CMU-CS-78-116

The Design and Analysis of Algorithms for Asynchronous Multiprocessors

Gérard M. Baudet

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

April 28, 1978

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was partly supported by the National Science Foundation under Grant MCS 75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, and partly by a Research Grant from the Institut de Recherche d'Informatique et d'Automatique (IRIA), Rocquencourt, France.

Acknowledgements

The advice and assistance of H. T. Kung have been instrumental in the development of this thesis. He has been more than an advisor to me, and I would like to express very special thanks to him for reading numerous drafts, for making many suggestions, and for his continual encouragement.

I am also especially grateful to Joe Traub for his comments and support. As Chairman of the Computer Science Department, he has contributed greatly to the development of an atmosphere favorable to carrying out my research.

I would also like to thank the two other members of my committee, Bill Wulf and Sam Fuller, for their help and cooperation.

Chapter II was initially written as a technical report in conjunction with Richard Brent, from The Australian University at Canberra, and H. T. Kung. I am also grateful to Peter Oleinick for helping me implement algorithms on C.mmp, and to Levy Raskin for running the same experiments on Cm*. I would also like to thank John Robinson and Bruce Weide, and Henryk Woźniakowski, from the University of Warsaw, for useful comments and discussions.

Last but not least, I would like to thank my wife for her inspiration, understanding and TLC throughout this ordeal.

Abstract

The characteristic of an asynchronous multiprocessor is that it is composed of several processors capable of carrying out the execution of their own programs in a completely independent fashion. As a consequence, parallel algorithms for asynchronous multiprocessors present some unique aspects in both their design and their analysis. This thesis explores the issues raised by the design and the analysis of parallel algorithms for asynchronous multiprocessors and illustrates the various notions and concepts involved with these algorithms by considering problems in diverse areas. The thesis demonstrates that asynchronous multiprocessors can be used efficiently in different problem domains, provided that appropriate algorithms are used. It also illustrates various techniques useful in the analysis of such algorithms.

As evidenced by a series of experimental results, the computation time required by a process to execute several instances of the same task on an asynchronous multiprocessor cannot be regarded as constant and is actually subject to important fluctuations. These fluctuations in computation times have a negative effect on the performance of parallel algorithms when several processes cooperating in the solution of a problem communicate extensively among themselves. In this case, when synchronization is used, it tends to introduce a prohibitive overhead which decreases the parallelism. On the other hand, an algorithm is presented to illustrate that the fluctuations are not always a negative factor but can also be utilized advantageously. The algorithm demonstrates the seemingly counter-intuitive result that the execution of a purely sequential program can still be accelerated on an asynchronous multiprocessor without introducing any parallelism within the program itself, but only by taking advantage of the fluctuations in computation times. Two different parallel implementations of this algorithm are proposed (with and without critical section), and analyses are presented to measure the speed-up achievable.

In the domain of numerical applications, the class of *asynchronous iterative methods* is introduced to remove the need for synchronization in the implementation of iterations for solving a system of equations on a multiprocessor. This class includes iterations corresponding to parallel implementations in which the cooperating processes have a minimum of inter-communication and do not make any use of synchonization. The *Purely asynchronous method* is a typical example. A sufficient condition is established which guarantees the convergence of any asynchronous iterations. This condition is satisfied for systems of equations found in numerous practical applications.

Several asynchronous iterations have actually been implemented on an asynchronous multiprocessor. Experimental results are reported, and they show that the Purely Asynchronous method achieves an almost optimal speed-up. The experiments constitute an illustration of the various notions and concepts specific to the design and analysis of parallel algorithms for asynchronous multiprocessors. It is also shown how simple techniques drawn from order statistics and queueing theory can be used to predict the experimental results with a fair accuracy.

The α-β pruning algorithm serves as an example of a non-numerical application in this thesis. The sequential algorithm is first analyzed, and it is shown that the branching factor of the α-β pruning algorithm for a uniform game tree of degree n grows with n as $\Theta(n/\ln n)$. This confirms a claim by Knuth and Moore that deep cut-offs only have a second order effect on the behavior of the algorithm. The results obtained with the sequential algorithm are then used to derive an efficient parallel implementation of the α-β pruning algorithm on an asynchronous multiprocessor. An analysis of the parallel implementation with k processes shows, rather surprisingly, an improvement over the original algorithm by a factor larger than k.

To K

to Cunégonde

and

to my wife

La quatrième planète était celle du businessman. Cet homme était si occupé qu'il ne leva même pas la tête à l'arrivée du petit prince.

—Bonjour lui dit celui-ci. Votre cigarette est éteinte.

—Trois et deux font cinq. Cinq et sept douze. Douze et trois quinze. Bonjour. Quinze et sept vingt-deux. Vingt-deux et six vingt-huit. Pas le temps de la rallumer. Vingt-six et cinq trente-et-un. Ouf! Ca fait donc cinq cent un millions six cent vingt-deux mille sept cent trente-et-un.

—Cinq cent millions de quoi?

—Hein? Tu es toujours là? Cinq cent un millions de . . . je ne sais plus . . . j'ai tellement de travail! Je suis sérieux, moi, je ne m'amuse pas à des balivernes! Deux et cinq sept . . .

Antoine de Saint Exupéry, *Le Petit Prince*

Table of contents

V Experimental results with asynchronous multiprocessors

VI Conclusion

Chapter I

Introduction

1 - Introduction and motivation

Parallel computers and multiprocessors offer a natural solution to the ever-increasing demand for computing power. At the same time, their evolution has brought about the need for the development of efficient parallel algorithms. This need is now becoming more and more acute since recent advances in computer technology have drastically reduced the cost of components, and it is quite conceivable that parallel computers composed of *1000* or more processors will be built in the near future.

Parallelism is achievable in a variety of ways, as exemplified by the various architectures of parallel computers already existing. Following Flynn's classification [21], we mention below only a few among the more important ones. For a general overview, Stone [57] offers an introductory presentation of parallel computer architecture; Kuck [36] evaluates some parallel machine organizations in relation to their programming; and Enslow [19] surveys specifically multiprocessor organization, which is of central interest to us in this thesis.

The ILLIAC IV computer [5] is a typical example of an SIMD (Single Instruction stream Multiple Data stream) machine [21]. Often referred to as an *array processor*, the ILLIAC IV was designed explicitly for solving partial differential equations by the method of finite differences (typically, for weather forecast). It is composed of *64* identical processing elements, organized as an *8×8* array, which execute *synchronously* the same

instruction possibly operating on different data. The CDC STAR-100 [29] and the Cray-1 computer [54] are also SIMD machines in Flynn's classification. They are often referred to as *vector computers*, and they gain their efficiency by providing for vector-type instructions, capable of executing in parallel the same operation on all elements of a variable size vector rather than on a single scalar. Pipelined computers and associative processors also belong to the class of SIMD machines; a general presentation of their architectures can be found in [12] and [65], respectively.

This thesis is concerned with another type of parallel computer, classified by Flynn as an MIMD (Multiple Instruction stream Multiple Data stream) machine [21]. Throughout the thesis, this type of computer will be referred to as an *asynchronous multiprocessor*, since we think this term better reflects the view we are taking here.

Examples of asynchronous multiprocessors include commercially available computers like the UNIVAC 1108 bi-processor; special purpose computers like the D825 [1], produced for command and control military applications; and research products like C.mmp [63] and Cm* [59]. C.mmp and Cm* have been (and are being) built at Carnegie-Mellon University using mini-processors, slightly modified versions of the DEC PDP-11 and the DEC LSI-11. While C.mmp is truly a multiprocessor, in that each processor has a direct access to each memory bank through a cross-point switch, Cm* could also be considered as a local network, in which intercommunication takes place between *clusters* (each processor, however, can actually access the entire common memory through a sophisticated address mechanism [30], [59]).

We do not intend to go into the details of the architecture of any asynchronous multiprocessors. (See [19] for a general survey of the architectures of existing multiprocessors.) For the purpose of the thesis, it is sufficient to consider an asynchronous multiprocessor as composed of a set of *independent* processors sharing a common memory, each processor being able to carry out the execution of its own program. In this respect the execution of programs on an asynchronous multiprocessor, unlike on an

SIMD machine, is made in a completely asynchronous fashion and takes on a chaotic appearance. This is especially true since the processors are not necessarily of the same type, as is the case with C.mmp (composed of both PDP-11/20 and PDP-11/40), and could actually have drastically different characteristics, particularly in speeds. Another reason is that access to memory is not necessarily uniform, as is the case with Cm*. Notice that, in this broad sense, a network of computers could be viewed as an asynchronous multiprocessor as well since, in this case, the computers can still be considered to share a common memory, although very indirectly. As a matter of fact, the algorithms that we propose in this thesis for asynchronous multiprocessors are also well suited for implementation over a network, especially if the time required for the intercommunication between the computers is not too high compared to the time required by the computation on each computer.

After this very brief presentation of parallel computer architecture, let us now turn our attention to the issue of parallel algorithms. From an algorithmic point of view, SIMD machines have been the most widely studied to date, and particularly the ILLIAC IV type of computer. Due to its specific structure, the efficient utilization of an array processor requires that a problem be decomposed into identical subtasks which communicate among each other in some *regular* fashion, and the range of possible applications is, therefore, limited (mainly to linear algebra oriented problems). Numerous examples of parallel algorithms for SIMD machines in the area of numerical linear algebra can be found in a recent survey by Heller [27]. Examples of non-numerical algorithms can be found, for instance, in [9], [58], and [61].

Being composed of a set of independent processors, an asynchronous multiprocessor allows for greater flexibility in its programming than does an SIMD machine. Although asynchronous multiprocessors have now been in existence for several years (the D825 [1], in fact, dates back to the early 60's), very little has been published so far on how to design parallel algorithms that run efficiently on an asynchronous multiprocessor. Until

recently, emphasis in the design of parallel algorithms for multiprocessors has been placed mainly on techniques for recognizing the *intrinsic parallelism* of existing sequential algorithms rather than on the direct construction of parallel algorithms. Some of these techniques have actually been implemented in a version of the Algol-68 compiler running on Cm* [28]. Typically, the transformation of a sequential program is accomplished by identifying independent subtasks within the program and introducing precedence relations between them; a parallel program then can execute the various subtasks according to the graph of the relations. However, a parallel program resulting directly from this automatic transformation requires considerable communication and extensive synchronization to control the flow of execution of the various subtasks. This ultimately reduces its efficiency.

In the domain of numerical analysis, a different approach in designing algorithms for asynchronous multiprocessors has proved to be more fruitful. Rather than adapting existing sequential algorithms, Chazan and Miranker [11] have presented a class of iterative methods for the solution of a linear system of equations which takes into account the asynchronous nature of multiprocessors.

Essentially initiated by a recent paper by Kung [37], a systematic study is now under way to explore some of the unique issues raised specifically by the design and the analysis of parallel algorithms for asynchronous multiprocessors. This study certainly benefits from an extensive research done on a different, but related, area concerning time-shared processors rather than true multiprocessors. However, results in the latter area deal mostly with special problems typically encontered in time-sharing or multiprogramming operating systems, e. g., resource allocation, co-ordination of independent devices (typically, I/O devices), and they address directly the issue of co-operation of processes without addressing general issues, such as problem decomposition, involved with the design of multiprocessor algorithms. (See, for example, [16] for an early presentation of this area, and [2] for some examples of typical problems.)

In addition to [37], a few examples of typical algorithms for multiprocessors have already appeared, and they illustrate several important notions unique in their design [6], [38], [39], [40] and in their analysis [3], [4], [8], [51].

This thesis is concerned specifically with the design and the analysis of parallel algorithms for asynchronous multiprocessors. In Section 2 of this chapter, we briefly discuss the main issues involved in their designs. The remaining chapters of the thesis study these issues in depth in several problem domains. These results are summarized in Section 3 of this chapter.

2 - The design of algorithms for asynchronous multiprocessors

Algorithms for SIMD machines and algorithms for asynchronous multiprocessors are similar in principle, in that they both rely on the decomposition of a problem into subtasks executed in parallel. This is, however, their only similarity, and these two types of parallel algorithms in general present drastic differences with respect to both their design and their analysis. Let us examine, in this section, some of the unique issues raised by parallel algorithms for asynchronous multiprocessors.

Most of the problems associated with the design of parallel algorithms for asynchronous multiprocessors have been clearly exposed by Kung [37]. Throughout the thesis, we use the notions and concepts introduced in his paper, and, below, we briefly review some of the more important ones. In particular, [37, p. 156]:

> "We define a parallel algorithm for multiprocessors as a collection of concurrent processes that may operate simultaneously for solving a given problem."

It is important to distinguish between the notion of *process*, which corresponds to the execution of a procedure or a piece of program, and the notion of *processor*, the physical entity which carries out the execution of a process. While we have control over the processes in the design of a parallel algorithm, we do not usually have control over the processors, which are administered by the operating system. In particular, the same

process is not necessarily executed by only one processor during its entire lifetime, and, upon decisions of the operating system, several processors might be assigned successively to its execution. As an immediate consequence, the time required for the execution of a process on an asynchronous multiprocessor can fluctuate in an almost unpredictable way. There are, in fact, numerous reasons contributing to this unpredictable behavior; we already mentioned the fact that the different processors of an asynchronous multiprocessor might have different speeds and that the access to memory is not necessarily uniform; several other features of an asynchronous multiprocessor or of its environment which also contribute to the *fluctuations* in the execution time of a process are listed in [37].

Communication is very likely to be required among the processes co-operating in the solution of a problem. Kung [37] regards a process as a sequence of *stages* defined between two consecutive *interaction points* at which the process communicates with other processes. Parallel algorithms for multiprocessors are then classified according to the way in which communication is accomplished. In a *synchronized parallel algorithm* (or, simply, a *synchronized algorithm*) processes explicitly use synchronization primitives, and, upon completion of a stage, a process may have to wait for the results of other processes before resuming its execution; a producer-consumer type of program is a typical example of a synchronized algorithm. In an *asynchronous parallel algorithm* (or, simply, an *asynchronous algorithm*) the processes communicate among themselves only through the use of global variables (possibly updated within a critical section), and, at the completion of a stage, a process either terminates or proceeds further, without any delay, according to the current contents of the global variables. Examples of asynchronous algorithms are presented in the following chapters.

Let us now address briefly (and informally) the issues of correctness and of efficiency, both of which we feel should always be dealt with in the design of any algorithms. These issues are not the only ones which should be taken into account, but, in

the case of parallel algorithms for asynchronous multiprocessors, these two issues become particularly interesting and important because of the a priori unpredictable behavior in the execution of these algorithms. For this very reason, however, we can anticipate that proving the correctness and analyzing the efficiency of an algorithm for multiprocessor are, in general, difficult tasks.

2.1 - Correctness

Correctness is obviously a requirement for any algorithm. Considerable research has been done on the proof of correctness of sequential programs, and a detailed treatment of some of the techniques available can be found, for example, in Dijkstra's recent text [17]. These techniques, however, are mostly applicable to sequential programs with a simple structure (with no complicated data structures, for instance), and their generalization to parallel programs (especially asynchronous parallel programs) is still quite limited.

An early paper by Dijkstra [16] contains the first major statement on the proof of correctness of parallel programs. Research in this area has been restricted mostly to proving the correctness of the solutions of small problems, which could be used for the implementation of some mechanisms in larger parallel programs (e. g., the readers and writers problem [13], or the producer-consumer scheme [26]). Several attempts have been made only very recently to extend some of the techniques to the proof of correctness of complete and more complex parallel programs [47], [20].

Despite the lack of a formal theory, we still feel that we have given with every algorithm presented in this thesis a convincing argument that it performs correctly. This *proof of correctness* can take on very different aspects. In Chapter II, for example, we give a proof of the correctness of a parallel program by verifying that global variables used in the program satisfy some property which holds during the entire execution of the program; this is achieved by checking the possible transitions of the global variables before and

after interaction points. In some respect, the proof resembles more, in this case, the formal proof of a sequential program using *assertions* and *invariants*; this is partly due to the simple structure of the particular parallel program we are dealing with. In Chapter III, on the other hand, the proof of the correctness (and of the termination) of the algorithm follows directly from the theorem of convergence which is derived through techniques of numerical analysis.

2.2 - Efficiency

In the design of any algorithm, efficiency is always an important issue. Since one of the primary goals in the design of a parallel algorithm is to achieve better efficiency than with a sequential algorithm, this issue must be considered very seriously in the case of an algorithm for asynchronous multiprocessor.

We would like to illustrate below that, because of the fluctuations in the execution times on an asynchronous multiprocessor, synchronized algorithms will generally show a very poor performance. This is true for several reasons. The execution time of the synchronization primitives themselves is often very time consuming (a typical execution time for these primitives is usually on the order of a couple of hundreds of additions). Also, and most importantly, the use of synchronization implies the blocking of the processes co-operating in a task, and, in turn, either causes some of the processors to be idle or entails the switching of contexts. In both cases, the use of synchronization may reduce the parallelism and decrease the speed-up that we hope to achieve by using an asynchronous multiprocessor.

To illustrate this point, let us consider Jacobi's method to solve the linear system of equations given by:

$$x' = Ax + b,$$

where A is an $n \times n$-matrix, and b and x are n-vectors. Let x_0 be an initial approximation to the solution of this system, Jacobi's method consists of computing the sequence of iterates x_i, for $i = 1, 2, \ldots$, through the recurrence:

$$x_i = A\,x_{i-1} + b\,.$$

This method is well suited for parallel computation since, at each step of the iteration, the computations of all components can be carried out in parallel. For example, assuming that n processors are available, a natural way to decompose the computation of a new iterate is to assign to each of the n processors the computation of one of the n components of the iterate. This implementation requires, however, that at the end of each step all processes be synchronized before they can start the computation of the next iterate. In case all processes take exactly the same amount of time to compute a component, the overhead introduced by the synchronization is reduced to the execution time of the synchronization primitives themselves. However, it follows from the discussion at the beginning of the section that it is more realistic to assume that the time taken by a process to compute a component is a random variable rather than a constant. In this case the time it takes to compute the whole set of components of a new iterate is given by the maximum of n randoms variables. In particular, to give an idea, assume that the time for the computation of any component is distributed according to the same exponential distribution with mean τ, then, simple calculus shows that the mean computing time for obtaining a new iterate is given by $H_n.\tau$, where $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ is the n-th harmonic number. The coefficient H_n represents the penalty imposed by the synchronization.

This simple example shows that the apparent parallelism in Jacobi's method for solving linear systems of equations is considerably reduced by the fact that this method implicitly requires synchronization at each step of the computation. In fact, it can be shown that the proportion of time wasted by the processes (while they are idle, waiting for the completion of the last computation) is given by:

$$\frac{1}{n}\,\frac{H_1 + H_2 + \ldots + H_{n-1}}{H_n} = 1 - \frac{1}{H_n} \sim 1 - \frac{1}{\ln n}$$

and tends to 1 as n tends to infinity, which means that *the processes are almost always idle waiting for each other*!

This example also shows that, when programming an asynchronous multiprocessor,

the problem of the fluctuations in the execution times requires much attention, and that synchronization should be used very carefully. In particular, the design of parallel programs for asynchronous multiprocessors should take into account the fact that the various processors execute their programs *independently* and possibly at very different speeds, and that, therefore, communication among the processes co-operating in a task should be reduced to a strict minimum.

3 - Thesis overview

This thesis explores the issues raised by the design and the analysis of parallel algorithms for asynchronous multiprocessors. The various notions and concepts involved with these algorithms are illustrated by considering very diverse problem areas for numerical as well as non-numerical applications. The thesis demonstrates, in particular, that asynchronous multiprocessors can be used very effectively in different problem domains, provided that appropriate algorithms are used. The thesis also illustrates various techniques useful in the analysis of such algorithms. The remaining chapters are briefly summarized below.

We have just shown, in Section 2.2, that the fluctuations in the execution times of programs that are run on an asynchronous multiprocessor could cause a very important degradation in the performance of synchronized algorithms, even for a problem which is, a priori, well suited for parallel implementation. In Chapter II, we show that we have the reverse phenomenon with asynchronous algorithms, even for a purely sequential problem. Namely, given a sequence of tasks to be performed serially, we propose an asynchronous algorithm to accelerate the execution of the tasks on an asynchronous multiprocessor without introducing parallelism within the tasks but *only by taking advantage of fluctuations in the execution times.* We give a parallel program requiring no critical section to implement the algorithm, and we prove its correctness. We also give a spacewise more efficient implementation, which requires the use of critical sections. We

then present an analysis for both implementations to estimate the speed-up achievable with the parallel algorithm, and we show that, when the execution times are exponentially distributed and no critical section is used, the algorithm with k processes yields a speed-up of order $\sqrt{k}$.

In Chapter III, we introduce the class of *asynchronous iterative methods* for solving a (linear or non-linear) system of equations. We identify existing iterative methods in terms of asynchronous iterations, and we propose new schemes corresponding to a purely asynchronous algorithm (with no synchronization between the co-operating processes). We give a sufficient condition (satisfied in most practical applications) to guarantee the convergence of any asynchronous iterations and extend the results to include *asynchronous iterative methods with memory*. We then evaluate asynchronous iterative methods from a computational point of view; we derive bounds for the efficiency and briefly compare the bounds with experimental results (see Chapter V).

Chapter IV deals with the α-β pruning agorithm. In the first part of Chapter IV, we analyze the sequential α-β pruning algorithm, using the number of terminal nodes examined by the algorithm as the cost measure. The analysis takes into account *both* shallow and deep cut-offs, and we also consider the possibility of ties between terminal positions: specifically, we assume that all bottom values are independent identically distributed random variables drawn from a discrete probability distribution. We show that the worst case of the algorithm can be achieved even when only two distinct values are assigned to the terminal nodes, and we deduce that the branching factor of the α-β pruning algorithm in a uniform game tree of degree n grows with n as $\Theta(n/\ln n)$, therefore confirming a claim by Knuth and Moore [35] that deep cut-offs only have a second order effect on the behavior of the algorithm.

In the second part of Chapter IV, we propose a parallel implementation of the α-β pruning algorithm requiring very little communication between the processes. In the parallel scheme, the processes work independently by searching for the solution of the

game tree over disjoint subintervals. We develop an analysis of the parallel algorithm, from which it follows that the parallel implementation with k processes shows an improvement over the sequential α-β pruning algorithm by a factor larger than k for $k = 2$ or 3. This leads to the rather surprising discovery that the sequential α-β pruning algorithm is not optimal.

In Chapter V, we present the results of measurements performed by running several asynchronous iterations (introduced in Chapter III) on C.mmp [63], an asynchronous multiprocessor at Carnegie-Mellon University. These experiments have proved to be an invaluable tool for providing us with some insight into the behavior of parallel algorithms, and, in particular, they constitute a clear illustration of the advantage of purely asynchronous algorithms over synchronized algorithms.

In Chapter VI, we show how the classical tools of queueing theory can be applied to the analysis of the performance of parallel algorithms for asynchronous multiprocessors, and, in particular, we develop a simple queueing model to account for the behavior of a parallel program which uses critical sections. We then compare the analytical results derived from the model with the experimental results presented in Chapter V, and the comparison shows an excellent agreement.

In the last chapter, we summarize the principal results of the thesis, mention some possible extensions and give some concluding remarks. We also present some topics for future research.

Chapter II

Parallel Execution of a Sequence of Tasks on an Asynchronous Multiprocessor

1 - Introduction

We are interested in the design and analysis of parallel algorithms for asynchronous multiprocessors such as C.mmp [63] or Cm* [59]. For any given task, the task execution time on such a system is dependent upon the properties of the operating system, effects of other users, processor-memory interference, and many other factors. As a result, it is necessary to assume that task execution times are random variables rather than constants. (See Chapter V for experimental results supporting this assumption.) In this chapter we propose a novel way of using asynchronous multiprocessors, which takes advantage of fluctuations in task execution times. We will present our result as a solution to the problem of executing a sequence of n tasks $w_1, ..., w_n$ under the following conditions:

C1. For $i = 2, ..., n$, task w_i cannot be started before the completion of task w_{i-1} (i. e., the tasks are linearly ordered).

C2. For $i = 1, ..., n$, no parallelism can be utilized in the execution of task w_i (i. e., we are not allowed to decompose a task).

C3. The execution time of a task is a random variable rather than a constant. (This condition corresponds to the asynchronous nature of the multiprocessor.)

We will view a parallel algorithm for asynchronous multiprocessors as a collection of asynchronous processes which communicate among each other through the use of global

variables. Such an algorithm will be defined by giving the procedure each of its processes executes when assigned to a processor. *While analyzing the algorithm, we will always assume that a processor is available for any of the runnable processes of the algorithm.* (See Kung [37] for a general discussion of asynchronous parallel algorithms.)

In Section 2 we give an algorithm which uses $k \geq 1$ asynchronous processes to solve the problem. The algorithm is interesting because at most one process is doing useful work at any given time. Nevertheless, by taking advantage of condition C3, the mean execution time is less for $k > 1$ than for $k = 1$, i. e., a speed-up is achieved.

As an example, consider the computation of $x_1, \ldots, x_n$ defined by

$$x_{i+1} = \varphi(x_i, \ldots, x_{i-d}),$$

where $x_0, x_{-1}, \ldots, x_{-d}$ are given and φ is some iteration function. Let w_{i+1} be the task of computing $\varphi(x_i,\ldots,x_{i-d})$. Our algorithm could be used to execute tasks $w_1, \ldots, w_n$, which is equivalent to evaluating $x_1, \ldots, x_n$.

The speed-up ratio $S_k(n)$ of a parallel algorithm using k processes is defined in Section 3, and some preliminary results are proved there. In Section 4 we give programs to implement our algorithm both with and without critical sections and prove informally their correctness. In Section 5 we consider the implementation without critical sections, and obtain an analytic expression for the speed-up under certain assumptions (A1 and A2 of Section 5). For large n and k, our result is $S_k(n) \sim \sqrt{2k/\pi}$. In Section 6 we consider the implementation which uses critical sections. Here the analysis is more difficult, and we can obtain analytic results only for $k \leq 2$. Some conclusions and open problems are stated in Section 7.

2 - The algorithm

For each positive integer k, we define an algorithm with k processes for executing tasks $w_1, \ldots, w_n$ under conditions C1 and C2 stated in the preceding section. The algorithm is specified as follows:

Whenever a process, P, is ready to execute a task,

(i) if no task has yet been completed by any process, process P starts executing task w_1,

(ii) otherwise, if the last task w_n has not yet been completed by any process, process P starts executing a task which is unfinished and ready for execution.

For simplicity, we will assume that no two tasks are completed at the same time. Then, due to the linear ordering of the tasks, condition (ii) defines without ambiguity a unique task to be executed by process P.

Let $t_1, t_2, t_3, \ldots$ with $t_i < t_{i+1}$ be the times of task completion by the processes. The diagram of Figure 2.1 illustrates a possible scheduling of the tasks when they are executed by the algorithm with three processes.

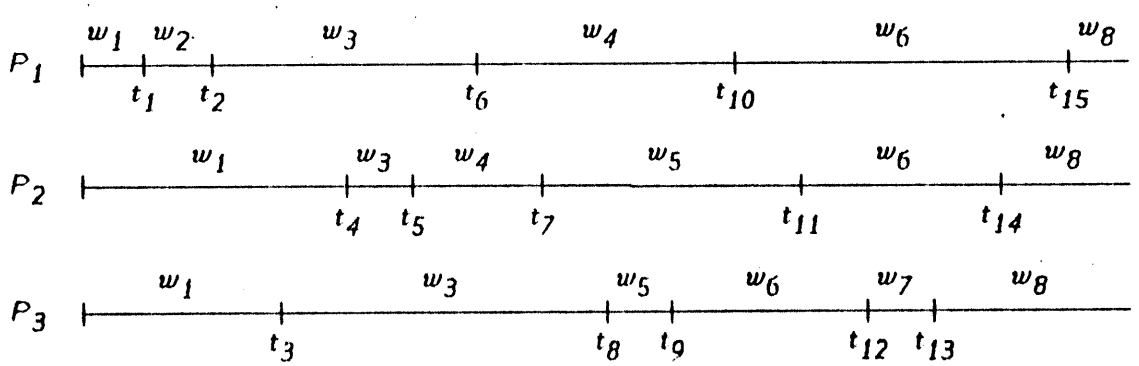


Figure 2.1 - A possible task scheduling with three processes

Note that, when process P_3 finishes task w_3 at time t_8, process P_2 has already completed task w_4. Thus, after P_3 completes w_3, it starts executing w_5 rather than w_4. Task w_4 is skipped by P_3. Similarly, tasks w_5 and w_7 are skipped by P_1, and tasks w_2 and w_7 by P_2. After any one of the three processes has executed six tasks, tasks w_1 through w_8 rather than tasks w_1 through w_6 are completed. A speed-up has been achieved!

Observe that at any given time at most one process is doing work useful for later computation. With respect to the scheduling given by Figure 2.1, the time intervals on which processes are doing useful computations are indicated in Figure 2.2.

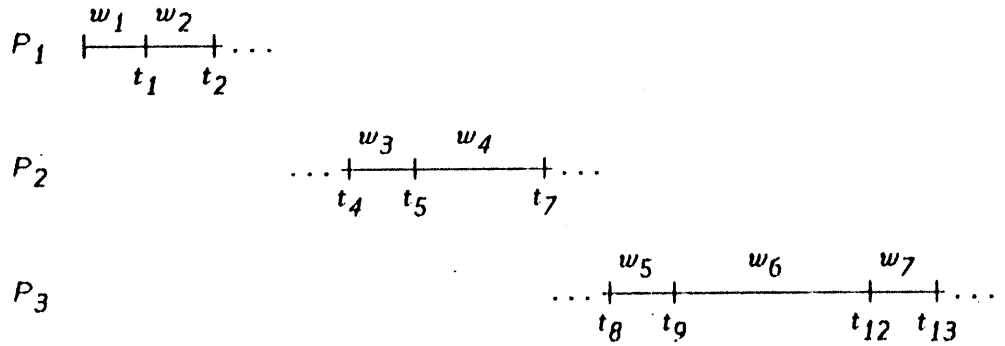


Figure 2.2 - Time intervals on which processes are doing useful work

Thus *the speed-up is not achieved by sharing work among the processes, but is achieved by taking advantage of fluctuations in the execution times.*

3 - A speed-up measure

Consider the algorithm with k processes as specified in the preceding section. The algorithm is said to be the sequential algorithm if $k = 1$ and to be a parallel algorithm if $k > 1$. Let $T_k(n)$ be the time to execute tasks $w_1, ..., w_n$ by the algorithm with k processes. Let $\overline{T}_k(n)$ be the mean of the random variable $T_k(n)$. We define the *speed-up ratio* of the algorithm with k processes to be

$$S_k(n) = \overline{T}_1(n) / \overline{T}_k(n) .$$

For each k and for each execution of the algorithm with k processes, we define $s_{k,i}$ to be the time of the *first* completion of task w_i, and define $s_{k,0} = 0$. For example, with respect to the scheduling of Figure 2.1, with $k = 3$, we have:

$$s_{3,1} = t_1 , \; s_{3,2} = t_2 , \; s_{3,3} = t_5 , \; s_{3,4} = t_7 ,$$
$$s_{3,5} = t_9 , \; s_{3,6} = t_{12} , \; s_{3,7} = t_{13} , \; \ldots .$$

The following theorem describes the relation between $\{s_{k,i}\}$ and $\{t_i\}$ in terms of the scheduling of the tasks. This theorem is important in Sections 5 and 6 for computing speed-up ratios.

Theorem 3.1:

Suppose that $s_{k,i} = t_r$ with $1 \le i \le n-1$. Then $s_{k,i+1} = t_{r+j}$ for some $1 \le j \le k$ if and only if

(a) the j processes completing tasks at times $t_r, t_{r+1}, \ldots, t_{r+j-1}$ are all distinct, and

(b) the process completing task w_{i+1} at time t_{r+j} is one of the j processes mentioned in (a).

Proof:

We will only prove the necessary condition since the proof for the sufficient condition is similar.

Suppose that some process P completes two tasks at times t_{r+h} and t_{r+m} for $0 \le h < m \le j-1$. Then, since at time t_{r+h} task w_i has already been completed, the task completed at time t_{r+m} by process P must be w_{i+1}. This contradicts the fact that w_{i+1} is completed for the first time at time t_{r+j}, since $t_{r+m} < t_{r+j}$. This proves (a).

Let P be the process completing task w_{i+1}, for the first time, at time t_{r+j}. Suppose that P does not complete any task in the interval $[t_r, t_{r+j-1}]$. Then the task completed by P at time t_{r+j} must be started before time t_r. But at any time before t_r, task w_i is not completed yet. Hence any task started before time t_r cannot be w_{i+1}. In particular, the task completed by P at time t_{r+j} cannot be w_{i+1}. This contradiction proves (b). ■

For $i = 1, \ldots, n$, let $\tau_k(i)$ be the random variable representing the quantity $s_{k,i} - s_{k,i-1}$. Then, since $T_k(n) = s_{k,n}$, we have

$$T_k(n) = \tau_k(1) + \tau_k(2) + \ldots + \tau_k(n) . \qquad (3.1)$$

Equation (3.1) will be used later to compute $\overline{T}_k(n)$, which is needed for evaluating the speed-up ratio $S_k(n)$.

4 - Parallel programs for the algorithm and their correctness

We give two programs to implement the algorithm with k processes: one without critical sections and one with critical sections.

4.1 - A program without critical sections

Program A:

```
global integer (or real) array U[1:n];
global boolean array M[1:n+1];

Initialization:
    begin
    for m := 1 to n+1 do M[m] := false;
    start processes P_1, ..., P_k
    end

Process P_j:
    begin integer m_j;
    m_j := 1;
    while M[m_j] do m_j := m_j + 1;                          (4.1)
    while m_j ≤ n do                                         (4.2)
        begin
        perform task w_{m_j};                                (4.3)
        write the output of task w_{m_j} on U[m_j];          (4.4)
        M[m_j] := true;                                      (4.5)
        while M[m_j] do m_j := m_j + 1                       (4.6)
        end
    end
```

Assume that the tasks are not allowed to alter the array M and integers m_j. We will prove that Program A is correct in the following sense:

P1. For $m = 2, ..., n$, task w_m is executed only if task w_{m-1} has been finished and its output has been written on $U[m-1]$.

P2. For $j = 1, ..., k$, process P_j can execute the loops at (4.1), (4.2) and (4.6) at most n times.

P3. All the tasks $w_1, ..., w_m$ will have been completed at the time when any one of the processes $P_1, ..., P_k$ terminates its execution.

Property P2 guarantees that the program will terminate. (Note that there is no possibility of deadlocks in the program.) Property P1 ensures that the linear ordering requirement of the executions of the tasks is maintained, and property P3 implies that when the program terminates all the tasks are completed.

Lemma 4.1:

(i) For $m = 1, ..., n$, if $M[m]$ is set to true, it remains true afterwards.

(ii) After being initialized to false, $M[n+1]$ is never modified.

Proof:

After initialization, M can only be modified through statement (4.5) executed by some process P_j. But, when entering the main while-loop (starting with statement (4.2)), m_j satisfies the condition $m_j \leq n$ and is not modified before execution of (4.5). Therefore $M[n+1]$ can never be modified. ■

Lemma 4.2:

For $j = 1, ..., k$, if m_j has the value $m \geq 2$, then $M[m-1]$ is true.

Proof:

Suppose that $m_j = m$ with $m \geq 2$ at time t. If m_j was incremented by 1 to the value m inside the while statement (4.1) or (4.6), then the test of $M[m_j]$ being true with $m_j = m-1$ must have been satisfied. Hence $M[m-1]$ was true at some time before t. Thus, by Lemma 4.1, $M[m-1]$ is true at time t. ■

Lemma 4.3:

For $m = 2, ..., n$, if $M[m]$ is true, then $M[m-1]$ is true.

Proof:

Suppose that $M[m]$ is true. Then $M[m]$ must have been assigned to true through instruction (4.5) by some process P_j with m_j having the value m. Therefore, by Lemma 4.2, $M[m-1]$ is true. ■

Lemma 4.4:

For $m = 1, ..., n$, if $M[m]$ is true, then task w_m is completed and its output is on $U[m]$.

Proof:

Suppose that $M[m]$ is true. Then $M[m]$ must have been assigned to true through instruction (4.5) by some process P_j with m_j having the value m. Since P_j executes instruction (4.5) only after the completion of task w_{m_j} and since m_j is not modified in between, we conclude that task w_m is completed. ■

We are now able to prove the following theorem.

Theorem 4.1:

Program A satisfies properties P1, P2 and P3.

Proof:

Suppose that process P_j is executing task w_m with $m = m_j \geq 2$. Then, by Lemma 4.2, $M[m-1]$ is true, and hence, by Lemma 4.4, task w_{m-1} is completed and its output is on $U[m-1]$. We conclude that Program A satisfies property P1.

Property P2 follows from statement (ii) of Lemma 4.1 since m_j is incremented by 1 in each execution of a loop.

Suppose that a process, say process P_j, terminates. This happens only when $m_j = n+1$. Thus, by Lemma 4.2, $M[n]$ is true for all $m = 1, ..., n$. Therefore, by Lemma 4.4, all tasks are completed. We have shown that Program A also satisfies property P3. ■

Program A is very reliable in the following sense. Property P3 implies that, even if some processes fail (for reasons external to the algorithm: e. g., crash of the processors executing the processes), the program may still continue executing tasks and eventually complete all tasks, provided that there remains at least one active process. We will not pursue this reliability issue any further, though we believe it is important.

4.2 - A program with critical sections

For problems where we are only interested in the output of the last task w_n, the use of the global arrays $U[1:n]$ and $M[1:n+1]$ in Program A can be avoided at the expense of using critical sections.

We will illustrate the idea with the following example. Consider the problem of generating the n-th iterate x_n by $x_i := \varphi(x_{i-1})$ given the initial iterate x_0. Suppose that we use Program A. Then, corresponding to the global array $U[1:n]$, we have the global array $x[0:n]$ where $x[i]$ keeps the value of the i-th iterate, and instructions (4.3) and (4.4) become

$$x[m_j] := \varphi(x[m_j-1]) .$$

Note that we only need $x[n]$. The use of the array $x[0:n]$ is wasteful in space, and might even be impractical (e. g., when n is large or when the elements $x[0]$, ..., $x[n]$ are themselves vectors or complicated structures). The following program eliminates this problem.

