	Observed	Error
10^{-5}	2.45	3.52	30.4%
5×10^{-6}	2.68	3.79	29.3%
10^{-6}	2.97	4.07	27.0%
5×10^{-7}	3.14	4.06	22.7%
10^{-7}	3.93	4.86	19.1%
5×10^{-8}	4.43	4.98	11.0%

number of iterations performed per algorithm phase.

The LDT allows us to estimate such probabilities as:

$$\mathcal{P}\{\Theta \leq \Theta_1 - \epsilon\} \quad \text{and} \quad \mathcal{P}\{\Theta \geq \Theta_1 + \epsilon\}.$$

Such estimations are explained in detail in Section 4.5.5. It is then possible to use those estimates to approximate such probabilities as:

$$\mathcal{P}\{x \leq \Theta \leq y\} \quad \forall x, y, \in \mathbb{R}.$$

This is how the dash-dot curve has been computed. According to LDT, this curve should be an upper bound of the empirical distribution. And this is exactly what can be observed on all the Figures 5.2 to 5.7. This upper bound is very loose for small deviations, as explained in Section 4.5.6, but should be a good approximation of the actual probabilities of large deviations of the execution time from its mean: Θ_1.

Furthermore, one can see on Figure 5.4 that the peak of the the level 3 characterization decreases as the estimate of the rate of convergence decreases. It is notably much lower for the rightmost characterization (corresponding to Baudet's convergence rate estimate) than for the leftmost one (our best case estimate). This is due to the fact that, as noted in Section 4.5.5, the Large Deviations computation depends on $\mathcal{N}$, which increases when the convergence rate estimate decreases.

This phenomenon will be observed in all following graphs that show performance characterizations for different convergence rate estimates.

5.1.7 Conclusion

Our different levels of characterizations give very satisfactory results for this simulation. The major issue is the accuracy of the convergence rate estimate. This was discussed in Section 5.1.4 as it is very noticeable for level 1. The level 2 characterization seems to provide a reasonable order of magnitude of the empirical standard deviation. The errors in level 2 were identified and explained in Section 5.1.5. Finally, level 3 characterization seems to be very easy to interpret, if not to compute. However, it is fairly difficult to check its validity for rare events. Indeed, rare events are very unlikely to occur in a simulation and their empirical probability will almost always be zero. Furthermore, if one rare event occurs during a simulation, its empirical probability is bound to be much higher than its actual probability due to a too small sample size. The simulation of rare event is in fact an active field of research and is outside the scope of this dissertation. (An important technique to obtain valid simulation results about the probability of occurrence of rare events is *Importance Sampling.* One can find further development and references on this subject in [26, 29].)

5.2 Experimental Results

This section presents actual experimental results obtained for a real implementation of an iterative algorithm running on a real network of workstations. After describing the setting of the experiment, we give results for the algorithm execution as observed during a time period of a week and a time period of 24 hours.

5.2.1 The Experiment

We implemented a parallel iterative algorithm in C using MPI [49] for inter-process communications. This implementation can be used on any number of processors, and its degree of asynchronism can be easily modified. Indeed, it takes as parameters the A_i and B_i values for $i = 1, .., p$ (see definition 3.4.1). It also takes as parameters the user's convergence criterion and other variables describing the iterative algorithm itself. Let us start with a description of the distributed environment.

The Distributed Environment

As for the simulation, we used three workstations. Those workstations are part of one of the laboratories available to students at the Department of Computer Science of the University of Tennessee. The workstations are Sun Sparc Ultra 1 interconnected by a standard 10 Mbps Ethernet network. Those workstations are being used by students for course-work as well as for personal research. The load of each of the processors and of the network therefore varies considerably throughout the day.

Figure 5.8 shows the empirical distributions of the solution vector update times as measured throughout our time period of a week (Nov. 17-24, 1997). One can see that the first processor is on average slower than the other two.

Figure 5.9 shows those empirical distribution as computed from measurements for a time period of only 24 hours (Nov. 26, 1997). The distributions are different from the ones on Figure 5.8. The first two processors have fairly similar distributions and are on average slower than the third one.

One can also see on the figure that we have discretized the distribution. This discretization will lead to approximations in our computations but is necessary in order to use our stochastic model. Increasing the level of refinement for the discretization leads to more accurate results as the discrete distribution approaches the real distribution. The level of refinement shown in the figure seems to be sufficient here to obtain interesting results. The messages exchanged among the processors are small (80 bytes) and the machines are on the same local area network, so it is not surprising that our measurement of the network traffic shows that the communication time distributions have *almost* a null variance, meaning that the communication times are hardly random. This is due to the fact that no user was saturating the network during our time period. We therefore used our model assuming that the network delivers constant performance.

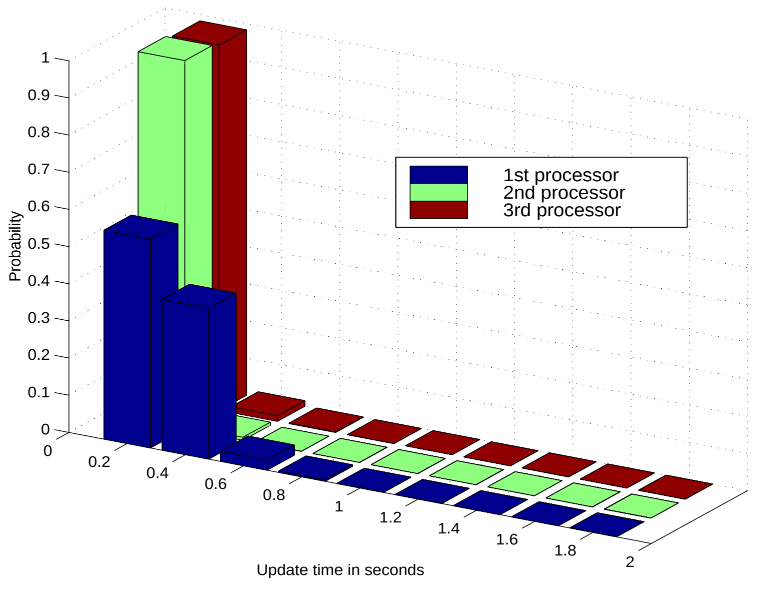

Figure 5.8: Measured update time distributions over a week

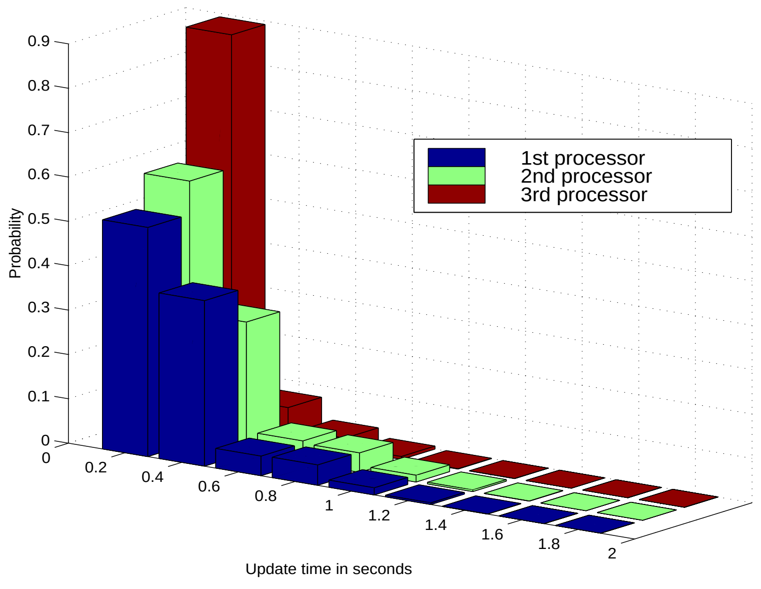

Figure 5.9: Measured update time distributions over 24 hours

Since the network communication times are 2 orders of magnitude smaller than the solution vector update times, this approximation should lead to no observable error.

The Iterative Algorithm

The iterative algorithm is the same as for the simulation: a Gradient descent with step-size 0.005 for a multi-polynomial function from $\mathbb{R}^{30}$ to $\mathbb{R}$. Here also, the initial guess on the solution vector is the same for each run of the algorithm. We measured the execution times for the same three implementations of the parallel iterative algorithm as for the simulation performed in Section 5.1. The different implementations were executed alternatively on the same three processors. It is therefore meaningful to compare their execution times throughout the time periods.

The following section offers fundamental observations on the raw measurements that we have collected and draws conclusions about the current limitations and domain of application of our stochastic model.

5.2.2 General Observations on the Measurements

Immediate Comparisons of the Different Implementations

Figure 5.10 shows the execution times observed throughout the one week time period for the synchronous implementation and for the first asynchronous imple-

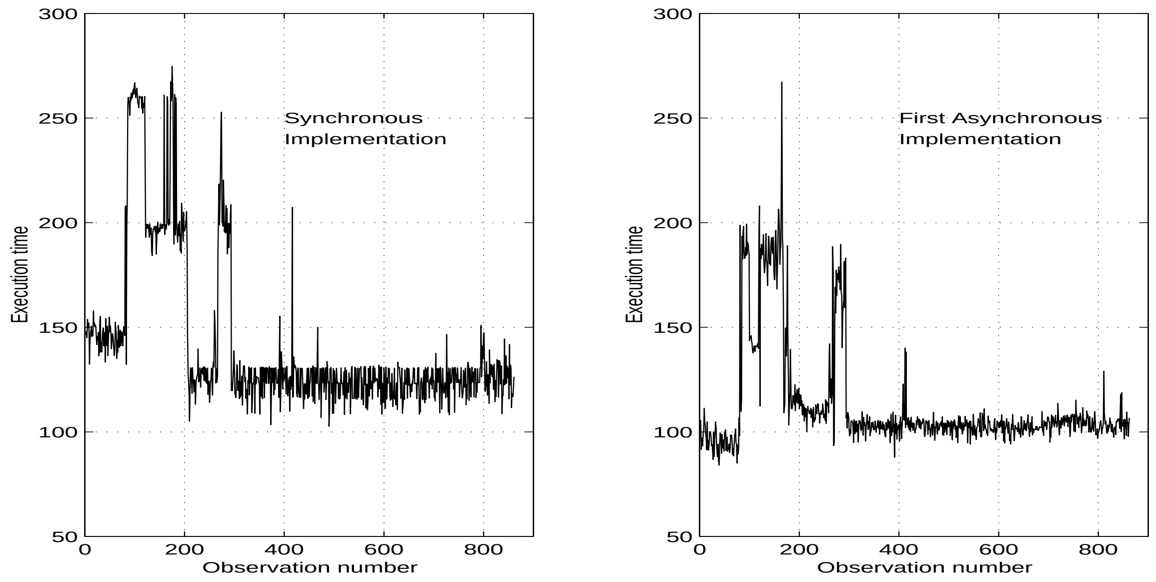

Figure 5.10: Execution time measurements over a week

mentation. This corresponds to 862 consecutive observations for each implementation. The measurements for the second asynchronous implementation are not shown because they would be difficult to distinguish from the ones of the first asynchronous implementation on that time scale. Instead, Figure 5.11 shows the differences in execution times throughout the time period between the first asynchronous implementation and the synchronous implementation, and between the first and the second asynchronous implementation.

The first observation to make is that the asynchronous implementations are generally more efficient than the synchronous one. According to the top graph in Figure 5.11, the first asynchronous implementation is up to 150 seconds faster than the synchronous implementation, and 30 seconds faster on average. Out of

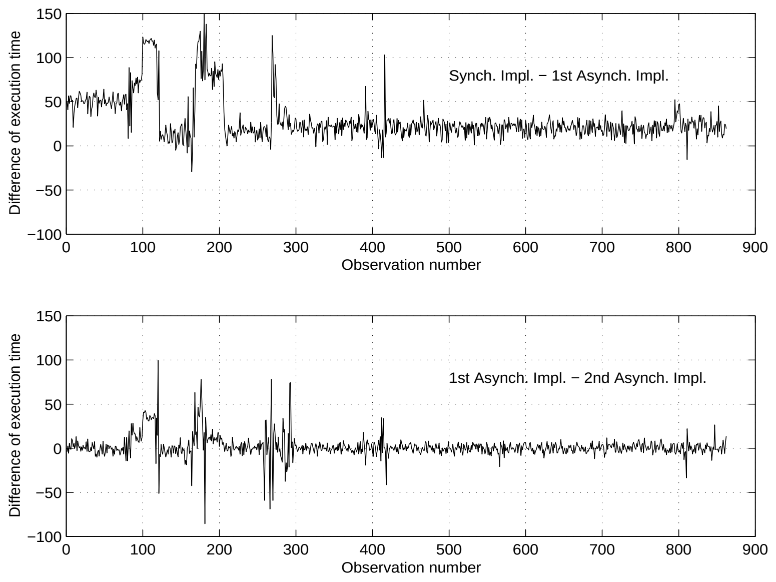

Figure 5.11: Differences in execution times

the 862 observations, only 13 are such that the synchronous implementation is actually faster than the asynchronous ones. This represents roughly 1.5% of the observations. In fact, since the experiment runs the implementations one after the other in a round-robin fashion, it is therefore highly likely that a quick change in the distributed environment will lead to a few incoherent comparisons between the different implementations.

The second observation concerns the two asynchronous implementations. Analyzing the data in the bottom graph of Figure 5.11, we compute that, "on average" the second implementation is faster than the first one by about 1.9 seconds. However, one can observe cases where the first implementation is 100 seconds slower than the second one, as well as cases where the second implementation is slower than the first one by up to 85 seconds. However, in only 15% of the observations is the absolute difference between between the two implementations more than 10 seconds.

It seems that, in this experiment, a good choice is to use an asynchronous implementation as opposed to a synchronous one. This is explained both by the nature of the distributed environment and by the nature of the iterative algorithm. Table 5.5 shows the observed mean and standard deviations of the execution times in seconds for each implementation: it seems that the mean and the standard deviation decrease with the asynchronicity, and in fact the operator of our iterative algorithm is smooth enough that the use of moderately out-of-date data is rarely detrimental to convergence. Several other research works include examples for which asynchronous implementations outperform synchronous ones [5, 7, 40]. Therefore, asynchronicity allows an implementation not only to achieve faster convergence, but also to adapt to the fluctuations of the distributed environment. This explains the decrease in mean and standard deviations witnessed in Table 5.5.

Table 5.5: Observed mean and standard deviations of the execution time

A_i	B_i	Mean	Std. Dev.
1	0	142.88	38.08
1	1	113.14	26.26
1	2	111.18	25.47

Burstiness of the Workload Distributions

Concerning the *shape* of the curves in Figure 5.10, a fundamental observation is that: the execution time is bursty. In fact, the distributed environment, and therefore the algorithm, behaves very differently at different times in our time period, for the system is in use for a variety purposes during the experimental runs.