Program B:

```
global integer m; global real x;

Initialization:
    begin
    m := 1; x := x0;
    start processes P1, ..., Pk
    end
```

Process P_j:

```
begin integer m_j; real y_j;
{m_j := m; y_j := x};                                   (4.7)
while m_j ≤ n do
    begin
    y_j := φ(y_j);
    {if m_j = m then (m := m_j; x := y_j)};             (4.8)
    {m_j := m; y_j := x}                                (4.9)
    end
end
```

It is crucial to assume that the statements enclosed within a pair of curly brackets (lines (4.7), (4.8) and (4.9)) are programmed as critical sections. (As a matter of fact, the two lines (4.8) and (4.9) can be programmed as one critical section.) With this assumption it is possible to prove the correctness of the above program. The proof is based on the observation that the global variable m is a non-decreasing function of time which takes on all integer values between 1 and $n+1$. The proof is relatively easy and hence is omitted here.

Note that, as was already mentioned, x and y_j may represent large amount of data. Hence the execution of $x := y_j$ or $y_j := x$ may take a significant amount of time. After presenting, in Section 5, an analysis for programs which do not have critical sections, we will give, in Section 6, an analysis for programs which do have critical sections.

5 - Speed-up ratios: Implementations without critical sections

Let $t_{i,j}$ be the random variable representing the time to execute task w_i by process P_j. *In this and the next section, we assume that the $t_{i,j}$, for $i = 1, ..., n$ and $j = 1, ..., k$, are independent and identically distributed.* The assumption is reasonable when all tasks are of the same complexity and executed by identical processors. We will use T to denote any of the random variables $t_{i,j}$, and use τ to denote the mean of T.

It is easy to obtain $\overline{T}_1(n)$. By equation (3.1) with $k = 1$, we have:

$$T_1(n) = \tau_1(1) + \tau_1(2) + \ldots + \tau_j(n) .$$

Since, in this case, the $\tau_1(i)$ are independent and identically distributed with mean τ, we deduce that

$$\overline{T}_1(n) = n\tau . \tag{5.1}$$

In the rest of the chapter, in order to evaluate $\overline{T}_k(n)$, we impose the following further assumptions:

A1. All processes start at the same time $t = 0$. (I. e, at t_0 all the k processes start with the execution of task w_1.)

A2. The random variable T is exponentially distributed with mean τ.

We observe that by the independence of the $t_{i,j}$ and by assumption A2 the quantities $\tau_k(i)$, $i = 1, \ldots, n$, are independent random variables. It follows, from equation (3.1), and assumption A2, that

$$\overline{T}_k(n) = \overline{\tau}_k(1) + \overline{\tau}_k(2) + \ldots + \overline{\tau}_k(n) , \tag{5.2}$$

where $\overline{\tau}_k(i)$ is the mean of $\tau_k(i)$.

In addition, by assumption A1, $\tau_k(1)$ is given by the minimum of k random variables distributed as T. Since T is exponentially distributed, the minimum has the mean:

$$\overline{\tau}_k(1) = \frac{\tau}{k} . \tag{5.3}$$

We now consider $\tau_k(i+1)$ for $i = 1, \ldots, n-1$. Define the distribution probability $p_{k,j}$, $j = 1, 2, \ldots$, as follows. (We use here the same notation as in Section 3.) Let $p_{k,j}$ be the probability that $s_{k,i+1} = t_{r+j}$, given that $s_{k,i} = t_r$ for some r. Hence for $j = 1, \ldots, k$, $p_{k,j}$ is the probability that conditions (a) and (b) of Theorem 3.1 hold. Using the same argument as used in the proof of Theorem 3.1, it is easy to show that $p_{k,j} = 0$ if $j > k$. In addition, assumption A2 implies that, from the memory-less property of the exponential distribution, $p_{k,j}$ is independent of i and r. We have:

$$\tau_k(i+1) = \begin{cases} t_{r+1} - t_r & \text{with probability } p_{k,1}, \\ (t_{r+1} - t_r) + (t_{r+2} - t_{r+1}) & \text{with probability } p_{k,2}, \\ \cdots & \cdots \\ (t_{r+1} - t_r) + \cdots + (t_{r+k} - t_{r+k-1}) & \text{with probability } p_{k,k}. \end{cases} \tag{5.4}$$

Since by assumption A2 the random variables $t_{r+1} - t_r$, $r = 1, 2, \ldots$, are independent (and identically distributed) random variables with mean $\frac{1}{k}\tau$, we derive from equation (5.4) that, for $i = 1, \ldots, n-1$, the mean of $\tau_k(i+1)$ is given by:

$$\bar{\tau}_k(i+1) = \sum_{1 \le j \le k} \left(j \frac{\tau}{k}\right) . p_{k,j} = \frac{\tau}{k} \sum_{1 \le j \le k} j . p_{j,k} \tag{5.5}$$

By equations (5.2), (5.3) and (5.5), we obtain that

$$\bar{T}_k(n) = \frac{1}{k} \tau \left(1 + (n-1) \sum_{1 \le j \le k} j . p_{j,k}\right) . \tag{5.6}$$

To evaluate $\bar{T}_k(n)$, we need to know the following quantity:

$$N_k = \sum_{1 \le j \le k} j . p_{j,k} .$$

Lemma 5.1:

For $j = 1, \ldots, k$:

$$p_{j,k} = \frac{j . k!}{k^{j+1}(k-j)!} . \tag{5.7}$$

Proof:

We first observe that, by assumption A2, for $r = 1, 2, \ldots$, any one of the k processes is equally likely to complete a task at time t_r. Suppose that $s_{k,i} = t_r$ and $s_{k,i+1} = t_{r+j}$. Then, by condition (a) of Theorem 3.1, the j processes completing tasks at time $t_r, t_{r+1}, \ldots, t_{r+j-1}$ are different. This occurs with probability

$$\frac{k}{k} \times \frac{(k-1)}{k} \times \cdots \times \frac{(k-j+1)}{k} = \frac{k!}{k^j (k-j)!} . \tag{5.8}$$

Moreover, by condition (b) of Theorem 3.1, the process completing a task at time t_{r+j} must be one of the j processes mentioned above. This occurs with probability j/k. Hence the probability that $s_{k,i} = t_r$ and $s_{k,i+1} = t_{r+j}$ is:

$$\frac{j}{k} \times \frac{k!}{k^j (k-j)!} . \qquad \blacksquare$$

The problem of computing the leading terms in the asymptotic series for N_k is rather difficult. Fortunately, some known results can be used here. Define

$$Q_k = \sum_{1 \le j \le k} \frac{k!}{k^j (k-j)!} .$$

We are now able to establish the following.

Lemma 5.2:

$$N_k = Q_k .$$

Proof:

We have

$$N_k = \sum_{1\le j\le k} j.p_{k,j} = \sum_{1\le j\le k} [k-(k-j)].p_{k,j}$$

$$= k \sum_{1\le j\le k} p_{k,j} - \sum_{1\le j\le k} (k-j).p_{k,j}$$

$$= \sum_{1\le j\le k} \frac{j.k!}{k^j(k-j)!} - \sum_{1\le j\le k-1} \frac{j.k!}{k^{j+1}(k-j-1)!}$$

$$= \sum_{1\le j\le k} \frac{j.k!}{k^j(k-j)!} - \sum_{1\le j\le k} \frac{(j-1)k!}{k^j(k-j)!}$$

$$= \sum_{1\le j\le k} \frac{k!}{k^j(k-j)!} .$$

■

The leading terms in the asymptotic series for Q_k are known [34, p. 118]:

$$Q_k = \sqrt{\frac{\pi k}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2k}} + O(\frac{1}{k}) .$$

Hence, by equations (5.1), (5.6) and Lemma 5.2, we have the following theorem.

Theorem 5.1

Using k processes, the speed-up ratio is given by

$$S_k(n) = \frac{n.k}{1 + (n-1)N_k} ,$$

where

$$N_k = \sqrt{\frac{\pi k}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2k}} + O(\frac{1}{k}) .$$

Asymptotically, when both n and k are large, we obtain:

$$S_k(n) \sim \sqrt{\frac{2k}{\pi}} \sim 0.798\sqrt{k} .$$

6 - Speed-up ratios: Implementations with critical sections

In this section, we analyze speed-up ratios achievable by the algorithms when they are implemented with critical sections.

The diagram of Figure 6.1 illustrates a portion of a possible scheduling of the tasks by the parallel algorithm with two processes.

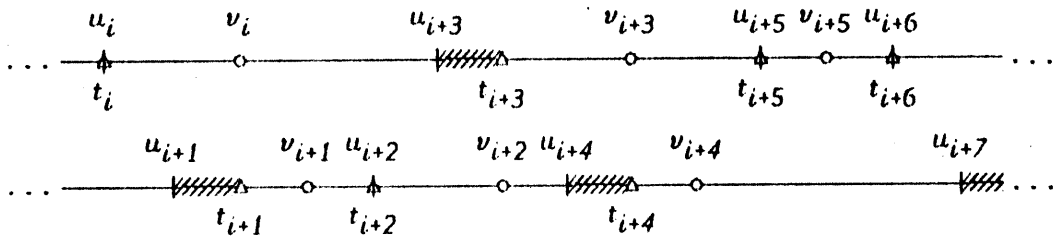


Figure 6.1 - A possible task scheduling with two processes

In the diagram, the marks '—|—' and '—o—' indicate the sequences of time instants u_i and v_i, $i = 1, 2, \ldots$, when a process completes a task and when the same process completes the subsequent critical section. Since, at any time, only one process can execute the critical section, a process may have to wait before entering the critical section. The periods of waiting times are indicated by the marks '////'. The time instants t_i when processes actually enter the critical section are indicated by the marks '—△—'.

As in the preceding section, we assume that the time a process takes to execute a task is a random variable independent of the process and of the task. Let F be its distribution function, and f its density function. Similarly, we assume that the time a process takes to execute the critical section is a random variable independent of the process. Let B be its distribution function and b its density function. Furthermore, let τ and β denote the average execution times for a task and for the critical section, respectively.

In the following we derive a general formula for evaluating the speed-up ratio achievable by the parallel algorithm with two processes for the case when F *is an exponential distribution function and* B *is a general distribution function.*

Observe that at time t_i when a process enters the critical section, the second process is necessarily performing some task (possibly just starting a task). Since the

distribution function F is exponential, at time t_i the remaining execution time for the task performed by the second process is distributed according to the same distribution function F. Therefore the evolution of the processes, from time t_i on, is independent of the past for any distribution B. In particular, the random variables $t_{i+1} - t_i$, for $i = 1, 2, \ldots$, are independent and identically distributed, and the same holds for the random variables $\tau_k(i+1)$, for $i = 1, 2, \ldots$, defined in Section 3.

In this section, let $T_1(n)$ and $T_2(n)$ denote the time to complete task w_n *and the subsequent critical section* by the sequential algorithm and the parallel algorithm with two processes, respectively. Let $\overline{T}_1(n)$ and $\overline{T}_2(n)$ denote their means. It follows from the above discussion that, for $k = 1$ and 2, we have:

$$\overline{T}_k(n) = \overline{\tau}(1) + \overline{\tau}(2) + \ldots + \overline{\tau}(n) + \beta , \tag{6.1}$$

where the last term, β, accounts for the time to execute the last critical section (after the completion of task w_n).

Consider first the sequential algorithm. In this case, we simply have $\overline{\tau}(1) = \tau$, and, for $i = 2, \ldots, n$, $\overline{\tau}(i) = \beta + \tau$. Therefore, by equation (6.1):

$$\overline{T}_1(n) = n(\tau + \beta) . \tag{6.2}$$

(Here we ignore the fact that in the sequential algorithm the critical section can be shortened, since there is no need to include synchronization primitives.)

Consider now the parallel algorithm. As with equation (5.3), we have:

$$\overline{\tau}_2(1) = \frac{1}{2}\tau . \tag{6.3}$$

For $j = 1$ and 2, let p_j be the probability that $s_{2,i+1} = t_{r+j}$, given that $s_{2,i} = t_r$ for some r. As in Section 5, by Theorem 3.1, we obtain, for $i = 1, \ldots, n-1$,

$$\tau_2(i+1) = \begin{cases} t_{r+1} - t_r & \text{with probability } p_1 , \\ (t_{r+1} - t_r) + (t_{r+2} - t_{r+1}) & \text{with probability } p_2 , \end{cases} \tag{6.4}$$

We have already mentioned that the random variables $t_{r+1} - t_r$, $r = 1, 2, \ldots$, are independent and identically distributed. Let μ denote their mean. It follows from equation (6.4) that the mean of $\tau_2(i+1)$ is given by:

$$\bar{\varepsilon}_2(i+1) = \mu.p_1 + 2\mu.p_2 = (2-p_1).\mu \,, \tag{6.5}$$

since $p_1 + p_2 = 1$.

The following lemma establishes the values of μ and p_1.

Lemma 6.1:

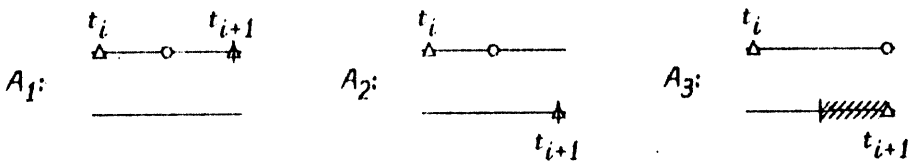
Let B^* denote the Laplace transform of the distribution function B. We have:

$$\mu = \beta + \frac{\varepsilon}{2} B^*\left(\frac{1}{\varepsilon}\right), \tag{6.6}$$

$$p_1 = \frac{1}{2} B^*\left(\frac{1}{\varepsilon}\right). \tag{6.7}$$

Proof:

We consider transitions for passing from time t_i to time t_{i+1}. Up to a permutation of the processes, there are three possible transitions as defined by the following diagrams:



where the notation of Figure 6.1 is assumed.

Let $H_j(t)$, $j = 1$, 2, and 3, be the probability that transition A_j takes place and that $t_{i+1} - t_i \le t$. We have:

$$H_1(t) = \int_0^t [1 - F(x)] \int_0^x b(y)\, f(x-y)\, dy\, dx \,,$$

$$H_2(t) = \int_0^t f(x) \int_0^x b(y)\, [1 - F(x-y)]\, dy\, dx \,,$$

$$H_3(t) = \int_0^t b(x)\, F(x)\, dx \,.$$

But we observe that $H(t) = H_1(t) + H_2(t) + H_3(t)$ is the distribution function for $t_{i+1} - t_i$ and that the same process enters the critical section at both times t_i and t_{i+1} only with transition A_1. Hence:

$$\mu = \int_0^\infty t\, dH(t) = \int_0^\infty [1 - H(t)]\, dt \,,$$

$$p_1 = \int_0^\infty dH_1(t) = \int_0^\infty [1 - F(x)] \int_0^x b(y)\, f(x-y)\, dy\, dx \,,$$

from which equations (6.6) and (6.7) follow easily. ■

By collecting the preceding results, we obtain the following theorem.

Theorem 6.1:

The speed-up ratio of the parallel algorithm with two processes is given by:

$$S_2(n) = \frac{n(\tau+\beta)}{(n-1)[2-\frac{1}{2}B^*(\frac{1}{\tau})][\beta+\frac{\tau}{2}B^*(\frac{1}{\tau})]+\frac{\tau}{2}+\beta}$$

$$= \frac{1}{2-\frac{1}{2}B^*(\frac{1}{\tau})} \times \frac{\tau+\beta}{\beta+\frac{\tau}{2}B^*(\frac{1}{\tau})} + O(\frac{1}{n}).$$

We give below $B^*(\frac{1}{\tau})$ for some distribution functions B.

(i) B is exponential (with parameter $1/\beta$):

$$B^*(\frac{1}{\tau}) = \frac{\tau}{\tau+\beta}.$$

(ii) B is uniform over $[a, b]$:

$$B^*(\frac{1}{\tau}) = \frac{e^{-a/\tau}-e^{-b/\tau}}{(b-a)/\tau}.$$

(iii) B is the Dirac function at the point β:

$$B^*(\frac{1}{\tau}) = e^{-\beta/\tau}.$$

In Figure 6.2, we have plotted the asymptotic speed-up ratio S_2 as a function of the ratio $\alpha = \tau/(\tau+\beta)$ for the three distributions mentioned above (in the second case, a and b have been chosen as $\beta/2$ and $3\beta/2$, respectively).

When α tends to 0 (or β tends to infinity), the algorithm approaches its worst case performance, since the evaluations of the two processes tend to be exactly interleaved. When $\alpha = 1$ (or $\beta = 0$), the critical section is non-existent and we have the results of Section 5.

We observe from Figure 6.2 that the best speed-up ratio is always obtained when B is an exponential distribution (the first case). We also note that the results obtained for the two other cases are very close to each other and close to the results obtained with the exponential distribution. This suggests that the results obtained with the exponential distribution could be used as approximations to results obtained with other distributions.

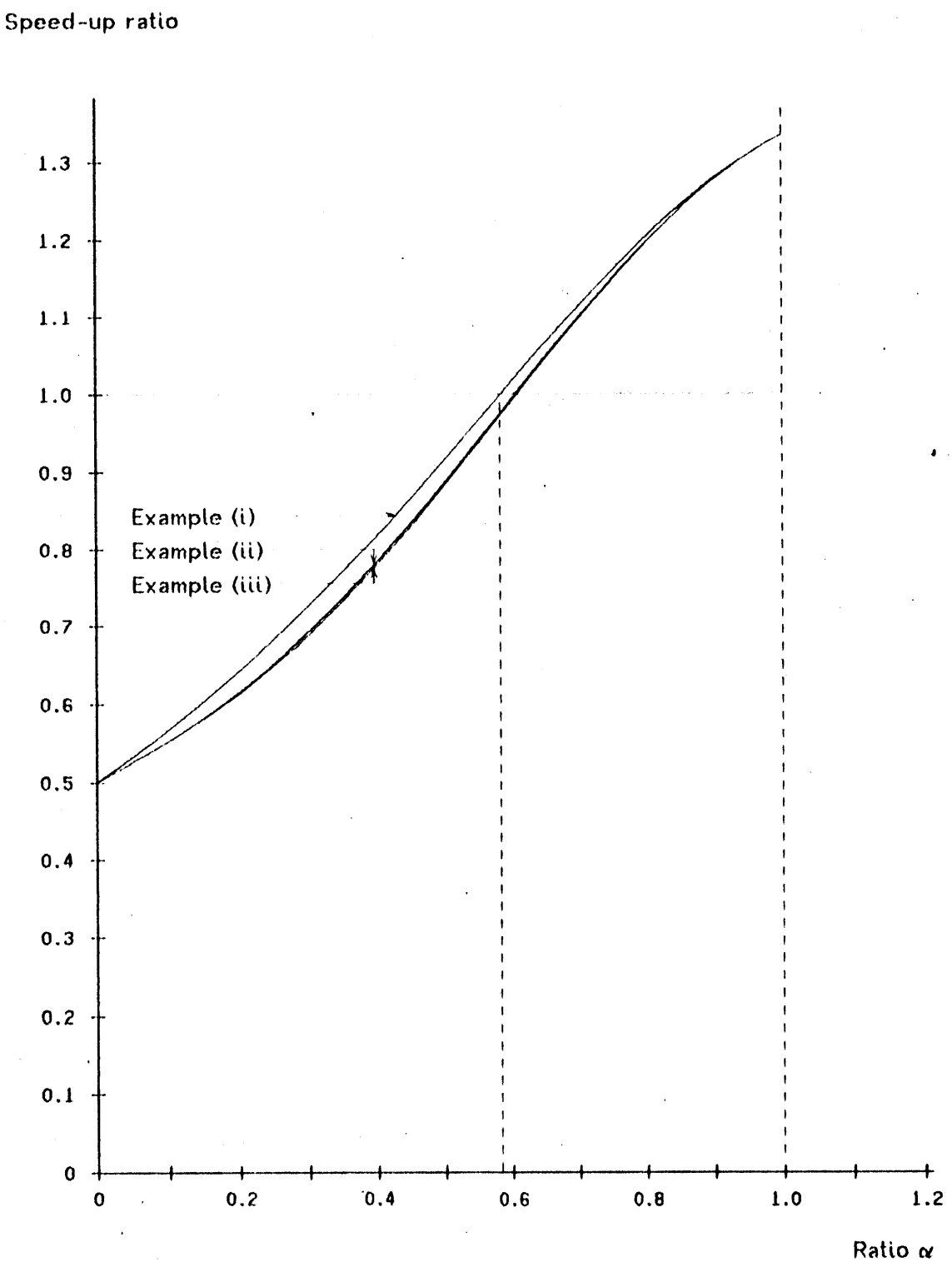


Figure 6.2 - Speed-up ratio with 2 processes for various distributions B

We can observe from Figure 6.2 that, unlike the implementation without critical section, better speed-up is not necessarily achieved by using more processes, though we assume that a processor is always available to each process! More precisely, the figure

indicates that (when B is an exponential distribution) in order to achieve the best speed-up when two processors are available, one should create two processes when $\alpha > 0.586$, but only one process when $\alpha \leq 0.586$. Similar results are useful in practice, since they can be used to determine the optimal number of processes to create in order to minimize the overall execution time.

7 - Conclusions and open problems

In recent years, research in parallel algorithms has dealt mostly with synchronized array or vector processors such as the ILLIAC IV or the CDC STAR, and there are very few results on the design and analysis of algorithms for asynchronous multiprocessors. In this chapter, we have proposed a novel method of using asynchronous multiprocessors which takes advantage of their asynchronous behavior. We have also presented analytic techniques to evaluate the performance of an asynchronous algorithm using the method. The algorithm is expected to achieve a large speed-up when the fluctuations in the task execution times are relatively large. Moreover, as noted in Section 4, the algorithm has a nice reliability property. The same idea may also be used to construct other reliable algorithms.

For the implementation with critical sections we obtained analytic results for two processes. The results show that the parallel algorithm using two processes is not necessarily faster than the sequential algorithm, because of the critical section overheads associated with the parallel algorithm. This confirms the practical experience that the speed-up ratio does not necessarily increase as the number of processes increases. It would be interesting to extend our analytic results for more than two processes. We have chosen to deal with a simple problem by imposing the condition that the tasks are linearly ordered. An interesting extension would be to consider a set of tasks (possibly generated dynamically) which are ordered by a directed graph (i. e., partially rather than linearly ordered). Another interesting extension would be to design algorithms where the

execution of a task by a process may be interrupted by another process. We expect that this approach would result in more efficient algorithms, since processes which are not doing useful work can be interrupted. A careful performance analysis including the additional overheads introduced by the interruption mechanism is needed here. This problem has been addressed in two recent papers by Barak and Downey [3] and [4].

Finally, we note that the results of this chapter are not restricted to multiprocessor systems. The ideas can be used to solve any problem in Operations Research which satisfies conditions similar to C1, C2 and C3.

Chapter III

Asynchronous Iterative Methods for Multiprocessors[1]

1 - Introduction

In this chapter we investigate the fixed point problem for an operator F from $\mathbb{R}^n$ into itself: we want to find a vector x in $\mathbb{R}^n$ which satisfies the system of equations represented by

$$x = F(x). \tag{1.1}$$

In [11], Chazan and Miranker introduced the *chaotic relaxation scheme*, a class of iterative methods for solving equation (1.1) where F is a linear operator given by $F(x) = Ax + b$. They showed that iterations defined by a chaotic relaxation scheme converge to the solution of equation (1.1) if and only if $\rho(|A|) < 1$. (If M is a real $n \times n$ matrix, $\rho(M)$ denotes its spectral radius and $|M|$ denotes the non-negative $n \times n$ matrix obtained by replacing the elements of M by their absolute values.)

In [41] and [43], Miellou generalized the chaotic relaxation scheme to include non-linear operators and obtained convergence results similar to those of [11] in the case of *contracting operators* (see, for example, [46, p. 433]).

In [11], [41] and [43], the motivation of defining chaotic relaxation is to account for the parallel implementation of iterative methods on a multiprocessor system so as to

 This chapter appeared in *Journal of the ACM*, Vol. 25, No. 2, April 1978, pp. 226-244.

reduce communication and synchronization between the cooperating processes. This reduction is obtained by not forcing the processes to follow a predetermined sequence of computations, but simply by allowing a process, when starting the evaluation of a new iterate, to choose dynamically not only the components to be evaluated but also the values of the previous iterates used in the evaluation.

The chaotic relaxation scheme does not, however, allow for a completely arbitrary choice of the antecedent values used in the evaluation of an iterate. A restriction is that there must exist a *fixed* positive integer s such that, in carrying out the evaluation of the i-th iterate, a process cannot make use of any value of the components of the j-th iterate if $j < i-s$. We will show that this condition can be replaced by a more general one, which still guarantees the convergence of the iteration.

In the next section we introduce the class of *asynchronous iterative methods* which relaxes the assumption mentioned above, and we show that existing iterative methods (and, in particular, the chaotic relaxation) can be represented as special cases of asynchronous iterations. Section 3 gives the definition and reviews some properties of *contracting operators*. Then the theorem of Section 4 generalizes the sufficient condition on the convergence of the chaotic relaxation obtained by Chazan and Miranker [11] and by Miellou [41] and [43]. This result is further extended, in Section 5, to include iterative methods with memory. In Section 6, we consider the complexity of asynchronous iterative methods, and we derive bounds on the efficiency. These bounds are then compared with actual measurements of asynchronous iterations. The experimental results, presented in Section 7, show a considerable advantage for iterations making no use of synchronization. Section 8 is devoted to the study of an asynchronous iteration showing super-linear convergence and, through a specific analysis, we give lower bounds on the order of convergence and on the efficiency. Possible extensions of the results are discussed in Section 9, and concluding remarks are presented in the last section.

2 - The class of asynchronous iterative methods

The following notations will be used throughout the chapter. If x is a vector of $\mathbb{R}^n$, its components will be denoted by x_i, $i = 1, ..., n$. To avoid confusion, a sequence of vectors of $\mathbb{R}^n$ will be denoted by $x(j)$, $j = 0, 1, ...$ If F is an operator of $\mathbb{R}^n$ into itself, $F(x)$ will also be represented in components by $f_i(x)$ or by $f_i(x_1, ..., x_n)$, $i = 1, ..., n$. We denote by $\mathbb{N}$ the set of all non-negative integers.

2.1 - Definition of asynchronous iterative methods

The definition of *chaotic iteration* is originally due to Chazan and Miranker [11], and the definition we give below for *asynchronous iteration* is similar to their definition.

Definition 2.1:

Let F be an operator from $\mathbb{R}^n$ to $\mathbb{R}^n$. An *asynchronous iteration* corresponding to the operator F and starting with a given vector $x(0)$ is a sequence $x(j)$, $j = 0, 1, ...,$ of vectors of $\mathbb{R}^n$ defined recursively by:

$$x_i(j) = \begin{cases} x_i(j-1) & \text{if } i \notin J_j \\ f_i(x_1(s_1(j)), ..., x_n(s_n(j))) & \text{if } i \in J_j , \end{cases} \tag{2.1}$$

where $\mathcal{J} = \{ J_j \mid j = 1, 2, ... \}$ is a sequence of non-empty subsets of $\{1, ..., n\}$ and $\mathcal{S} = \{ (s_1(j), ..., s_n(j)) \mid j = 1, 2, ... \}$ is a sequence of elements in $\mathbb{N}^n$.

In addition, $\mathcal{J}$ and $\mathcal{S}$ are subject to the following conditions:

for each $i = 1, ..., n$

(a) $s_i(j) \leq j-1$, $j = 1, 2, ...,$

(b) $s_i(j)$, considered as a function of j, tends to infinity as j tends to infinity,

(c) i occurs infinitely many often in the sets J_j, $j = 1, 2, ...$

An asynchronous iteration corresponding to F, starting with $x(0)$ and defined by $\mathcal{J}$ and $\mathcal{S}$ will be denoted by $(F, x(0), \mathcal{J}, \mathcal{S})$. ■

In the definition of *chaotic iterations*, Chazan and Miranker [11] use the following condition

(b') there exists a fixed integer s such that $j - s_i(j) \leq s$ for $j = 1, 2, \ldots$ and $i = 1, \ldots, n$,

in lieu of condition (b). Clearly, condition (b') implies condition (b), and, in this sense, asynchronous iterations provide a generalization of chaotic relaxations.

An asynchronous iteration $(F, x(0), J, S)$ may be thought of as corresponding to the following sequence of computations on an asynchronous multiprocessor.

Assume we have a pool of processors available. Let t_j, $j = 1, 2, \ldots$, be an increasing sequence of time instants. At time t_j processor P is idle and is assigned to the evaluation of the iterate $x(j)$, $x(j)$ differs from $x(j-1)$ by the set of components $\{ x_i \mid i \in J_j \}$ and P starts computing these components using values of components known from previous iterates, namely the r-th component of the $s_r(j)$-th iterate, for $r = 1, \ldots, n$. The choice of the components may be guided by any criterion, and, in particular, a natural criterion is to pick up the most recently available values of the components. This scheme does not require any synchronization between the processes. At some time t_k, later on ($k > j$), P will finish its computations and will be assigned to a new evaluation: $x(k)$.

The use of asynchronous iterative methods is obviously not restricted to multiprocessor systems, and the scheme is also well suited for execution on a network of computers, in particular, when the communication between elements of the network is not too expensive as opposed to the computation itself.

We notice that, in the evaluation of an iterate, nothing is imposed on the use of the values of the previous iterates. The only thing required, by condition (b) of the definition, is that, eventually, the values of an early iterate cannot be used any more in further evaluations, and more and more recent values of the components have to be used instead. On a multiprocessor, this condition can be satisfied as long as no processor crashes (and eventually completes its computation).

Condition (a) of the definition states the fact that only components of previous iterates can be used in the evaluation of a new iterate. Condition (c) guarantees that no component be abandoned forever.

2.2 - Examples and particular cases of asynchronous iterations

Classical iterative methods: point or block Jacobi, Gauss-Seidel, etc., as well as others introduced more recently: *chaotic relaxation scheme* [11], *periodic chaotic scheme* [18], *itération chaotique à retards* [41] and [43], *itération chaotique série-parallèle* [50], can all be seen as particular cases of asynchronous iterations.

For example, the point-Jacobi method defined on the operator F with the initial approximation $x(0)$ can be represented by the asynchronous iteration $(F,x(0),\mathcal{J},\mathcal{S})$ where $\mathcal{J}$ and $\mathcal{S}$ are defined by:

$$J_j = \{ 1, \ldots, n \} \text{ for } j = 1, 2, \ldots ,$$

$$s_i(j) = j-1 \text{ for } j = 1, 2, \ldots \text{ and } i = 1, \ldots, n .$$

The same point-Jacobi method can equivalently be represented by the asynchronous iteration where $\mathcal{J}$ and $\mathcal{S}$ are defined by:

$$J_j = \{ 1 + (j-1 \bmod n) \} \text{ for } j = 1, 2, \ldots ,$$

$$s_i(j) = n \lfloor (j-1)/n \rfloor \text{ for } j = 1, 2, \ldots \text{ and } i = 1, \ldots, n .$$

Although those two representations correspond to the same point-Jacobi method, they differ by the implicit information they contain about the decomposition of the computations. In the first case, all components are evaluated at once and this, presumably, will be done by one computational process. In the second case, however, each component is evaluated separately, and up to n processes can be used to perform the evaluations. Between the two extreme representations of the point-Jacobi method, in terms of asynchronous iterations, several others can be proposed, each of which can be interpreted in terms of decomposition into computational processes and in terms of implementation by concurrent processes.

The iterative method proposed by Robert, Charnay and Musy (*itération chaotique série-parallèle* [50]) can be obtained as a special case of an asynchronous iteration in which $s_i(j) = j-1$ (for all $i = 1, ..., n$ and $j = 1, 2, ...$). This corresponds to a strictly sequential computation of sets of components. The choice of the components within a set is arbitrary and the calculations of their values can be done simultaneously but the evaluation of a new set of components cannot be started before all components of the previous set have been computed and their new values relaxed. The goal of their research was to show that, for example, in the iterative solution of linear systems resulting from the application of the method of finite differences to partial differential equations, it is possible to concentrate the computations more on those points of the grid where the convergence is slower than on other nodes. This is not the case with ordinary iterative methods for which any component is iterated as many times as any other component.

Chazan and Miranker [11] have proposed a *chaotic relaxation scheme* to solve a linear system. As we have already mentioned, our definition of an asynchronous iterative method is similar to the definition they give for a chaotic iterative scheme. Our definition, however, does not require the condition that $j-s_i(j)$ has to be uniformly bounded by some fixed integer, say s, (for all $i = 1, ..., n$ and $j = 1, 2, ...$). This assumption, however, happens to be satisfied in most usual implementations, with small values for s. It will be useful in Sections 6 and 7, and we will use this assumption explicitly in order to derive bounds on the rate of convergence and on the efficiency of various methods implemented on an asynchronous multiprocessor.

Although all chaotic relaxation methods (as presented in [11], [41] and [43]) can be identified as asynchronous iterations, the converse is not true as is illustrated by the following example. Let F be an operator from $\mathbb{R}^2$ into itself. Assume we have two processes P_1 and P_2 attached to the evaluations of the first and second components, respectively. To avoid synchronization, the processes always use in an evaluation the

values of the components currently available at the begining of the computation. If we assume that it always takes 1 unit of time for P_1 to perform the evaluation of x_1 and it takes k units of time for P_2 to perform the k-th evaluation of x_2, then the quantity $j - s_2(j)$ grows as $\sqrt{j}$ which is unbounded. This iteration is a legitimate asynchronous iteration, it is not, however, allowed in the setting of [11], [41] and [43].

3 - Contracting operators

In the next section we shall give a sufficient condition on the operator F for the convergence of any asynchronous iteration. Needed definitions are given in this section.

3.1 - Lipschitzian and contracting operators

Contracting operators, to be defined below, correspond to *P-contractions* in [46, p. 433]. They seem to have been first introduced by Kantorovitch, Vulich and Pinsker in [31], and they have been further studied by Robert [49]. The notion was used in particular to obtain the results of [10], [41], [43] and [50].

Definition 3.1:

An operator F from $\mathbb{R}^n$ to $\mathbb{R}^n$ is a *Lipschitzian operator* on a subset D of $\mathbb{R}^n$ if there exists a non-negative $n \times n$ matrix A such that:

$$|F(x)-F(y)| \leq A|x-y| \ , \ \forall \ x, y \in D \ , \tag{3.1}$$

where, if z is a vector of $\mathbb{R}^n$ with components z_i, $i = 1, ..., n$, $|z|$ denotes the vector with components $|z_i|$, $i = 1, ..., n$, and the inequality holds for every component.

The matrix A will be called a *Lipschitzian matrix* for the operator F. ■

From this definition we can see that any Lipschitzian operator is continuous and, in fact, uniformly continuous on D. However, this definition is too broad and, in particular, we are not guaranteed of the existence and of the uniqueness of a fixed point as is shown by the following example. Take the operator F from $\mathbb{R}$ to $\mathbb{R}$ defined by $F(x) = \sqrt{x^2+a^2}$, this operator is Lipschitzian on $\mathbb{R}$ because

$$|F(x)-F(y)| = |(x-y)[(x+y)/(\sqrt{x^2+a^2} + \sqrt{y^2+a^2})]| \leq |x-y| , \quad \forall\, x, y \in \mathbb{R} .$$

However, the equation $x = \sqrt{x^2+1}$ (corresponding to $a = 1$) has no solution. On the other hand, the equation $x = |x|$, (corresponding to $a = 0$) has an infinity of solutions, and, in fact, a continuum of solutions.

We will, therefore, restrict ourselves to the following class of operators.

Definition 3.2:

An operator F from $\mathbb{R}^n$ to $\mathbb{R}^n$ is a *contracting operator* on a subset D of $\mathbb{R}^n$ if it is a Lipschitzian operator on D with a Lipschitzian matrix A such that $\rho(A) < 1$ (where $\rho(A)$ is the spectral radius of A).

The matrix A will be called a *contracting matrix* for the operator F. ■

The fact that, unlike Lipschitzian operators, contracting operators are guaranteed to have a unique fixed point in the subset D can be easily derived from the definition. In addition, if we assume, for example, that D is closed and that $F(D)$ is a subset of D, we are also guaranteed of the existence of a fixed point in the subset D. A proof can be found in [46, pp. 433-434].

3.2 - Examples of contracting operators

Let F be a linear operator given by $F(x) = Ax + b$, where A is an $n \times n$ matrix and b is a vector of $\mathbb{R}^n$. We observe that F is a contracting operator if and only if $\rho(|A|) < 1$. Therefore, in the case of linear operators, the notion of contracting operators coincides with the property stated by Chazan and Miranker for their convergence result [11], and their result will appear as a particular case of the theorem of the next section.

We could have considered a more general definition for asynchronous iterative methods by introducing a relaxation factor $\omega > 0$. This would simply consist of replacing, in equations (2.1), the operator F by the operator $F_\omega = \omega F + (1-\omega)E$, where E is the identity operator of $\mathbb{R}^n$. It follows that

$$|F_\omega(x)-F_\omega(y)| \le \omega|F(x)-F(y)| + |1-\omega||x-y| ,$$

and, if F is a contracting operator with a contracting matrix A, F_ω is a Lipschitzian operator with the Lipschitzian matrix $A_\omega = \omega A + |1-\omega|I$. The matrix A being non-negative we have $\rho(A_\omega) = \omega\rho(A) + |1-\omega|$, and, if we choose

$$0 < \omega < 2/[1+\rho(A)] , \tag{3.2}$$

F_ω is also a contracting operator. In particular, as long as condition (3.2) is satisfied, the results of the next section also apply to asynchronous iterative methods with relaxation. Condition (3.2) is classical and is mentioned, in particular, in [11, p. 221], [43, p. 62], and [50, p. 31].

If we consider a linear system of equations derived from a linear elliptic differential equation by the method of finite differences, we note that the system is represented by $Ax = b$, where b is a vector of $\mathbb{R}^n$ obtained from the boundary conditions and A is an $n\times n$ M-matrix (see, for example, [62, p. 85]). Therefore the system can be written into the form of equation (1.1) in which F is the contracting operator given by $F(x) = (I - D^{-1}A)x + D^{-1}b$, where D is the matrix composed of the diagonal elements of A. This example shows, in the case of linear operators, the importance of contracting operators.

On the other hand, non-linear contracting operators, too, constitute a very important class. A first example is directly derived from the previous one. Elliptic partial differential equations, obtained by the addition of a small non-linear perturbation to a linear partial differential equation, can also be shown to give rise to (non-linear) contracting operators.