In order to illustrate these different behaviors, Figures 5.12(a), (b) and (c) show three close-ups of the execution times for each implementation during short sub-periods of the one week time period. Table 5.6 shows the means and standard deviations for each implementation corresponding to the three sub-periods that we will call sub-periods (a), (b) and (c). Each of them is about two hours long. The standard deviations for the small sub-periods are much lower than those for the whole week shown in Table 5.5. Let us analyze the three sub-periods.

During sub-period (a), the gaps between the three implementations are large and the execution time of the synchronous implementation is high. This tends to suggest that the processors were heavily loaded, with a great load imbalance among them. The more asynchronous the implementation, the more resistant it is to this imbalance. During sub-period (b), the three implementations perform comparably. This suggests that the processor loads were uniform during the sub-period since the asynchronous implementations were not able to take advantage of

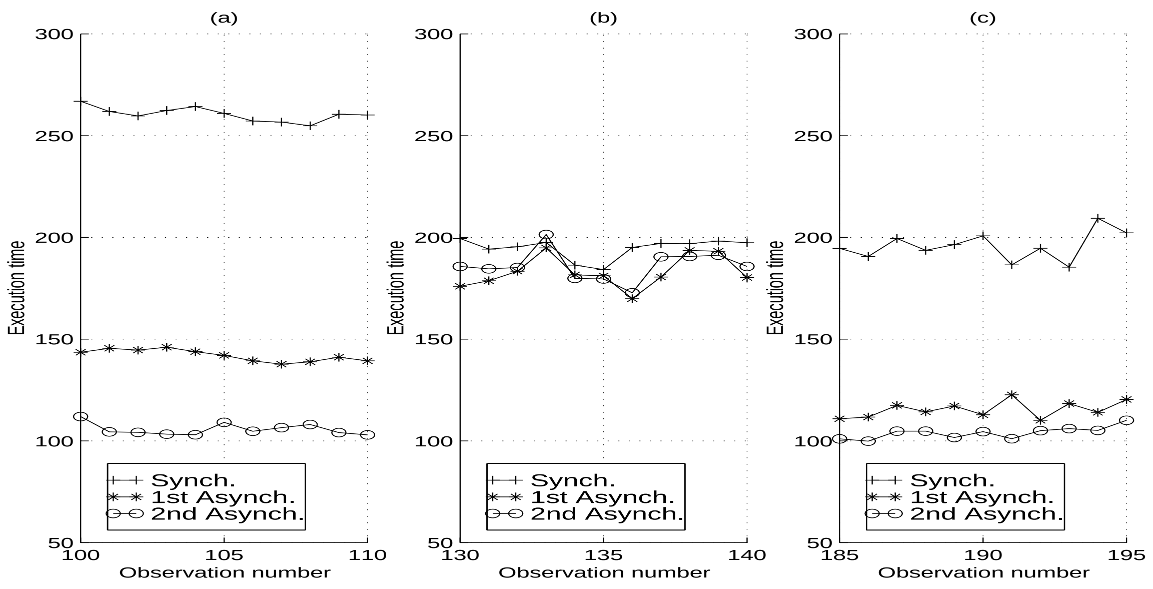

Figure 5.12: Different experimental behaviors throughout one week

Table 5.6: Observed means and standard deviations of the execution time

Impl.		sub-period (a)		sub-period (b)		sub-period (c)	
A_i	B_i	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
1	0	260.51	3.30	194.73	4.68	195.80	6.70
1	1	141.99	2.77	183.01	7.47	115.44	3.88
1	2	105.67	2.76	186.12	7.16	104.02	2.79

wasted CPU cycles. Sub-period (c) seems to be more "typical". The asynchronous implementations are close in performance and both faster than the synchronous one. The load imbalance among the processors was not as extreme as during sub-period (a) but more important than during sub-period (c). It is interesting to note that the performance of the synchronous implementation stays the same during sub-periods (b) and (c), wasting a lot of CPU cycles during sub-period (c). It is clear that the distribution depicted in Figure 5.8 does not characterize the workload of the processors during the whole week. The processor workload is bursty, causing the execution times to be bursty as well. It would be interesting to characterize precisely the burstiness of the workload. In [58, 34] it is shown that the Ethernet traffic exhibits a self-similar behavior [36, 39]. In [43], it is shown that the superposition of network traffics generated by Markov-modulated bursty sources generates self-similar traffic. If the workload on the different processors is modeled as such Markov-modulated sources, then it is possible that some elements in our model could be seen as self-similar random processes, meaning that they would be bursty on every time scale (or at least a great number of them as an approximation). Such a study is outside the scope of this dissertation and would require a much larger and detailed sample of measurements for a much longer time period. Such extensive data-sets are used in [58, 34]. Furthermore, precise measurements of the processor workloads fluctuations are needed in order to construct the Markov-modulated random process. These considerations are left for future work.

In what follows, we present results obtained with our model. We use the model to characterize the performance of the different implementations for the two time periods: first for the whole week, using the distribution shown in Figure 5.8; and second,, a time period of 24 hours during which the distributed environment exhibits more stable behavior, using the distribution shown in Figure 5.9. For reasons explained earlier, we expect the model to be fairly inaccurate for the first time period, whereas the second time period should lead to more satisfactory results. In the following sections, we present and comment on some of the results for both time periods. Many observations on the results are identical to the ones we have already made on the simulation results in Section 5.1. We will only describe here new phenomena not observed in the simulation.

5.2.3 The One Week Time Period

The Results

Figure 5.13(a) shows our characterization for the synchronous implementation versus the experimental distribution of the execution time. Figure 5.13(b) shows the same characterization when its convergence rate estimate is exactly equal to the observed convergence rate. The experimental distribution, shown as a bar diagram, is composed of three parts. Each of these parts corresponds to a different typical mean execution time. Such typical behaviors were seen clearly on Figure 5.10. Let us analyze each characterization level for those two figures.

The level 1 characterization on Figure 5.13(a) is around 119 seconds, whereas the observed mean is 143 seconds. This is a an error of approximatively 17%. This is explained by the burstiness of the workload distributions. Indeed, if the distributed environment were behaving as assumed by our model, meaning that the solution vector update times are i.i.d. with the distribution depicted in Figure 5.8, then the mean execution time for the parallel iterative algorithm would

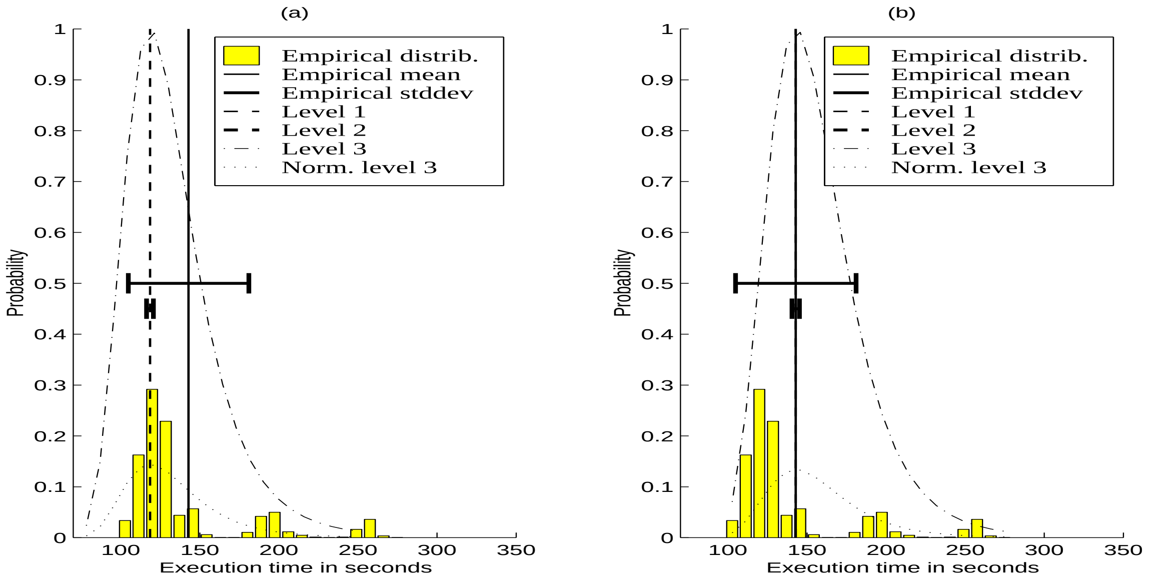

Figure 5.13: Experiment vs. Characterization for the synchronous implementation

be smaller than the observed one. Due to the workload burstiness, the algorithm can run extremely slowly for a large number of observations (see Figure 5.12(a)), contributing to increasing the mean execution time over the one week time period.