More important, if G is a non-linear operator from $\mathbb{R}^n$ into itself with the simple root ξ, superlinear iterative methods have been devised to find the root ξ of G, provided that an initial approximation $x(0)$ sufficiently close to ξ is already known. For example, Newton iterative method generates the sequence of iterates

$$x(i+1) = F(x(i)) = x(i) - [G'(x(i))]^{-1}G(x(i)) , \quad \text{for } i = 0, 1, \dots ,$$

which converges quadratically to the root ξ of G. In this particular example, we can easily derive, under usual assumptions (for example, G' satisfies some Lipschitz condition in a neighborhood of ξ), that the Newton operator F corresponding to G is a contracting operator. (This result will be derived in a more general context in Section 8.)

In fact this result is very general. Let F be an operator from $\mathbb{R}^n$ into itself with a fixed point ξ. If we assume that F is continuously differentiable in the set $D_r = \{ x \mid \|x-\xi\| < r \}$ and that the derivative F' vanishes at ξ and satisfies a Lipschitz condition

$$\|F'(x)-F'(x)\| \leq M\|x-y\| , \quad \forall\, x, y \in D_r ,$$

then it can be easily shown that

$$\|F(x)-F(y)\| \leq 2Mr\|x-y\| , \quad \forall\, x, y \in D_r .$$

Therefore, by choosing the vector norm $\|x\| = |x_1| + \ldots + |x_n|$ (which only changes the constant M), the operator F is certainly a Lipschitzian operator with the Lipschitzian matrix $A = [a_{ij}]$ where $a_{ij} = 2Mr$, for $i, j = 1, \ldots, n$. In particular, if we know a sufficiently close approximation to the fixed point ξ (i. e., if r is small enough), the operator F is also a contracting operator. This shows that the class of contracting operators contains, under weak conditions, all iterative functions occurring in the classical superlinear iterative methods.

4 - Convergence theorem

Before stating a sufficient condition ensuring the convergence of an asynchronous iteration, we give a characterization of a non-negative matrix with spectral radius less than unity. The result is classical and an algebraic proof of this characterization can be found in [11, p. 218]. A shorter proof, based on the continuity of the spectral radius of a matrix as a function of its coefficients, is given below.

Lemma 4.1:

Let A be a non-negative square matrix. Then $\rho(A) < 1$ if and only if there exists a positive scalar ω and a positive vector v such that:

$$Av \leq \omega v \quad \text{and} \quad \omega < 1 . \tag{4.1}$$

Proof:

We first assume that (4.1) holds. In this case we note that $\|A\|_v \leq \omega < 1$, where the matrix norm $\|.\|_v$ is induced by the vector norm defined by:

$$\|x\|_v = \max\{ |x_i|/v_i \mid i = 1, ..., n \} .$$

Therefore the matrix A is convergent which implies $\rho(A) < 1$ (see, for example, [62, p. 13]).

Now assume that $\rho(A) < 1$. Let t be a non-negative scalar and A_t be the matrix obtained by adding t to all null coefficients of A. Clearly, for any positive vector x, we have $Ax \leq A_t x$. On the other hand, $\rho(A_t)$ is a continuous function of t. In particular, since $A_0 = A$ and $\rho(A) < 1$, we can always choose $t > 0$ small enough so that $\rho(A_t) < 1$ (in fact, we also have $\rho(A) \leq \rho(A_t)$). Then let $\omega = \rho(A_t)$. As $A_t > 0$, from Perron's theorem (see, for example, [62, p. 30]), there exists a positive eigenvector v corresponding to the eigenvalue ω. The positive scalar ω and the positive vector v verify $Av \leq A_t v = \omega v$ with $\omega < 1$. And this completes the proof. ■

This proof shows, in particular, that $\omega \geq \rho(A)$. But, we also see easily that the positive scalar ω can be chosen arbitrarily close to $\rho(A)$.

We are now able to state a sufficient condition on the operator F for the convergence of any asynchronous iteration corresponding to F. Similar results were first established for chaotic iterations, i. e., under condition (b'), by Chazan and Miranker [11] in the case of linear operators, and by Miellou [41] and [43] in the case of contracting operators. The proof given here follows the same idea as in [11, pp. 217-218].

Theorem 4.1:

If F is a contracting operator on a closed subset D of $\mathbb{R}^n$ and if $F(D)$ is a subset of D, then any asynchronous iteration $(F,x(0),J,\Delta)$ corresponding to F and starting with a vector $x(0)$ in D converges to the unique fixed point of F in D.

Proof:

Let ξ be the unique fixed point of F. By considering the operator $F(x+\xi)-\xi$, we may assume, without loss of generality, that $\xi = F(\xi) = 0$. By setting $y = \xi$ in equation (3.1), the Lipschitz condition on the operator F gives:

$$|F(x)| \leq A|x| , \quad \forall\, x \in D .$$

Let A be a contracting matrix for F and let ω and v be as defined in Lemma 4.1. Since v is a positive vector, for any starting vector $x(0)$ we can find a positive scalar α such that $|x(0)| \leq \alpha v$.

We will show that we can construct a sequence of indices j_p, $p = 0, 1, \ldots$, such that the sequence of iterates of $(F,x(0),J,\mathcal{S})$ satisfies:

$$|x(j)| \leq \alpha\omega^p v \quad \text{for} \quad j \geq j_p . \tag{4.2}$$

As $0 < \omega < 1$, this shows that $x(j) \to 0$ as $j \to \infty$ and this will prove the theorem.

We first show that inequality (4.2) holds for $p = 0$ if we choose $j_0 = 0$. That is, for $j \geq 0$ we have:

$$|x(j)| \leq \alpha v . \tag{4.3}$$

From the choice of α, inequality (4.3) is true for $j = 0$. Assume, for induction, that it is true for $0 \leq j < k$ and consider $x(k)$. Let z denote the vector with components $z_i = x_i(s_i(k))$, for $i = 1, \ldots, n$. From Definition 2.1, the components of $x(k)$ are given either by $x_i(k) = x_i(k-1)$ if $i \notin J_k$, in which case $|x_i(k)| = |x_i(k-1)| \leq \alpha v_i$, or by $x_i(k) = f_i(z)$ if $i \in J_k$. In this latter case, we note that, as $s_i(k) < k$ (condition (a) of Definition 2.1), we have:

$$|F(z)| \leq A|z| \leq \alpha A v \leq \alpha\omega v$$

and in particular:

$$|x_i(k)| = |f_i(z)| \leq \alpha\omega v_i .$$

As $0 < \omega < 1$, in this case too we obtain $|x_i(k)| \leq \alpha v_i$ and (4.3) is proved by induction, which shows that (4.2) is true for $p = 0$ if we choose $j_0 = 0$.

Now assume that j_p has been found and that inequality (4.2) holds for $0 \le p < q$. We want to find j_q and show that (4.2) also holds for $p = q$.

First define r by

$$r = \min\{ k \mid \forall j \ge k \;\; s_i(j) \ge j_{q-1} \text{ , for } i = 1, ..., n \} .$$

We see, from condition (b) of Definition 2.1, that this number exists, and we note that, from condition (a), we have $r > j_{q-1}$ which shows, in particular, that $|x(r)| \le \alpha\omega^{q-1}v$.

Then take $j \ge r$ and consider the components of $x(j)$. As above, let z be the vector with components $z_i = x_i(s_i(j))$. From the choice of r, we have $s_i(j) \ge j_{q-1}$, for $i = 1, ..., n$, and this shows that $|z| \le \alpha\omega^{q-1}v$. In particular, using the contracting property of the operator F we obtain:

$$|F(z)| \le A|z| \le \alpha\omega^{q-1}Av \le \alpha\omega^{q}v .$$

This inequality shows that, if $i \in J_j$, $x_i(j)$ satisfies:

$$|x_i(j)| = |f_i(z)| \le \alpha\omega^{q}v_i .$$

On the other hand, if $i \notin J_j$ the i-th component is not modified. Therefore, as soon as the i-th component is updated between the r-th and the j-th iteration we have:

$$|x_i(j)| \le \alpha\omega^{q}v_i . \qquad (4.4)$$

Now, define j_q as:

$$j_q = \min\{ j \mid j \ge r \text{ and } \{1, ..., n\} = J_r \cup ... \cup J_j \}$$

(this number exists by condition (c) of Definition 2.1), then for any $j \ge j_q$ every component is updated at least once between the r-th and the j-th iteration and therefore inequality (4.4) holds for $i = 1, ..., n$. This shows that inequality (4.2) holds for $p = q$ and this proves the theorem. ■

Considering only the class of linear operators, $F(x) = Ax + b$, Chazan and Miranker [11] have established a stronger result, namely, that the condition $\rho(|A|) < 1$ is also a necessary condition for the convergence of chaotic iterations.

5 - The class of asynchronous iterative methods with memory

The idea behind the definition of asynchronous iterations, as presented in Section 2, is to allow, in the evaluation of $F(x)$, different (and independent) processes to compute different subsets of the components. This corresponds to a natural decomposition for the evaluation of $F(x)$ when the operator F is known explicitly by the set of functions $f_1, \ldots, f_n$. This is not, however, always so. For example, if F is the Newton operator corresponding to a non-linear operator G, i. e.: $F(x) = x - [G'(x)]^{-1}G(x)$, usually only the operator G is given and the operator F is not known explicitly. In this particular case, when two processors are available, a more natural decomposition, as proposed by Kung in [37], is to have one process computing the value of G' while the other process uses this value for the evaluation of F. More precisely, if x and y are two global variables containing the current values of the iterate and of the reciprocal of the derivative of G, respectively, the two processes correspond to the two following programs.

Process 1: <u>while</u> (termination criterion not satisfied)

<u>do</u> $x := x - y \times G(x)$.

Process 2: <u>while</u> (termination criterion not satisfied)

<u>do</u> $y := [G'(x)]^{-1}$.

Starting with the initial values $x(0)$ and $[G'(x(0))]^{-1}$ for x and y respectively, the two processes execute their programs asynchronously and use for x and y whatever values are currently available when needed. They implicitly define the sequence of iterates $x(j)$, for $j = 0, 1, \ldots$, through formulas of the form:

$$x(j) = H[x(j-1), x(k_j)], \text{ with } k_j \leq j-1, \tag{5.1}$$

where

$$H(x,y) = x - [G'(y)]^{-1}G(x).$$

This iteration, however, is not allowed in the setting of Definition 2.1, because, in equation (5.1), $x(j)$ is defined in terms of two previous iterates. This motivates the need for a generalization of the class of asynchronous iterative methods.

5.1 - Asynchronous iterations with memory

A generalization to Definition 2.1 can be obtained by noting that, if, for $j = 2, 3, \ldots$, it happens that $k_j = j-2$ in equation (5.1), this equation defines a sequence of iterates which corresponds exactly to the sequence generated by an iterative method with one memory. This remark suggests the following generalization for the problem stated in equation (1.1).

Given an operator F from $[\mathbb{R}^n]^m$ into $\mathbb{R}^n$, the problem is now to find a vector ξ in $\mathbb{R}^n$ such that:

$$\xi = \lim_{\{x^1 \to \xi, \ldots, x^m \to \xi\}} F(x^1, \ldots, x^m) . \tag{5.2}$$

The vector ξ will still be called a *fixed point* for the operator F.

In very much the same way as we introduced the class of asynchronous iterative methods to solve equation (1.1), we now introduce the class of *asynchronous iterative methods with memory* to solve equation (5.2).

Definition 5.1:

Let F be an operator from $[\mathbb{R}^n]^m$ into $\mathbb{R}^n$. An *asynchronous iteration with memory* corresponding to the operator F and starting with a given set of vectors $x(0), \ldots, x(m-1)$ is a sequence $x(j)$, $j = 0, 1, \ldots$, of vectors of $\mathbb{R}^n$ defined for $j = m, m+1, \ldots$ by:

$$x_i(j) = \begin{cases} x_i(j-1) & \text{if } i \notin J_j \\ f_i(z^1, \ldots, z^m) & \text{if } i \in J_j , \end{cases}$$

where z^r, $1 \le r \le m$, is the vector with components $z_i^r = x_i(s_i^r(j))$, $1 \le i \le n$. As in Definition 2.1, $\mathcal{J} = \{ J_j \mid j = m, m+1, \ldots \}$ is a sequence of non-empty subsets of $\{1, \ldots, n\}$ which correspond to the subsets of components evaluated at each step of the iteration. But the sequence $\mathcal{S}$ is now to be replaced by:

$$\mathcal{S} = \{ (s_1^1(j), \ldots, s_n^1(j), s_1^2(j), \ldots, s_n^m(j)) \mid j = m, m+1, \ldots \} ,$$

a sequence of elements in $[\mathbb{N}^n]^m$. In addition, while condition (c) of Definition 2.1 remains the same, conditions (a) and (b) now become:

for each $i = 1, ..., n$

(a) $\max\{ s_i^r(j) \mid 1 \le r \le m \} \le j-1$, for $j = m, m+1, ...,$

(b) $\min\{ s_i^r(j) \mid 1 \le r \le m \}$ tends to infinity as j tends to infinity.

An asynchronous iteration with memory corresponding to F, starting with a set X of m vectors and defined with $\mathcal{J}$ and $\mathcal{A}$ will be denoted by $(F,X,\mathcal{J},\mathcal{A})$. ■

For practical reasons (e. g., stability in the implementation on a computer), we might want to have the additional condition that the vectors $x^1, ..., x^m$ are all distinct. But this restriction is not essential for our purpose here if we assume, for example, that the operator F is defined by continuity when two or more vectors are identical. This will be the case with the class of operators we will consider.

In order to obtain, for asynchronous iterations with memory, a convergence result similar to the result stated in Theorem 4.1, we need to generalize the notion of contracting operators to operators from $[\mathbb{R}^n]^m$ into $\mathbb{R}^n$.

In the remainder of the section, we will use the following notation. If $\{x^1, ..., x^m\}$ is a set of vectors in $\mathbb{R}^n$, $z = \max[x^1, ..., x^m]$ denotes the vector in $\mathbb{R}^n$ with components $z_i = \max\{ x_i^r \mid 1 \le r \le m \}$, $i = 1, ..., n$. A natural generalization to the notion of contracting operators is given in the following.

Definition 5.2:

An operator F from $[\mathbb{R}^n]^m$ into $\mathbb{R}^n$ is an *m-contracting operator* on a subset D of $\mathbb{R}^n$ if there exists a non-negative $n \times n$ matrix A with spectral radius less than unity satisfying, for all $x^1, ..., x^m, y^1, ..., y^m$ in D,

$$|F(x^1, ..., x^m) - F(y^1, ..., y^m)| \le A \max[|x^1-y^1|, ..., |x^m-y^m|] .$$

The matrix A will be called a *contracting matrix* for the operator F. ■

When $m = 1$, the preceding definition corresponds exactly to Definition 3.2, and m-contracting operators have all the properties we have already mentioned for

contracting operators. In particular, it is clear from the definition that m-contracting operators are continuous and, in fact, uniformly continuous on D^m. The uniqueness of a fixed point in D is also easily derived. In addition, if we assume that D is a closed subset of $\mathbb{R}^n$ such that $F(D^m)$ is a subset of D, then we are guaranteed the existence of a fixed point in D: the fixed point is, for example, obtained as the limit of the sequence $x(j)$, $j = 0, 1, \ldots$, defined by:

$$x(j) = F(x(j-1), \ldots, x(j-m)) , \quad j = m, m+1, \ldots ,$$

which is independent of the set of starting vectors $x(0), \ldots, x(m-1)$ in D.

We are now able to state the analogue of Theorem 4.1 for m-contracting operators in the following.

Theorem 5.1:

If F is an m-contracting operator on a closed subset D of $\mathbb{R}^n$ such that $F(D^m)$ is a subset of D, then any asynchronous iteration with memory corresponding to the operator F and starting with an arbitrary set of m vectors in D converges to the unique fixed point of F in D.

Proof:

With slight modifications, the proof of this theorem is identical to the proof of Theorem 4.1. ■

5.2 - Examples of asynchronous iterations with memory

In the beginning of this section, we considered the *Asynchronous Newton's method* to find the simple root ξ of a non-linear operator G. This method led to the sequence of iterates generated by the asynchronous iteration with memory $(H,\{x(0),x(0)\},J,s)$, where:

$$J_j = \{1, \ldots, n\} \quad \text{for} \quad j = 2, 3, \ldots ,$$

$$s_i^1(j) = j-1 , \quad s_i^2(j) = k_j \quad \text{for} \quad j = 2, 3, \ldots \text{ and } i = 1, \ldots, n .$$

In addition, as the operator H can easily be shown to be a 2-contracting operator (assuming, for example, some Lipschitz condition for the derivative of G in a small

neighborhood of the root ξ), we see that the sequence defined by equation (5.1) converges to ξ, provided that k_j tends to infinity with j (which simply states the fact that the processes eventually complete each step of their computations).

Let F be an operator from $[\mathbb{R}^n]^m$ into $\mathbb{R}^n$, and let ω be a positive scalar. Consider the operator F_ω from $[\mathbb{R}^n]^{m+1}$ into $\mathbb{R}^n$ obtained from the operator F by the introduction of the relaxation factor ω, and defined as

$$F_\omega(x^0, x^1, ..., x^m) = (1-\omega)x^0 + \omega F(x^1, ..., x^m) .$$

We first note that both F and F_ω have the same fixed points (if any). We also note that, if F is an m-contracting operator on some subset D of $\mathbb{R}^n$ with the contracting matrix A, then, for all $x^0, x^1, ..., x^m, y^0, y^1, ..., y^m$ in D, the operator F_ω satisfies:

$$\begin{aligned} |F_\omega(x^0, ..., x^m) - F_\omega(y^0, ..., y^m)| &\le |1-\omega||x^0-y^0| + \omega|F(x^1, ..., x^m) - F(y^1, ..., y^m)| \\ &\le |1-\omega||x^0-y^0| + \omega A \max[|x^1-y^1|, ..., |x^m-y^m|] \\ &\le [|1-\omega|I + \omega A]\max[|x^0-y^0|, |x^1-y^1|, ..., |x^m-y^m|] , \end{aligned}$$

and, provided that $0 < \omega < 2/[1+\rho(A)]$, F_ω is an $(m+1)$-contracting operator on D with the contracting matrix $A_\omega = |1-\omega|I + \omega A$. This reestablishes, in a more general setting, the result mentioned in Section 3.2 for asynchronous iterative methods with relaxation.

In [42], Miellou introduced a generalization of the idea of *itérations chaotiques à retards* for the problem of finding the fixed point of an operator F from $[\mathbb{R}^n]^2$ into $\mathbb{R}^n$. His generalization is a particular case of an asynchronous iteration with memory corresponding to the operator F (with $m = 2$). Miellou, in addition, gives convergence results under different assumptions on the operator F (monotony, continuity and existence of a fixed point).

Many more examples of asynchronous iterations with memory can be given and, in particular, all classical iterative method with memory can be expressed in this way. In addition, all usual super-linear iterative methods with m memories can be shown (under weak conditions) to correspond to some $(m+1)$-contracting operator, therefore ensuring the convergence of any asynchronous iterations corresponding to this operator.

6 - On the complexity of asynchronous iterations

Let F be an operator from $\mathbb{R}^n$ to itself with a fixed-point ξ and satisfying the assumptions of Theorem 4.1. We now investigate some measures of complexity for the convergence of the asynchronous iteration $(F,x(0),\mathcal{J},\mathcal{S})$ toward the fixed-point ξ of F.

We will first derive, in Section 6.1, results applicable to asynchronous iterations in general, then, in Section 6.2, using condition (b') in Definition 2.1, we will derive more specific results for the particular case of chaotic iterations.

The constructive proof of the theorem already provides us with bounds for the error vector $x(j) - \xi$. And, in fact, if F is a contracting operator with the contracting matrix A, we note that an estimate of the error committed with the asynchronous iteration $(F,x(0),\mathcal{J},\mathcal{S})$ is directly obtainable from the asynchronous iteration $(A,|x(0)-\xi|,\mathcal{J},\mathcal{S})$. This estimate is used in this section to derive bounds for the complexity of asynchronous iterations corresponding to contracting operators. However, since $(A,|x(0)-\xi|,\mathcal{J},\mathcal{S})$ can only reflect linear convergence, this estimate is certainly not adequate to deal with all asynchronous iterations, and, in Section 8, using an example, we present an analysis for an asynchronous iteration with super-linear convergence.

For convenience, we only consider the convergence in norm of the error vector $x(j) - \xi$. By choosing, for example, the norm $\|x\| = \max\{ |x_i| \mid i = 1, ..., n \}$, this corresponds to the worst possible case for the convergence of the components.

To measure the linear convergence of the sequence $x(j)$, $j = 0, 1, ...$, toward its limit ξ, we consider the following complexity measures often referred to in the literature. The rate of convergence of the sequence is defined as:

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

In addition, if c_j is the cost associated with the evaluations of the first j iterates, $x(1), ..., x(j)$, we define the complexity of the sequence by:

$$E = \liminf_{j\to\infty} [(-\log\|x(j)-\xi\|)/c_j] .$$

If all logarithms are taken to the base 10, $1/\hat{\mathcal{R}}$ measures the asymptotic number of steps required to divide the error by a factor of 10, whereas $1/E$ measures the corresponding cost. We note that, if c_j/j tends to some finite limit $\mathcal{C}$ (which corresponds to the average cost per step), then the complexity is simply given by $E = \hat{\mathcal{R}}/\mathcal{C}$.

The costs c_j, $j = 1, 2, \ldots$, can be chosen according to any convenient measure. In our case, we consider the cost to correspond either to the number of evaluations of the operator F, or to the time to perform the evaluations. In the former case, if each component is equally as hard to compute, the cost can be directly evaluated from the sequence $\mathcal{J}$ by considering

$$c_j = (|J_1| + \ldots + |J_j|)/n \,, \tag{6.1}$$

where $|J_j|$ is the cardinality of the set J_j, i. e., the number of components evaluated at the j-th step of the iteration. In the latter case, the cost is better suited to deal with parallel algorithms, and can be evaluated through the classical tools of queueing theory. When it is necessary to indicate which cost measure is used in the evaluation of the complexity, we use the notations E_e if the cost is measured in number of evaluations of F, and E_t if the cost is measured by the time needed to perform (sequentially) one evaluation of F.

6.1 - General bounds: asynchronous iterations

We return to the proof of Theorem 4.1, and we use the same notations. The proof simply consists of constructing an increasing sequence of indices j_p, $p = 0, 1, \ldots$, satisfying

$$\|x(j) - \xi\| \leq \alpha\omega^p \quad \text{for} \quad j \geq j_p \,,$$

where the positive constant α can be taken to be $\alpha = \|x(0)-\xi\|$. From the construction of this sequence we note that

$$j_{p+1} = j_p + r_p + t_p \quad \text{for} \quad p = 0, 1, \ldots \,,$$

where r_p and t_p are integers chosen to satisfy: (1) starting with the index j_p+r_p, all evaluations of iterates do not make any more use of values of components corresponding to iterates with indices smaller than j_p; and (2) all components are evaluated at least once between the (j_p+r_p)-th and the $(j_p+r_p+t_p)$-th iterates.

Now let

$$p_j = \sup\{\ p \mid r_0 + t_0 + \ldots + r_{p-1} + t_{p-1} \le j\ \} \quad \text{for} \quad j = 0, 1, \ldots . \tag{6.2}$$

Then, if we know r_p and t_p for $p = 0, 1, \ldots$, we can deduce a bound on $\|x(j)-\xi\|$ since

$$\|x(j)-\xi\| \le \alpha\omega^{p_j} \quad \text{for} \quad j = 0, 1, \ldots ,$$

which shows that the sequence $x(j)$, $j = 0, 1, \ldots$, converges at least as fast as the sequence ω^{p_j}, $j = 0, 1, \ldots$, with a rate of convergence $\mathcal{R}$ such that

$$\mathcal{R} \ge - [\liminf_{j\to\infty} (p_j/j)] \log\omega .$$

And, if c_j is the cost associated with the evaluations of the first j iterates, we have the following bound for the complexity:

$$E \ge - [\liminf_{j\to\infty} (p_j/c_j)] \log\omega .$$

In addition, as was noticed earlier, if A is a contracting matrix for the operator F, ω can be chosen arbitrarily close to $\rho(A)$. This shows that in the bounds we have just obtained we can simply replace ω by $\rho(A)$, and this yields the following.

Theorem 6.1:

Let F satisfy the condition of Theorem 4.1, and let A be a contracting matrix for the operator F. Then the asynchronous iteration $(F,x(0),\mathcal{J},\mathcal{S})$ converges to the fixed point of F with a rate of convergence

$$\mathcal{R} \ge - [\liminf_{j\to\infty} (p_j/j)] \log\rho(A) ,$$

and a complexity

$$E \ge - [\liminf_{j\to\infty} (p_j/c_j)] \log\rho(A) ,$$

where the sequence p_j is defined from $\mathcal{J}$ and $\mathcal{S}$ by equation (6.2).

An example

As an illustration, we consider the parallel implementation of Jacobi's method with k processes. For simplicity, we assume that n is a multiple of k, and we set $q = n/k$.

To avoid an overhead in the selection of the components to be updated at each step of the iteration, each process is assigned to the evaluation of a fixed subset of the components. In particular, when all components are equally as hard to compute, and when

all processors are equally as fast, it is natural to decompose the set of components into subsets of equal sizes, and, for example, to assign the first process to the evaluation of the first q components, the second process to the evaluation of the next q components, and so forth. Corresponding to this decomposition, a parallel implementation of Jacobi's method with k processes can be represented by the asynchronous iteration $(F,x(0),\mathcal{J},\mathcal{S})$, where $\mathcal{J}$ and $\mathcal{S}$ are defined by:

$$J_j = \{ i \mid 1 + (j-1 \bmod k)q \le i \le q + (j-1 \bmod k)q \} \text{ for } j = 1, 2, \ldots ,$$

$$s_i(j) = \lfloor (j-1)/k \rfloor q \text{ for } j = 1, 2, \ldots \text{ and } i = 1, \ldots, n .$$

The two asynchronous iterations we introduced in Section 2.2 to represent Jacobi's method correspond to the particular cases $k = 1$ and $k = n$.

It is easy to check that r_p and t_p are given by 1 and k, respectively, for $p = 0, 1, \ldots$. This shows that $p_j = \lfloor j/k \rfloor$ and therefore

$$\mathcal{R}(k) \ge -(\log \rho(A))/k .$$

Now, if c_j measures the number of evaluations of F required to compute the first j iterates, using equation (6.1), we have $c_j = j/k$. This gives for the complexity:

$$E_e(k) \ge -\log \rho(A) . \tag{6.3}$$

For all values of k, we obtain the same bound for the complexity. In particular, when F is the linear operator defined by $F(x) = Ax + b$, where A is a non-negative $n \times n$ matrix with spectral radius less than unity, then A can be chosen as a contracting matrix for F and the bound (6.3) is known to be sharp.

Since the asynchronous iteration we are considering corresponds to a parallel implementation of Jacobi's method, instead of measuring the cost by the number of evaluations of F, it is more natural to use the average time to perform the evaluations as a measure of the cost. Let the time unit be the average time to perform (sequentially) one evaluation of F. Then, if $pk \le j \le (p+1)k$, we have $c_{pk} \le c_j \le c_{(p+1)k}$ and $c_{pk} = p[\lambda_k/k]$. The expression λ_k/k corresponds to the time for the k processes to execute in parallel their computations and to synchronize their executions. The factor λ_k is the *penalty factor*

introduced by Kung in [37]; it measures the overhead due to the fluctuations in the computing times of the k processes, and can be evaluated if we know, for example, the distribution function for the time to evaluate F. In particular, we have $\lambda_1 = 1$ and, for $k \geq 2$, $\lambda_k \geq 1$ with the equality only when it always take the same constant time to evaluate F (i. e., there are no fluctuations in the computing time). This cost measure yields the following bound for the complexity:

$$E_t(k) \geq -[k/\lambda_k]\log_{\rho(A)} .$$

Again, these bounds are sharp for the linear operator we mentioned above, and the ratio $E_t(k)/E_t(1) = k/\lambda_k$ measures the speed-up achieved by using a parallel implementation with k processes. We would expect the implementation with k processes to be k times as efficient as the sequential implementation (with $k = 1$), but this is not so because of the overhead introduced by synchronizing the k processes and measured by the penalty factor λ_k.

6.2 - Additional assumptions: chaotic iterations

In the preceding example, we have been able to carry out the analysis for Jacobi's method (and even obtain sharp bounds on the complexity) because the representation in terms of asynchronous iterations is known explicitly and follows a very regular pattern. This is not, however, generally so. For example, in a parallel implementation with several processes using no synchronization (as presented in Section 2.1), the sequences $\mathcal{A}$ and $\mathcal{J}$ (and, therefore, the sequences r_p and t_p, $p = 0, 1, \ldots$) are not known directly but are only defined implicitly by the processes in the course of their executions.

Below, we present alternate bounds for $\mathcal{R}$ and E under conditions often satisfied in usual implementations of asynchronous iterations. We assume that we know bounds on r_p and t_p, and we restrict the definition of the class of asynchronous iterative methods by replacing conditions (b) and (c) of Definition 2.1 with the following:

(b') There exists a positive integer r such that, for $j = 1, 2, \ldots$ and $i = 1, \ldots, n$,

$$s_i(j) \geq j-r,$$

(c') there exists a non-negative integer t such that, for $j = 1, 2, \ldots$,

$$J_j \cup \ldots \cup J_{j+t} = \{1, \ldots, n\}.$$

As was already mentioned, condition (b') was proposed by Chazan and Miranker in the definition of the chaotic relaxation scheme [11]. Although the convergence result obtained under condition (b) of Definition 2.1 is mathematically more satisfactory, condition (b') is very often satisfied in practical applications, in particular, when the computations of all components have the same complexity (which is the case with a linear operator). Condition (c') is also satisfied for most of the usual implementations of asynchronous iterations, since it is natural that (1) a process evaluates a component by using the most recently updated values of all components; and (2) two processes never evaluate the same component at the same time; in this case it follows directly that, by taking $r = t+1$, conditions (b') and (c') are equivalent.

Under the additional conditions (b') and (c'), we clearly have $r_p \leq r$ and $t_p \leq t$, for $p = 0, 1, \ldots$, and, therefore, $p_j \leq \lfloor j/(r+t) \rfloor$. From the bounds stated in Theorem 6.1, we immediately obtain the following.

Corollary:

Let F satisfy the condition of Theorem 4.1, and let A be a contracting matrix for F. If the asynchronous iteration $(F, x(0), J, A)$ satisfies the additional conditions (b') and (c'), then it converges to the fixed point of F with a rate of convergence

$$R \geq - [1/(r+t)] \log \rho(A) ,$$

and a complexity

$$E \geq - [\lim_{j \to \infty} j/(r+t)c_j] \log \rho(A) .$$

7 - Experimental results

The results of this section are reported in detail in Chapter V. A very brief presentation is given below as an immediate illustration of asynchronous iterative methods.

Several asynchronous iterations have been experimented with on C.mmp, the Carnegie-Mellon multiprocessor [63], they are described in Section 7.1, and the actual measurements are presented in Section 7.2. Although asynchronous iterative methods are applicable to non-linear problems, the experiments reported here deal only with linear problems. More specific treatments for non-linear problems will be reported elsewhere.

7.1 - Experiments with asynchronous iterations

All asynchronous iterations we have experimented with consist of the parallel execution of k processes. As we did with the parallel implementation of Jacobi's method, we assign to each of the processes the evaluation of a fixed subset of the components. Each process computes cyclically new values for the components in its subset, and the methods only differ by the choices of the values used in the evaluations.

Asynchronous Jacobi's method (AJ): For the evaluations of all components, a process uses only values of the components known at the beginning of a *cycle*, and the process releases all new values at the end of each cycle.

Asynchronous Gauss-Seidel's method (AGS): Same as the AJ method except that the process uses new values of the components in its subset as soon as they are known for further evaluations in the same cycle. Again, it releases the new values (for the other processes) at the end of its cycle.

Purely Asynchronous method (PA): A process computes the new values of each component by using the most recent values of all components and releases each new value immediately after its evaluation.

The PA method is certainly the easiest method to implement, and, as far as space is concerned, is clearly the most efficient one, whereas the AJ method is the worst one, since it requires from each process not only a complete duplication of all components (as of the beginning of its cycle) but still another copy of the components in its own subset. This can hardly be justified but experimental results give useful comparisons between the AJ

method and the actual Jacobi's method (also between the AGS and Gauss-Seidel's methods).

In addition, both the AJ and AGS methods also require the need for a critical section in order to read all components at the beginning of a cycle and to update the values at the end of a cycle, whereas no critical section is needed with the PA method. However, C.mmp has the drawback that no indivisible instructions exist to read or write floating point numbers (implemented on two consecutive words of memory), therefore, if we are to implement the PA method on C.mmp, only the first 8 bits of the mantissa can be considered significant, and the admissible error in the termination criterion has to be chosen accordingly.

7.2 - Results

The three methods just described, as well as Jacobi's method, have been implemented on C.mmp to solve the Dirichlet problem for Laplace's equation on a rectangular domain of $\mathbb{R}^2$. Using the method of finite differences, an approximate solution to this problem can be found by solving a linear system of equations. In the experiments reported here, a regular grid has been chosen with 21×24 interior points, resulting in a linear system of size $n = 504$. This system can be represented in the form $x = F(x) = Ax + b$, where the vector b is obtained from the boundary conditions, and the matrix A is a (very sparse) non-negative matrix with spectral radius $\rho(A) = 0.991$. Since $\rho(|A|) = \rho(A) < 1$, this shows that A is a contracting matrix for the operator F, and, therefore, that the result of Theorem 4.1 can be applied to F to ensure the convergence of each iterative method.

At the time the measurements have been taken, the configuration of C.mmp included six processors, and all iterative methods have been run with a number of processes $k = 1, 2, 3, 4$, and 6. Each of the results reported here is the average of three measurements, but, since C.mmp was used in stand-alone mode during the experiments, very little difference was noted from one run to the next.

In Table 7.1, we report for the four methods the average number of vector evaluations required to reduce (asymptotically) the error vector by a factor of *10*: this corresponds to the cost measure $1/E_e$. And, in Table 7.2, we report the average time (expressed in seconds) required to achieve this reduction: this corresponds to the cost mesure $1/E_t$.

The bounds obtained from the results of the previous sections are mentioned in parentheses along with the measurements. The parameters in these bounds have been evaluated either directly (e. g., $\rho(A) = 0.991$), or through measurements by tracing the executions of the processes. In particular, for the AJ, AGS and PA methods, the bounds r and t, defined in Section 6.2, have been determined by observing the sequencing of the tasks performed by the different processes. Similarly, the penalty factor in Jacobi's method and the overhead due to the critical section in the AJ and AGS methods have been obtained by direct measurements: they are presented in Tables 7.3 and 7.4.

	Jacobi	AJ	AGS	PA
$k = 1$	254 (254)	254 (254)	127 (254)	127 (254)
$k = 2$	254 (254)	266 (888)	142 (888)	127 (762)
$k = 3$	254 (254)	267 (846)	149 (846)	127 (762)
$k = 4$	254 (254)	273 (825)	166 (825)	129 (762)
$k = 6$	254 (254)	285 (804)	196 (804)	128 (762)

Table 7.1 - Number of evaluations required to divide the error by a factor of *10*

	Jacobi	AJ	AGS	PA
$k = 1$	337 (337)	337 (337)	168 (337)	168 (337)
$k = 2$	241 (241)	211 (705)	113 (705)	84 (506)
$k = 3$	178 (178)	149 (471)	83 (471)	56 (337)
$k = 4$	153 (153)	123 (372)	75 (372)	43 (253)
$k = 6$	131 (131)	102 (289)	70 (289)	28 (169)

Table 7.2 - Time required to divide the error by a factor of *10*

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 6$
λ_k	1	1.43	1.59	1.82	2.34
%	0	29.9	37.1	45.1	57.3

Table 7.3 - Penalty factor with Jacobi's method and percentage of time wasted

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 6$
λ_k	1	1.20	1.26	1.35	1.62
%	0	16.6	20.8	26.0	38.2

Table 7.4 - Critical section overhead cost with the AJ and AGS methods and percentage of time wasted

These results must only be considered to illustrate the behavior of asynchronous iterations, since, in particular, the two cost measures reported in Tables 7.1 and 7.2 strongly depend on both the problem (i. e., the matrix A) and the multiprocessor system. Yet, they show a clear advantage of asynchronous methods over synchronized methods.

We note, for example, from Table 7.3 that, with Jacobi's method, when $k = 6$ processes are used, the penalty factor is as big as $\lambda_6 = 2.34$. This means that about 57 percent of the time is spent by a process waiting for the other processes to finish their computations. This limits the possible speed-up to 2.6 rather than 6.

We also note that the use of critical sections, too, should be avoided, since, with the AJ or AGS methods, when 6 processes are used, about 38 percent of the time is spent waiting for entering the critical section, again limiting the possible speed-up to 3.7 rather than 6.

The measurements for the PA method, on the other hand, indicate that we achieve an almost full speed-up with this method (at least with a small number of processes). An obvious reason for this speed-up is the total absence of any form of synchronization;

another reason, specific to the problem we have experimented with and indicated by the results of Table 7.1, is due to the sparsity of the matrix A.

The bounds derived in Section 6 have been obtained in a very general case. Yet Tables 7.1 and 7.2 show that they are always within a factor between 3 and 6 of the actual measurements (except for Jacobi's method where they are sharp). In addition, we certainly could obtain much sharper bounds by carrying out the analysis for the specific problem we have experimented with (for example, by taking into account the sparsity of the matrix). In particular, a specific analysis for the PA method can easily explain the fact that $1/E_e$ is almost independent of the number of processes (see Table 7.1).

8 - Asynchronous iterations with super-linear convergence

As we already noticed, the bounds established in Section 6 are certainly not adequate to measure the complexity of iterations with super-linear convergence. In this section, we use as an example the iterative method we have mentioned at the beginning of Section 5 to show how an analysis of the complexity can be done for this case.