The level 2 characterization, as seen on Figure 5.13(b) is most striking. It is much smaller than the observed standard deviations, by a factor of 50! This was also expected since level 2 is very sensitive to our assumption about the distributed environment. Let us recall that level 2 is in fact the standard deviation of a bivariate Normal distribution of known covariance matrix (see Section 4.5.4). We have already seen in Table 5.4 that some of the errors in level 2 characterizations are due to too small a number of samples of the RV $S(k)$. In other words, the algorithm needs to go through "enough" phases for level 2 characterization to be accurate. Furthermore, level 2 assumes that the samples of $S(k)$ are independent so that the Central Limit Theorem is applicable. In this experiment, due again to the burstiness of the workloads, the samples of $S(k)$ are also bursty and therefore hardly independent.

Level 3 characterization seems to yield more satisfactory results than level 2. The upper bound on the distribution suggested by the dash-dot curve in Figure 5.13(b) is non-symmetric. It suggests that the distribution should have a heavier tail towards $+\infty$, meaning that extreme observations of the execution time correspond to slow executions. The experimental distribution appears to be roughly under the level 3 characterization, but it is difficult to interpret this observation due to the shape of that distribution. Here, the normalized level 3 curve (dotted line) does not fit the data as well as for the simulation in Section 5.1. It seems difficult to really assess the accuracy of level 3 from this experiment because of the extremely unstable distributed environment behavior throughout one week.

Figure 5.14 is similar to Figure 5.4. It shows the four characterizations of the performance of the first asynchronous implementation for each convergence rate estimate. As in the simulation, the experimental mean is located between the level 1 characterizations for our "best case" and "average case" convergence rate estimates. Figure 5.15 shows the adjusted characterization for the first asynchronous implementation where the level 1 characterization is equal to the observed mean execution time. As for the synchronous implementation, and for the same reasons,

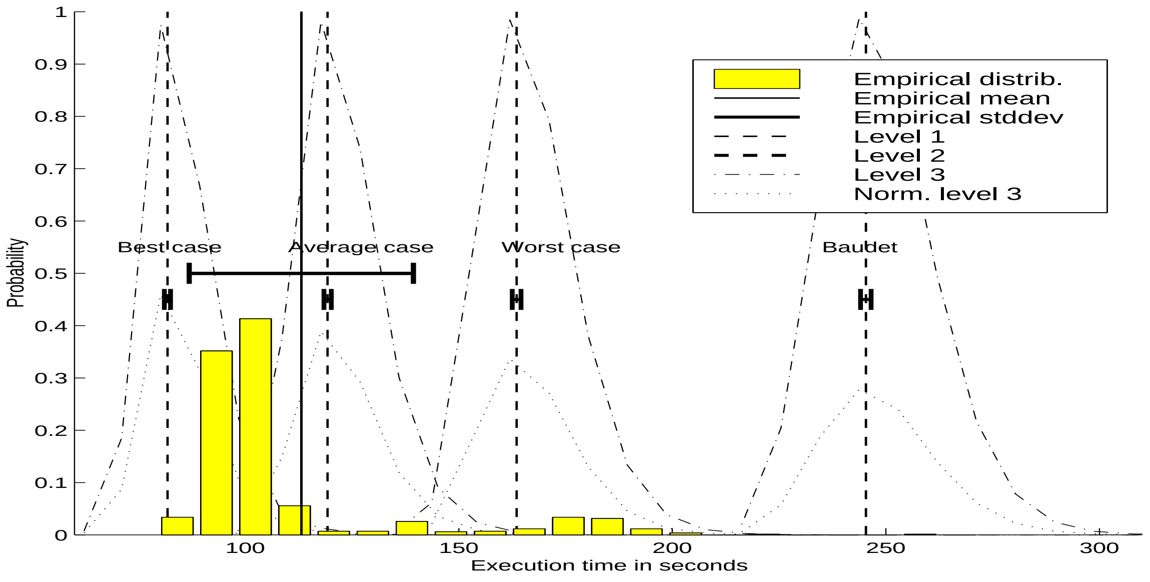

Figure 5.14: Experiment vs. Characterization for the first asynchronous implementation

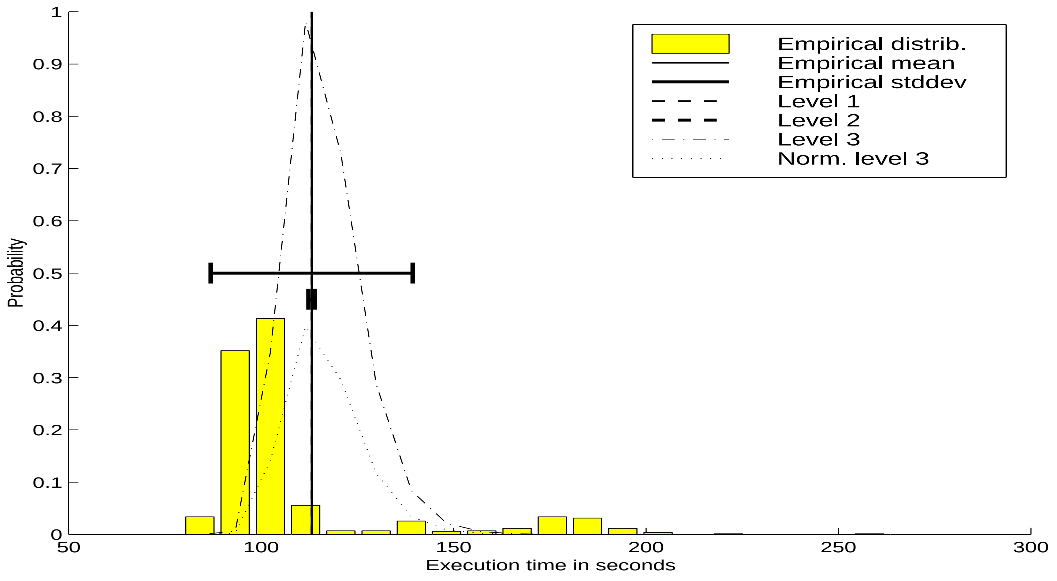

Figure 5.15: Adjusted curve for the first asynchronous implementation

level 2 characterization is much smaller than the observed standard deviation. The level 3 characterization exhibits the same dissymmetry as for the synchronous implementation. We can observe that the experimental distribution has a much heavier tail than predicted by the level 3 characterization due again to the bursty workloads of the processors.

The results for the second asynchronous implementation are not shown here. They are fairly similar to the results for the first asynchronous implementation. The only interesting observation has already been made in Section 5.1: the gaps between the level 1 characterizations for the different convergence rate estimates increase with asynchronicity.

Conclusion

The burstiness in workload for the one week time period is responsible for most of the phenomena that differentiate this experiment from the simulation. Level 2 characterization is most sensitive to this violation of our fundamental assumptions of stochastic independence. In the next section, we describe results obtained for a 24 hour time period where the burstiness of the workload is less dramatic.

5.2.4 The 24 Hour Time Period

The Results

Figure 5.16 shows the execution times for the parallel iterative algorithm throughout our 24 hour time period. The distributed environment exhibited a fairly stable behavior similar to the one shown in Figure 5.12(c), leading to much smoother measurements than the ones presented in Figure 5.10. Our model should yield more accurate results than for the one week time period.

Figure 5.17(a) shows the results of our model for the synchronous implementation. Figure (b) is the usual adjusted curve where the level 1 characterization is equal to the observed mean execution time. One can make three new observations on these figures.

First, the level 1 characterization in Figure (a) is larger than the observed mean. This can be explained by a comparison with the simulation results of Section 5.1 and the experimental results for the one week time period. For the simulation our independence assumption was completely satisfied and the level 1 was larger than the observed mean by 8%. For the one week time period, our independence assumption was largely violated, and the level 1 characterization was smaller than the observed mean by 16%. Here, the independence assumption is not satisfied since the workload is still bursty (but not so bursty as for the entire week). This explains why the level 1 characterization can be here larger (by 4%)

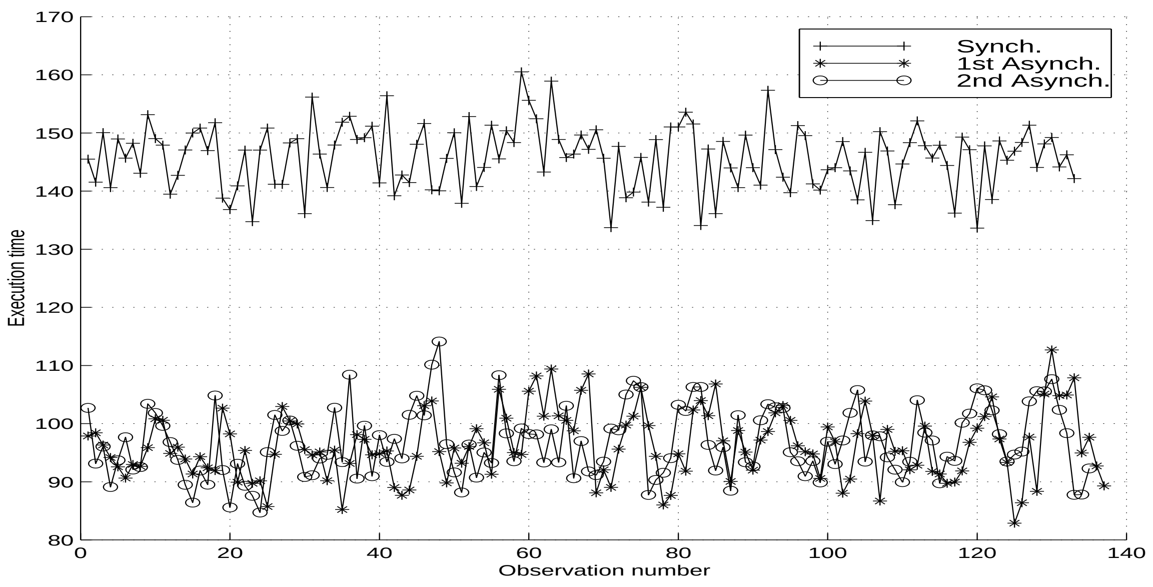

Figure 5.16: Measurements during 24 hours for the three implementations

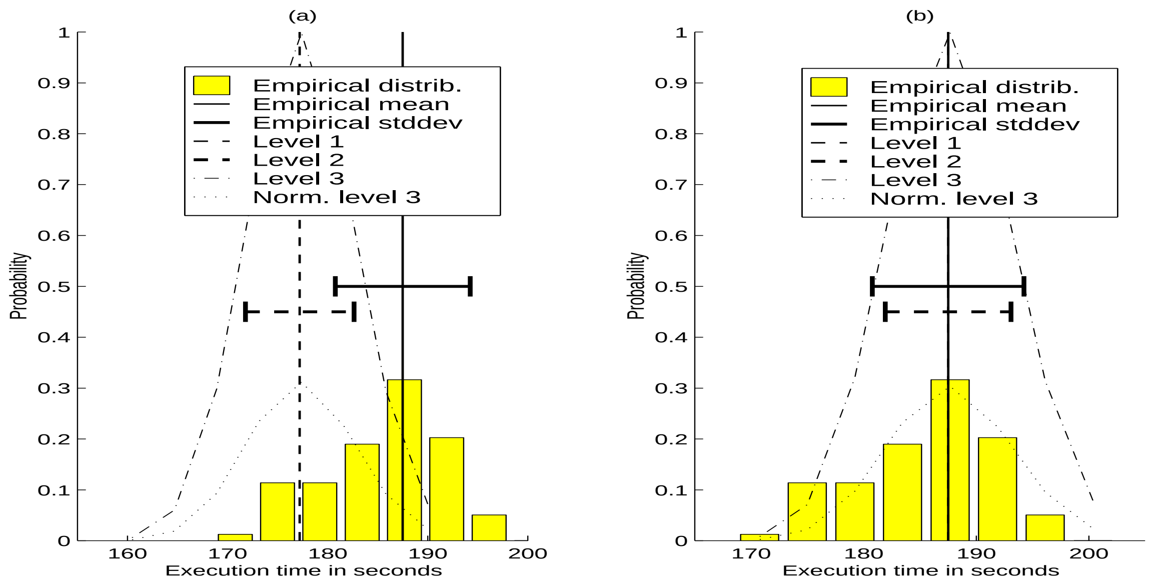

Figure 5.17: Experiment vs. Characterization for the synchronous implementation

than the observed mean. This statement is valid because the iterative algorithm is the same in all the simulations and experiments. Second, the level 2 error is much smaller than for the one week time period (about 27%) for the adjusted curve. This improvement is due to the decreased workload burstiness during this shorter time period. Third, level 3 characterization seems to be fairly accurate. The tail of the experimental distribution is a little heavier than suggested by level 3. Once again, this is to be attributed to workload burstiness and this explains why the level 3 error is much less important than for the one week time period.

Figure 5.18 shows our model's results for the first asynchronous implementation and can be analyzed exactly like Figure 5.14. Notice that the observed mean execution time is closer to our level 1 characterization for the "best" case convergence rate estimate than to the one for the "average" case estimate. This differs from both the simulation and the one week time period. It is difficult to quantify this behavior since it depends on the workload patterns of the processors in the distributed environment. Figure 5.19 shows the adjusted characterization for the first asynchronous implementation. One can observe that the error in level 2 characterization is here much smaller than for the one week time period ("only" 58%). As for the synchronous implementation, this is due to less extreme violations of our stochastic independence assumptions. Level 3 characterization is also more accurate than for the one week time period for the same reason.