To study the convergence of a sequence $x(j)$, $j = 0, 1, \ldots$, toward its limit ξ, we now use the following usual measures of complexity. The order of convergence is defined as

$$\rho = \liminf_{j\to\infty} [(-\log\|x(j)-\xi\|)^{1/j}] ,$$

and, as before, if c_j is the cost associated with the evaluations of the first j iterates, $x(1), \ldots, x(j)$, we define the complexity of the sequence by:

$$E = \liminf_{j\to\infty} [(\log-\log\|x(j)-\xi\|)/c_j] ,$$

Again, we note that, if the average cost per step c_j/j tends to some finite limit τ when j tends to infinity, the complexity is simply given by $E = (\log\rho)/\tau$. In the remainder of the section, we assume that the limit τ exists.

In order to find the simple root ξ of an operator G from $\mathbb{R}^n$ into itself, we use the *Asynchronous Newton's method*, AN, as implemented by the two processes described at the

beginning of Section 5. Let r_i, $i = 1, 2, \ldots$, be the number of iterates evaluated by the first process, P_1, during the i-th evaluation of the derivative G' by the second process, P_2. Let $j_0 = 0$ and $j_i = r_1 + \ldots + r_i$, for $i = 1, 2, \ldots$, then $x(j_i)$, $i = 0, 1, \ldots$, is the iterate used by P_2 for the $(i+1)$-st evaluation of the derivative. Starting with the two initial values $x(0)$ and $G'(x(0))$, the AN method generates with the two processes P_1 and P_2 the sequence of iterates $x(j)$, $j = 1, 2, \ldots$, defined by

$$x(j+1) = x(j) - [G'(x(j_{i-1}))]^{-1}G(x(j)) \text{ , for } i = 1, 2, \ldots \text{ and } j_i < j \le j_{i+1} . \quad (8.1)$$

The following theorem gives the measures of complexity for this sequence if we know some bounds on the sequence r_i, $i = 1, 2, \ldots$.

Theorem 8.1:

Let the initial approximation $x(0)$ be close enough to the root ξ, that is

$$x(0) \in D_\epsilon = \{ x \mid \|x-\xi\| < \epsilon \} ,$$

and let the derivative G' satisfy some Lipschitz condition on D_ϵ:

$$\|G'(x)-G'(y)\| \le M\|x-y\| , \quad \forall\, x, y \in D_\epsilon .$$

If ϵ satisfies the condition

$$M\|G'(\xi)^{-1}\|\epsilon < 2/5 ,$$

and if there exist some positive integers p and q such that

$$p \le r_i \le q , \text{ for } i = 1, 2, \ldots ,$$

then the order of convergence, ρ, and the complexity, E, of the sequence defined by equation (8.1) satisfy:

$$\rho \ge \lambda_p^{1/q} , \quad (8.2)$$

and

$$E \ge (\log\lambda_p)/(q\alpha) , \quad (8.3)$$

where λ_p is the largest root of the equation $z^3 - z^2 - (p-1)z - 1 = 0$ (for which we can check easily that $0.4 + \sqrt{p} < \lambda_p < 0.5 + \sqrt{p}$, $p = 1, 2, \ldots$).

Proof:

The proof is easy but technical, and below we only give an outline for this proof.

Let $\alpha = M\|G'(\xi)^{-1}\|$, and let $c = 3\alpha/[2(1-\alpha\epsilon)]$. From the choice of ϵ, we first note that, starting with $x(0) \in D_\epsilon$, the sequence $\|x(j)-\xi\|$, $j = 0, 1, \ldots$, is strictly decreasing and satisfies:

$$\|x(j_i+1)-\xi\| \le c\|x(j_{i-2})-\xi\|\|x(j_i)-\xi\| \text{ , for } i = 2, 3, \ldots ,$$

and

$$\|x(j+1)-\xi\| \le c\|x(j_{i-1})-\xi\|\|x(j)-\xi\| \text{ , for } i = 2, 3, \ldots \text{ and } j_i < j < j_{i+1} = j_i + r_i .$$

By substitution, it follows that, for $i = 2, 3, \ldots$,

$$\|x(j_{i+1})-\xi\| \le c^{r_i}\|x(j_{i-1})-\xi\|^{r_i-1}\|x(j_{i-2})-\xi\|\|x(j_i)-\xi\| ,$$

and, if we set $u_i = -\log c\|x(j_i)-\xi\|$, we obtain:

$$u_{i+1} \ge u_i + (r_i-1)u_{i-1} + u_{i-2} \text{ , for } i = 2, 3, \ldots .$$

Therefore, by using the lower bound on r_i, we deduce that

$$u_{i+1} \ge u_i + (p-1)u_{i-1} + u_{i-2} \text{ , for } i = 2, 3, \ldots .$$

This shows that u_i tends to infinity at least as fast as $\lambda_p{}^i$. Therefore, the order of convergence, ρ', of the subsequence $x(j_i)$, $i = 0, 1, \ldots$, must verify $\rho' \ge \lambda_p$. The bounds (8.2) and (8.3) are derived directly from this last inequality. ■

In particular, if the cost c_j measures the number of evaluations of the operator G, we simply have $c_j = j$, and, therefore, $E_e \ge (\log\lambda_p)/q$. On the other hand, if the cost corresponds to the execution time, the complexity will depend on the implementation itself. For example, an implementation corresponding strictly to the generation of the sequence described by equation (8.1) requires the use of a critical section for reading and writing, in a block, the values of the iterates and of the derivative. The use of a critical section introduces an overhead, but, as is done with the PA method, the overhead can be avoided if a process uses whatever values are currently available when needed. In this case the bounds of Theorem 8.1 still holds, and τ can be given the value $\tau = 1$.

The parameters p and q, too, depend on the particular implementation of the AN method, and, especially, on the relative speeds of the processors executing the processes P_1 and P_2. In practice, if the processors are equally as fast, we expect, with small

variations, r_i to be close to n, and the values $p = q = n$ can predict good estimates for the complexity of the AN method implemented with two processes.

The AN method is easily generalizable to more than two processes. If k processes are available, k_1 might be assigned to the evaluation of the sequence of iterates, while $k_2 = k - k_1$ are assigned to the evaluation of the derivative. The bounds of Theorem 8.1 still holds for this case as well, only with different values for the sequence r_i, $i = 1, 2, \ldots$ (or for the bounds p and q), determined by the parallel implementations of the two evaluations. Further results in this direction will be reported elsewhere.

9 - Extensions of the results

We mention below some direct extensions of the results presented in this chapter and some points subject to further development.

A straighforward generalization of the results can be obtained if, instead of $\mathbb{R}^n$, we consider the product P of n Banach spaces B_i with norms $|.|_i$, $i = 1, \ldots, n$. In this case, if x is an element of P, x is determined by its components $x_i \in B_i$, $i = 1, \ldots, n$. And $|x|$ represents the non-negative vector of $\mathbb{R}^n$ with components $|x_i|_i$, $i = 1, \ldots, n$.

Considering only the class of linear operators, $F(x) = Ax + b$, we have noted that the notion of contracting operators coincides with the condition that $\rho(|A|) < 1$. In [11], Chazan and Miranker have shown that this condition is not only sufficient but also necessary for the convergence of all chaotic iterations. This implies, in particular, that all asynchronous iterations corresponding to a linear operator F are convergent if and only if F is a contracting operator. The necessity of this condition, however, seems to be inherent to the linear nature of the problem, and when we also consider non-linear operators the proof given by Chazan and Miranker does not apply any more. It would be of interest to obtain conditions on the class of operators for which all asynchronous iterations are guaranteed to converge. Similar conditions for the convergence of a more

restricted class of iterations would also be of interest, in particular, for the subclass of asynchronous iterative methods corresponding to the additional assumptions introduced in Section 6.2.

The bounds we have obtained to estimate the rate of convergence of asynchronous iterations have been derived by considering the worst possible case, and, compared to actual measurements, these bounds are very conservative. It would certainly be very useful to obtain bounds (or estimates) corresponding to the average behavior of asynchronous iterations, for example, given the probability distributions of the two sequences $\mathcal{J}$ and $\mathcal{S}$, or, more generally, given the distribution functions for the time it takes the different processes to evaluate the components.

We have already mentioned the possibility of introducing a relaxation factor in asynchronous iterations, and, for contracting operators, we have derived a possible range that guarantees the convergence of all asynchronous iterations. Nothing is known, however, about the *optimal* choice of the relaxation factor, for example, given directly the asynchronous iteration through $\mathcal{J}$ and $\mathcal{S}$, or, again, given the distribution functions for the evaluation times.

10 - Concluding remarks

In the implementation of most parallel algorithms, synchronization seems to be required to assure the communication between the processes, and to guarantee their correct executions. However, the main drawback with synchronization is that it degrades considerably the performance of the algorithms because it is very time consuming. The class of *asynchronous iterative methods* avoids this drawback. It includes iterations corresponding to a parallel implementation in which the cooperating processes have a minimum of intercommunication and do not make any use of synchronization. The *Purely Asynchronous method* described in Section 7.1 is a typical example of an asynchronous iterative method. Asynchronous iterations follow the same goal as chaotic relaxations [11]: to eliminate the need for synchronization in a parallel computation.

Asynchronous iterations generalize to *asynchronous iterations with memory* which allow different values of the same variable to be used within the same computation. Using the notions of *contracting operators* and of *m-contracting operators*, Theorems 4.1 and 5.1 state sufficient conditions to guarantee the convergence of any asynchronous iterations and asynchronous iterations with memory. These conditions are satisfied for a large class of operators.

In the second part of the chapter, asynchronous iterations are evaluated from a computational point of view, then the results of a series of actual measurements (obtained by running asynchronous iterations on a multiprocessor) are presented. These results fully justify the use of asynchronous iterative methods.

General bounds on the complexity of asynchronous iterations are first derived directly from the proof of the convergence theorem. Although these bounds are sharp for a parallel implementation of Jacobi's method, they are of little applicability since they require to know *a priori* the exact specification of each step of the iteration. Alternate bounds are then derived under additional conditions which are usually satisfied in practical applications. These bounds are consistent with actual measurements; for the experiments we have run, they are always within a factor of 6 of the measurements. In addition, it is our feeling that these bounds can be largely improved if we take into account specific characteristics of the problem being solved, therefore leading to a better understanding of asynchronous iterations. In Section 8, for example, we have made a first step in this direction, and we have presented an analysis for the *Asynchronous Newton's method*.

A series of experiments has been conducted on C.mmp, a multiprocessor system (with 6 processors at the time the experiments have been run), and several asynchronous iterative methods have been implemented to solve a large linear system of equations. They range from Jacobi's method, requiring a full synchronization of all the processes at each step of the iteration, to the PA method, which requires no synchronization at all. In

between, the A.J and AGS methods are derived from the usual Jacobi's and Gauss-Seidel's methods, and they require the use of a critical section.

The experimental results show a considerable advantage for the iterative method with no synchronization at all. For a number of processes up to the number of processors available on C.mmp, the PA method exhibits full parallelism and has an optimal speed-up compared to Gauss-Seidel's method, the best sequential method experimented with. The A.J and AGS methods have a very similar behavior, and when 6 processes are used the overhead caused by the critical section implies that *38* percent of the time a process is waiting for entering the critical section. As is intuitively expected, Jacobi's method has the worst behavior of all the methods considered, and, with 6 processes, the overhead, due to the synchronization of all the processes at each step of the iteration, is about *57* percent (i. e., more than half the time a process is waiting for the other processes to finish their computations).

On the basis of these experimental results, and for the problem we have considered, there does not seem to be any alternatives: the PA method is obviously the most efficient one. In addition, another advantage of the PA method is that it is the easiest one to implement, and, spacewise, it is also the most efficient one.

Finally, another possibility, which has only been outlined in this chapter, is the introduction of a relaxation factor. Based only on a few experimental results (not reported here), it is our belief that we can expect an improvement of the *Purely Asynchronous Over-Relaxation method* over the PA method similar to the improvement of the SOR method over the Gauss-Seidel's method, if we choose the relaxation factor in an optimal way. The optimal choice of the relaxation factor depends not only on the system being solved, but also on the probability distributions of the various execution times by the different processes.

Chapter IV

On the Alpha-Beta Pruning Algorithm

Part 1: The sequential algorithm

1 - Introduction

Most so-called intelligent programs use some form of tree searching; among them, most game playing programs are built around an efficient tree searching algorithm known as the *alpha-beta pruning algorithm*. In the first part of this chapter, we investigate the efficiency of this algorithm with respect to a cost measure first introduced by Knuth and Moore in [35] and given in Definition 1.1 below. The second part of the chapter is devoted to the study of a parallel implementation of the algorithm on an asynchronous multiprocessor.

Definition 1.1:

Let $N_{n,d}$ be the number of terminal positions examined by some algorithm A in searching a uniform tree of degree n and depth d. The quantity

$$\mathcal{R}_A(n) = \lim_{d\to\infty} (N_{n,d})^{1/d}$$

is called the *branching factor* corresponding to the search algorithm A. ■

Analyses of the α-β pruning algorithm have been attempted in two recent papers by Fuller, Gaschnig and Gillogly [23] and by Knuth and Moore [35]. Both papers address the problem of searching a uniform game tree of degree n and depth d with the α-β pruning algorithm under the assumptions that the n^d static values assigned to the terminal nodes are independent identically distributed random variables and that they are *all distinct*. We

immediately observe that, in order to evaluate the branching factor, the last assumption requires that the n^d distinct values assigned to the terminal positions be taken from an infinite range. For most practical applications this is, however, unrealistic.

Fuller, Gaschnig and Gillogly developed in [23] a general formula for the average number of terminal positions examined by the α-β procedure. Their formula, however, is computationally intractable and leads to undesirable rounding errors for large trees (i. e., for large n and d) since it involves, in particular, a $2d-2$ nested summation of terms with alternating signs and requires on the order of n^d steps for its evaluation. Then they gave some empirical results based on a series of simulations, and compared the results with actual measurements obtained by running a modified version of the Technology Chess Program [24], [25].

In [35], Knuth and Moore have analyzed, under the same conditions, a simpler version of the full α-β pruning algorithm by not considering the possibility of deep cut-offs; they have shown, in particular, that the branching factor of the resulting algorithm is $\Theta(n/\ln n)$. Knuth and Moore also considered other assumptions to account for dependencies among the static values assigned to the terminal positions and developed analytic results under those assumptions. Their paper gives, in addition, an excellent presentation and historical account of the α-β pruning algorithm.

Departing from the assumptions of the two papers we just mentioned, we first consider the effect of possible equalities between the values assigned to the terminal nodes of a uniform tree, assuming that these values are independent identically distributed random variables drawn from any *discrete* probability distribution. In Section 2, we establish some notations and preliminary results, and in Section 3, we derive a general formula for the number of terminal nodes examined by the α-β pruning algorithm when we take into account both shallow and deep cut-offs. The evaluation of this formula requires only a finite summation over the range of possible values assigned to the terminal nodes and is relatively easy. We show, in particular, that, when the terminal nodes can only take

on two distinct values, the branching factor of the α-β pruning algorithm can grow with n as $O(n/\ln n)$ for some choice of the probability distribution. In Section 4, we show that, when the discrete probability distribution tends to a continuous probability distribution, the summation derived in Section 3 can be replaced by an integral, which constitutes the worst case over all discrete probability distributions. In Section 5, an analysis of this integral shows that the branching factor of the α-β pruning algorithm for a uniform tree of degree n grows with n as $\Theta(n/\ln n)$, therefore confirming a claim by Knuth and Moore [35] that deep cut-offs have only a second order effect on the average behavior of the α-β pruning algorithm. In Section 6, we propose a parallel implementation of the α-β pruning algorithm in which several processes search for the solution (i. e., the value associated with the game tree) within different subintervals. This parallel implementation is analyzed in Section 7; the parallel implementation with 2 processes, in particular, turns out to be more than twice as efficient as the original α-β pruning algorithm, which is consequently shown not to be optimal. Some concluding remarks and open problems are given in the last section.

2 - Presentation and initial properties of the α-β pruning algorithm

There are two usual approaches for dealing with searching a game tree. In [23], Fuller, Gaschnig and Gillogly adopted the *Min-Max* approach, while, in [35], Knuth and Moore chose the *Nega-Max* approach. We will briefly present, in Section 2.1, the two approaches and introduce the α-β procedure in terms of the Nega-Max model. Then, in Section 2.2, we will reestablish an initial result of [23] which was stated in terms of the Min-Max approach.

2.1 - The α-β procedure

Let us consider a game (like chess, checkers, tic-tac-toe or kalah) played by two players who take turns. It is common to represent the evolution of the game by means of

a *game tree*, where each position of the game is represented by a node. If the position is a dead-end, the node is terminal, otherwise all possible moves from that position are represented as the successors of the node. The structure of the tree is preserved by not generating moves leading to some positions already generated (thus, avoiding cycles); this is the function of the *move generator*. The *evaluation function* is another important function in game playing programs; it assigns to each terminal position a *static value* by estimating various parameters such as piece counts, occupation of the board, etc. The evaluation function evaluates the terminal nodes from one player's viewpoint, giving higher values to positions more favorable to this player. It is convenient at this point to name the two players Max and Min. Hence, Max's strategy is to lead the game towards positions with higher values, while Min's strategy is to lead the game towards positions with lower values.

The *minimax procedure* is directly based on this formulation and can be used by either Max or Min to decide on his next move from a given position, assuming that his opponent will respond with his best move. Using a rather brute force approach, the minimax procedure assigns values to all nodes of a game tree. It first assigns to terminal nodes the results of the evaluation function, then it backs-up to internal nodes corresponding to a position from which it is Max's (Min's) turn to play the maximum (minimum) of the values assigned to its successors.

Suppose it is Max's turn to play from an initial position (corresponding to the root of the game tree), then it is his turn to play from any positions at even depth and Min's turn to play from any positions at odd depth. Therefore, the minimax procedure will back-up values to the nodes of the game tree through a succession of Minimazing/Maximazing operations. This corresponds to the *Min-Max* approach.

By observing that:

$$\max\{ \min\{ x_1, x_2, \dots \}, \min\{ y_1, y_2, \dots \}, \dots \} =$$
$$\max\{ -\max\{ -x_1, -x_2, \dots \}, -\max\{ -y_1, -y_2, \dots \}, \dots \},$$

the Min-Max approach can be directly reformulated into the *Nega-Max* approach. In the Nega-Max formulation, a terminal node of a game tree should be assigned the result of the evaluation function only if it is at an even depth (assuming it is initially Max's turn to play) and it should be assigned the opposite of the result of the evaluation function if it is at an odd depth. The Nega-Max approach requires the same operator at all levels of a game tree, and the uniformity of the notation will make it easier to carry out an analysis. This approach will be used throughout.

Figure 2.1 shows the effect of the minimax procedure in a uniform tree of degree *2* and depth *4*. The values assigned to the terminal nodes have been chosen arbitrarily. The path indicated by a darker line shows the sequence of moves selected by the procedure.

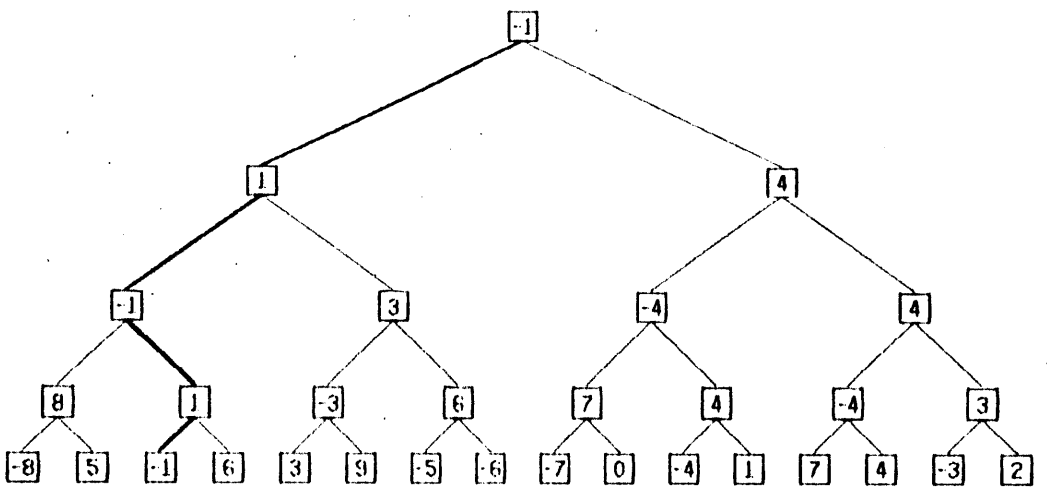


Figure 2.1 - Searching a game tree with the minimax procedure

The minimax procedure is clearly a brute force search and, when exploring a node, it uses none of the information already available from the nodes previously explored. Obviously, by taking advantage of the information previously acquired we can easily improve on the brute force search. Figure 2.2 presents some simple patterns in which the distribution of the information could lead to such improvements. In the figure, the circled nodes have already been explored, and they are labeled with their backed-up values; the values of the other nodes are yet to be determined. We are interested in the value v of the top level node in both patterns (a) and (b).

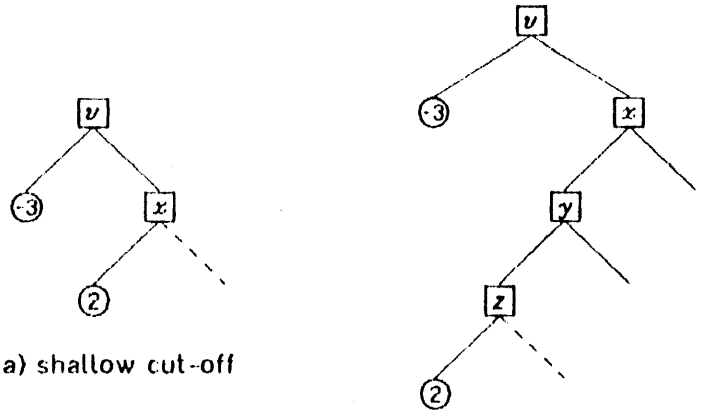


Figure 2.2 - Examples of possible cut-offs

Let us consider the pattern of Figure 2.2 (a) first. From the definition of the minimax procedure, the values v and x satisfy:

$$v = \max\{ 3, -x \} , \quad x = \max\{ -2, \dots \} ,$$

which shows that $x \geq -2$ or $2 \geq -x$. Since $3 \geq 2 \geq -x$, it follows that *independent of the exact value of* x, we will have $v = 3$. This shows that we need not explore further the successors of the node labeled x if we are only interested in the value of v. This leads to a first type of *cut-offs* known as *shallow cut-offs*.

The pattern of Figure 2.2 (b) illustrates a *deeper cut-off*. As with the previous example, there are immediate relations between the values of the nodes. In particular, we have $y \geq -z$, which leads us to consider two cases. Either $y > -z$, and this means that the value y is determined by its right son(s) and certainly does not depend on the right son(s) of z. Or $y = -z$, in which case, since $x \geq -y$ and $z \geq -2$, we deduce $x \geq -2$ or $-x \leq 2$; but since $v = \max\{3, -x\}$ it follows that $v = 3$, independent of the exact value of x and, a fortiori, independent of the exact value of z. This shows that in either case the successors of the node labeled z need not be further explored since the final value of v would in no way be affected.

The two examples presented in Figure 2.2 indicate that a reduction of the search

can be achieved if a node passes down to its sons the current value backed-up so far (3 in the case of the two above examples) as a bound for pruning branches *2, 4, 6,* ... levels below; the bound can, of course, be improved as the search progresses down the tree (leading to more and more possible cut-offs).

Using two bounds for even and odd levels of a tree, these improvements are implemented in the following procedure adapted from [35].

```
integer procedure ALPHABETA(position P, integer alpha, integer beta):
    begin integer j, t, n;
    determine the successor positions: P_1, ..., P_n;
    if n = 0 then
        ALPHABETA := f(P)
    else
        begin
        for j := 1 step 1 until n do
            begin
            t := -ALPHABETA(P_j,-beta,-alpha);
            if t > alpha then alpha := t;
            if alpha ≥ beta then goto done                    (2.1)
            end;
done:   ALPHABETA := alpha
        end
    end
```

The Alpha-Beta procedure (from [35])

The function denoted by *f* is the evaluation function which assigns *static values* to terminal positions.

Knuth and Moore [35] have shown this procedure to be correct in the sense that the call ALPHABETA(P,$-\infty$,$+\infty$) assigns to position P the value MINIMAX(P), assigned by the minimax procedure. More generally, they showed [35, p. 297] that, if alpha < beta:

ALPHABETA(P,alpha,beta) ≤ alpha, if MINIMAX(P) ≤ alpha, (2.2)

ALPHABETA(P,alpha,beta) = MINIMAX(P), if alpha < MINIMAX(P) < beta, (2.3)

ALPHABETA(P,alpha,beta) ≥ beta, if MINIMAX(P) ≥ beta. (2.4)

The same tree used in Figure 2.1 to illustrate the minimax procedure is shown in Figure 2.3 to illustrate the effects of the α-β procedure. The branches pruned by the

procedure are indicated with dashed lines, and the nodes marked with a circle have not been completely explored.

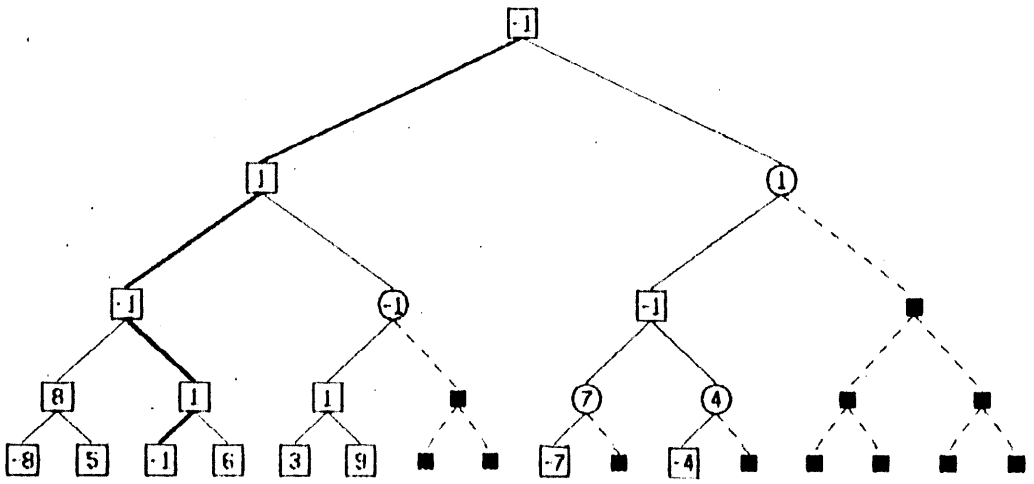


Figure 2.3 - Searching a game tree with the α-β procedure

We observe that only *8* out of the *16* terminal positions and *19* out of all the *31* nodes are examined by the α-β pruning algorithm in this example, reducing greatly the cost of searching the tree. As is seen by comparing Figures 2.1 and 2.3, the values backed-up by the α-β procedure to some internal nodes are not necessarily the same as the values backed-up by the minimax procedure, as reflected by the indetermination in equations (2.2) and (2.4). The top value, however, is not affected by this indetermination.

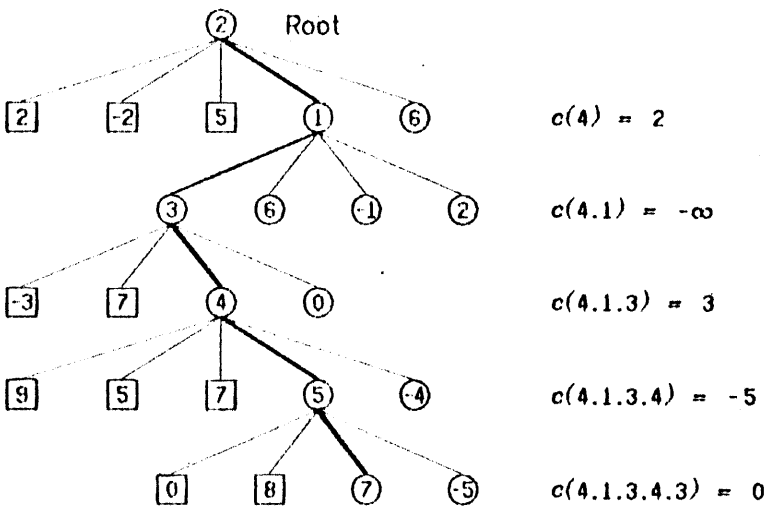
2.2 - Some properties of the α-β pruning algorithm

In this section, we will introduce some notations which will be used throughout, and we will reestablish, in terms of the Nega-Max approach, an initial result of [23] giving a necessary and sufficient condition for any node of a game tree to be examined by the α-β pruning algorithm.

2.2.1 - Notations

As in [35], we will use the Dewey decimal notation to represent a node in a tree.

More precisely, let ε, the empty sequence, denote the root of the game tree. Then, if $\mathcal{J}$ denotes some internal node of the tree with n sons, $\mathcal{J}.j$ will denote the j-th son of node $\mathcal{J}$, for $j = 1, ..., n$. In Figure 2.4, node *4.1.3.4.3* is the node at depth 5 whose path from the root is indicated with a darker line.



$$\alpha(4.1.3.4.3) = \max\{\, c(4.1.3.4.3),\ c(4.1.3),\ c(4) \,\} = 3$$

$$\beta(4.1.3.4.3) = -\max\{\, c(4.1.3.4),\ c(4.1) \,\} = 5$$

Figure 2.4 - Portion of a game tree showing the path to node <*4.1.3.4.3*>

The value associated with some node $\mathcal{J}$ of a game tree by the minimax procedure (see Section 2.1) will be denoted by $v(\mathcal{J})$. Then, if $\mathcal{J}$ is a terminal node, $v(\mathcal{J})$ is the *static value* asigned to that terminal position, and, if $\mathcal{J}$ is an internal node, $v(\mathcal{J})$ is the value backed-up to node $\mathcal{J}$ by the minimax procedure. In the latter case, if node $\mathcal{J}$ has n sons, $v(\mathcal{J})$ is given by:

$$v(\mathcal{J}) = \max\{\, -v(\mathcal{J}.j) \mid 1 \le j \le n \,\}. \qquad (2.5)$$

In Figure 2.4, the nodes on the path from the root to node *4.1.3.4.3* are evaluated through formula (2.5) while the other nodes (including *4.1.3.4.3*) are shown as terminal nodes and are assigned arbitrary values. (Nodes are labeled with their values.)

While the values $v(\mathcal{J})$ deal with the static aspect of a game tree, the quantities we

will introduce next deal more with the dynamic aspect of the tree when being searched by the α-β procedure.

For any node $\mathcal{J}.j$ at depth $d \geq 1$, we define:

$$c(\mathcal{J}.j) = \max\{ -v(\mathcal{J}.i) \mid 1 \leq i \leq j-1 \} .$$

(By convention, the maximum over an empty set is defined to be $-\infty$; in particular, $c(\mathcal{J}.1) = -\infty$.) For the root of the tree we also define $c(\epsilon) = -\infty$. The quantity $c(\mathcal{J})$ accounts for the information provided to node $\mathcal{J}$ by its *elder* brothers. These values are indicated to the right of the game tree shown in Figure 2.4 for all nodes on the path to node *4.1.3.4.3*; only the nodes indicated with squares are used in computing these values.

We finally define for any node $\mathcal{J} = j_1 . \ldots . j_d$ at depth $d \geq 1$ in a game tree two quantities directly associated with node $\mathcal{J}$ by the α-β procedure. For $i = 0, \ldots, d-1$, let $\mathcal{J}_i = j_1 . \ldots . j_{d-i}$. We define:

$$\alpha(\mathcal{J}) = \max\{ c(\mathcal{J}_i) \mid i \text{ is } even, 0 \leq i \leq d-1 \} ,$$

$$\beta(\mathcal{J}) = -\max\{ c(\mathcal{J}_i) \mid i \text{ is } odd, 0 \leq i \leq d-1 \} .$$

It is convenient to define these two quantities for the root of the game tree by $\alpha(\epsilon) = -\infty$ and $\beta(\epsilon) = +\infty$ (which is consistent with the definition). These α- and β-values are shown in Figure 2.4 for the node *4.1.3.4.3* along with their definitions.

2.2.2 - Necessary and sufficient condition for a node to be explored by the α-β procedure

The following lemma justifies the notations we just introduced in the preceding section.

Lemma 2.1:

Assume that, initially, the root of a game tree is explored by the α-β procedure through the call

ALPHABETA(root,$-\infty$,$+\infty$) . (2.6)

Then, if node $\mathcal{J}$ is examined, it is through a call of procedure ALPHABETA in which the parameters alpha and beta satisfy:

$$\text{alpha} = \alpha(\mathcal{J}), \tag{2.7}$$

$$\text{beta} = \beta(\mathcal{J}). \tag{2.8}$$

Proof:

If $\mathcal{J} = j_1.\ldots.j_d$ denotes some node explored by the procedure at depth $d \geq 1$, let, as before, $\mathcal{J}_i = j_1.\ldots.j_{d-i}$, for $0 \leq i \leq d-1$. Thus node $\mathcal{J}_1$ is the father of node $\mathcal{J}$, while, if $j_d \geq 2$, node $\mathcal{J}_1.(j_d-1)$ is the brother of $\mathcal{J}$ immediately preceding $\mathcal{J}$ (and explored just before $\mathcal{J}$). Observe that, if $j_d = 1$, $c(\mathcal{J}_0) = c(\mathcal{J}) = -\infty$ and therefore:

$$\begin{aligned}\alpha(\mathcal{J}) &= \max\{\, c(\mathcal{J}_i) \mid i \text{ is } even,\ 0 \leq i \leq d-1 \,\} \\ &= -\,[\,-\max\{\, c(\mathcal{J}_{i+1}) \mid i \text{ is } odd,\ 0 \leq i \leq d-2 \,\}\,] \\ &= -\,\beta(\mathcal{J}_1)\end{aligned}$$

(similarly, $\beta(\mathcal{J}) = -\alpha(\mathcal{J}_1)$). Observe also that, if $j_d \geq 2$:

$$\begin{aligned}\alpha(\mathcal{J}) &= \max\{\, \alpha(\mathcal{J}_2),\ c(\mathcal{J}) \,\} \\ &= \max\{\, \alpha(\mathcal{J}_2),\ c[\mathcal{J}_1.(j_d-1)],\ -v[\mathcal{J}_1.(j_d-1)] \,\}\end{aligned}$$

and that $\beta(\mathcal{J}) = \beta[\mathcal{J}_1.(j_d-1)]$.

By the call of line (2.6), relations (2.7) and (2.8) certainly hold for the root of the game tree, since $\alpha(\varepsilon) = -\infty$ and $\beta(\varepsilon) = +\infty$. Then the proof follows by induction from inspection of the procedure ALPHABETA, and from the relations we derived above. ■

The following theorem states a useful relation that characterizes the fact that a node of a tree is explored by the α-β pruning algorithm. This relation was first established by Fuller, Gaschnig and Gillogly [23] with different notations in terms of the Min-Max model.

Theorem 2.1:

Assume that, initially, the root of a game tree is explored by the α-β procedure through the call

ALPHABETA(root,$-\infty$,$+\infty$).

Then, an arbitrary node $\mathcal{J}$ of the game tree is subsequently explored if and only if

$$\alpha(\mathcal{J}) < \beta(\mathcal{J}). \tag{2.9}$$

Proof:

Because of the presence of line (2.1) in the procedure ALPHABETA, the result follows directly from the result of Lemma 2.1. ■

Since it will be more convenient in the following sections, rather than $\alpha(\mathcal{J})$ and $\beta(\mathcal{J})$, we will use the quantities:

$$A(\mathcal{J}) = \max\{ c(\mathcal{J}_i) \mid i \text{ is } even,\ 0 \le i \le d-1 \},$$

$$B(\mathcal{J}) = \max\{ c(\mathcal{J}_i) \mid i \text{ is } odd,\ 0 \le i \le d-1 \},$$

where $\mathcal{J}_i$ is defined as before. The definitions of $A(\mathcal{J})$ and $B(\mathcal{J})$ are more symmetrical, and relation (2.9) can also be rewritten in a more symmetrical way:

$$A(\mathcal{J}) + B(\mathcal{J}) < 0 . \qquad (2.10)$$

3 - Number of nodes explored by the α-β procedure: discrete case

As in [23] and [35], we will evaluate in this and the following section the amount of work performed in searching a *random uniform game tree* using the α-β pruning algorithm. The definition and some properties of random uniform game trees are given in Section 3.1. The amount of work performed by the α-β procedure is measured by the number of terminal nodes examined during the search and is evaluated in Section 3.2.

3.1 - Random uniform game trees

In order to perform an analysis of the α-β pruning algorithm, we will limit ourselves and consider the following class of game trees.

Definition 3.1:

A game tree in which

(a) all internal nodes have exactly n sons, and

(b) all terminal nodes (or *bottom positions*) are at depth d

is called a *uniform game tree* of degree n and depth d.

A uniform game tree which satisfies the additional condition

(c) the values assigned to all terminal nodes (or *bottom values*) are independent identically distributed random variables

is called a *random uniform game tree*, or, for short, a *rug tree*. ■

Unless otherwise specified, we will only consider throughout a rug tree of degree n and depth d.

Since the value backed-up to a node by the minimax procedure only depends on the backed-up values of its sons, we immediately observe that, by condition (c), the backed-up values of all nodes at the *same* depth are also independent identically distributed random variables. In the remainder of the section, we will assume that the bottom values are drawn from the finite set $\{ x_k = k/m \mid -m \le k \le m \}$, for some $m > 0$, and we will denote by $\{p_i(k)\}_{-m \le k \le m}$ or simply $\{p_i(k)\}$ the common probability distribution for the backed-up values of all nodes at depth $d - i$ (i. e., $p_i(k)$ is the probability that the value, $v(\mathcal{J})$, backed-up by the minimax procedure to some node $\mathcal{J}$ at depth $d-i$ be k/m). In particular, $\{p_0(k)\}$ is the common probability distribution for all bottom values, and $\{p_d(k)\}$ is the probability distribution for the value backed-up to the root of the rug tree.

The following lemma states the relations between these probability distributions.

Lemma 3.1:

For $i = 0, \ldots, d-1$, we have:

$$p_{i+1}(-m) + \ldots + p_{i+1}(k) = [p_i(-k) + \ldots + p_i(m)]^n . \tag{3.1}$$

Proof:

Let $\mathcal{J}$ be some internal node at depth $d-i-1$, then by equation (2.5), $v(\mathcal{J}) \le k$ if and only if $-v(\mathcal{J}.j) \le k$, for $j = 1, \ldots, n$. Equation (3.1) follows easily from the fact that all variables $v(\mathcal{J}.j)$ are independent. ■

Since the quantity $p_i(-k) + \ldots + p_i(m)$ will occur again later on, we define for $i = 0, 1, \ldots$ and $-m \le k \le m$:

$$\varphi_i(k) = p_i(-k) + \ldots + p_i(m) .$$

For convenience, we also define $\varphi_i(-m-1) = 0$. Note that $\varphi_i(k)$ is a non-decreasing function of k which satisfies $\varphi_i(-m-1) = 0$ and $\varphi_i(m) = p_i(-m) + \ldots + p_i(m) = 1$. By rewriting equation (3.1), we see that φ_i satisfies:

$$\varphi_{i+1}(-k-1) = 1 - [\varphi_i(k)]^n \quad \text{for } i = 0, 1, \ldots, \tag{3.2}$$

and, therefore:

$$\varphi_{i+2}(k) = 1 - \{1 - [\varphi_i(k)]^n\}^n \quad \text{for } i = 0, 1, \ldots . \tag{3.3}$$

The following quantities will also be useful in Section 3.2. For $i = 0, 1, \ldots$ and $-m-1 \le k \le m$, define:

$$\rho_i(k) = 1 + [\varphi_i(k)] + \ldots + [\varphi_i(k)]^{n-1}, \tag{3.4}$$

and

$$\sigma_i(k) = 1 + [\varphi_i(-k-1)] + \ldots + [\varphi_i(-k-1)]^{n-1}. \tag{3.5}$$

Observe that $\rho_i(-m-1) = \sigma_i(m) = 1$ and $\rho_i(m) = \sigma_i(-m-1) = n$.

Lemma 3.1 establishes the probability distributions for all the values in the nodes of a rug tree. The next lemma establishes a similar result for the quantities $c(\mathcal{J})$ defined in Section 2.

Lemma 3.2:

Let $\mathcal{J}.j$ denote any node at depth i, where $i = 1, \ldots, d$. If $j = 1$, $c(\mathcal{J}.j) = -\infty$. If $j \ge 2$, then the probability distribution of $c(\mathcal{J}.j)$, denoted by $\{q_k(\mathcal{J}.j)\}_{-m \le k \le m}$, satisfies:

$$q_{-m}(\mathcal{J}.j) + \ldots + q_k(\mathcal{J}.j) = [\varphi_{d-i}(k)]^{j-1}. \tag{3.6}$$

Proof:

When $j = 1$, $c(\mathcal{J}.j) = -\infty$ by definition. When $j \ge 2$, equation (3.6) follows from the same argument given in the proof of Lemma 3.1. ■

In order to evaluate, through equation (2.10), the probability that a terminal node is explored, we first need to determine the probability distributions for the two quantities $A(\mathcal{J})$ and $B(\mathcal{J})$. This is done in the following.

Lemma 3.3:

Let $\mathcal{J} = j_{d-1} \cdot \ldots \cdot j_1 \cdot j_0$ denote any terminal node.

(1) If $j_i = 1$ for all *even* integers i in the range $0 \le i \le d-1$, then $A(\mathcal{J}) = -\infty$.

(2) Otherwise, the probability distribution for $A(\mathcal{J})$, denoted by $\{a_k(\mathcal{J})\}_{-m \le k \le m}$, satisfies:

$$a_{-m}(\mathcal{J}) + \ldots + a_k(\mathcal{J}) = \prod_e [\varphi_i(k)]^{j_i - 1}, \tag{3.7}$$

where the product denoted by $\prod_e$ is extended to all *even* integers in the range $0 \le i \le d-1$.

Similarly,

(1') If $j_i = 1$ for all *odd* integers i in the range $1 \le i \le d-1$, then $B(\mathcal{J}) = -\infty$.

(2') Otherwise, the probability distribution for $B(\mathcal{J})$, denoted by $\{b_k(\mathcal{J})\}_{-m \le k \le m}$, satisfies:

$$b_{-m}(\mathcal{J}) + \ldots + b_k(\mathcal{J}) = \prod_o [\varphi_i(k)]^{j_i - 1}, \tag{3.8}$$

where the product denoted by $\prod_o$ is extended to all *odd* integers in the range $1 \le i \le d-1$.

Proof:

We will only consider $A(\mathcal{J})$ since the proof relative to $B(\mathcal{J})$ is the same. Part (1) follows directly from the definition. For part (2), let $\mathcal{J}_i$ denote the node $j_{d-1} \cdot \ldots \cdot j_i$. We note that $A(\mathcal{J}) \le k$ if and only if $c(\mathcal{J}_i) \le k$ for all even integers i in the range $0 \le i \le d-1$ such that $j_i \ge 2$. Since the variables $c(\mathcal{J}_i)$ are independent, equation (3.7) follows from equation (3.6) by observing that, in the product $\prod_e$, a factor corresponding to $j_i = 1$ amounts to 1. ■

The last lemma in this section states the probability of exploring a terminal node.

Lemma 3.4:

Let $\mathcal{J} = j_{d-1} \cdot \ldots \cdot j_1 \cdot j_0$ denote any terminal node. The probability $\pi(\mathcal{J})$ that node $\mathcal{J}$ is examined by the α-β procedure is given by:

$\pi(\mathcal{J}) = 1$ if $j_i = 1$ for all even integers i in the range $0 \le i \le d-1$,

$\pi(\mathcal{J}) = 1$ if $j_i = 1$ for all odd integers i in the range $1 \le i \le d-1$,

$$\pi(\mathcal{J}) = \sum_{-m \le k \le m-1} a_k(\mathcal{J}) \, [b_{-m}(\mathcal{J}) + \ldots + b_{-k-1}(\mathcal{J})] \quad \text{otherwise.} \tag{3.9}$$

Proof:

When $j_i = 1$ for all even integers i in the range $0 \le i \le d-1$, by Lemma 3.3 $A(\mathcal{J}) = -\infty$. Hence $A(\mathcal{J}) + B(\mathcal{J}) = -\infty$ too, and by Theorem 2.1 node $\mathcal{J}$ is certainly explored. Similarly when $j_i = 1$ for all odd integers in the range $1 \le i \le d-1$.

Otherwise, both $A(\mathcal{J})$ and $B(\mathcal{J})$ are finite. Let $A(\mathcal{J}) = x_k$. We observe that $A(\mathcal{J}) + B(\mathcal{J}) < 0$ if and only if $-m \le k \le m-1$ and $-x_m \le B(\mathcal{J}) \le x_{-k-1}$. Hence, equation (3.9) follows from Theorem 2.1 and the fact that $A(\mathcal{J})$ and $B(\mathcal{J})$ are independent variables. ■

Using equations (3.7) and (3.8), equation (3.9) can be rewritten as:

$$\pi(\mathcal{J}) = \sum_{-m \le k \le m-1} a_k(\mathcal{J}) \prod_o [\varphi_i(-k-1)]^{j_i - 1} ,$$

$$\pi(\mathcal{J}) = \sum_{-m \le k \le m-1} \{\prod_e [\varphi_i(k)]^{j_i - 1} - \prod_e [\varphi_i(k-1)]^{j_i - 1}\} \prod_o [\varphi_i(-k-1)]^{j_i - 1} \tag{3.10}$$

(recall that $\varphi_i(-m-1) = 0$).

3.2 - Number of terminal nodes examined by the α-β pruning algorithm: discrete case

We are now able to evaluate the amount of work performed by the α-β procedure while searching a rug tree. As in [23] and [35], we have chosen to measure the amount of work by the number of terminal nodes examined by the procedure. (We will also consider briefly, at the end of the section, the total number of internal and terminal nodes explored by the procedure as a measure of performance.)

Theorem 3.1:

The average number, $N_{n,d}(m)$, of bottom positions examined by the α-β procedure in searching a rug tree of degree n and depth d, for which the bottom values are distributed according to the discrete probability distribution $\{p_0(k)\}_{-m \le k \le m}$, is given by:

$$N_{n,d}(m) = n^{\lfloor d/2 \rfloor} + \sum_{-m \le k \le m} [\prod_e \rho_i(k) - \prod_e \rho_i(k-1)] \prod_o \sigma_i(k) , \tag{3.11}$$

where the quantities $\rho_i(k)$ and $\sigma_i(k)$ are defined by equations (3.4) and (3.5), and where the products denoted by $\prod_e$ and $\prod_o$ are defined in Lemma 3.3.

Proof:

By definition of the probability $\pi(j)$, the average number of bottom positions examined by the α-β procedure is

$$N_{n,d}(m) = \sum \pi(j) ,$$

where the sum is extended to all terminal nodes $j = j_{d-1} \cdot \ldots \cdot j_1 \cdot j_0$, and is actually a d-nested summation over the range $1 \le j_0 \le n$, $1 \le j_1 \le n$, ..., $1 \le j_{d-1} \le n$. The summation can be rearranged as:

$$N_{n,d}(m) = \sum_e \pi(j) + \sum_o \pi(j) + \sum' \pi(j) - \pi(1. \ldots .1) ,$$

where the three summations $\sum_e$, $\sum_o$ and $\sum'$ correspond to the three expressions for $\pi(j)$ given in Lemma 3.4. The fourth term $\pi(1. \ldots .1)$ is subtracted from the sum since it is counted by both $\sum_e$ and $\sum_o$. These two sums are easily evaluated since all the terms $\pi(j)$ are 1. As $\pi(1. \ldots .1)$ itself is 1, we obtain:

$$N_{n,d}(m) = n^{\lceil d/2 \rceil} + n^{\lfloor d/2 \rfloor} - 1 + \sum' \pi(j) . \qquad (3.12)$$

It is to be noted that the first three terms correspond exactly to the number of terminal nodes examined by the α-β procedure under optimal ordering of the bottom values (see [56, p. 201]).

We now evaluate the sum $\sum'$. Inside the sum the terms $\pi(j)$ can be evaluated through equation (3.10). We note that all the summations relative to j_i, for $i = 0, 1, \ldots, d-1$, can be done independently, each one being the sum of a geometric series. Using the quantities $\rho_i(k)$ and $\sigma_i(k)$ defined by equations (3.4) and (3.5), we obtain:

$$\sum' \pi(j) = \sum_{m \le k \le m-1} [\prod_e \rho_i(k) - \prod_e \rho_i(k-1)] \prod_o \sigma_i(k) - \prod_e \rho_i(m-1) + 1 .$$

The theorem follows from this last equation and equation (3.12), using the facts that $\rho_i(m) = n$ and that $\sigma_i(m) = 1$. ■

The formula of equation (3.11) can be easily evaluated and provides us with a measure of performance for the α-β pruning algorithm. For some applications, however (especially when the cost of generating moves is greater than the cost of evaluating positions), it is more convenient to use the total number of nodes (internal and terminal)

explored by the procedure as a measure of performance. Let $T_{n,d}(m)$ denote the average of this number. The same way we evaluated $N_{n,d}(m)$, we can evaluate $T_{n,d}(m)$ by summing the probabilities $\pi(j)$ over all nodes of the tree. We obtain:

$$T_{n,d}(m) = N^0_{n,d}(m) + N^1_{n,d}(m) + \ldots + N^d_{n,d}(m) ,$$

where $N^i_{n,d}(m)$ is the average number of nodes examined at depth i, and is directly derived from the expression of $N_{n,d}(m)$ in equation (3.11) by replacing d by i and $\{p_0(k)\}$ by $\{p_{d-i}(k)\}$ (recall that $\{p_0(k)\}$ is the probability distribution for the values assigned to the terminal nodes and that $\{p_{d-i}(k)\}$ is the probability distribution for the values backed-up to nodes at depth i).

3.3 - Bi-valued rug trees

Although it is relatively easy in most game playing programs to obtain (by inspection of the evaluation function) an accurate bound for the range of distinct values assigned to the various positions of the game, it is usually not so easy to derive a good estimate for the probability distribution of these values. In the remainder of the section we will study rug trees in which the terminal nodes can only take on two distinct values, and we will see, in particular, that a change in the probability distribution of these values can lead to very important differences in the growth rate of $N_{n,d}(m)$.

We will assume in the following that the values assigned to the terminal nodes of a rug tree can only be either -1 or $+1$ with respective probabilities $1-p$ and p, for some $p \in [0, 1]$. Under these conditions, the number, $T_{n,d}(p)$, of terminal nodes examined by the α-β procedure can be obtained as a particular case of equation (3.11) in which $m = 1$ and $\{p_0(k)\}_{-m \le k \le m}$ is defined by $p_0(-1) = 1-p$, $p_0(0) = 0$, $p_0(1) = p$.

Theorem 3.2:

Let $p_0 = p$, and, for $i = 1, 2, \ldots$, let $p_i = 1 - p^n_{i-1}$.

$$T_{n,d}(p) = n^{\lceil d/2 \rceil} + n^{\lfloor d/2 \rfloor} - 1 + (P_e - 1)(P_o - 1) , \qquad (3.13)$$

with

$$P_e = \prod_e \frac{p_{i+1}}{1 - p_i} , \quad P_o = \prod_o \frac{p_{i+1}}{1 - p_i} ,$$

where the products Π_e and Π_o are defined as before.

Proof:

Choose $m = 1$ and define the probability distribution $\{p_0(k)\}_{-m \le k \le m}$ by $p_0(-1) = 1-p$, $p_0(0) = 0$ and $p_0(1) = p$. Hence $\varphi_0(-2) = 0$, $\varphi_0(-1) = \varphi_0(0) = p = p_0$ and $\varphi_0(1) = 1$. By equation (3.2) we obtain:

$$\varphi_i(-2) = 0 \ , \ \varphi_i(-1) = \varphi_i(0) = p_i \ , \ \varphi_i(1) = 1 \ , \ \text{for} \ \ i = 0, 1, \ldots .$$

Then equation (3.13) follows directly from Theorem 3.1 and equations (3.4) and (3.5). ■

Equation (3.13) can be evaluated very easily and, in particular, we note that for $0 < p < 1$:

$$T_{n,d}(p) > T_{n,d}(0) = T_{n,d}(1) = n^{\lfloor d/2 \rfloor} + n^{\lfloor d/2 \rfloor} - 1 . \tag{3.14}$$

This last equation shows that $T_{n,d}(p)$ reaches its minimum $n^{\lfloor d/2 \rfloor} + n^{\lfloor d/2 \rfloor} - 1$ for $p = 0$ and $p = 1$. This is in agreement with the result of Slagle and Dixon [56, p. 201] since it corresponds to the case when all terminal nodes are assigned the same value and therefore all possible cut-offs do occur. Equation (3.14) also shows that $T_{n,d}(p)$ admits a maximum for $p \in (0, 1)$; although the exact maximum cannot be readily obtained, we will derive a lower bound in the following. We first establish a preliminary result.

Lemma 3.5:

The unique positive root, ξ_n, of the equation

$$x^n + x - 1 = 0$$

is in the interval $(0, 1)$. Asymptotically (for large n) it satisfies:

$$1 - \xi_n \sim \frac{1}{n} \ln n . \tag{3.15}$$

Proof:

As there is no ambiguity, we will drop the index n from ξ_n in the following.

Let $g(x) = x^n + x - 1$, note that $g(0) = -1 < 0$ and $g(1) = 1 > 0$. Since $g(x)$ is continuous and strictly increases for x positive, the equation $g(x) = 0$ admits a unique positive root, ξ, which is in the interval $(0, 1)$.

We observe that equation $\xi^n + \xi - 1 = 0$ can be rewritten as

$$1 - \xi = \frac{1}{1 + (1 + \xi + \ldots + \xi^{n-1})},$$

from which we deduce that

$$1 - \xi > \frac{1}{n+1}. \tag{3.16}$$

On the other hand, since $\xi^n = 1 - \xi$, we obtain

$$n(\xi - 1) > n \ln \xi = \ln(1-\xi),$$

which shows, along with equation (3.16), that

$$1 - \xi < \frac{1}{n} \ln(n+1) = \frac{1}{n} \ln n + O(n^{-2}). \tag{3.17}$$

Similarly, taking the logarithm of both sides of equation (3.17), and using the facts that $1 - \xi = \xi^n$ and that $\ln \xi > 1 - \frac{1}{\xi}$, we obtain:

$$\xi < \frac{1}{1 + \ln(n/\ln n+1)},$$

hence:

$$1 - \xi > \frac{1}{n} \ln(n/\ln n+1) + O[(\frac{1}{n} \ln n)^2] = \frac{1}{n} \ln n + O(\frac{1}{n} \ln \ln n).$$

Equation (3.15) follows directly from the previous equation and equation (3.17). ■

When $p = \xi_n$ we obtain immediately that, for $i = 0, 1, \ldots, p_i = \xi_n$. Hence

$$P_e = [\xi_n/(1-\xi_n)]^{\lfloor d/2 \rfloor} \text{ and } P_o = [\xi_n/(1-\xi_n)]^{\lfloor d/2 \rfloor}.$$

From equations (3.13) and (3.15) it follows that, for large n:

$$T_{n,d}(\xi_n) \sim [n/\ln n]^d, \tag{3.18}$$

while equation (3.14) shows that

$$T_{n,d}(0) = T_{n,d}(1) \sim O(n^{\lfloor d/2 \rfloor}). \tag{3.19}$$

Equations (3.18) and (3.19) indicate that $T_{n,d}(p)$ can be largely influenced by the variations of the probability distribution for the static values. This result can be easily generalized to $N_{n,d}(m)$. In the next section, we will derive an approximation to $N_{n,d}(m)$ which corresponds to its worst case behavior.

4 - Number of nodes explored by the α-β procedure: continuous case

In this section, we derive an approximation to $N_{n,d}(m)$ by considering the limit of the finite series of equation (3.11) when m tends to infinity while the discrete probability

distribution $\{p_0(k)\}_{-m\le k\le m}$ tends to a continuous probability distribution. This corresponds to the case studied by Fuller, Gaschnig and Gillogly [23] and by Knuth and Moore [35] when the terminal nodes of a rug tree are all assigned distinct values. In particular, we will reestablish (with a much simpler formula) a result of [23].

4.1 - Notations and preliminary results

We first introduce the sequence of functions $\{f_i\}$ mapping the interval $[0, 1]$ into itself, and defined recursively by:

$$f_0(x) = x ,$$

$$f_i(x) = 1 - \{1 - [f_{i-1}(x)]^n\}^n \quad \text{for } i = 1, 2, \ldots .$$

It is readily verified by induction on i that all functions f_i are strictly increasing on $[0, 1]$ and satisfy $f_i(0) = 0$ and $f_i(1) = 1$, i. e., 0 and 1 are two fixed points of the functions f_i, for all n and i. The function f_i will be shown to be related to the quantities $\rho_{2i}(k)$ defined in Section 3.1. Similarly, in relation to the quantities $\rho_{2i}(k)$ and $\sigma_{2i+1}(k)$, we define the following functions on $[0, 1]$: for $i = 1, 2, \ldots$, let

$$r_i(x) = \frac{1-[f_{i-1}(x)]^n}{1-f_{i-1}(x)} ,$$

$$s_i(x) = \frac{f_i(x)}{[f_{i-1}(x)]^n} .$$

If we define $r_i(1) = n$ and $s_i(0) = 1$, we observe that all functions r_i and s_i are continuous on $[0, 1]$ (they are actually polynomials in x), and that r_i is strictly increasing while s_i is strictly decreasing.

In relation to the two products Π_e and Π_o, we also introduce, for $i = 1, 2, \ldots$, the following functions on $[0, 1]$:

$$R_i(x) = r_1(x) \times \ldots \times r_{\lfloor i/2 \rfloor}(x) ,$$

$$S_i(x) = s_1(x) \times \ldots \times s_{\lfloor i/2 \rfloor}(x) ,$$

where $S_1(x) = 1$. Observe here, too, that functions R_i and S_i are polynomials, and that, when x increases from 0 to 1, $R_i(x)$ increases from 1 to $n^{\lfloor i/2 \rfloor}$ while $S_i(x)$ decreases from $n^{\lfloor i/2 \rfloor}$ to 1.

Lastly, for $k = 0, 1, \ldots, 2m+1$, let

$$\tau_k = \varphi_0(k-m-1) .$$

Lemma 4.1:

For $i = 1, 2, \ldots$ and $k = 0, \ldots, 2m+1$, we have:

$$r_i(\tau_k) = \rho_{2i-2}(k-m-1) , \tag{4.1}$$

$$s_i(\tau_k) = \sigma_{2i-1}(k-m-1) . \tag{4.2}$$

Proof:

We first show that for $i = 0, 1, \ldots$ and $k = 0, \ldots, 2m+1$:

$$f_i(\tau_k) = \varphi_{2i}(k-m-1) . \tag{4.3}$$

Since $f_0(x) = x$, it follows from the definition of τ_k that equation (4.3) holds when $i = 0$. Assume, for induction, that equation (4.3) holds for $i = h$. Then by equation (3.3)

$$\varphi_{2h+2}(k-m-1) = 1 - \{1 - [f_h(\tau_k)]^n\}^n ,$$

which shows that equation (4.3) also holds for $i = h+1$ (from the definition of f_{h+1}).

Observe that $r_i(\tau_k) = 1 + [f_{i-1}(\tau_k)] + \ldots + [f_{i-1}(\tau_k)]^{n-1}$, then equation (4.1) follows from equations (4.3) and (3.4). Similarly, if we note that $s_i(x)$ can be rewritten as

$$s_i(x) = \frac{1 - \{1 - [f_{i-1}(x)]^n\}^n}{1 - \{1 - [f_{i-1}(x)]^n\}} ,$$

equation (4.2) follows from equations (3.2), (4.3) and (3.5). ∎

4.2 - Number of bottom positions examined by the α-β procedure: continuous case

Let us return to the definition of the sequence $T_m = \{\tau_k\}_{0 \le k \le 2m+1}$. As was observed in Section 3.1 with the sequence $\{\varphi_i(k)\}$, the sequence T_m is non-decreasing and defines a partition of the interval $[0, 1]$, i. e.:

$$0 = \tau_0 \le \tau_1 \le \ldots \le \tau_{2m} \le \tau_{2m+1} = 1 .$$

The norm of the partition T_m is

$$\|T_m\| = \max\{ \tau_k - \tau_{k-1} \mid 1 \le k \le 2m+1 \} = \max\{ p_0(k) \mid -m \le k \le m \} .$$

In the remainder of the section we require the following.

Assumption:

$$\text{(A1)} \quad \lim_{m\to\infty} \max\{\, p_0(k) \mid -m \le k \le m \,\} = 0. \qquad \blacksquare$$

This assumption ensures that the norm of the partition T_m tends to 0 when m tends to infinity. It also shows that, as m tends to infinity, the probability of two terminal nodes being assigned the same value vanishes. This corresponds to the case studied by Fuller, Gaschnig and Gillogly [23], and by Knuth and Moore [35].

With this assumption, we will now see that the finite series of equation (3.11) can be replaced by an integral when $m\to\infty$. This is established in the following.

Theorem 4.1:

Under assumption (A1), we have:

$$\lim_{m\to\infty} N_{n,d}(m) = n^{\lfloor d/2 \rfloor} + \int_0^1 R'_d(t).S_d(t).dt\,, \qquad (4.4)$$

where $R'_d(x)$ is the first derivative of $R_d(x)$.

Proof:

Since there is no risks of confusion, we will drop, in the following, the index d from the functions R_d and S_d.

It follows directly from Lemma 4.1 that for $k = 0, \ldots, 2m+1$:

$$R(\tau_k) = \prod_e \rho_i(k-m-1)\,,$$

$$S(\tau_k) = \prod_o \sigma_i(k-m-1)\,,$$

which shows that equation (3.11) can be simply rewritten as:

$$N_{n,d}(m) = n^{\lfloor d/2 \rfloor} + \sum_{1\le k\le 2m+1} [R(\tau_k) - R(\tau_{k-1})]\, S(\tau_k)\,.$$

Let A_m denote the series defined in this last equation.

Recall that $R(x)$ is a polynomial. By considering the Taylor development of $R(\tau_{k-1})$, we obtain for $k = 1, \ldots, 2m+1$:

$$R(\tau_k) - R(\tau_{k-1}) = [\tau_k - \tau_{k-1}]\, R'(\tau_k) + \frac{1}{2}\,[\tau_k - \tau_{k-1}]^2\, R''(t_k)\,,$$

where $\tau_{k-1} \le t_k \le \tau_k$. Hence:

$$A_m = \sum_{1\le k\le 2m+1} [\tau_k - \tau_{k-1}]\, R'(\tau_k)\, S(\tau_k) + \sum_{1\le k\le 2m+1} \frac{1}{2}\,[\tau_k - \tau_{k-1}]^2\, R''(t_k)\, S(\tau_k)\,. \qquad (4.5)$$

Since R and S are polynomials, the quantity $|R''(x)S(y)/2|$ is bounded by some constant, say M, for any x and y in $[0, 1]$. In particular, the second sum in equation (4.5) is bounded in module by $M.\|T_m\|.[x_{2m+1}-x_0] = M.\|T_m\|$ and therefore tends to 0 when $m \to \infty$ since, from assumption (A1), $\|T_m\| \to 0$.

As for the first sum in equation (4.5), we observe that it corresponds to a Riemann sum for the function $R'(x)S(x)$ over the partition T_m of $[0, 1]$. Therefore since, in particular, this function is continuous and since $\|T_m\|$ tends to 0, the sum tends to the integral of equation (4.4). This proves the theorem. ■

In the remainder of the section we will reinterpret the limit of $N_{n,d}(m)$ established in Theorem 4.1.

Let G be the distribution function of some continuous probability density function g, and assume, to simplify the discussion, that $G(-1) = 0$ and $G(1) = 1$ (therefore, $G(x) = 0$ for $x \le -1$ and $G(x) = 1$ for $x \ge 1$). We define a sequence of functions G_m for $m = 0, 1, \ldots$ as follows. For $-m \le k \le m$, let $x_k = k/m$. Function G_m is defined as the following step function:

$$G_m(x) = \begin{cases} 0 & \text{if } x < x_{-m} = 0\,, \\ G(x_k) & \text{if } x_k \le x < x_{k+1}\,, \text{ for } -m \le k \le m-1\,, \\ 1 & \text{if } 1 = x_m \le x\,. \end{cases}$$

The sequence of functions $\{G_m\}$ constitutes a sequence of approximations to the continuous function G. (It should be noted that the convergence of the sequence is uniform on the interval $[0, 1]$.) The function G_m corresponds to the cumulative distribution of the discrete probability distribution $p_0(k) = G_m(x_k^+) - G_m(x_k^-)$ associated with the points $x_k = k/m$, for $k = -m, \ldots, m$.

Using the approximation $\{p_0(k)\}_{-m \le k \le m}$ to the density function g, equation (3.11) provides us with an approximation to the average number of bottom positions examined by the α-β procedure in a rug tree in which the bottom values are drawn from the continuous

probability density function g. When m becomes larger, the approximation becomes better, and (due to the uniform convergence of the sequence G_m) it can actually be shown (in a rather technical way) that the limit of $N_{n,d}(m)$ when $m \to \infty$ corresponds exactly to the average number of bottom positions examined by the α-β procedure in the continuous case. As a matter of fact, equation (4.4) could be derived directly by considering a continuous probability distribution rather than a discrete one in very much the same way we derived equation (3.11) in Section 3. This result is stated in the following.

Theorem 4.2:

Let $f_0(x) = x$, and, for $i = 1, 2, \ldots$, define:

$$f_i(x) = 1 - \{1 - [f_{i-1}(x)]^n\}^n ,$$

$$r_i(x) = \frac{1-[f_{i-1}(x)]^n}{1-f_{i-1}(x)} ,$$

$$s_i(x) = \frac{f_i(x)}{[f_{i-1}(x)]^n} ,$$

$$R_i(x) = r_1(x) \times \ldots \times r_{\lfloor i/2 \rfloor}(x) ,$$

$$S_i(x) = s_1(x) \times \ldots \times s_{\lfloor i/2 \rfloor}(x) .$$

The average number, $N_{n,d}$, of terminal nodes examined by the α-β pruning algorithm in a rug tree of degree n and depth d for which the bottom values are drawn from a continuous distribution is given by:

$$N_{n,d} = n^{\lfloor d/2 \rfloor} + \int_0^1 R'_d(t).S_d(t).dt . \qquad (4.6)$$

It is to be noted that, unlike the case of a discrete probability distribution, when the bottom values are drawn from a continuous distribution, the number of terminal positions examined by the α-β procedure does not depend on the distribution function.

4.3 - Discrete case versus continuous case

Since equation (4.6) has been derived as the limit of equation (3.11), it is reasonable to investigate the validity of the approximation of $N_{n,d}(m)$ by $N_{n,d}$. As was seen in Section 3.3, $N_{n,d}(m)$ strongly depends on the probability distribution $\{p_0(k)\}_{-m \le k \le m}$ and,

therefore, we cannot expect $N_{n,d}$ to be a close approximation of $N_{n,d}(m)$ in all cases. We will see below, however, that $N_{n,d}$ provides us with a good insight into the behavior of the α-β pruning algorithm. Namely, we will see that it constitutes the worst case of $N_{n,d}(m)$ over all discrete probability distributions.

Since $N_{n,d}$ was obtained as the limit of $N_{n,d}(m)$, it is sufficient to show that, for all probability distributions $\{p_0(k)\}_{-m \le k \le m}$, we have:

$$N_{n,d} \ge N_{n,d}(m) . \tag{4.7}$$

In order to prove inequality (4.7), it is convenient to give a geometric interpretation of both $N_{n,d}$ and $N_{n,d}(m)$.

Consider the curve $(\mathcal{L})$ defined by the Cartesian coordinates (x, y) through the parametric equations

$$(\mathcal{L}): \ [\ x = R_d(t)\ ,\ y = S_d(t)\ ]\ ,$$

where the parameter t varies in the interval $[0, 1]$. The integral of equation (4.6) represents the area delimited by the curve $(\mathcal{L})$, the x-axis and the parallels to the y-axis at the abscissas $R_d(0) = 1$ and $R_d(1) = n^{\lfloor d/2 \rfloor}$ (see Figure 4.1). Since $R_d(0) = 1$ and $S_d(0) = n^{\lfloor d/2 \rfloor}$, the term $n^{\lfloor d/2 \rfloor}$ of equation (4.6) can be accounted for by the area of the rectangle delimited by the x-axis, the y-axis and the lines $x = 1$ and $y = n^{\lfloor d/2 \rfloor}$ (the latter line extends the curve $(\mathcal{L})$ in a continuous way). Figure 4.1 represents the curve $(\mathcal{L})$ and its *extension* in the case $n = 3$, $d = 6$. The area below the unbroken lines represents the quantity $N_{n,d}$.

The sum of equation (3.11) can also be represented along with the curve $(\mathcal{L})$. It follows directly from the relations of equations (4.1) and (4.2) that the terms of the sum represent the areas of the rectangles delimited by the lines $x = R(\tau_{k-1})$, $x = R(\tau_k)$, $y = 0$ and $y = S(\tau_k)$, for $k = 1, 2, \ldots, 2m-1$. The quantity $N_{n,d}(m)$ represents therefore the area of Figure 4.1 shown below the broken lines.

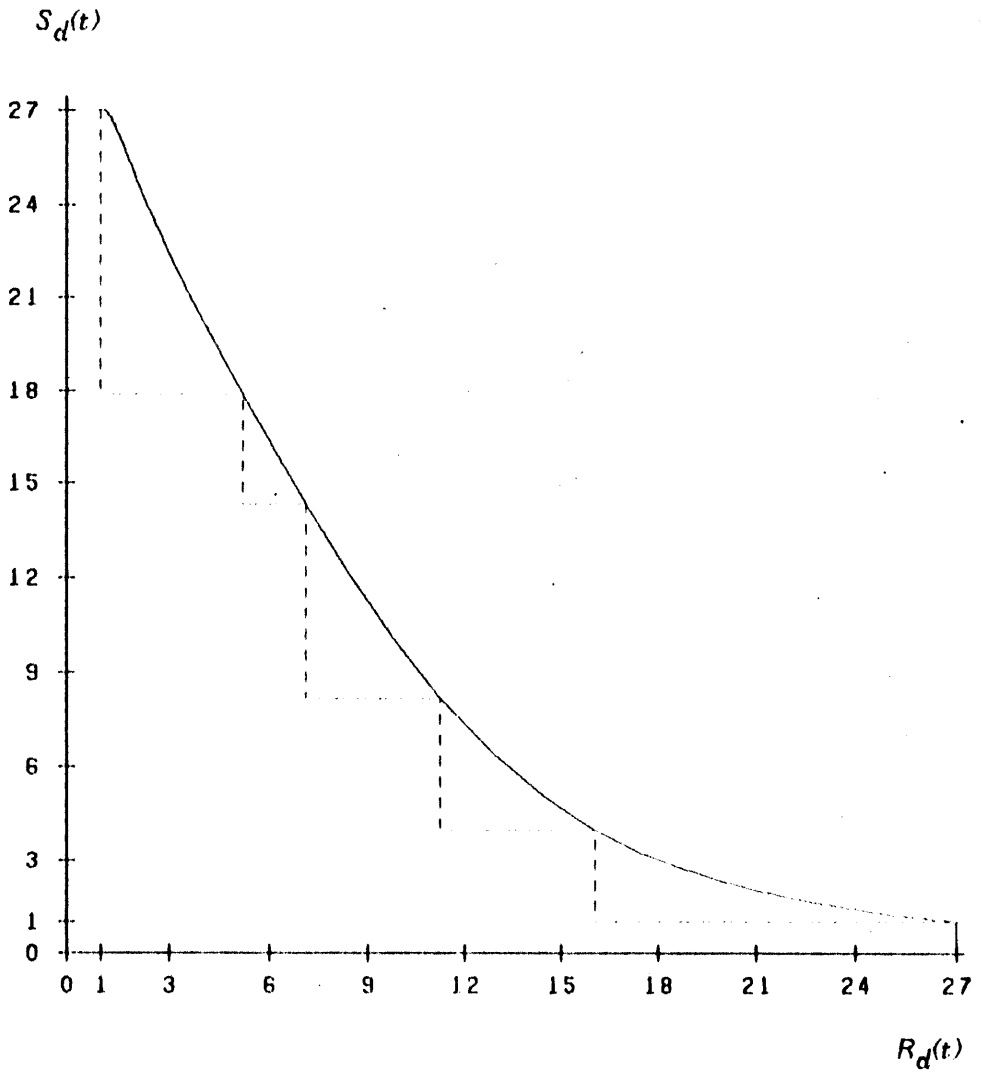


Figure 4.1 - Geometric interpretation of $N_{n,d}$ and $N_{n,d}(m)$

Inequality (4.7), then, follows directly from the fact that, when t increases in $[0, 1]$, $R(t)$ increases while $S(t)$ decreases.

5 - On the branching factor of the α-β pruning algorithm

We have deliberately chosen to introduce first the case when the bottom values of a game tree are drawn from a discrete probability distribution since it is of most interest in practical applications. The case of a continuous distribution, however, lends itself more easily to an analysis, and, since it constitutes the worst case over all discrete probability distributions, we will, in this section, examine the integral of equation (4.6) rather than the series of equation (3.11).

5.1 - Previous results

In Section 1, we introduced the branching factor as a cost measure for the work involved in searching a tree. Rather than considering the number, $N_{n,d}$, of terminal positions examined by a search algorithm, as a measure of performance of the algorithm, we could have considered the total number, $T_{n,d}$, of nodes (terminal and internal) explored during the search. In the case of the α-β pruning algorithm, since $N_{n,d}$, given by equation (4.6), does not depend on the distribution function of the bottom values, we deduce that $T_{n,d}$ satisfies:

$$T_{n,d} = 1 + N_{n,1} + \ldots + N_{n,d} .$$

It can be checked easily that $0 \le N_{n,i-1} \le N_{n,i}$, therefore $N_{n,d} \le T_{n,d} \le dN_{n,d}$, and:

$$\lim_{d\to\infty} (T_{n,d})^{1/d} = \lim_{d\to\infty} (N_{n,d})^{1/d} = \mathcal{R}_{\alpha\text{-}\beta}(n) .$$

Thus, Definition 1.1 provides us with a measure of performance useful to compare search algorithms. In the following, we review some of the results which have already been presented in the literature.

Minimax search

The minimax search examines systematically all nodes of a tree. It, therefore, examines $N_{n,d} = n^d$ terminal nodes in a uniform tree of degree n and depth d, leading to a branching factor

$$\mathcal{R}_{\text{minimax}}(n) = n .$$

α-β procedure under optimal ordering

Slagle and Dixon [56, p. 201] have shown that, when all possible α- and β-cut-offs occur, the α-β procedure examines

$$N_{n,d} = n^{\lceil d/2 \rceil} + n^{\lfloor d/2 \rfloor} - 1$$

terminal positions. In this case, the corresponding branching factor is

$$\mathcal{R}_{\text{opt}}(n) = n^{1/2} .$$

α-β procedure (experimental results from [23])

Based on a series of simulation results, Fuller, Gaschnig and Gillogly [23] have argued that the formula

$$N_{n,d} = c(d).n^{0.72d + 0.277}$$

constitutes a reasonable approximation to the number of bottom positions examined by the α-β procedure for small values of n and d, and that $1 \leq c(d) \leq 2$ (at least for the range of values they considered). For purposes of comparison, let us assume that their approximation can be extrapolated for any n and d. Provided that $c(d)^{1/d} \rightarrow 1$ when $d \rightarrow \infty$, we obtain

$$\mathcal{R}_{\alpha\text{-}\beta}(n) \sim n^{0.72}.$$

In view of the results of Section 3.3, we can question the accuracy of the approximation for large n since it follows from Theorem 3.2 that

$$\lim_{d\rightarrow\infty} [T_{n,d}(\xi_n)]^{1/d} = O(n/\ln n).$$

α-β procedure without deep cut-offs

Knuth and Moore [35] have analyzed a simpler version of the α-β procedure by not considering the possibilities of deep cut-offs. This *β-procedure* is the same as the α-β procedure except that no α-values are passed to the α-β procedure; instead, the lower value α is always set to $-\infty$ before exploring the successors of a node. Knuth and Moore have shown that the branching factor of this procedure satisfies

$$\mathcal{R}_{\beta}(n) = \Theta(n/\ln n).$$

Note that, since the β-procedure always explores more nodes at any depth in a tree than the full α-β procedure does in the same tree, $\mathcal{R}_{\beta}(n)$ provides us with an upper bound for $\mathcal{R}_{\alpha\text{-}\beta}(n)$.