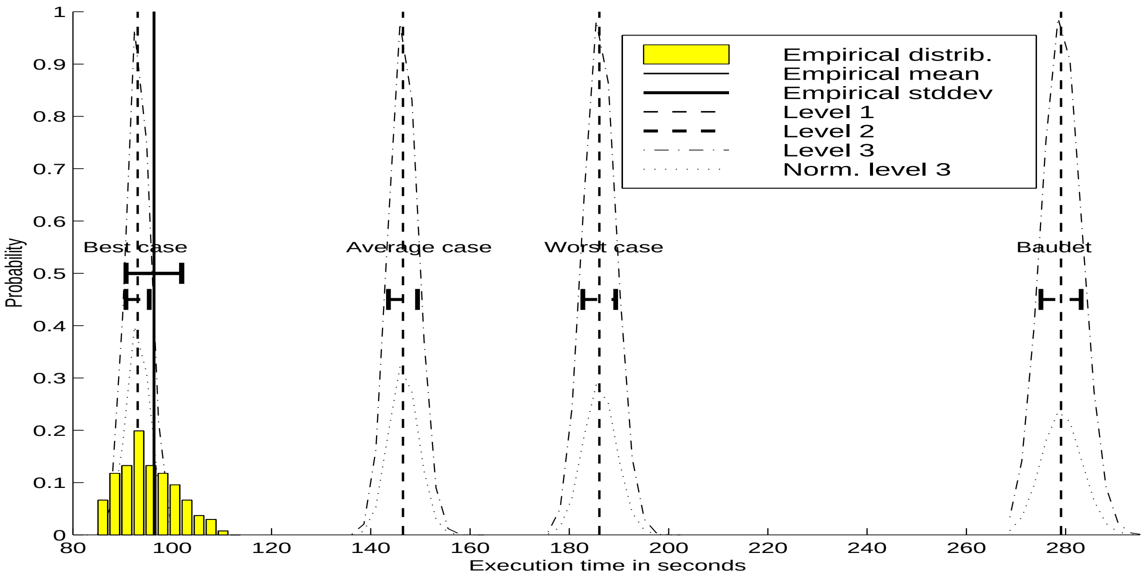

Figure 5.18: Experiment vs. Characterization for the first asynchronous implementation

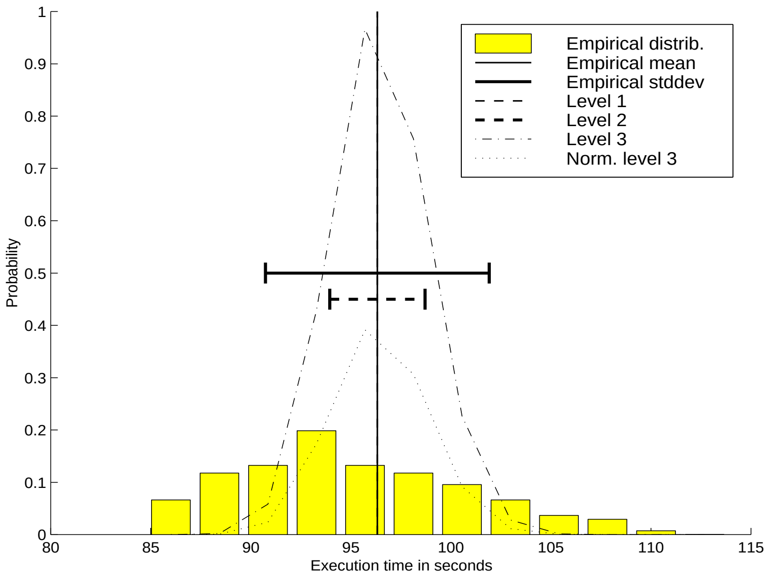

Figure 5.19: Adjusted curve for the first asynchronous implementation

However, the tails of the experimental distributions are still heavier than the ones suggested by LDT. This is still due to violations of the stochastic independence assumptions.

Conclusion

The experiments conducted over the 24 hour time period confirm some of our interpretations of the simulation and of the experiment over one week. The results given by our model for the 24 hour period can be considered *between* the ones for the simulation and the one week experiment. The level 1 characterization was "pessimistic" for the simulation, for the same reason that Baudet's convergence rate estimate is pessimistic: there is no precise control over the shape of the cost function. By contrast, level 1 was "optimistic" for the one week time period because our model is based on fundamental assumptions on the independence of the solution vector update times. This assumption was violated since the processor workloads exhibit bursty distributions. The level 2 characterization was useless for the one week time period as it was several orders of magnitude smaller than the observed standard deviations for the execution time. For the 24 hour time period it yields much more acceptable results. However, it suffers from the same errors identified during the simulation (see Section 5.1.6), in addition to the still present (but reduced) workload burstiness. Finally, level 3 is not as accurate for the 24 hour period as it was for the simulation. This is due to violations of the independence assumption. As we have already noted, it is difficult to interpret level 3 for the one week time period.

5.3 Conclusion

In this chapter we have presented three validations of the stochastic model presented in Chapter 3 and of the performance characterization described in Chapter 4. First, in Section 5.1, we conducted a simulation which has the advantage of matching our main assumptions exactly. The results were very conclusive and we discussed their interpretations in detail. In particular, we identified the sources of error in our performance characterizations. This simulation supports the hypothesis that our stochastic approach is reasonable and that it produces useful results when the necessary assumptions are satisfied.

In Section 5.2, we showed experimental results. The experiment was conducted on a real network of workstations used by students at the University of Tennessee. Three implementations(one synchronous and two asynchronous) of a parallel iterative algorithm were run on this network during a time period of a week, and then during an additional 24 hour period. The main differences between the experiment and the simulation are explained by bursty processor workloads. This burstiness is in violation of our stochastic independence assumption and impacts on our performance characterization. The level 2 characterization seems to be the most sensitive one. This impact was clear for the one week time period as the distributed environment exhibits different general behaviors throughout the week (see Figure 5.12). Our performance characterization gives very acceptable results for the 24 hour time period as the distributed environment behaves more consistently. Nevertheless, some workload burstiness can still be observed, leading to heavier than expected distribution tails for the experimental execution time distribution.

Chapter 6

Conclusion

The research described in this document contributes theoretical tools, techniques, and interpretations in quantitative performance analysis of different parallel implementations of some iterative algorithms. A broad class of algorithms has been considered: those that can be reduced to solving a fixed point problem for a linear or non-linear operator.

In Chapter 2, we described popular strategies that have been used for implementing a parallel version of these iterative algorithms. The two schemes of interest are the synchronous implementation and the partially asynchronous implementation. These schemes have been studied in the past, and several theoretical results concerning their conditions for convergence and their rates of convergence have become available. However, these results are still a step removed from a performance analysis of the algorithms in a given distributed environment. As explained in Section 2.4, stochastic models appear to be natural tools to use in conjunction with the convergence results to extend the study of these iterative algorithms. We examined some earlier stochastic approaches to such a study and illuminated their shortcomings.

In Chapter 3, we showed how to produce a stochastic model for the distributed environment that can be easily used to analyze the execution of a parallel iterative algorithm in that environment. Our model is based on an underlying Markov chain. This Markov chain, the *wavefront*, has been clearly identified and described in Section 3.5: it depends on the distributed environment and on the implementation of the algorithm, and can be easily computed. The wavefront can be exploited to obtain a variety of results concerning the behavior of the algorithm implementation in the distributed environment.

In Chapter 4, we first presented three examples of actual wavefront computations and made several remarks on its structure and properties. We then proposed three new estimates of the convergence rate of the algorithm. These new estimates are stochastic extensions of Baudet's estimate [5]. By contrast with Baudet's estimate, our estimates take into account the random behavior of the algorithm's execution in a distributed environment. Baudet's estimate has been replaced by our own "worst case" convergence rate estimates, and we introduced "average" and "best" case estimates.