5.2 - Bounds on the branching factor of the α-β procedure

In this section we will derive some lower and upper bounds on the branching factor of the α-β pruning algorithm. In particular, since the lower bound we derive grows with n

as $n/\ln n$, we will be able to conclude, using the result on the branching factor of the α-β procedure *without* deep cut-offs established by Knuth and Moore in [35], that the branching factor of the α-β procedure is $\Theta(n/\ln n)$.

We introduced in Section 4.1 the sequence of functions f_i, $i = 0, 1, \ldots$, from $[0, 1]$ to itself, and we observed that all functions f_i share the two fixed points 0 and 1 (independent of n). Another common fixed point, which depends on n, was introduced in Section 3.3.

Lemma 5.1:

For a given n, all functions f_i, for $i = 0, 1, \ldots$, share the common fixed point $\xi_n \in (0, 1)$, the unique positive root of the equation

$$x^n + x - 1 = 0 .$$

Proof:

For clarity, we will drop the index n from ξ_n in the following.

Since $f_0(x) = x$, ξ is certainly a fixed point of f_0; assume, for induction, that $f_{i-1}(\xi) = \xi$, then from the definition of f_i we have

$$f_i(\xi) = 1 - \{1 - [f_{i-1}(\xi)]^n\}^n = 1 - (1-\xi^n)^n = 1 - \xi^n = \xi ,$$

which shows that ξ is a fixed point common to all functions f_i, $i = 0, 1, \ldots$. ■

Since ξ_n is a fixed point common to all functions f_i, $i = 0, 1, \ldots$, it is easy to evaluate at this point the functions r_i and s_i defined in Section 4.1. For $i = 1, 2, \ldots$, we deduce that:

$$r_i(\xi_n) = s_i(\xi_n) = \xi_n/(1 - \xi_n) . \tag{5.1}$$

In particular, it follows from Lemma 3.5 that, for large n:

$$r_i(\xi_n) = s_i(\xi_n) \sim n/\ln n . \tag{5.2}$$

Equations (5.1) and (5.2) will be useful to obtain the desired bounds in the remainder of the section.

The geometric representation of equation (4.6), given in Figure 4.1, makes it easy to derive bounds on the quantity $N_{n,d}$. They are stated in the following.

Theorem 5.1:

The branching factor of the α-β pruning algorithm in the search of a rug tree of degree n satisfies:

$$n/\ln n \sim \xi_n/(1-\xi_n) \leq \mathcal{R}_{\alpha\text{-}\beta}(n) \leq \sqrt{n\xi_n/(1-\xi_n)} \sim n/\sqrt{\ln n}\,, \qquad (5.3)$$

for $n = 2, 3, \ldots$.

Proof:

Since, when t increases in $[0, 1]$, $R_d(t)$ increases while $S_d(t)$ decreases, it follows directly that for any α in $[0, 1]$ we have the following inequalities:

$$R_d(\alpha).S_d(\alpha) < N_{n,d} < R_d(\alpha).S_d(0) + [R_d(1) - R_d(\alpha)].S_d(\alpha)\,. \qquad (5.4)$$

If we choose $\alpha = \xi_n$, we have $R_d(\alpha) = [\xi_n/(1-\xi_n)]^{\lceil d/2 \rceil}$ and $S_d(\alpha) = [\xi_n/(1-\xi_n)]^{\lfloor d/2 \rfloor}$. Since $R_d(1) = n^{\lceil d/2 \rceil}$ and $S_d(0) = n^{\lfloor d/2 \rfloor}$, inequality (5.3) follows immediately from inequality (5.4) and the results of Lemma 3.5. ■

As an immediate consequence, we obtain the following.

Theorem 5.2:

The branching factor of the α-β pruning algorithm in the search of a rug tree of degree n satisfies, for large n:

$$\mathcal{R}_{\alpha\text{-}\beta}(n) = \Theta(n/\ln n)\,.$$

Proof:

The result comes directly from the lower bound $\xi_n/(1-\xi_n) \sim n/\ln n$ of Theorem 5.1, and from the upper bound $\mathcal{R}_{\beta}(n)$ obtained for the α-β procedure *without* deep cut-offs, which Knuth and Moore have shown to be $\Theta(n/\ln n)$. ■

This results confirms, as was suggested by Knuth and Moore [35, p. 310], that deep cut-offs have only a second order effect on the behavior of the α-β pruning algorithm. On the other hand, it shows that the formula proposed by Fuller, Gaschnig and Gillogly in [23] and mentioned in Section 5.1, if it constitutes a reasonable approximation for small values of n and d (the range of values they considered is $n + d \leq 12$), is certainly not adequate for large values.

We note that the bounds of Theorem 5.1 were obtained without difficulty by conveniently choosing just one point, ξ_n, on the curve $(\mathcal{L})$ since it was easy to evaluate both $R_d(\xi_n)$ and $S_d(\xi_n)$. In the next section, using a different approach, we will derive a tighter upper bound for $N_{n,d}$, and hence for $\mathcal{R}_{\alpha\text{-}\beta}(n)$.

5.3 - Improved upper bound

Since, for $d = 1, 2, \ldots$, $N_{n,d} \leq N_{n,d+1} \leq nN_{n,d}$, then, if $(N_{n,d})^{1/d}$ tends to some limit when d tends to infinity as an *even* integer, this quantity tends to the same limit when d tends to infinity as an *odd* integer. Therefore, without loss of generality, we will only consider, in this section, the case when d is an even integer. Let $d = 2h$.

For x in $[0, 1]$ and for $i = 1, 2, \ldots$, we define $p_i(x) = r_i(x)s_i(x)$.

Lemma 5.2:

All functions p_i, for $i = 1, 2, \ldots$, have the same absolute maximum, M_n, in the interval $[0, 1]$.

Proof:

From the definitions of $r_i(x)$ and $s_i(x)$ we have for $i = 1, 2, \ldots$:

$$r_i(x) = r_1[f_{i-1}(x)],$$

and

$$s_i(x) = s_1[f_{i-1}(x)].$$

Therefore, for $i = 1, 2, \ldots$, we also have, from the definition of $p_i(x)$:

$$p_i(x) = p_1[f_{i-1}(x)].$$

The lemma follows by observing that, for $i = 1, 2, \ldots$, f_{i-1} is a one-to-one function from $[0, 1]$ to itself. ■

Lemma 5.2 shows that, in order to study the maximum of $p_i(x)$, when $x \in [0, 1]$, it is sufficient to study the maximum of the polynomial

$$p_1(x) = \frac{1 - x^n}{1 - x} \frac{1 - (1 - x^n)^n}{x^n}, \quad \text{for } x \in [0, 1].$$

Observe that $M_n \geq p_1(\xi_n) = [\xi_n/(1-\xi_n)]^2$, in particular, since it can be checked easily that, for $n = 2, 3, \ldots, \xi_n > \sqrt{n}/(1+\sqrt{n})$, it follows that

$$M_n > n \quad \text{for} \quad n = 2, 3, \ldots . \tag{5.5}$$

Theorem 5.3

The branching factor of the α-β pruning algorithm for a rug tree of degree n satisfies:

$$\mathcal{R}_{\alpha\text{-}\beta}(n) \leq \sqrt{M_n}\,, \tag{5.6}$$

where M_n is defined in Lemma 5.2.

Proof:

From the definition of $R_{2h}(t)$, we obtain for $h = 2, 3, \ldots$:

$$R'_{2h}(t) = R'_{2h-2}(t).r_h(t) + R_{2h-2}(t).r'_h(t)\,.$$

By multiplication by $S_{2h}(t)$ it follows that

$$R'_{2h}(t).S_{2h}(t) = R'_{2h-2}(t).S_{2h-2}(t).p_h(t) + R_{2h-2}(t).S_{2h-2}(t).r'_h(t).s_h(t)\,.$$

Since, for $t \in [0, 1]$, all factors in this equation are non-negative, we deduce, using the results of Lemma 5.2 and the fact that $s_h(t) \leq n$ when $t \in [0, 1]$, that:

$$R'_{2h}(t)S_{2h}(t) \leq M_n R'_{2h-2}(t)S_{2h-2}(t) + n\,M_n{}^{h-1}\,r'_h(t)\,.$$

Since, in addition,

$$R'_2(t)\,S_2(t) = r'_1(t)\,s_1(t) \leq n\,r'_1(t)\,,$$

it follows that for $t \in [0, 1]$ and $h = 1, 2, \ldots$:

$$R'_{2h}(t)\,S_{2h}(t) \leq n\,M_n{}^{h-1}\,[r'_1(t) + \ldots + r'_h(t)]\,. \tag{5.7}$$

Let $I_{n,d}$ be the integral defined in equation (4.6). By integrating inequality (5.7) over $[0, 1]$ we see that $I_{n,d}$ satisfies:

$$I_{n,2h} \leq n\,M_n{}^{h-1}\,[h(n-1)] = n(n-1)\,h\,M_n{}^{h-1}$$

since $r_i(0) = 1$ and $r_i(1) = n$ for $i = 1, 2, \ldots$. This shows that

$$N_{n,2h} \leq n^h + n(n-1)\,h\,M_n{}^{h-1}\,.$$

Equation (5.6) now follows directly from inequality (5.5). ■

5.4 - Numerical results

Table 5.1 summarizes the results of this section. It presents the various lower and upper bounds we have derived for the branching factor of the α-β pruning algorithm from equations (5.3) and (5.6).

	lower bound	upper bounds		
n	$\xi_n/(1-\xi_n)$	$\sqrt{M_n}$	$\sqrt{n\xi_n/(1-\xi_n)}$	from [35]
2	1.618	1.622	1.799	1.884
3	2.148	2.168	2.538	2.666
4	2.630	2.678	3.243	3.397
5	3.080	3.166	3.924	4.095
6	3.506	3.638	4.587	4.767
7	3.915	4.098	5.235	5.421
8	4.309	4.549	5.872	6.059
9	4.692	4.993	6.498	6.684
10	5.064	5.430	7.116	7.298
11	5.427	5.862	7.726	7.902
12	5.782	6.290	8.330	8.498
13	6.130	6.713	8.927	9.086
14	6.473	7.133	9.519	9.668
15	6.809	7.549	10.107	10.243
16	7.141	7.963	10.689	10.813
17	7.468	8.373	11.268	11.378
18	7.791	8.782	11.842	11.938
19	8.110	9.188	12.413	12.494
20	8.425	9.591	12.980	13.045
21	8.736	9.993	13.545	13.593
22	9.045	10.393	14.106	14.137
23	9.350	10.791	14.665	14.678
24	9.653	11.188	15.221	15.215
25	9.952	11.583	15.774	15.748
26	10.250	11.976	16.325	16.265
27	10.545	12.369	16.873	16.778
28	10.838	12.759	17.420	17.288
29	11.128	13.149	17.964	17.796
30	11.416	13.537	18.507	18.300
31	11.703	13.924	19.047	18.802
32	11.987	14.310	19.586	

Table 5.1 - Bounds on the branching factor of the α-β pruning algorithm

Although we have not been able to give an estimate for the asymptotic growth of $\sqrt{M_n}$, we can easily derive an upper bound for this quantity by studying rug trees of depth 2 since:

$$M_n \leq N_{n,2} \leq 2n\xi_n/(1-\xi_n) - [\xi_n/(1-\xi_n)]^2 \sim 2n^2/\ln n ,$$

which shows that $\sqrt{M_n} \leq O(n/\sqrt{\ln n})$. The numerical results of Table 5.1 indicate that $\sqrt{M_n}$ is a much better upper bound for $\hat{\mathcal{R}}_{\alpha\text{-}\beta}(n)$ than $\sqrt{n\xi_n/(1-\xi_n)}$ for the range of values we have considered.

Part 2: A parallel implementation of the algorithm

6 - A parallel α-β pruning algorithm

When several processes are available a solution that comes naturally to mind for implementing the α-β pruning algorithm is to have each process explore in parallel a different subtree of the entire game tree. Each subtree would be explored using the α-β procedure to back-up its value to its root, say some node P, then the value should be reported to the father of node P in order to decide if the remaining brothers of node P can be pruned.

A possible implementation for this solution is to have the parallel algorithm organized around a *static decomposition* of the game tree, for example, by generating first all nodes at, say, depth *1* or depth *2* before starting all processes in parallel. As is shown in [37], however, static decomposition is not well adapted for execution on an asynchronous multiprocessor; this is especially true when processes have different speeds and the various subtasks have different sizes.

A *dynamic decomposition* of the game tree, on the other hand, is better suited for the processes to adjust their loads according to their own speeds. We immediately observe, however, that a dynamic implementation will require a global data structure for the processes to communicate among themselves. Since this data structure has to be updated by more than one process in parallel, synchronization will almost necessarily be required to preserve the validity of the structure at any time; in consequence, this will create a large (and unwanted) overhead.

Most important is that, by exploring in parallel and independently different subtrees of the game tree, we loose the power of the α-β pruning algorithm. By looking back at the original algorithm, we observe that it's efficiency is mainly achieved by the fact that, at any point during the search, the decision of pruning branches is based upon all the information previously acquired during the search. Obviously, when different subtrees are explored independently in parallel rather than sequentially, less information is available to each process, and, consequently, in the overall more nodes have to be explored. As will be seen, the parallel algorithm we propose below for the α-β pruning does not suffer from the loss of information communicated between the various processes.

6.1 - A parallel implementation for the α-β pruning algorithm

While proving the correctness of the ALPHABETA procedure, Knuth and Moore [35] have established equations (2.2), (2.3) and (2.4) mentioned in Section 2. We now reinterpret these equations. Let V = ALPHABETA(P,α,β), and let V_0 = MINIMAX(P). It follows directly from equations (2.2), (2.3) and (2.4) that when $\alpha < \beta$:

$$\text{if } V \le \alpha \quad \text{then } V_0 \le \alpha, \tag{6.1}$$

$$\text{if } \alpha < V < \beta \quad \text{then } V_0 = V, \tag{6.2}$$

$$\text{if } V \ge \beta \quad \text{then } V_0 \ge \beta. \tag{6.3}$$

The value V_0 (and the path in the game tree associated with that value) is the solution we are seeking when the node P is the root of the game tree. Equations (6.1) to (6.3) suggest that the problem of finding the solution V_0 can be viewed as the problem of locating the root of a monotonic function over some interval using only asynchronous parallel evaluation of the function. (This root finding problem has been studied by Hyafil and Kung, see [37] and [44].) Several differences are, however, immediately noticeable. In the root finding problem we are only looking for an approximation to the root and each evaluation of the function takes place at a single point. In the game tree searching problem, on the other hand, we are interested in the exact solution and each intermediate search, or *partial search*, executed through the call ALPHABETA(P,α,β), examines an open

interval: (α, β). Equation (6.2) shows that, provided the exact value lies in this open interval, the call returns the exact solution, and this terminates the entire search. The following program gives a parallel implementation of the α-β pruning algorithm based on this decomposition.

Program A:

```
global integer GALPHA, GBETA;

Initialization:
    begin
    GALPHA := -∞; GBETA := +∞;
    start processes P_1, ..., P_k
    end

Process P_j:
    begin
    integer A_j, B_j, V_j;
    {(A_j, B_j) := SELECTNEWINTERVAL};
    while A_j < B_j do
        begin
        V_j := AB(Root,A_j,B_j,true);
        if V_j ≤ A_j then
            begin
            {GBETA := min(GBETA,A_j+1);                          (6.4)
            (A_j, B_j) := SELECTNEWINTERVAL}
            end
        else
        if V_j ≥ B_j then
            begin
            {GALPHA := max(GALPHA,B_j-1);                        (6.5)
            (A_j, B_j) := SELECTNEWINTERVAL}
            end
        else
            begin
            {GALPHA := GBETA := V_j};                            (6.6)
            return the solution: V_j;
            terminate
            end
        end;
    terminate
    end
```

The two global variables *GALPHA* and *GBETA* define the current *open* interval known to contain the solution V_0. (When this solution is found, however, both *GALPHA* and *GBETA* are set to V_0.) The interval *(GALPHA, GBETA)* is initialized to $(-\infty, +\infty)$ and is updated each time a process finishes a partial search over the game tree. The procedure

SELECTNEWINTERVAL uses, without modifying them, the variables *GALPHA* and *GBETA* (as well as $A_1, ..., A_k$ and $B_1, ..., B_k$) to determine a new interval (A_j, B_j) over which process P_j will proceed to a new partial search. This procedure is critical to the efficiency of Program A and will be discussed in more detail in Section 7. For the time being, we will only assume that it meets the following specifications. Given the variables *GALPHA* and *GBETA* (and the variables $A_1, ..., A_k$ and $B_1, ..., B_k$), let (A, B) := SELECTNEWINTERVAL:

(a) $A = B$ if $GALPHA = GBETA$;

(b) $A < B$ otherwise.

As we are only dealing with integers, condition (b) is equivalent to the condition $A \leq B-1$.

Since the two global variables *GALPHA* and *GBETA* are updated in parallel by several processes, their use is restricted within critical section (indicated in Program A with curly brackets); the use of the procedure SELECTNEWINTERVAL also occurs within critical section.

Theorem 6.1:

At any time in the execution of Program A (outside a critical section), the solution V_0 satisfies either one of the following two conditions:

$$GALPHA < V_0 < GBETA\,, \tag{6.7}$$

$$GALPHA = V_0 = GBETA\,. \tag{6.8}$$

Proof:

After initialization, at time t_0, the variables *GALPHA* and *GBETA* are only modified (in a critical section) through one of the instructions (6.4), (6.5) or (6.6) executed at the time instants $t_1, t_2, ..., t_i, ...$ (with $t_i \geq t_{i-1}$ for $i \geq 2$). After t_0, $GALPHA = -\infty$ and $GBETA = +\infty$, therefore condition (6.7) is certainly satisfied. Assume that after t_{i-1}, for $i \geq 1$, condition (6.7) or (6.8) is satisfied. If instruction (6.6) is executed at time t_i by process P_j, it follows from equation (6.2) that $V_j = V_0$, therefore condition (6.8) is satisfied after t_i. If instruction (6.4) is executed at time t_i by process P_j, it follows from equation (6.1) that $V_0 \leq A_j$, or eqivalently $V_0 < A_j+1$ (recall that both V_0 and A_j are

integers); if, prior to t_i, condition (6.7) were satisfied, then $V_0 < GBETA$, which shows that $V_0 < \min(GBETA, A_j+1)$ and condition (6.7) remains satisfied after t_i; if, prior to t_i, condition (6.8) were satisfied, then $GBETA = V_0 < A_j+1$, which shows that $\min(GBETA, A_j+1) = GBETA$ and condition (6.7) remains satisfied. The same holds when instruction (6.5) is executed. ■

Theorem 6.1, along with the specifications (a) and (b) of the procedure SELECTNEWINTERVAL, proves the correctness of Program A in the sense that if the program terminates it generates the correct solution.

Proving the termination of Program A, on the other hand, requires additional specification of the procedure SELECTNEWINTERVAL. Observe, for example, that, if we always have $A_j = B_j-1$, the *open* interval (A_j, B_j) does not contain any integer (A_j and B_j are integers themselves) and no solution can ever be found. If, however, we replace condition (b) above by:

(b') $A \leq B-2$ otherwise,

it can be shown easily that the length of the interval $(GALPHA, GBETA)$ decreases at least by 1 each time a process completes a partial search. Since in a practical implementation the interval $(-\infty, +\infty)$ is actually a finite interval in which we know that the solution V_0 is to be found, we are guaranteed of the termination of Program A under condition (b').

6.2 - Some improvements on Program A

A feature of the parallel implementation presented in Section 6.1 is that intercommunication between processes is reduced to a minimum, and confined to the selection of a new interval over which a partial search is to take place next. As a consequence, once a process has initiated a partial search, it runs until completion oblivious of the results of the other processes. This can obviously be overly wasteful since the interval searched by a process might be ruled out by some other process very soon after the beginning of the search.

This shortcoming can be eliminated in several ways. First, a process completing a partial search could check all other processes, causing them, if necessary, either to abort their searches or to readjust their intervals. This solution, however, requires a lot of book-keeping and becomes unpractical when a large number of processes are cooperating.

Another solution is to have each process modify its own interval by regularly checking possible changes of the variables *GALPHA* and *GBETA* during the search. Let $A' \le A < B \le B'$, and consider the two calls:

ALPHABETA(Root,A',B') and ALPHABETA(Root,A,B) .

It is easy to check, by induction, that if node P is explored by the second call, through ALPHABETA(P,α,β), node P is also explored by the first call, through ALPHABETA(P,α',β'). Moreover, the bounds α, β, α' and β' satisfy:

$$\alpha = \max\{\alpha', A\}\ , \qquad \beta = \min\{\beta', B\}\ , \qquad \text{if } P \text{ is at } \textit{even} \text{ depth,} \tag{6.9}$$

$$\alpha = \max\{\alpha', -B\}\ , \qquad \beta = \min\{\beta', -A\}\ , \qquad \text{if } P \text{ is at } \textit{odd} \text{ depth.} \tag{6.10}$$

The procedure AB, below, is a modification of the procedure ALPHABETA, in which the bounds alpha and beta are regularly updated according to the relations (6.9) and (6.10) to take into account the changes of the two variables *GALPHA* and *GBETA*.

```
integer procedure AB(position P, integer alpha, integer beta, boolean even):
    begin
    determine the successor positions: P_1, ..., P_n;
    if n = 0 then
        AB := f(P)
    else
        begin
        for j := 1 step 1 until n do
            begin
            t := -AB(P_j,-beta,-alpha,not even);
            if t > alpha then alpha := t;
            if even then
                (alpha := max{alpha,GALPHA}; beta := min{beta,GBETA})
            else
                (alpha := max{alpha,-GBETA}; beta := min{beta,-GALPHA});
            if alpha ≥ beta then goto done
            end;
done:   AB := alpha
        end
    end
```

A modified Alpha-Beta procedure

Relations similar to the relations (6.1), (6.2) and (6.3) hold for the procedure AB as well. Consider the call:

$$V := \mathrm{AB}(P,\alpha,\beta,\underline{\mathrm{true}}) , \tag{6.11}$$

and as before define $V_0 := \mathrm{MINIMAX}(P)$. Also, let A and B denote the values of the two variables *GALPHA* and *GBETA* when returning from the call (6.11) (i. e., as of the last time they are used during the execution of the call). For A' and B' satisfying $A' \geq A$ and $B' \leq B$, define $\alpha' = \max\{\alpha,A'\}$ and $\beta' = \min\{\beta,B'\}$. We have the following.

Theorem 6.2

With the above notations, provided that:

$$A' \leq V_0 \leq B' \quad \text{and} \quad \alpha' < \beta' ,$$

we have:

$$\text{if } V \leq \alpha' \quad \text{then } V_0 \leq \alpha' ,$$

$$\text{if } \alpha' < V < \beta' \quad \text{then } V_0 = V ,$$

$$\text{if } V \geq \beta' \quad \text{then } V_0 \geq \beta' .$$

Proof:

The proof follows easily (by induction on the depth of node P) from the relations (6.1), (6.2) and (6.3) and the relations (6.9) and (6.10). ■

Program B, below, directly implements the relations stated in this theorem. Since the analog of Theorem 6.1 can be proved for Program B as well, its *correctness* is a direct consequence of Theorem 6.2.

Program B:

```
global integer GALPHA, GBETA;

Initialization:
    begin
    GALPHA := -∞; GBETA := +∞;
    start processes P_1, ..., P_k
    end
```

```
Process P_j;
    begin
    integer A_j, B_j, V_j;
    {(A_j, B_j) := SELECTNEWINTERVAL};
    while A_j < B_j do
        begin
        V_j := AB(Root,A_j,B_j,true);
        A_j := max(A_j,GALPHA); B_j := min(B_j,GBETA);
        if A_j < B_j then
            begin
            if V_j ≤ A_j then
                begin
                {GBETA := min(GBETA,A_j+1);
                (A_j, B_j) := SELECTNEWINTERVAL}
                end
            else
            if V_j ≥ B_j then
                begin
                {GALPHA := max(GALPHA,B_j-1);
                (A_j, B_j) := SELECTNEWINTERVAL}
                end
            else
                begin
                {GALPHA := GBETA := V_j};
                return the solution: V_j;
                terminate
                end
            end
        else
            {(A_j, B_j) := SELECTNEWINTERVAL}
        end;
    terminate
    end
```

Procedures ALPHABETA and AB implement two extreme alternatives in which the bounds alpha and beta are never updated and in which they are updated each time they are used. A more efficient implementation would be to update alpha and beta only when changes have been made on the variables *GALPHA* and *GBETA*. This can be achieved very easily by introducing a global counter incremented by *1* inside the critical section *after* each of the instuctions of Program B modifying *GALPHA* and/or *GBETA*, and by introducing a counter local to each process to check if the latest modifications of *GALPHA* and *GBETA* have been taken into account. Since the counters can only increase, no additional critical section is required. We will not present the implementation details, but the point, here, is mainly to show that it is possible to implement (at a very low extra cost) each process so that it is continuing a partial search only if the result of the search can produce the

solution or, at at least, a reduction of the interval in which the solution can lie. In particular, we note that, in Program B, process P_j will terminate its search as soon as, for example, $GALPHA \geq B_j$ or $GBETA \leq A_j$, either condition ruling out the original interval (A_j, B_j). This property will be taken into account in the analysis presented in Section 7.

7 - Analysis of the parallel α-β pruning algorithm

We will proceed in this section to the analysis of the parallel algorithm described in the preceding section. Since the algorithm is organized around parallel executions of *partial searches*, it is the first thing we want to analyze. Most of this analysis differs very slightly from the analysis developed in Sections 2 and 3, and we will only present in Section 7.1 and 7.2 the main results leading to the evaluation of a partial search. The overall evaluation of the algorithm depends upon the procedure SELECTNEWINTERVAL and will be derived in Section 7.3.

7.1 - Condition for a node to be examined under a partial search

As in Section 2, let $J = j_1 \cdot \ldots \cdot j_d$ denote a node at depth d in a game tree and, for $0 \leq i \leq d-1$, let $J_i = j_1 \cdot \ldots \cdot j_{d-i}$. The notations for $v(J)$ and $c(J)$ remaining the same, we now define:

$$\alpha'(J) = \max\{ c(J_{d-i}) \mid i \text{ is } odd,\ 1 \leq i \leq d \},$$

$$\beta'(J) = \max\{ c(J_{d-i}) \mid i \text{ is } even,\ 1 \leq i \leq d \}.$$

Given the two bounds a and b, we also define:

$$A'(J) = \max\{ a, \alpha'(J) \},$$

$$B'(J) = \max\{ -b, \beta'(J) \}.$$

The analog of Theorem 2.1 for a partial search can now be stated in the following.

Theorem 7.1

Assume that the root of a game tree is explored through the call

ALPHABETA(Root,α,β)

by some process executing the parallel procedure of Section 6.1. Then, with the above notations and provided that $a < b$, an arbitrary node $\mathcal{J}$ of the game tree will be subsequently explored if and only if:

$$A'(\mathcal{J}) + B'(\mathcal{J}) < 0 . \qquad (7.1)$$

Proof:

The proof is immediate by induction. ■

Observe that, when the procedure AB of Section 6.2 is used instead of the procedure ALPHABETA, condition (7.1) only remains a necessary condition for node $\mathcal{J}$ to be explored through the call AB(Root,α,β). It is no longer a sufficient condition since, by updating the bounds α and β during the execution of the procedure AB, additional pruning might occur.

In the following evaluation of a partial search we will assume that the process executes the procedure ALPHABETA, and we will utilize condition (7.1) to characterize the fact that node $\mathcal{J}$ is explored.

7.2 - Average number of nodes explored under a partial search

As before, we will consider a rug tree of degree n and depth d, and we will assume first that the bottom values are independent identically distributed random variables distributed according to some *discrete* probability distribution $\{p_0(k)\}_{-m \le k \le m}$, where $p_0(k)$ is the probability that a bottom value be assigned the value $x_k = k/m$, for $-m \le k \le m$.

Given two bounds α and β, we define k_1 and k_2 by:

$$\alpha = x_{k_1} , \quad \beta = x_{k_2} .$$

Since the values α and β could be unbounded, it is convenient to define $x_{-m-1} = -\infty$ and $x_{m+1} = +\infty$. Throughout we will only consider the partial search corresponding to the call ALPHABETA(Root,α,β), and we will assume that $\alpha < \beta$, which can equivalently be expressed as $-m-1 \le k_1 < k_2 \le m+1$.

Using arguments identical to those of Section 3.1, the probability distributions for the quantities $A'(\mathcal{J})$ and $B'(\mathcal{J})$ can be obtained immediately as a function of the quantities $\varphi_i(k)$, for $0 \le i \le d$ and $-m-1 \le k \le m$. Then the probability $\pi(\mathcal{J})$ that some node $\mathcal{J}$ of the game tree be explored under a partial search can be derived from these results using the characterization given by condition (7.1). As with Theorem 3.1, the following theorem results directly from the expression for $\pi(\mathcal{J})$. In order to present a uniform result (independent of the parity of d) in this theorem, we depart slightly from the notations of Section 3.1, and the products denoted by $\prod_e$ and $\prod_o$ are now extended over all *even* and *odd* integers i, respectively, in the range $1 \le i \le d$.

Theorem 7.2:

The average number, $N_{n,d}(m,\alpha,\beta)$, of bottom positions examined under a partial search is given by:

$$N_{n,d}(m,\alpha,\beta) = \prod_o \rho_{d-i}(k_1) \times \prod_e \sigma_{d-i}(k_1) + \sum_{k_1+1 \le k \le k_2-1} [\prod_o \rho_{d-i}(k) - \prod_o \rho_{d-i}(k-1)] \times \prod_e \sigma_{d-i}(k) \,. \qquad (7.2)$$

Proof:

As with the proof of Theorem 3.1, the result follows directly by summing the probabilities $\pi(\mathcal{J})$ over all terminal positions $\mathcal{J}$. ■

When assuming that all bottom values are distributed according to some continuous probability distribution (or, similarly, are all distinct), again we can obtain, as in Section 4, the average number of bottom positions examined under a partial search by considering the limit of $N_{n,d}(m,\alpha,\beta)$ in equation (7.2). At this point it is convenient to consider the cumulative distribution for the value v(Root) with respect to the two points α and β. Namely, given the probability distribution $\{p_d(k)\}_{-m \le k \le m}$ (or equivalently $\{p_0(k)\}_{-m \le k \le m}$) and given $\alpha = x_{k_1}$ and $\beta = x_{k_2}$, we introduce:

$$a_m = p_d(-m) + \ldots + p_d(k_1) = 1 - \varphi_d(-k_1-1) \,,$$

$$b_m = p_d(-m) + \ldots + p_d(k_2) = 1 - \varphi_d(-k_2-1) \,.$$

If, in general, we let:

$$t = p_d(-m) + \ldots + p_d(k) = 1 - \varphi_d(-k-1) \,,$$

and define in an obvious way the functions P and Q on $[0, 1]$ by the correspondence:

$$P(t) = \prod_o p_{d-i}(k) ,$$

$$Q(t) = \prod_e q_{d-i}(k) ,$$

we can state the limit of equation (7.2) in the following theorem.

Theorem 7.3:

Provided that:

$$\lim_{m\to\infty} \max\{ p_0(k) \mid -m \le k \le m \} = 0$$

and that:

$$\lim_{m\to\infty} a_m = a , \quad \lim_{m\to\infty} b_m = b ,$$

the limit of $N_{n,d}(m,\alpha,\beta)$, when $m \to \infty$, is given by:

$$N_{n,d}(a,b) = P(a).Q(a) + \int_a^b P'(t).Q(t).dt . \qquad (7.3)$$

Both Theorem 7.2 and Theorem 7.3 provide us with a cost of executing a partial search, measured by the number of terminal positions examined during the search, when the bottom values are distributed according to either a discrete or a continuous probability distribution.

In Figure 7.1, we have plotted, for $x \in [0, 1]$, the two quantities

$$G(x) = P(x).Q(x) ,$$

$$H(x) = \int_0^x P'(t).Q(t).dt .$$

We deduce from equation (7.3) that $N_{n,d}(a,b)$ can be expressed directly from these two quantities as:

$$N_{n,d}(a,b) = G(a) + H(b) - H(a) ,$$

with an immediate interpretation in Figure 7.1. If we consider the case when the bottom values are distributed according to a discrete probability distribution, then $N_{n,d}(m,\alpha,\beta)$, as given by equation (7.2), can be expressed similarly as a function of a_m and b_m. The functions G and H are, in this case, simply replaced by step functions, which coincide with the continuous functions G and H at the points $t_k = 1 - \varphi_d(-k-1)$, for $-m \le k \le m$.

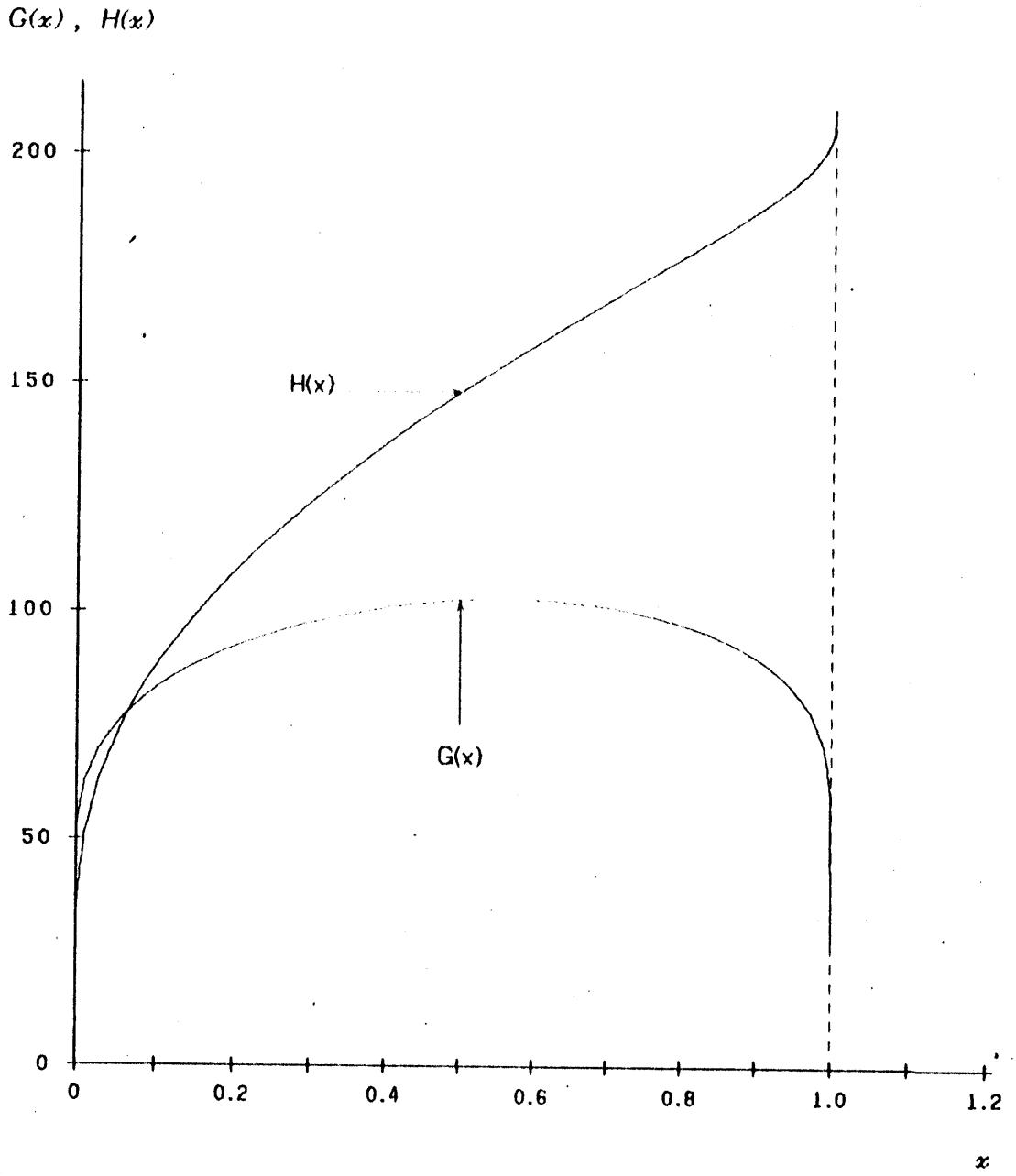


Figure 7.1 - An interpretation for $N_{n,d}(a,b)$

7.3 - The analysis of the parallel α-β pruning algorithm

The results of Section 7.2 show that the cost of executing the partial search corresponding to the call

ALPHABETA(Root,α,β)

can be expressed by:

$$c(a,b) = G(a) + [H(b) - H(a)] ,$$

with

$$a = \text{Proba}\{ V \leq \alpha \} , \quad b = \text{Proba}\{ V \leq \beta \} ,$$

where V is the random variable representing the value backed-up to the root of the game tree (by the MINIMAX procedure). Given the probability distribution for the random variable V, we have a one-to-one correspondence between intervals (α, β) of $(-\infty, +\infty)$ and intervals (a, b) of $(0, 1)$. Using this correspondence, we will only talk in the following about partial searches over intervals of $(0, 1)$.

Although the two functions G and H are readily computed numerically, they do not lend themselves very easily to analysis and, in the remainder of the section, we will consider an approximation suggested by Figure 7.1. We notice in the example depicted in this figure that $G(x)$ remains nearly constant when x varies in the interval $[0, 1]$ and that $H(x)$ varies almost linearly on the same interval. While the numerical results presented in Figure 7.1 correspond to a partial search of a rug tree of degree $n = 3$ and depth $d = 6$, numerical results obtained with other values of n and d actually show that the approximation of G by a constant and of H by a linear function is even better for large values of n and d. This is especially true in an open interval contained in $[0, 1]$. In consequence, we will assume in the following that the cost of executing a partial search over any interval (a, b) of $[0, 1]$ is exactly given by:

$$c(a,b) = p + q[b - a] , \tag{7.4}$$

where p and q only depend on the rug tree itself (i. e., on n and d). Numerical results, not presented here, have been run for $n = 3, 4, 8, 16$ and 32 and for $2 \leq d \leq 8$, it turns out that, if, obviously, p and q are very dependent on n and d, the ratio $\lambda = p/q$ does not show a large variation and lies typically in the range $0.2 \leq \lambda \leq 0.4$.

Without loss of generality, we will normalize the cost $c(a,b)$ of equation (7.4) by assuming that $q = 1$ (hence $p = \lambda$) and we will consider throughout that:

$$c(a,b) = \lambda + b - a ,$$

or, equivalently, with $b = a + h$, that:

$$c(a,a+h) = \lambda + h . \tag{7.5}$$

This cost will also be taken, in the following section, as the time for a process to execute a partial search over the interval $(a, b) = (a, a+h)$.

7.3.1 - An analysis of the parallel implementation: Optimal decomposition

Given the cost of a partial search through equation (7.5), we will determine in this section the optimal decomposition of the interval $[0, 1]$ and, with this result, the optimal procedure SELECTNEWINTERVAL, introduced in Section 6.1 for $k \geq 2$, processes can be defined.

As an example, we first examine the special case when the interval $[0, 1]$ is split into k subintervals $I_1, ..., I_k$ searched in parallel by processes $P_1, ..., P_k$, respectively. Let s_i be the size of I_i, for $i = 1, ..., k$, with $s_1 + ... + s_k = 1$. Under this decomposition, process P_i will find the solution, with probability s_i, after a cost $\lambda + s_i$. Therefore, the average cost (or time) to find the solution is, in this case, simply given by:

$$\begin{aligned} t &= s_1.(\lambda + s_1) + ... + s_k.(\lambda + s_k) \\ &= \lambda + s_1^2 + ... + s_k^2 , \end{aligned}$$

for which the minimum, T_0, is achieved when $s_i = \frac{1}{k}$, for $i = 1, ..., k$ (recall that $s_1 + ... + s_k = 1$). This yields:

$$T_0 = \lambda + \frac{1}{k} .$$

The decomposition of the interval $[0, 1]$ presented in this example is the simplest one, and it does not allow any feedback between the processes since the k partial searches cover the whole interval $[0, 1]$. The example confirms, however, the obvious fact that, in order to achieve the minimum cost, the k subintervals searched by the k processes should be of equal length.

In order to introduce some feedback between the processes, we now consider a further decomposition of the interval $[0, 1]$ illustrated in the diagram of Figure 7.2 in the case of two processes.

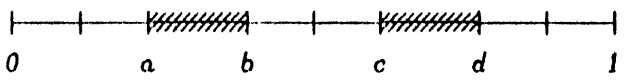


Figure 7.2 - A decomposition of $[0,1]$

The two processes P_1 and P_2 start exploring in parallel the two subintervals $[a, b]$ and $[c, d]$, respectively. If either process finds the solution at the completion of this first search, with probability $(b-a)$ or $(d-c)$, the execution terminates with a cost of either $(\lambda+b-a)$ or $(\lambda+d-c)$. Otherwise, consider that process P_1 finishes first. If it finds out that the solution lies in the interval $[0, a]$, we know that, with the implementation proposed in Section 6.2, process P_2 will terminate its search immediately after and, therefore, both processes can start simultaneously new partial searches within the interval $[0, a]$. If, on the other hand, process P_1 finds out that the solution lies in the interval $[b, 1]$, it will start arbitrarily a partial search over an interval within $[b, c]$ or $[d, 1]$ while waiting for process P_2 to complete its initial partial search and, possibly, will readjust its search as soon as process P_2 finishes. If we assume that both intervals $[a, b]$ and $[c, d]$ are of equal length, both processes will finish their initial searches roughly at the same time. We will neglect in the following the delay involved in making the decision as to which subinterval actually contains the solution, and we will assume that, if the solution has not yet been found, the processes restart a new partial search simultaneously.

According to this decomposition, k subintervals are initially searched by the k processes and, if the solution is not found during this first trial, it is known to lie in 1 of $k+1$ subintervals depending upon the outcomes of the first partial searches. Thus k subintervals will be searched during the second trial out of a total of $k(k+1)$ possible subintervals. In general, if not successful after the i-th trial, the k processes will start simultaneously k new partial searches over $a_i = k(k+1)^i$ possible subintervals during the $(i+1)$-st trial.

Let $h_0 = 1$, and, for $i = 1, 2, \ldots$, let h_i be the total length of the interval $[0, 1]$ that still could be explored after the i-th trial. Then, for $i = 1, 2, \ldots$, $h_{i-1} - h_i$ measures the

total length of all a_{i-1} subintervals that could be searched during the i-th trial. It also measures the probability that the solution be found at that time after a cost c_i given by:

$$c_i = [\lambda + (h_0 - h_1)/a_0] + \ldots + [\lambda + (h_{i-1} - h_i)/a_{i-1}] ,$$

assuming that the a_{i-1} subintervals that could be searched during the i-th trial have all the same length: $(h_{i-1} - h_i)/a_{i-1}$.

The total average cost, T, follows immediately. We have:

$$T = \sum_{i \geq 1} (h_{i-1} - h_i).c_i ,$$

$$T = \sum_{i \geq 1} i\lambda(h_{i-1} - h_i) + \sum_{i \geq 1} [(h_{i-1} - h_i) \sum_{1 \leq j \leq i} (h_{j-1} - h_j)/a_{j-1}] ,$$

$$T = \lambda \sum_{i \geq 0} h_i + \sum_{i \geq 0} h_i (h_i - h_{i+1})/a_i . \qquad (7.6)$$

The following theorem states the optimal decomposition $\{h_i\}_{i \geq 0}$ leading to the minimum average cost of expression (7.6). For $k \geq 2$, we will consider the following sequence of intervals (recall that $a_j = k(k+1)^j$):

$$A_0 = [1/a_0, +\infty) ,$$

$$A_j = [1/a_j, (k-1)/a_j) , \quad \text{for } j = 1, 2, \ldots ,$$

and

$$B_j = [(k-1)/a_j, 1/a_{j-1}) , \quad \text{for } j = 1, 2, \ldots .$$

Theorem 7.4:

Assume $k \geq 2$, and let $C_k(\lambda)$ denote the minimum of expression (7.6) over all possible decompositions $\{h_i\}_{i \geq 0}$.

(a) If $\lambda \in A_j$, for some $j = 0, 1, \ldots$, the minimum of expression (7.6) is achieved for:

$$h_0 = \ldots = h_j = 1 \text{ and } h_{j+1} = h_{j+2} = \ldots = 0 ,$$

yielding:

$$C_k(\lambda) = (j+1)\lambda + \frac{1}{a_j} ,$$

(b) Otherwise, if $\lambda \in B_j$, for some $j = 1, 2, \ldots$, the minimum is achieved for:

$$h_0 = \ldots = h_{j-1} = 1 , \; h_j = \frac{1}{2} a_j \left(\frac{1}{a_{j-1}} - \lambda\right) \text{ and } h_{j+1} = h_{j+2} = \ldots = 0 ,$$

yielding:

$$C_k(\lambda) = j\lambda + \frac{1}{a_{j-1}} - \frac{1}{4} a_j \left(\frac{1}{a_{j-1}} - \lambda\right)^2 .$$

Proof:

Observe first that the decomposition $\{h_i\}_{i\geq 0}$ satisfies:

$$1 = h_0 \geq h_1 \geq \ldots \geq h_{i-1} \geq h_i \geq \ldots \geq 0 .$$

Assume that $\lambda \geq 1/a_j$, for some $j \geq 0$. Given any decomposition $\{h_i\}_{i\geq 0}$, consider another decomposition $\{g_i\}_{i\geq 0}$ defined by:

$$g_i = \begin{cases} h_i & \text{if } i \leq j , \\ 0 & \text{if } i \geq j+1 , \end{cases}$$

and let T' denote the expression (7.6) where $\{h_i\}_{i\geq 0}$ is replaced by $\{g_i\}_{i\geq 0}$. We have:

$$\begin{aligned} T - T' &= \lambda \sum_{i\geq j+1} h_i + \sum_{i\geq j+1} \frac{1}{a_i} h_i (h_i - h_{i+1}) \\ &\qquad + [\lambda - \frac{1}{a_j} h_j + \frac{1}{a_{j+1}} h_{j+1} (h_{j+1} - h_{j+2})] \\ &\geq 0 + 0 + [\frac{1}{a_j} - \frac{1}{a_j} + \frac{1}{a_{j+1}} h_{j+1} (h_{j+1} - h_{j+2})] \\ &= \frac{1}{a_{j+1}} h_{j+1} (h_{j+1} - h_{j+2}) \geq 0 , \end{aligned}$$

which shows that T is minimized when $h_i = 0$ for $i \geq j+1$.

Assume now that $\lambda < (k-1)/a_j$ for some $j \geq 1$. Assume furthermore that $h_{i-1} = 1$ for some i, $1 \leq i \leq j$ (recall that $h_0 = 1$). We have:

$$\lambda < (k-1)/a_j \leq (k-1)/a_i ,$$

which shows that the derivative, t_i, of T with respect to h_i satisfies:

$$\begin{aligned} t_i &= 2 \frac{1}{a_i} h_i + \lambda - \frac{1}{a_{i-1}} h_{i-1} - \frac{1}{a_i} h_{i+1} \\ &= 2 \frac{1}{a_i} h_i + \lambda - \frac{1}{a_{i-1}} - \frac{1}{a_i} h_{i+1} \\ &< -2 \frac{1}{a_i} (1 - h_i) - \frac{1}{a_i} h_{i+1} \leq 0 \end{aligned}$$

This last inequality shows that T decreases when h_i increases from 0 to 1 and that, therefore, the minimum of T is achieved when $h_i = 1$. Since $h_0 = 1$, we have shown part (a) of the theorem.

Assume now that $\lambda \in B_j$ for some $j \geq 1$, i. e.:

$$(k-1)/a_j \leq \lambda < 1/a_{j-1} .$$

In particular, since $k \geq 2$, $\lambda \geq 1/a_j$ and $\lambda < (k-1)/a_{j-1}$. It follows from the above proof that $h_0 = \ldots = h_{j-1} = 1$ and that $h_{j+1} = h_{j+2} = \ldots = 0$. Hence, expression (7.6) becomes:

$$T = j\lambda + \frac{1}{a_{j-1}} - (\frac{1}{a_{j-1}} - \lambda) h_j + \frac{1}{a_j} h_j^2 ,$$

from which part (b) of the theorem follows directly. ■

Theorem 7.4 states as a function of λ, the initial cost for a partial search, the optimal decomposition of the interval $[0, 1]$ and the corresponding optimal average cost $C_k(\lambda)$ to find the solution using the parallel implementation with k processes. In Figure 7.3, we compare the cost $C_k(\lambda)$ with the cost $C(\lambda)$ of the original (sequential) algorithm as presented in Section 2.

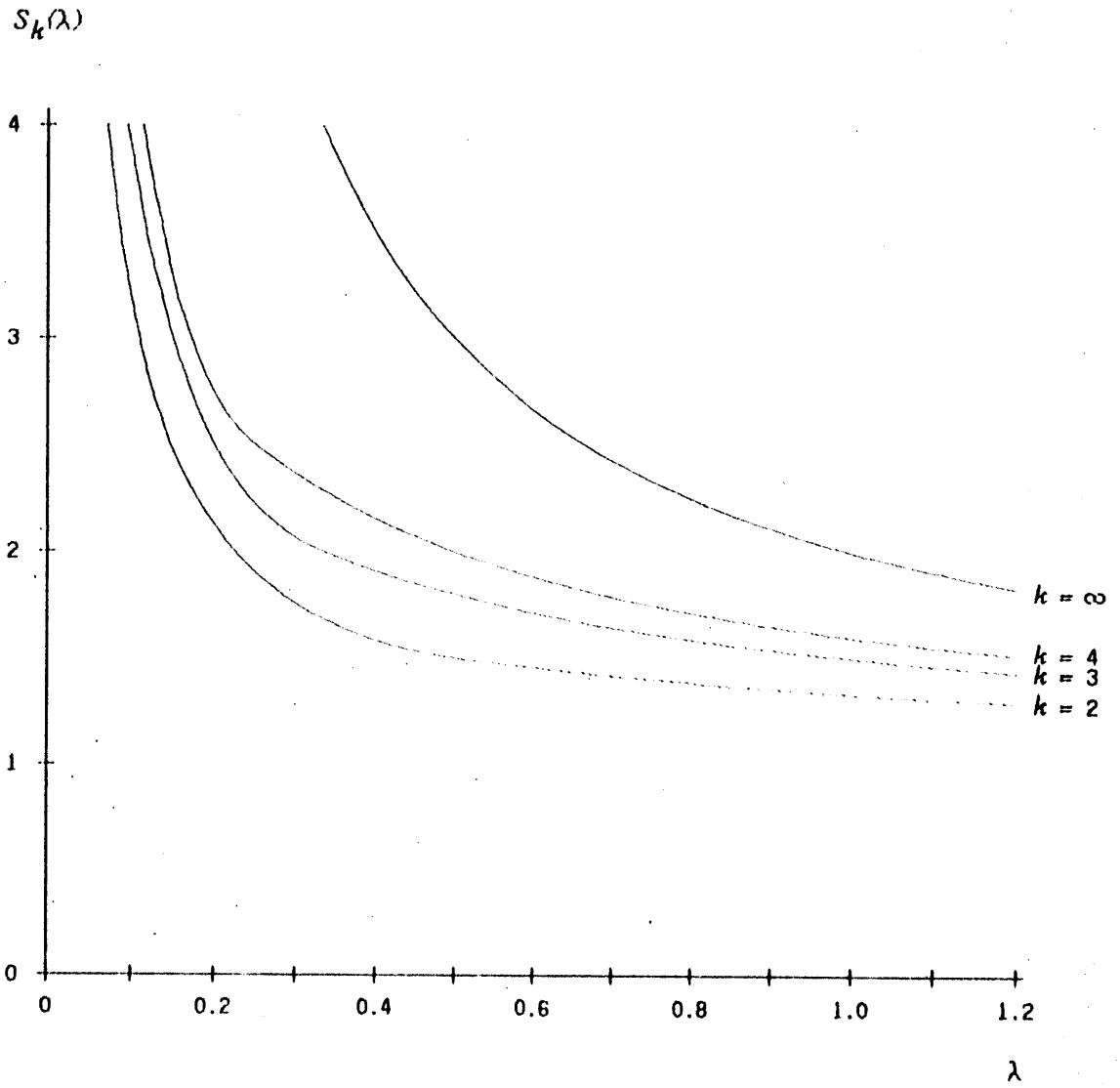


Figure 7.3 - Relative speed-up of the parallel implementation

Since in the original α-β pruning algorithm the whole interval $[0, 1]$ is searched at once, $C(\lambda)$ can be obtained directly from equation (7.5) and is given by:

$$C(\lambda) = c(0,1) = \lambda + 1 .$$

The various curves of Figure 7.3 represent the speed-up $S_k(\lambda) = C(\lambda)/C_k(\lambda)$ achieved by the parallel implementation with k processes over the original algorithm for $k = 2, 3, 4$ and for the limiting case $k = \infty$. In this latter case $1/a_0$ simply reduces to 0 and we always have $\lambda \in A_0$. It follows from Theorem 7.4 that $C_\infty(\lambda) = \lambda$ and therefore

$$S_\infty(\lambda) = \frac{\lambda + 1}{\lambda} = 1 + \frac{1}{\lambda}.$$

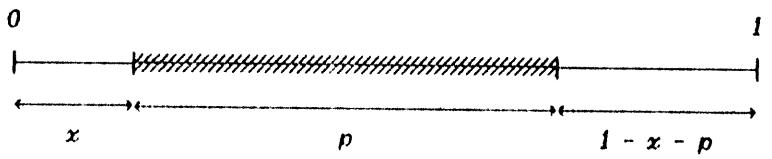
7.3.2 - Implications of the results and validity of the assumptions

Let us examine the results of the preceding section as illustrated in Figure 7.3. We noticed earlier that the initial cost of a partial search, λ, typically lies in the range $(0.2, 0.4)$. We observe from Figure 7.3 that when $k = 2$, for example, the parallel implementation can improve upon the original (sequential) α-β pruning algorithm by a factor which can be larger than 2 when λ lies in the range of practical interest. Moreover, when λ becomes small, the improvement actually becomes unbounded, as can be seen by choosing $\lambda = 1/a_j$ for which we have: $S_k(\lambda) = (a_j + 1)/(j + 2)$. An immediate consequence of the results of Section 7.3.1, therefore, is that the α-β pruning algorithm (as described in Section 2) is not optimal. The same strategy used for the parallel implementation with two or more processes is obviously also suitable to the case of only one process, and, in a similar fashion, we can deduce an optimal decomposition of the interval $[0, 1]$ for this case as well. Although the results of Theorem 7.4 are not applicable for the sequential case (only the first part of the proof is relevant when $k = 1$), simple calculus shows that when $\lambda \in (0.2, 0.4)$ an improvement between 15% and 25% can be achieved over the original algorithm, and this constitutes a substantial gain.

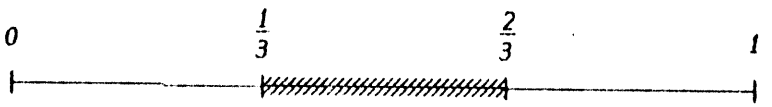
The analysis developed in Section 7.3.1 relies implicitly on the knowledge of the distribution for the value V_0 backed-up to the root of the game tree. In particular, when we state, in Theorem 7.4, the optimal decomposition of the interval $[0, 1]$ in terms of $\{h_i\}_{i \geq 0}$, we really need to know the distribution of V_0 to actually implement the procedure SELECTNEWINTERVAL according to this optimal decomposition. When nothing is known

about the distribution of V_0, the results of Theorem 7.4 stating the optimal cost $C_k(\lambda)$ can be simply reinterpreted as a lower bound on the cost achievable by an algorithm using this strategy of decomposition with partial searches.

In practice, however, although the distribution of V_0 is not known exactly, some information is actually available from the evaluation of the game tree at previous moves. In chess, for example, unless an important capture was hidden from the horizon of the search, successive evaluations of the game tree will yield closely related values, and it is common to be able to predict a priori an interval which contains the solution V_0 with some probability p, where, typically, $p = 80\%$. In the actual implementation of a chess program, this interval is examined first, and, if the solution is not found after this trial, the whole interval to its left (or to its right, depending on the outcome of the first search) is examined next. See Figure 7.4 (a).



(a) Actual decomposition



(b) Optimal decomposition ($\lambda = \frac{1}{3}$)

Figure 7.4 - Comparison of the actual and optimal decompositions of $[0,1]$

Under these conditions, let us consider the cost of finding the solution V_0 with 1 process, and let us assume, to give an idea, that $\lambda = 1/3$. For purposes of comparison, the optimal decomposition can be shown, in this case, to be $h_0 = 1$, $h_1 = 2/3$ and $h_2 = h_3 = \ldots = 0$, see Figure 7.4 (b), yielding the minimum cost $T_0 = 10/9 \sim 1.11$, while the cost of the original algorithm is simply given by $T_1 = \lambda + 1 = 4/3 \sim 1.33$ (an increase of 20% over the optimal cost).

The cost associated with the actual decomposition is easily evaluated and is given by:

$$T = p.(\lambda + p) + x.(\lambda + p + \lambda + x) + (1 - x - p).(\lambda + p + \lambda + 1 - x - p)$$
$$= (2 - p)\lambda + p + x^2 + (1 - p - x)^2 ,$$

from which we deduce that the worst case, achieved for $x = 0$ or $x = 1 - p$, is given by:

$$T_2 = (2\lambda + 1) - (\lambda + 1)p + p^2 ,$$

corresponding to $T_2 = 1.24$ when $\lambda = 1/3$ and $p = 0.8$. Although this worst case still corresponds to an increase of 11.6% over the optimal cost, it is an improvement of 7% over the cost of the original algorithm. Yet, in view of the optimal case, one could think of improving the cost by reducing the first interval so as to have $p = 1/3$, but then this would increase the worst case, which would, in fact, correspond in this case to the cost of the original algorithm, therefore, showing no improvement. (Looking at the best case, however, we could achieve the optimal case in this way, but only with the risk of aggravating the worst case.)

The results we have developed rely on several simplifying assumptions, and we would like to conclude this section by examining their validity. While equations (7.2) and (7.3) provide us with the exact cost of a partial search over some interval (α, β) (or (α, b) equivalently), measured by the number of terminal positions examined during the search, we have used the approximation given by equation (7.5) to derive the results of Section 7.3.1. As we have mentioned, however, this approximation seems to be reasonable and more and more accurate as the game tree becomes larger, and we do not feel that this approximation leads to a large error in the analysis. In order to check on the validity of this approximation, however, we have run a series of simulations and compared the results with the results predicted by Theorem 7.4, where λ was computed numerically by using a least square approximation to the functions $G(x)$ and $H(x)$ on the interval $[0, 1]$ (see Figure 7.1). The simulation results were very consistent with the analytical results and showed an actual improvement over the original algorithm between 5% and 10% better than the improvement predicted by the theory.

The simulation was also aimed at verifying another simplifying assumption we have used in the analysis. While equation (7.5) provides us with the *unconditional* average cost of a partial search over an interval (a, b), what we really need to derive equation (7.6) is the cost of a partial search over an interval (a, b) *conditionned* by the fact that the solution lies in some interval (a', b') (possibly the same interval). Here, too, the simulation results were useful to validate this simplifying assumption.

8 - Conclusions and open problems

We have presented in the first part of the chapter an analysis of the performance of the α-β pruning algorithm for searching a uniform tree of degree n and depth d when the values assigned to the terminal nodes are independent identically distributed random variables. The analysis takes into account both shallow and deep cut-offs, and we have also considered the effect of equalities between the values assigned to the terminal nodes.

A simple formula was derived, in Section 3, to measure the number of terminal nodes examined by the α-β procedure when the bottom values are drawn from a finite range according to an arbitrary *discrete* probability distribution. Although the formula can be easily computed numerically, a direct analysis is made difficult by the presence of the probability distribution. In the case when only two distinct values can be assigned to the terminal nodes, it is shown that, by choosing appropriately their probability distribution, the number of terminal nodes examined by the α-β procedure can grow at least as $O[(n/\ln n)^d]$, which, in fact, corresponds to the worst case behavior of the algorithm (over all possible probability distributions).

A formula was then presented in the form of an integral to measure the number of terminal nodes explored by the α-β procedure when the bottom values are all distinct. An analysis of the integral shows that the branching factor of the α-β pruning algorithm is $\Theta(n/\ln n)$, a result which confirms a claim by Knuth and Moore [35] that deep cut-offs only have a second order effect on the behavior of the α-β pruning algorithm.

We think that the main contribution of this analysis is to give a better understanding of the α-β pruning algorithm. In particular, we have shown that the a priori unrealistic assumption that all the values assigned to the terminal nodes of a game tree be distinct corresponds, in fact, to the worst case performance of the algorithm. Moreover, we have shown that this worst case performance can be attained even in the very simple case when the bottom values can only take on two distinct values, by choosing appropriately their probability distribution. We think that this can be important in practice because, it is relatively easy in most game playing programs to obtain (by inspection of the evaluation function) an accurate bound for the range of distinct values assigned to the various positions of the game, but it is usually not so easy to derive a good estimate for the probability distribution of these values.

Similarly, the branching factor analyzed in Section 5 provides us only with an asymptotic measure of performance for the α-β pruning algorithm (i. e., for trees of large depth). As indicated by the results of Section 3.3, however, the branching factor can also be used as a realistic measure of the worst case even for small trees.

We have measured the efficiency of the α-β pruning algorithm by the average number of terminal nodes explored during the search. It would be interesting to also obtain an estimate for the standard deviation of this number.

The scheme we have considered for assigning values to terminal nodes of a uniform tree lent itself easily to analysis; it is, however, very simplistic. Different schemes for assigning static values have been proposed in [23], [35] and [45]. Analyses of these schemes would be helpful for various applications; a step in this direction was presented in [45] for game trees of depth 2 and 3.

In the second part of this chapter we have investigated the possibilities of implementing the α-β pruning algorithm in parallel. Due to the intrinsically sequential character of the algorithm, it seems difficult to achieve a high efficiency with a parallel

implementation based on a direct reformulation of the original algorithm. Rather than having the processes search in parallel various subtrees of a game tree for the solution, we have proposed, in Section 6, a parallel implementation in which the processes work independently by searching the entire game tree for the solution over disjoint subintervals. The idea is similar to the notion of *aspiration level* implemented (sequentially) in the Technology Chess Program [24], [25].

In Section 7, we have developed an analysis of our parallel implementation of the α-β pruning algorithm, and Theorem 7.4 states an optimal sequence of intervals (which depends on the degree k of parallelism, i. e., the number of processes cooperating in the search) for minimizing the average cost of the algorithm. It follows, in particular, that, when the degree of parallelism k is small ($k = 2$ or 3), the parallel algorithm shows an improvement over the original algorithm by a factor which is larger than k. A surprising consequence of the results, therefore, is that the α-β pruning algorithm is not optimal. This fact has been confirmed through a series of simulations, and for a typical tree (with a degree of about 30, and a depth of about 5) the results show that the α-β pruning algorithm can be improved by 15% to 25%. It is to be noted that these figures are very consistent with empirical measurements of the Technology Chess Program [25] showing that the implementation of the aspiration level reduces the search by 23%.

The analysis we have developed relies on several simplifying assumptions, and it would be interesting to develop a more accurate analysis, for example, by using a closer approximation for the cost of a partial search, or by evaluating the cost of a partial search over some interval (a, b) given that the solution lies in some interval (a', b'). The analysis could also be refined by not assuming that the processes cooperating in the search restart new partial searches simultaneously.

Although the parallel implementation we have proposed appears to be efficient with a small number of processes, the maximum speed-up achievable is limited typically to 5 or 6 (see Figure 7.3 with $k = \infty$). We feel that a better way to implement in parallel the

α-β pruning algorithm with a large number of processes would be to combine both the strategy of decomposition we have proposed and the independent exploration of different subtrees of the entire game tree. For example, we could have two groups of processes, each group executing a partial search over a different subinterval, and each process in a group exploring a different subtree. We think, however, that the results are very important and should be used systematically in a sequential implementation, in conjunction with some dynamic evaluation of the probability distribution of the value of a game tree.

Chapter V

Experimental Results with Asynchronous Multiprocessors

1 - Introduction

By simulating a multiprocessor system, Rosenfeld [52] and Rosenfeld and Driscoll [53] have reported a series of results to measure the effectiveness of programming an asynchronous multiprocessor for the solution of the Dirichlet problem using chaotic iterations [11]. The problem consists of solving the set of linear equations associated with Laplace's equation through the method of finite differences.

In this chapter, we describe a series of experiments in which various asynchronous iterative methods (see Chapter III) are implemented on an asynchronous multiprocessor (C.mmp under the operating system Hydra [63], [64]) to solve the Dirichlet problem. We first present the results of measurements obtained with these experiments. We then show how very simple techniques from order statistics (see, for example, [14]) and from queueing theory (see, for example, [33]) can be used effectively to explain and predict with a fair accuracy the experimental results.

In Section 2, we briefly describe C.mmp and Hydra, and we outline the solution of the Dirichlet problem. In Section 3, we introduce the various asynchronous iterative methods that we have implemented on C.mmp. In Section 4, we report the results of the experiments, and, in Section 5, we present simple analytical techniques to account for these experimental results. Concluding remarks are given in the last section.

2 - Description of the experiments

In Section 2.1, we only present the main characteristics of C.mmp and of Hydra which are relevant to our purpose here; a formal presentation of C.mmp is given in [63] and of Hydra in [64]. Likewise, a full treatment of the use of the method of finite differences for solving the Dirichlet problem can be found, for example, in [22], and we only briefly describe the method in Section 2.2.

2.1 - The environment

The following description corresponds to a very simplified version of C.mmp under the operating system Hydra but will be sufficient to provide a reasonable model for our experiments.

C.mmp is a multiprocessor composed of p processors (p is currently *16*, but, at the time the experiments were run, it was oscillating between *4* and *9*), p_1 of those processors are PDP-11 model 20 and $p_2 = p - p_1$ are PDP-11 model 40. For purpose of comparison, we will indicate with the results the number and type of processors used in the experiments. Those processors are connected to m memory blocks (each with 1M words) through an $m \times p$ cross-point switch; m is currently *16* (it was *13* at the time of the experiments), but, since we are not limited by the size of the memory in our experiments, the exact value of m is irrelevant here. In addition, each processor is also connected to its own local memory (4K words). Although the memory available is very large, because of the small address field of an instruction (16 bits), only a small fraction (32K words) is directly addressable by a process at a given time. The Hydra system, however, provides the user with the facility of modifying the address registers in order to access the entire memory.

The Hydra system also provides the user with a set of macro-instructions for the manipulation of processes (creation, synchronization, etc.). In addition, the *policy module*

ensures some critical functions of the system (process scheduling, processor allocation, etc.); in particular, it ensures that each active process receives its fair share of processor time and a processor is allocated to a process only for some fixed quantum of time: at the end of a quantum the processor is deallocated from the process, and the latter is put back for re-scheduling into the pool of processes waiting for a processor.

2.2 - The problem

We consider a well-known problem, namely, the so-called *Dirichlet problem* for Laplace's equation (see, for example, [22, Section 20.9]).

The problem is to solve the partial differential equation:

$$u_{xx} + u_{yy} = 0 \qquad (2.1)$$

in a rectangular domain D of $\mathbb{R}^2$: $D = \{ (x,y) \mid 0 \le x \le \alpha , 0 \le y \le \beta \}$, when values of u on the boundary S of D are specified by the condition:

$$u = g , \qquad (2.2)$$

for some given function g defined on S. Many applications require solving this partial differential equation (or very similar ones) [22].

An approximation to the solution of equation (2.1) can be obtained through the *method of finite differences*. Assume that $\alpha = (n+1)h$ and $\beta = (m+1)h$, and define a regular grid on the domain D with mesh size h. This induces the set of points $\{ M_{i,j} (x_i=ih, y_j=jh) \mid 0 \le i \le n+1 , 0 \le j \le m+1 \}$. Let $u_{i,j}$ denote $u(M_{i,j})$; the values $u_{0,j}$, $u_{n+1,j}$, $u_{i,0}$ and $u_{i,m+1}$, on the boundary S, are known from equation (2.2). Using, for the second order derivative u_{xx} at the point (x,y), the approximation

$$u_{xx}(x,y) = [u(x+h,y) + u(x-h,y) - 2u(x,y)]/h^2$$

and a similar approximation for $u_{yy}(x,y)$, it can be shown (see, for example, [22, Section 23.4]) that a solution to the set of linear equations:

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = 0 , \quad 1 \le i \le n , \ 1 \le j \le m , \qquad (2.3)$$

gives an approximation to the solution of equation (2.1) for the points $M_{i,j}$ within an error

of order h^3 (assuming bounded properties of the fourth order derivatives of the solution u). A piecewise linear approximation for the solution u on the domain D can then be deduced from the solution of system (2.3).

The set of equations (2.3) constitutes a linear system for which we are investigating the solution. This system can be written, in matrix form, as:

$$A\,x = a\,. \tag{2.4}$$

When x is the nm-vector corresponding to the row-major ordering of the grid points:

$$x = [u_{1,1},\ \ldots,\ u_{n,1},\ u_{1,2},\ \ldots,\ u_{n,m}]^T\,,$$

we deduce from this ordering the $nm \times nm$-matrix A and the nm-vector a of equation (2.4), the latter being known from the values of the function g giving the boundary conditions.

Different iterative schemes have been implemented on C.mmp to solve this system. They are described in the following section.

3 - Some implementations of asynchronous iterations

The matrix A of equation (2.4) is a very sparse matrix (at most five elements are not zero in any given row), and, in this case, iterative methods, although they do not provide us with the exact solution, are usually advantageous.

The first two methods we have considered are two basic iterative methods: the point Jacobi and the Gauss-Seidel's methods. These two methods have been widely studied and will be useful as a basis of comparison. These and other iterative methods that we have implemented are described in the following sections. Throughout, we discuss parallel implementations with k processes ($k = 1$ corresponding to a sequential implementation), and, for simplicity, we assume that the size nm of the matrix A is a multiple of k and let $q = nm/k$. In all implementations, we make use of a global vector, called X, to contain the current value of the solution vector.

3.1 - Jacobi's method and Asynchronous Jacobi's method

Since all diagonal elements of the matrix A have the same value of 4, the point Jacobi matrix is readily obtained. Let $x(i)$ denote the i-th iterate computed by Jacobi's method. We simply deduce from equation (2.4) that:

$$x(i+1) = (I - \frac{1}{4} A)\, x(i) + \frac{1}{4} a = B\, x(i) + b .$$

The matrix

$$B = I - \frac{1}{4} A$$

is the *Jacobi matrix* associated with our problem. This matrix has been extensively studied, and its spectral radius, which determines the rate of convergence of Jacobi's method, is given by:

$$\rho(B) = \frac{1}{2} \left(\cos \frac{\pi}{n+1} + \cos \frac{\pi}{m+1} \right) . \qquad (3.1)$$

We see that with Jacobi's method all components of an iterate are computed simultaneously using the values of the previous iterate, and that parallelism can, therefore, be introduced easily. A natural parallel implementation with k processes is to simply decompose the evaluation of an iterate into k subcomputations, each one corresponding to the evaluation of a subset of $q = nm/k$ components, and to have the k processes carrying out the evaluation of the k subsets of components in parallel. When a process completes its computation, it must then block itself and wait until the completion of all other subcomputations before starting the evaluation of the next iterate. Our implementation corresponds to this description, in which process P_1 always evaluates the first q components of the iterate, process P_2 the next q components, ... and process P_k the last q components. After each subcomputation all processes synchronize themselves using a semaphore, and, after having updated the components, they all resume their executions for the evaluation of the next iterate.

The complete synchronization of all processes at each step of the iteration is an evident drawback in the parallel implementation of Jacobi's method, and we can anticipate that this will result in a substantial overhead. The *Asynchronous Jacobi's method* (or *AJ*

method) is a variation of Jacobi's method in which a process never waits for the other processes to complete their computations. As soon as a process completes the evaluation of its subset of components, it releases the new values for the other processes by updating the corresponding components of the global vector X, and, immediately after, the process starts re-evaluating its subset, using in the computation, the values of the components as they are known at the *beginning* of the re-evaluation. The AJ method has been implemented using a critical section for updating the components of the global vector X at the end of an evaluation, and for copying the components of X required for the next evaluation.

It can be seen easily that, if a process is never suspended indefinitely, the AJ method can be expressed as an asynchronous iterative method relative to the linear operator corresponding to the Jacobi matrix B. Since B is a non-negative matrix with a spectral radius less than unity, it is a contracting matrix, and the convergence of the AJ method for our problem is a direct consequence of the results of Chapter III.

3.2 - Gauss-Seidel's method and Asynchronous Gauss-Seidel's method

Gauss-Seidel's method differs from Jacobi's method in that the components of an iterate are evaluated in sequence and the value of $x_r(i)$ is used in the computation of $x_s(i)$ when $s > r$ (that is, as soon as it is available). Let L and U be the strictly lower and upper triangular matrices defined from:

$$B = I - \frac{1}{4} A = L + U .$$

The sequence of iterates, for Gauss-Seidel's method, satisfies:

$$x(i+1) = L\, x(i+1) + U\, x(i) + b .$$

The matrix

$$\mathcal{L} = (I - L)^{-1} U$$

defines $x(i+1)$ directly as a function of $x(i)$. Its spectral radius determines the rate of convergence of Gauss-Seidel's method and is given by:

$$\rho(\mathcal{L}) = [\rho(B)]^2 , \qquad (3.2)$$

where $\rho(B)$ is the spectral radius of the Jacobi matrix and is given by equation (3.1).

We notice that Gauss-Seidel's method is intrinsically sequential, and that parallelism cannot be easily introduced. The method has been implemented sequentially (i. e., with *1* process) as a particular case of the *Asynchronous Gauss-Seidel's method.*

The *Asynchronous Gauss-Seidel's method* (or *AGS method*) is similar to the AJ method except that a process evaluates the components in its subset sequentially and uses the new value of a component within the *same* subset as soon as it becomes available. In this respect, the AGS method resembles Gauss-Seidel's method for the computation within a subset of components, and, in particular, when the AGS is implemented with only one process, it simply reduces to Gauss-Seidel's method.

As in the case of the AJ method, the AGS method can be shown to correspond to an asynchronous iterative method relative to the Jacobi matrix B, and, in this case too, the convergence of the AGS method follows from the results of Chapter III since the matrix B (in the particular case of our problem) is a contracting matrix.

3.3 - Purely Asynchronous iterative method

The *Purely Asynchronous method* (or *PA method*) is the simplest method we have implemented. It basically resembles the AGS method, but it uses no critical section for releasing the values of the components in its subset of for copying the values of the components required in the computations. Rather, a process fetches directly from the global vector X the values of the components as they are needed and releases new values of the components *one by one, immediately after the evaluation of each component.* Again, the PA method can be easily expressed as an asynchronous iteration relative to the linear operator corresponding to the contracting matrix B, and the convergence of the PA method, for our problem, follows directly from the results of Chapter III.

In addition to being the simplest method to implement from a programming point of

view, the PA method is also, spacewise, the most efficient method since no extra variable is required to copy the values of an iterate as of the beginning of an evaluation or to contain the new values of the components before being released. The main advantage of the PA method, however, is the total absence of any form of synchronization, which, therefore, makes it very attractive for implementation on an asynchronous multiprocessor.

An apparent disadvantage of the PA method is that all processes frequently access the common global vector X, therefore possibly causing memory conflicts. This is not so for the particular problem we are considering in case of a large system of equations (i. e., for large n and m). Because of the sparsity and the special form of the matrix associated with our system, accesses to the vector X by a given process will be mostly confined to accesses of components within its own subset and only a few accesses to components in the two adjacent subsets. Moreover, this is the general case for the solution of linear systems resulting from the application of the method of finite differences to partial differential equations. Therefore, this apparent problem can be solved easily simply by allocating different memory banks to differents subsets of components of the global vector X.

Another problem with the PA method is specific to C.mmp (and Cm*) and is due to the absence of uninterruptible double word instructions on the PDP-11 (or the LSI-11). In particular, since a floating point number is implemented on two consecutive *16* bit words, simultaneous updating and reading of the same component by two processes might result in a lost of precision of the last *16* bits of the mantissa. Although this problem is very unlikely to occur, it is real, and the precision achievable on the solution vector has to be chosen accordingly.

3.4 - Other possible implementations

The methods we have introduced are intended to be an illustration of the issues raised by the implementation of parallel algorithms on an asynchronous multiprocessor,

and they are not necessarily the most efficient way to solve a linear system of equations by iteration. In this section, we mention several techniques which should be used in the practical implementation of asynchronous iterative methods.