Next, we then characterized the speed of the implementation in the distributed environment in terms of the number of iterations it performs per time unit. This number is computed thanks to the wavefront Markov chain. Large Deviation Theory appeared as a natural tool to perform a more in-depth analysis of the implementation speed. Basic concepts of this theory were given in Section 4.4 along with examples in connection with our stochastic model. Finally, we summarized the results of Chapter 3 in Section 4.5 and thereby extracted our three level performance characterization. Level 1 is a characterization of the mean execution time of an implementation of an iterative algorithm in a given environment. Level 2 is a characterization of the standard deviation of that execution time. It is based on a bi-variate Normal approximation to the sample average of a random vector. Level 3, thanks to Large Deviation Theory, characterizes the extreme behaviors of the execution time and, as such, proposes estimates of the tail of the execution time distribution.

The next step was to validate our performance model. This was done in Chapter 5 in the form of simulations and experiments. Section 5.1 described a simulation and obtained conclusive results about the validity of our stochastic performance modeling. This simulation was designed to comply with all our model requirements, especially stochastic independence assumptions for the solution vector update times. The accuracy of each level of characterization was studied and the sources of errors were identified. We also presented experimental results obtained for a real implementation of an iterative algorithm on a real distributed system at the University of Tennessee. Two separate experiments were conducted for two different time periods: (i) one week and (ii) 24 hours. The most instructive observation to draw from these experiments is that our model is sensitive to the processor workload burstiness because it violates the stochastic independence assumption. This is very observable for the one week time period where the algorithm execution times exhibit various behaviors throughout the week (see Section 5.2.2). The level 2 characterization is the most sensitive to this burstiness for reasons explained in Sections 5.1.5 and 5.2.3. The distributed environment behaves much more consistently during our 24 hour time period, and our performance characterization gives more accurate results than for the one week time period. Intuitively, the results for the 24 our time period are *between* the simulation results and the one week results.

6.1 Contribution of this Dissertation

Parallelizing iterative algorithms for the solution of large or complex optimization problems is a crucial issue. Indeed, the amount of computation required to solve such problems can be prohibitive for a sequential implementation, especially in non-linear cases. We examined different popular parallelization strategies as well as the corresponding theoretical results available in the literature and came to the conclusion that there is a gap between those results and what the end-user needs to know about the execution of his parallel iterative algorithm. This dissertation showed that this gap can be filled using stochastic models that take into account the distributed environment used to run the algorithm. Such models have been developed and used to obtain a variety of performance characterizations that are directly meaningful to the end-user. Our results can be used to make informed choices about what combinations of distributed environment and implementation style should lead to the shortest execution times.

6.2 Future Research Directions

There are multiple ways in which our research can be further expanded and improved. Some important research directions are suggested here.

In Chapter 4, we have given examples of "wavefront" computations. The study of the structure and properties of the wavefront transition matrix, P_X, seems to be very interesting and should lead to general results concerning the way processors lose and recover synchronization throughout the execution of the algorithm. We have given a few guidelines for such a study, but did not pursue it further as P_X is systematically computed by the implementation of our model.

Throughout Chapter 4, we have used several times an approximation to replace conditional probabilities by unconditional ones. The conditional distribution of a RV, A, conditioned on the observed state of the wavefront Markov chain can be replaced by an unconditional distribution, as seen in Section 4.2.2 for instance. This approximation is only valid for long-run observations of A as it uses the steady state distribution of the wavefront Markov chain. However, it is possible to use more sophisticated approximation techniques that should lead to better results. For instance, one can use one level of *Hidden Markov Model*(HMM) [23, 44] to approximate the RV A as the state of a new Markov chain. This approximation still uses the steady state distribution of the wavefront, but includes more information about the evolution of A than a simple distribution function. In fact, there is no limit to the number of levels that can be used and each additional level should provide a more accurate description of A, still as a Markov chain. Such Markov chains can be generated for all the RVs that we have introduced at the end of Chapter 3 and used in the computations that we have described in Chapter 4.

In Section 4.2, we have described how we used Baudet's convergence rate estimate to generate three new estimates that take into account the behavior of the distributed environment. Our "worst case" estimate is very easy to compute and is a clear extension of Baudet's work. The computation of our other two estimates is an attempt at giving a meaning to the limit $\liminf_{t\to\infty}(k_t/t)$ where k_t is a random variable. The experience proved that those two estimates are useful in analyzing the behavior of the parallel iterative algorithm. However, there are certainly many ways of improving the estimation of the algorithm convergence rate. Such improvement will require a better understanding of the behavior of the

RV k_t and of its impact on the algorithm execution.

Finally, and maybe most importantly, our work can be extended to take into account the bursty workload patterns described in Chapter 5. In Section 5.2.2 we showed experimental results obtained over a one week time period. Those results suggest that the solution vector update times are not i.i.d., violating one of our model's assumptions. Better suited for the update times would be the use of Markov-modulated random processes [41] rather than simple i.i.d. observations of a single RV. A popular and simple Markov-modulated process is one driven by a two-state Markov chain and is often called an *ON/OFF* or *zero/one* source [43, 22, 45, 31]. Markov-modulated random processes are generally used to model bursty behaviors. The idea would then be to model the processor workloads (and thereby the solution vector update times) as bursty sources. This would certainly lead to more accurate results than the current version of our model, but will require many changes to the performance characterization described and derived in Chapter 4. Furthermore, the statistical inference necessary to generate suitable Markov-modulated processes for a given distributed environment will be much more complicated than the one we have performed to generate simple distributions. One possibility is to limit the bursty sources to be ON/OFF, but this hardly seems a good choice since several processes can contribute to increase the load of one processor simultaneously. Computing the appropriate number of states of the underlying Markov chain for each bursty source, as well as its transition probabilities, will be one of the challenges of this new approach.

Bibliography

Bibliography

[1] M. Abdelaziz and I. Stavrakakis. Some Optimal Traffic Regulation Schemes for ATM Networks: A Markov Decision Approach. *IEEE Transactions on Networking*, 2(5):508–519, October 1994.

[2] V. Adve and M. Vernon. The influence of Random Delays on Parallel Execution Times. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 61–73, May 1993.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.

[4] R. Ash. *Information Theory.* Dover Publications, Mineola, N.Y., 1990.

[5] G. Baudet. Asynchronous Iterative Methods for Multiprocessors. *Journal of the Association for Computing Machinery*, 25:226–244, April 1978.

[6] D. El Baz. M-functions and parallel asynchronous algorithms. *SIAM Journal of Numerical Analysis*, 27:136–140, 1990.

[7] Didier El Baz, Pierre Spiteri, Jean-Claude Miellou, and Didier Gazen. Asynchronous Iterative Algorithms with Flexible Communication for Nonlinear Network Flow Problems. *Journal of Parallel and distributed Computing*, 38:1–15, 1996.

[8] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1992.

[9] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[10] Dimitri P. Bertsekas. Distributed asynchronous computation of fixed point. *Math. Programming*, 27:107–120, 1983.

[11] Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the Int. Conf. on Supercomputing*, pages 461–470, 1989.

[12] A. Bharucha-Reid. *Elements of the Theory of Markov Processes and their Applications.* Dover Publications, Mineola, N.Y., 1997.

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker,

and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

[14] L. Brochard, J.-P. Prost, and F. Fauire. Synchronization and load unbalance effects of parallel iterative algorithms. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume III, pages 153–160, 1989.

[15] James A. Buckllew. *Large Deviation Techniques in Decision, Simulation, and Estimation.* Wiley-Interscience, New York, NY, 1990.

[16] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Applications*, 2:199–222, 1969.

[17] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statist.*, 23:493–507, 1952.

[18] D. Comer and D. Stevens. *Internetworking with TCP/IP.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[19] H. Cramer. Sur un nouveau théorème-limite de la théorie des probabilités. In Hermann, editor, *Colloque consacré à la théorie des probabilités*, volume 736, pages 5–23, Paris, 1938.

[20] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of*

the Fourth ACM Symposium on Principles and Practice of Parallel Programming, May 1993.

[21] David E. Culler, Kim Keeton, Cedric Krumbein, Lok T. Liu, Alan Mainwaring, Richard P. Martin, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. Technical report, Department of Computer Science, University of California, Berkeley, November 1994.

[22] N. Duffield, J. Lewis, N. O'Connel, R. Russell, and F. Toomey. Entropy of ATM Traffic Streams: A Tool for Estimating QoS Parameters. *IEEE Journal on Selected Areas in Communications*, 13(6):981–989, August 1995.

[23] Robert J. Elliott, Lakhdar Aggoun, and John B. Moore. *Hidden Markov Models*. Springer-Verlag, New York, NY, 1995.

[24] A. Frommer. On asynchronous iterations in partially ordered spaces. *Numerical Funct. Anal. Optimization*, 12(3 & 4):315–325, 1991.

[25] A. Greenbaum. Synchronization costs on multiprocessors. *Parallel Computing*, 10:3–14, 1989.

[26] P. Heidelberger. Fast Simulation of Rare Events in Queueing and Reliability Models. *ACM transactions on Modeling and Computer Simulation*, 5(1):43–85, 1995.

[27] N. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, 1996.

[28] The Math Works Inc. *MATLAB Reference Guide.* The Math Works Inc., 1992.

[29] S. Junejas. *Efficient Rare Event Simulation of Stochastic Systems.* PhD thesis, Dept. of Operations Research, Stanford University, Palo Alto, CA, 1993.

[30] S. Karlin and H. Taylor. *A First Course in Stochastic Processes.* Academic Press, New York, NY, second edition, 1975.

[31] K. Kawahara, Y. Oie, M. Murata, and H. Miyahara. Performance Analysis of Reactive Congestion Control for ATM Networks. *IEEE Journal on Selected Areas in Communications*, 13(4):651–661, May 1995.

[32] E.S. Keeping. *Introduction to Statistical Inference.* Dover Publications, Mineola, N.Y., 1995.

[33] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 236–240, 1984.

[34] W. Leland, M. Raqqu, and W. Willinger D. Wilson. On the Self-Similar Nature of Ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb. 1994.

[35] Boris Lubachevsky and Debasis Mitra. Chaotic Asynchronous Algorithm for Computing the Fixed Point of a Nonnegative Matrix of Unit Spectral Radius. *Journal of the ACM*, 33(1):130–150, January 1986.

[36] Benoit B. Mandelbrot and Murad S. Taqqu. Robust R/S Analysis of Long-run serial Correlation. In *Proceedings of the 42nd Session of the ISI*, volume 48, pages 69–99, 1979.

[37] J.C. Miellou. Algorithmes de relaxation à retards. *Revue d'Automatique, Informatique et Recherche Opérationnelle*, 9:55–82, 1970.

[38] J.C. Miellou. Itérations chaotiques à retards. *Comptes Rendus de l'Acad, Sci. Paris*, 278:957–960, 1974.

[39] Ilkka Norros. A storage model with self-similar input. *Queuing systems*, 16:387–396, 1994.

[40] Soeren P. Olesen. *Parallel Computation for Positron Emission Emission Tomography with Reduced Processor Communications.* PhD thesis, Dept. of Computer Science, University of Tennessee, Knoxville, TN, 1996.

[41] R. Onvural. *Asynchronous Transter Mode Networks, Performance Issues.* Artech House, Inc., Norwood, MA, second edition, 1995.

[42] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, NY, 1970.

[43] P. Pruthi and A. Erramilli. Heavy-Tailed ON/OFF Source Behavior and Self-Similar Traffic. *Unknown IEEE Journal*, 2:445–450, 1995.

[44] Lawrence R. Rabiner. Mathematidcal Foundations of Hidden Markov Models. *Recent Advances in Speech Understanding and Dialog Systems*, F46:183–205, 1988. NATO ASI Series.

[45] J-F. Ren, J. Mark, and J. Wong. End-to-End Performance in ATM Networks. *Unknown IEEE Journal*, pages 996–1002, 1994.

[46] John T. Robinson. Some Analysis Techniques for Asynchronous Multiprocessor Algorithms. *IEEE Transactions on Software Engineering*, SE-5(1):24–31, January 1979.

[47] I.N. Sanov. On the probability of large deviations of random variables. *Selected Translations in Mathematical Statistics and Probability I*, pages 213–244, 1961.

[48] Adam Shwartz and Alan Weiss. *Large Deviation for Performance Analysis.* Chapman & Hall, London, UK, 1995.

[49] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*. The MIT Press, Cambridge, MA, 1996.

[50] W. Stalling. *Local and Metropolitan Area Networks*. Macmillan Publishing Company, New York, NY, fourth edition, 1993.

[51] El Tarazi. Some convergence results for asynchronous algorithms. *Numerical Mathematics*, 39:325–340, 1982.

[52] J. N. Tsitsiklis and G.D. Stamoulis. On the average communication complexity of asynchronous distributed algorithms. Technical Report LIDS-P-1986, Laboratory for Information and Decision Systems, 1990.

[53] A. Üresin and M. Dubois. Sufficient conditions for the convergence of asynchronous distributed algorithms. *Parallel Computing*, 10:83–92, November 1989.

[54] A. Üresin and M. Dubois. Parallel asynchronous algorithms for discrete data. *Journal of the ACM*, 37(3):588–606, 1990.

[55] Aydin Üresin and Michel Dubois. Effects of Asynchronism on the Convergence Rate of Iter ative Algorithms. *Journal of Parallel and Distributed Computing*, 34:66–81, 1996.

[56] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. In *Proceedings of Hot Interconnects II*, August 1994.

[57] Alan Weiss. An Introduction to Large Deviations for Communication Network. *IEEE Journal on Selected Areas in Communications*, 13(6):938–952, 1995.

[58] W. Willinger, M. Raqqu, W. Leland, and D. Wilson. Self-Similarity in High-Speed Packet Traffic: Analysis and Modeling of Ethernet Traffic Measurements. *Statistical Science*, 10(1):67–85, 1995.

[59] M. Woodward. *Communication and Computer Networks.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

Vita

Henri Casanova was born in Marseille, France on December 19, 1970. He received his high-school education from Collège Edgard Quinet and the Lycée Montgrand in Marseille, France. He graduated in 1988. From 1988 until 1990 he attended the classes préparatoire at the Lycée Thiers in Marseille, France. In June 1990 he entered the Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), Toulouse, France. In 1993 he received the Applied Mathematics and Computer Science Engineer degree from the ENSEEIHT and the Diplôme d'Etudes Approfondies in Parallel Architectures and Software Engineering from the Université Paul Sabatier, Toulouse, France. In January 1995, he entered the PhD program in Computer Science at the University of Tennessee, Knoxville.

In 1992-1993 he was a trainee at the Institut de Recherche en Informatique de Toulouse (IRIT), in Toulouse, France. From November 1993 until November 1994 he did his military service working from the French Ministry of Defense (DGA) as Advisor for the Computer Science division of the DPAG. From January 1995 until 1998, he worked as a Graduate Research Assistant in the Computer Science Department of the University of Tennessee, Knoxville. He was awarded the Doctor of Philosophy degree in Computer Science from the University of Tennessee in May of 1998.