3.4.1 - Asynchronous iterations with relaxation

The introduction of a relaxation factor is a well known technique for improving the performance of iterative methods, and, although we do not report here any results concerning iterative methods using relaxation, we have run some experiments which show that the introduction of a relaxation factor is a very promising way to accelerate asynchronous iterative methods.

Let F be an operator, and let ω be a positive scalar. An iteration relative to F with the relaxation factor ω defines the sequence of iterates through:

$$x(i+1) = \omega F x(i) + (1-\omega) x(i) .$$

In particular, when $\omega = 1$, this corresponds directly to the iteration relative to F. This technique is very useful, in general, since the relaxation factor ω can be chosen to maximize the efficiency of the iteration.

As particular cases, let us examine the methods we have implemented. The *Jacobi Over-Relaxation method* (or *JOR method*) produces the sequence of iterates defined by:

$$x(i+1) = \omega \left[(I - \frac{1}{4} A) x(i) + \frac{1}{4} a \right] + (1-\omega) x(i) ,$$

and, therefore, corresponds to Jacobi's method with the Jacobi matrix:

$$B_\omega = I - \frac{1}{4} \omega A = \omega B + (1-\omega) I .$$

It follows that, in our case,

$$\rho(B_\omega) = |1-\omega| + \omega \rho(B) ,$$

therefore, $\omega = 1$ minimizes $\rho(B_\omega)$, which means that Jacobi's method cannot be improved using relaxation.

The *Successive Over-Relaxation method* (or *SOR method*) is derived from Gauss-Seidel's method. The SOR method defines the sequence of iterates:

$$x(i+1) = \omega [L\, x(i+1) + U\, x(i) + b] + (1-\omega)\, x(i) ,$$

and it can be shown (see, for example, [62, p. 203]) that the spectral radius of the SOR matrix

$$\mathcal{L}_\omega = (I - \omega L)^{-1}[(1-\omega)I + \omega U]$$

is minimized when:

$$\omega = \frac{2}{1 + \sqrt{1-\rho^2(B)}} .$$

Similarly we can define the *AJOR*, *ASOR* and *PAOR methods* from the AJ, AGS and PA methods, respectively. All three methods are easily shown to correspond to asynchronous iterative methods relative to the linear operator associated with the matrix B_ω. In particular, since

$$\rho(|B_\omega|) = |1-\omega| + \omega\, \rho(B) ,$$

provided that:

$$0 < \omega < \frac{2}{1 + \rho(B)} , \tag{3.3}$$

the matrix B_ω is a contracting matrix, and we are guaranteed of the convergence of all three methods in the particular case of our problem. Nothing, however, is known in general as to the best ω, and further results in this direction would certainly be of interest. Note that condition (3.3) only represents a sufficient condition for convergence, and that the methods can still converge outside of this range.

3.4.2 - Adaptative asynchronous iterations

All of the implementations that we have proposed are based on a *static decomposition* of the computation involved in the evaluation of an iterate, and, in all cases, each process is assigned to the evaluation of a fixed subset of components. With Jacobi's method, this results in a substantial overhead since all processes have to wait for each other at the end of each step of the iteration. A possibility for reducing this overhead is to decompose the components of an iterate into more subsets than processes, and to let the processes adjust their own speeds by evaluating more or fewer subsets of

components. For example, the parallel implementation of Jacobi's method with 2 processes which seems the best suited for execution on an asynchronous multiprocessor is to have one process update the components starting with the first one and to have the second process update the components starting with the last one; an iteration step terminates when the two processes meet (not necessarily exactly in the middle). With this implementation, the difference in execution times between the two processes is limited at most to the time to evaluate only one component, which obviously reduces significantly the waiting time.

Another way to take into account the different speeds of the processes would be to subdivide the components into subsets of different sizes, and assign the computation of a larger subset of components to a faster process. The speed of a process, however, depends mainly on the speed of the processor on which the system decides to execute the process, and this is usually not known a priori.

There is another advantage of not pre-assigning to a process the evaluation of a fixed subset of components since, at each step of the iteration this allows for some flexibility in the selection of the subset to be evaluated next. Many criteria can be used for this selection, in particular:

(1) LRU: the subset selected is the one which has been the Least Recently Updated among those not currently updated.

(2) GRE: the subset selected is the one which carries the Greatest Relative Error (also among those which are not currently updated).

The GRE selection, for instance, should increase the efficiency of an iterative method by reducing the number of iterations required to achieve some given admissible error. The selection of a new subset at each step of the iteration might, however, introduce additional overhead and, in particular, will almost necessarily require the use of a critical section. We do not think that this should be used, therefore, in conjunction with the PA method.

3.5 - Organization of the program

Before presenting the results we give a brief description of the programs. All of the different methods have been implemented in BLISS-11 [15] and all programs have basically the same following structure.

Master process:

```
Initialization: read in n, m, ε, k;
for i = 1, ..., k do
    Create and start process i;
for i = 1, ..., k do
    P(completion);
Output the statistics about the run;
```

Computational process *i*:

```
⌈ P(mutex);
|    Read all necessary components of X;
⌊ V(mutex);
repeat
    Evaluate all components of subset i;
    ⌈ P(mutex);
    |    Update all components in subset i;
    |    Read all necessary components of X;
    ⌊ V(mutex);
until global error < ε;
V(completion);
```

The method implemented by this program is embedded in the instruction "Evaluate all components of subset *i*." From the program each process can be thought of as a succession of identical *cycles*; each cycle being composed of an *evaluation section* followed by a *critical section*.

The programs for Jacobi's method and for the PA method are slightly different but follow basically the same structure.

4 - The results of the experiments

We report, in this section, the measurements obtained by running on C.mmp the various iterative methods that we have introduced in Section 3. We discuss, in Section 4.1, the different parameters of the program and the decisions leading to their choices. In Section 4.2, we present the local behavior of the processes within each cycle, and, in Section 4.3, we present the global results and compare the different methods.

4.1 - Choice of the parameters

All of the experiments have been run under the same conditions, and, before presenting the results of the measurements, we briefly discuss below the choices we have made for the various parameters of our problem.

4.1.1 - Size of the system

We want to choose the size of the system to be solved (i. e., to choose n and m) large enough so that the problem be realistic, but, on the other hand, since we do not want to deal here with problems of memory addressing, we have limited ourselves to a size that permits all of the data to be directly addressable. The main restriction, in this case, comes from the fact that the size of the data local to a computational process has to fit into the stack of local variables (contained in page 0), i. e., in about 3K words. With the AJ method, for instance, each process has to have the values of the components it is updating and a copy of the values of the components used in the evaluation, as of the starting time of the computation. There may be up to $2nm$ elements each of which fits into two words of memory. Therefore nm has to be chosen below *700*. The number *504* has been chosen (mainly because it is divisible by *1*, *2*, *3*, *4*, *6*, *7*, *8*, *9* ... and almost by 5 too!), and n and m have been chosen to be *21* and *24*, respectively, in the series of experiments reported here.

4.1.2 - Error of the solution vector

An experiment is stopped when some norm of the error vector is smaller, in magnitude, than a given admissible error ϵ. (The norm we have chosen is $\|.\|_\infty$, the maximum over all components.) Since we want to be able to compare the experimental results with the results of a theoretical analysis, we want to choose ϵ small enough so that asymptotic rates of convergence can be estimated through experimental results. For our purposes, the asymptotic rate of convergence for a method $\mathcal{M}$ can be defined as:

$$R(\mathcal{M}) = \lim_{i \to \infty} - \frac{\log \|e_i\|}{n_i}, \tag{4.1}$$

where ϵ_i is the error vector after the i-th *sub-iteration* (a *sub-iteration* corresponds to an evaluation by *one* process so that k sub-iterations are carried out simultaneously in a parallel implementation with k processes), and where n_i is the mean number of times each component has been evaluated up to the i-th sub-iteration. For all the implementations we have considered the components are divided into k equal subsets, and n_i is simply given by $n_i = i/k$. (The norm in equation (4.1) is the same norm as the one used in the termination criterion.) This definition of asymptotic rate of convergence corresponds to the classical definition and, in particular, we have $\hat{R}(\text{Jacobi}) = -\log \rho(B)$.

The interpretation of the rate of convergence is that $1/\hat{R}(\mathcal{M})$ is an asymptotic measure of the average number of times each component has to be updated in order to decrease the norm of the error vector by a factor of *10* (if the log of equation (4.1) is base *10*). In particular, when ϵ tends to *0*, the average number of iterations (per component) required to solve the system with an error less than ϵ grows linearly like $-\log(\epsilon)/\hat{R}(\mathcal{M})$. In Figure 4.1 we have plotted the number, $N(\epsilon)$, of iterations required to solve our system ($n = 21$, $m = 24$) within an error ϵ, versus $-\log(\epsilon)$ for both the AJ and the AGS methods when $k = 1$ and *3* processes are used. This shows clearly that the asymptotic rate of convergence is reached very fast since, when $-\log(\epsilon) > 0.25$ (i. e., $\epsilon < 0.56$), $N(\epsilon)$ varies linearly with $-\log(\epsilon)$.

When $k = 1$ the AJ and AGS methods reduce to Jacobi's and Gauss-Seidel's methods, respectively, and the slopes obtained from Figure 4.1 can be compared to the theoretical values $[-\log \rho(B)]^{-1}$ and $[-\log \rho(\mathcal{L})]^{-1}$, respectively, where:

$$\rho(B) = \frac{1}{2}\left(\cos\frac{\pi}{n+1} + \cos\frac{\pi}{m+1}\right) \sim 0.99097\,,$$

$$\rho(\mathcal{L}) = [\rho(B)]^2 \sim 0.98202\,.$$

In Table 4.1, we report the observed and theoretical number of iterations required to asymptotically divide the norm of the error vector by a factor of *10*.

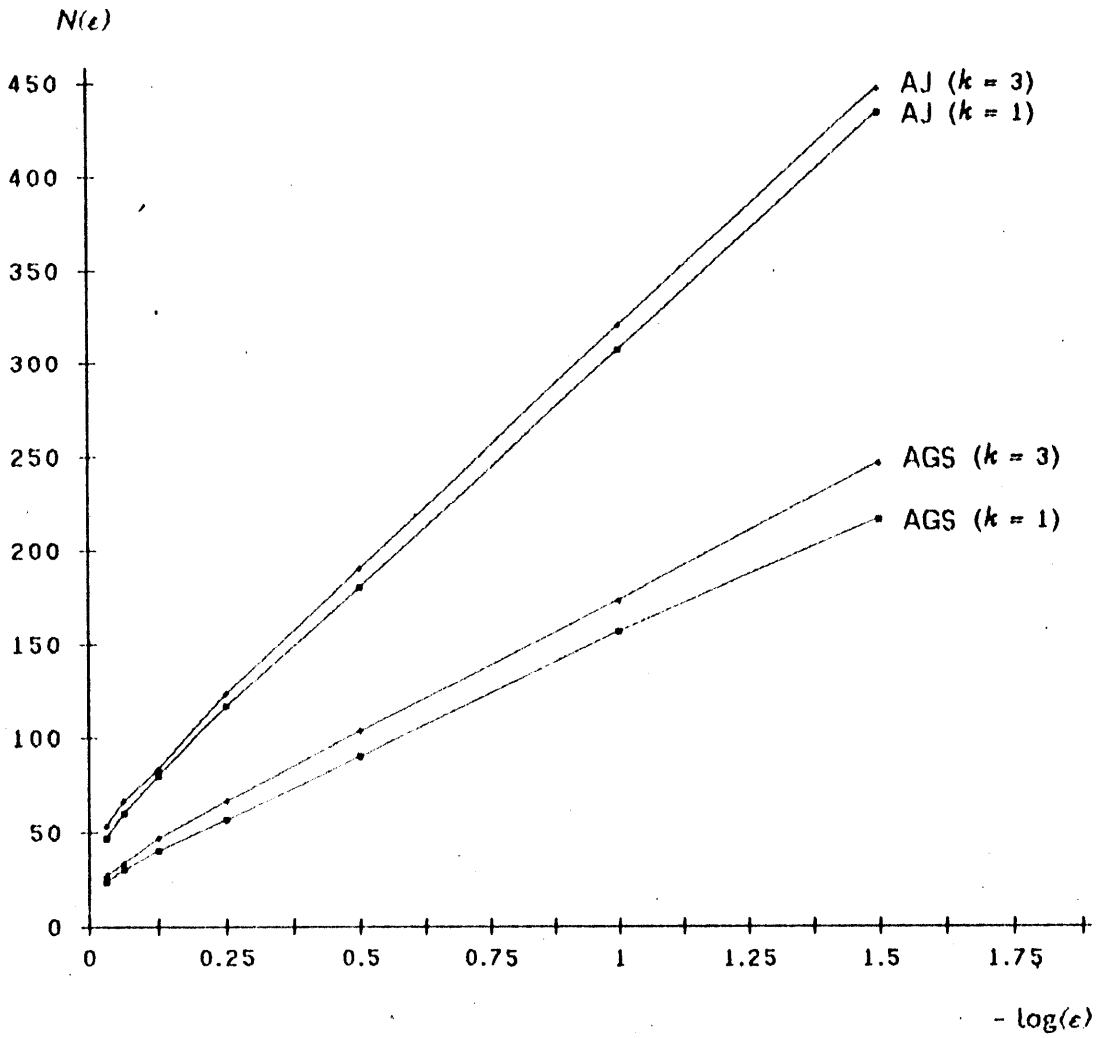


Figure 4.1 - Number of iterations required with the AJ and AGS methods

	AJ		AGS	
	$k = 1$	$k = 3$	$k = 1$	$k = 3$
Observed:	*254*	*257*	*127*	*143*
Theoretical:	*254.79*	-	*127.89*	-

Table 4.1 - Comparison of the rates of convergence for the AJ and AGS methods

In all the experiments reported below, the termination criterion uses $\epsilon = 0.1$ for the value of the admissible error. This value corresponds to a reasonable execution time, in the order of 3 min., and allows us to base our measurements on more experiments.

4.1.3 - Other parameters

Since we are mainly interested in comparing the different methods with respect to their rates of convergence toward the solution vector, we simply set the displacement vector b to be 0 so that the solution is known to be $\xi = 0$. As the system we are studying is linear, we do not loose any generality, but this will result in a simpler test for the termination criterion since, in this case, the current iterate is exactly the error vector. Lastly, in all the experiments, the initial approximation has been chosen as the vector with all components equal to 1.

4.2 - Local behavior of the program

We present, in this section, the local behavior of the computational processes by looking at the time they spend during each cycle in the evaluation section and (except with the PA method) in the critical section of the program. In Section 4.2.1, we present the results of the measurements, and, in Section 4.2.2, we give an interpretation.

4.2.1 - Results of the measurements

The results presented in this section have been derived from the information given by the tracer David Lamb implemented on C.mmp. (Among many other things, each P and V operation is reported by the tracer along with the time instant when it was executed, the process executing the operation and the processor carrying out the execution.) Since the code of the programs for the different methods are identical (with respect to these measurements) we limited ourselves to take measurements on the AJ method. Four experiments have been run with $k = 1, 3, 6,$ and 12 processes. In all of them $p = 7$ processors were available: 5 PDP-11/20 and 2 PDP-11/40. The histograms for the distribution of the time spent in the evaluation section as well as the distribution of the time spent in the critical section, for each of the experiments, are plotted in Figures 4.2 through 4.9. (In the case of the critical section, the results presented in these figures also include, when $k > 1$, the possible waiting time before entering the critical section.)

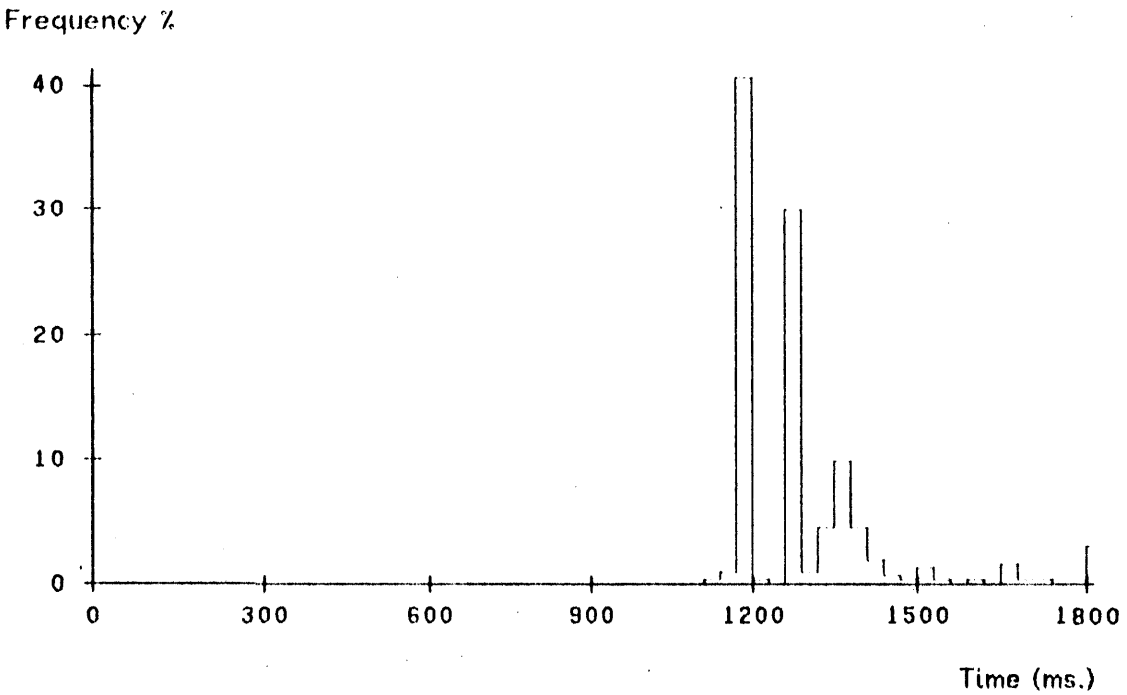


Figure 4.2 - Time spent in the evaluation section ($k = 1$)

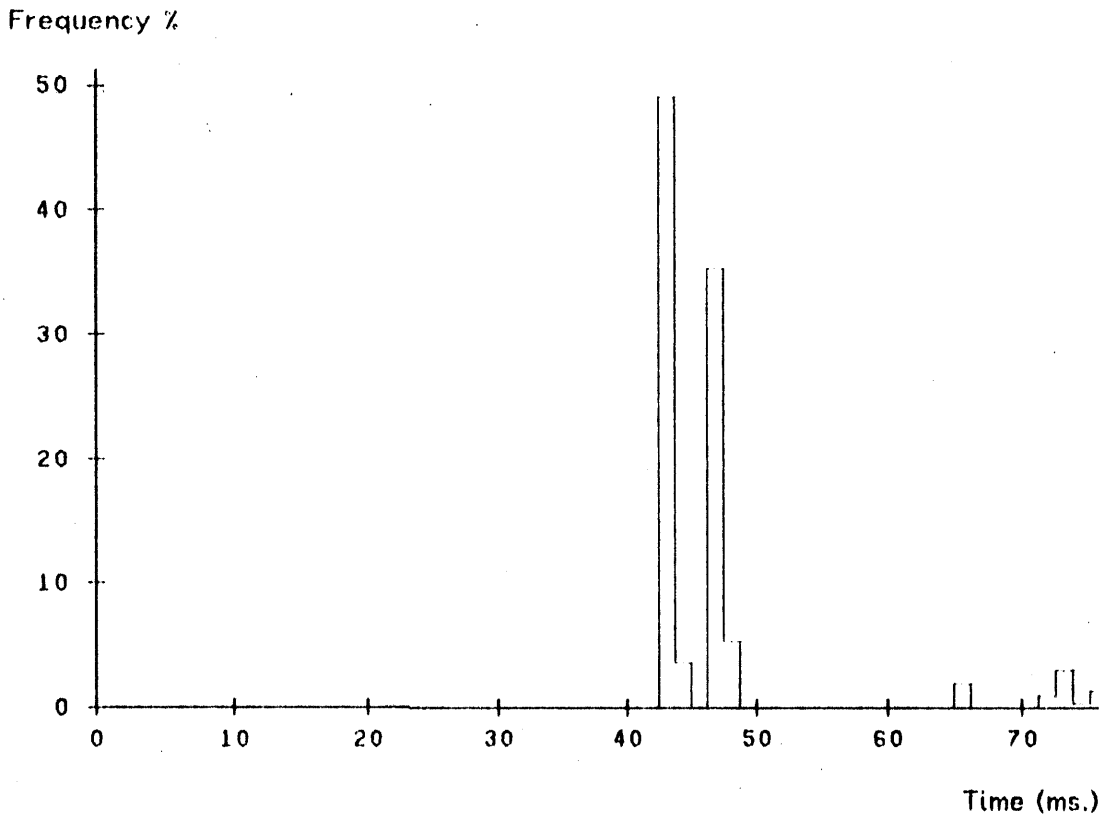


Figure 4.3 - Time spent in the critical section ($k = 1$)

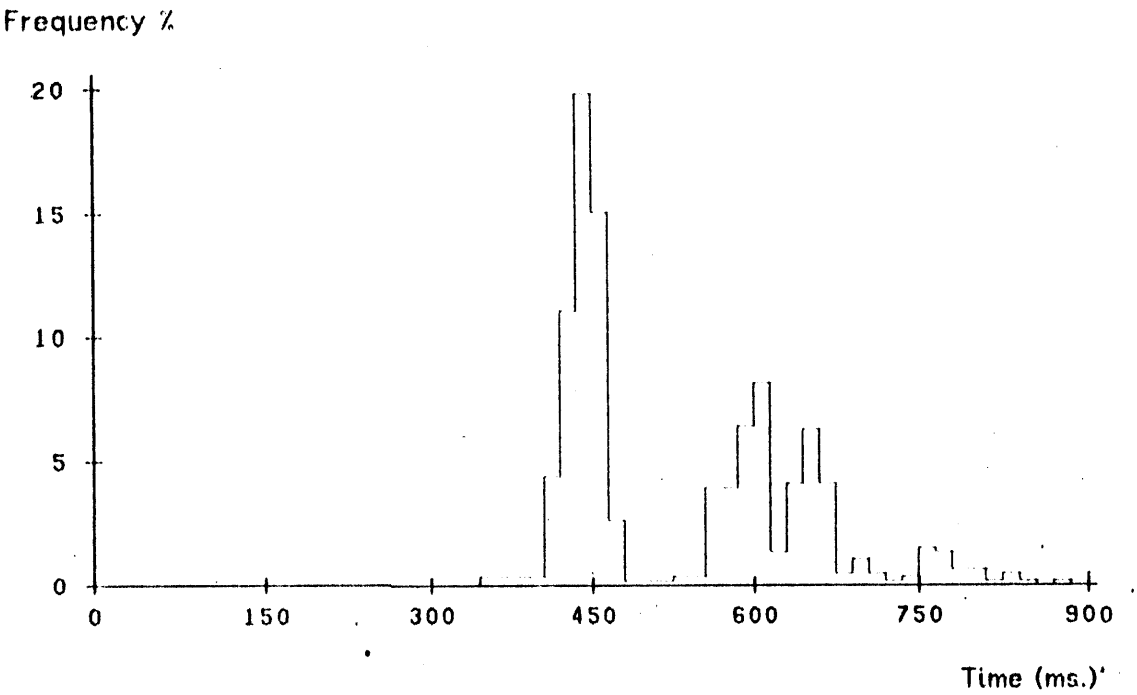


Figure 4.4 - Time spent in the evaluation section ($k = 3$)

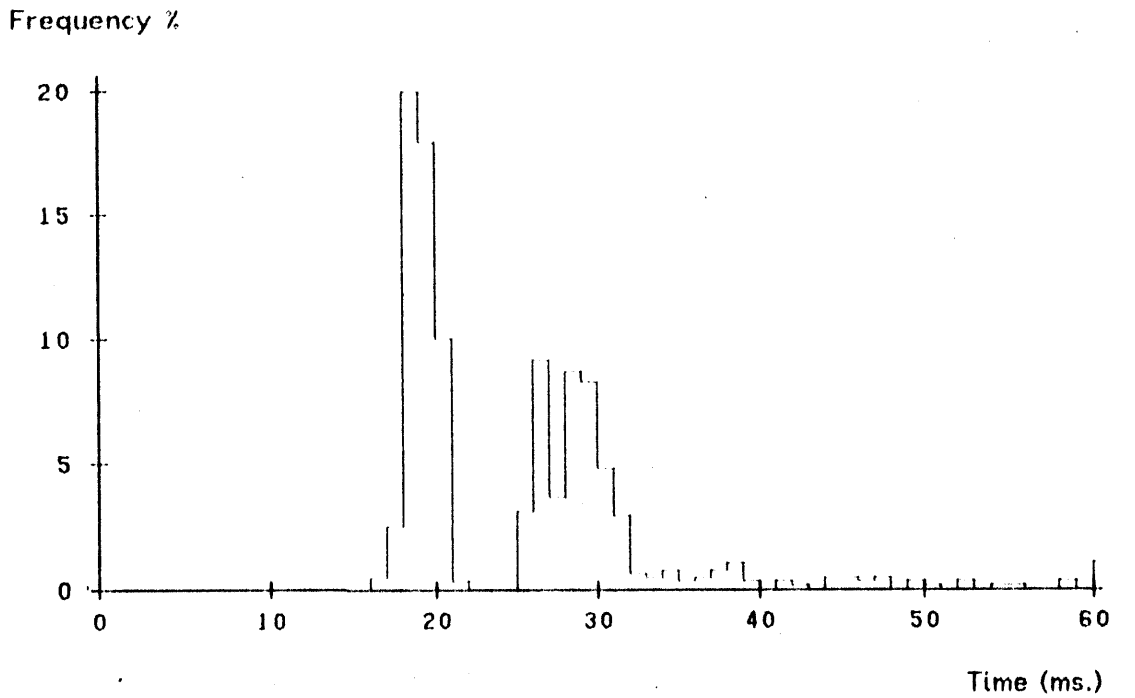


Figure 4.5 - Time spent in the critical section ($k = 3$)

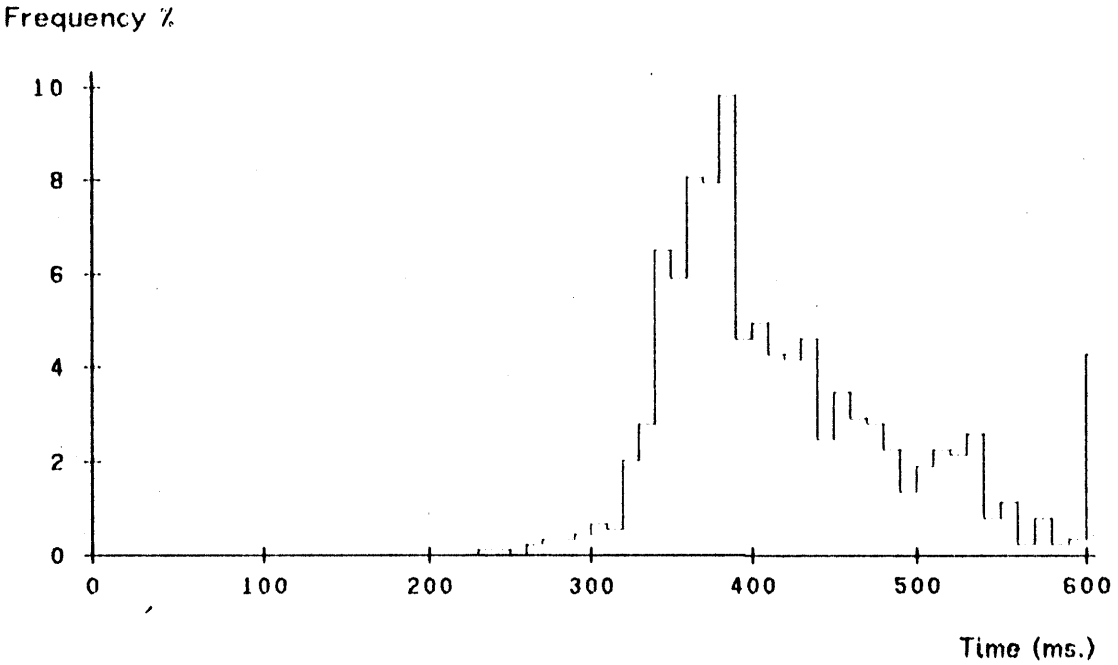


Figure 4.6 - Time spent in the evaluation section ($k = 6$)

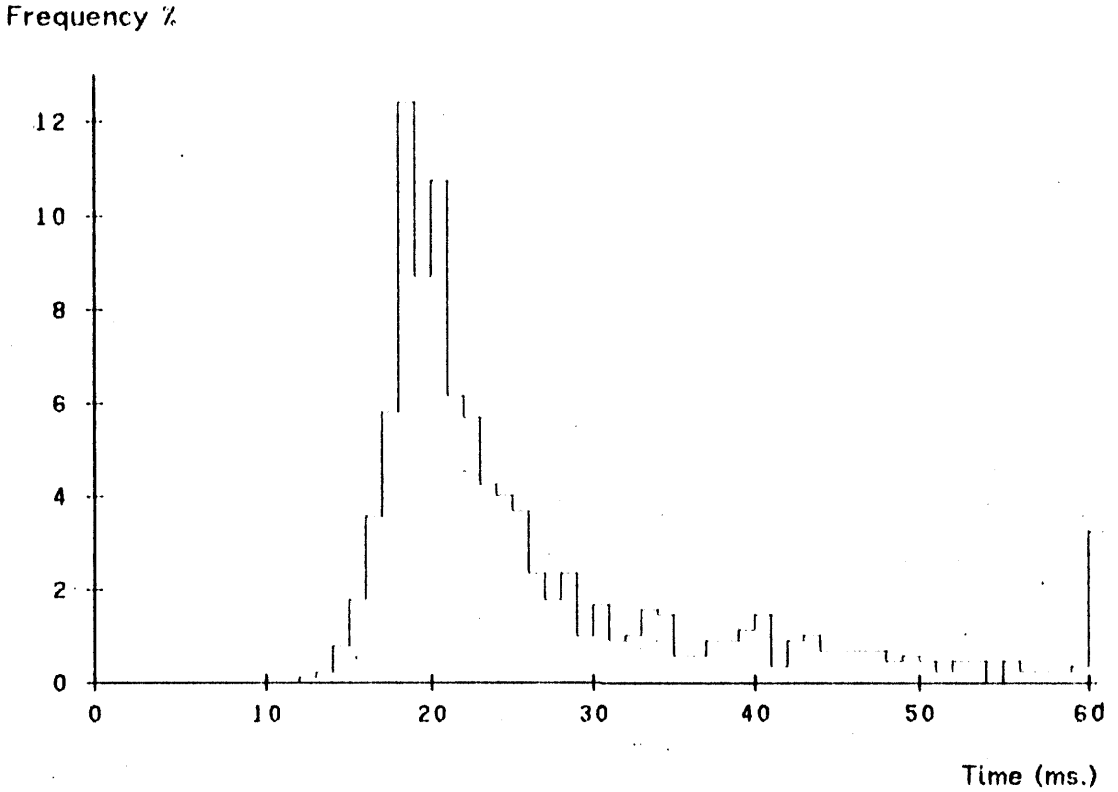


Figure 4.7 - Time spent in the critical section ($k = 6$)

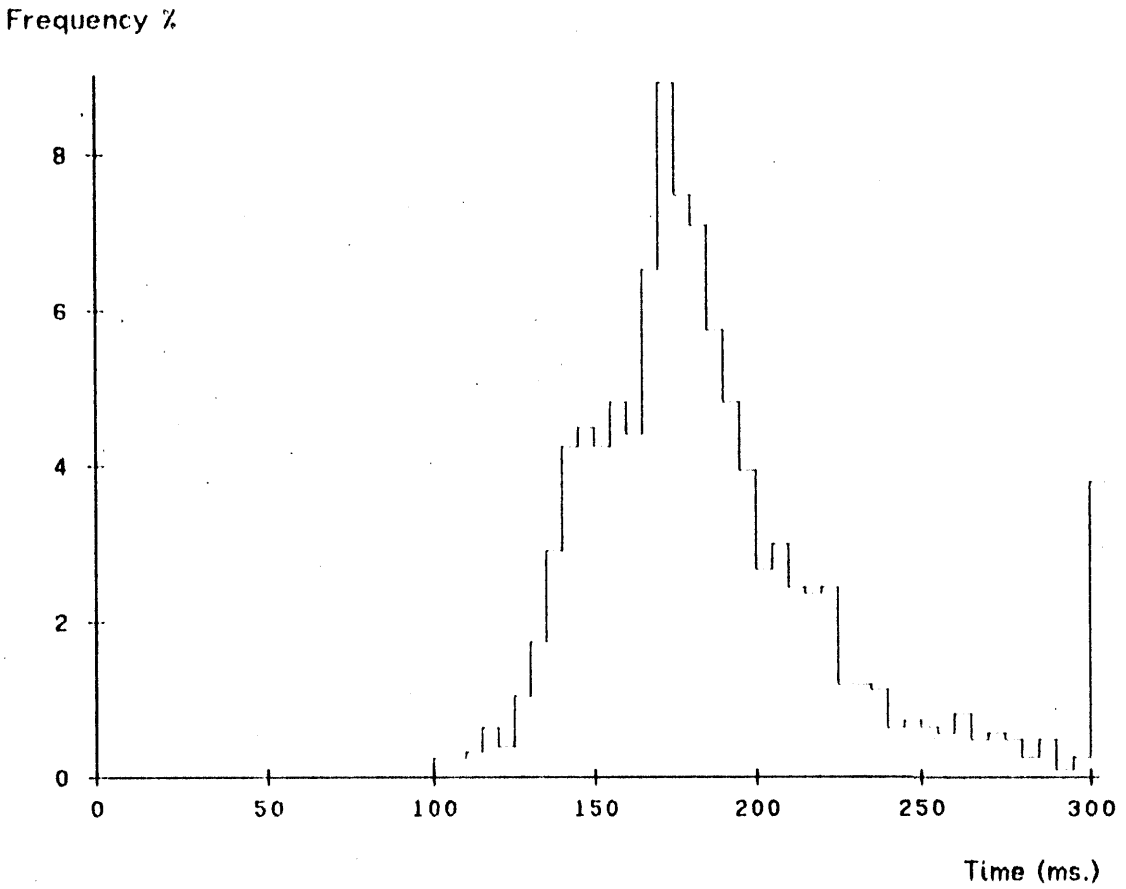


Figure 4.8 - Time spent in the evaluation section ($k = 12$)

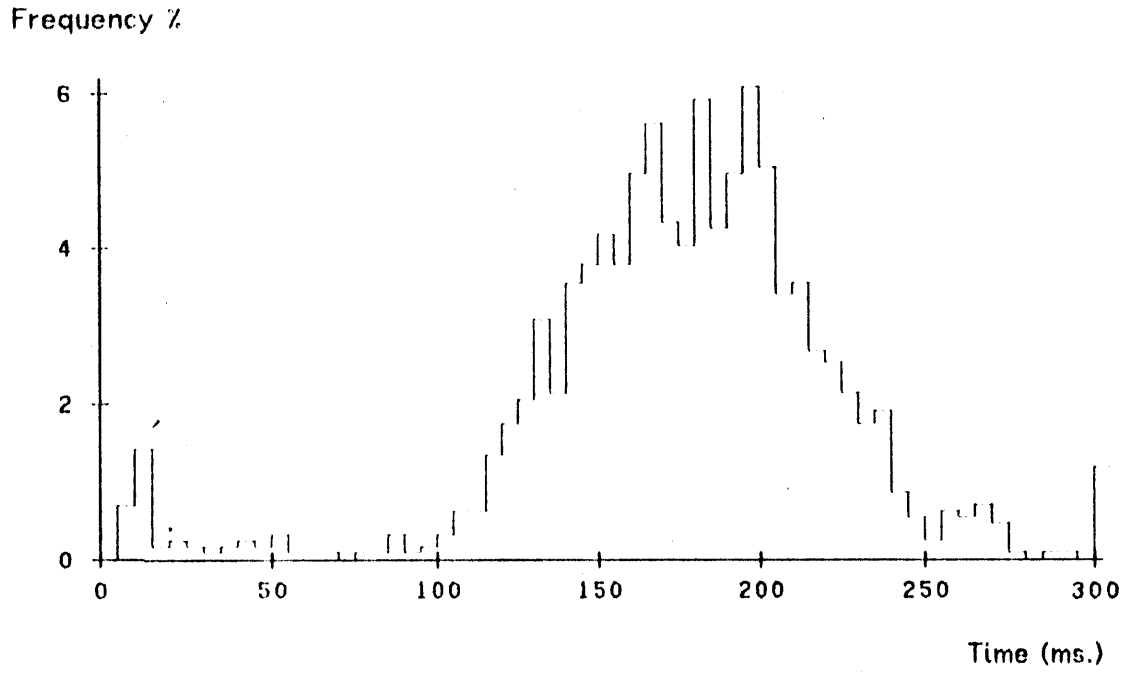


Figure 4.9 - Time spent in the critical section ($k = 12$)

These figures show clearly that two different types of processors are used. When $k = 3$, for example, the distributions have two main peaks (at about 18 ms. and 28 ms. in Figure 4.5), and, in particular, we can derive from our results an estimate for the relative speeds of the PDP-11/20 and the PDP-11/40. The ratio of the speeds is certainly problem dependent but, in our case, *1* second on a PDP-11/40 corresponds to about *1.4* seconds on a PDP-11/20, i. e., the use of a PDP-11/40 instead of a PDP-11/20 corresponds to a gain of about *30%* in running time. If we look more closely, we can see that each main peak is composed of several subpeaks corresponding to each processor; two different processors, even of the same type, actually have different speeds. This is particularly evident in Figures 4.2 and 4.3, where the two main peaks correspond to the executions on each of the *2* PDP-11/40. Since it is the policy of Hydra to allocate first the PDP-11/40, the third peak in Figure 4.2 does not correspond to to an execution on a PDP-11/20 but, in fact, corresponds to executions on a PDP-11/40 which include some overhead due to the re-scheduling of a process at the end of a quantum.

4.2.2 - An interpretation of the results

The main statistics about the distributions presented in the figures of Section 4.2.1 are collected in Table 4.2 (a) and (c) for the evaluation section and the critical section (including the possible waiting time), respectively. In addition, Table 4.2 (b) contains the same statistics concerning the critical section by itself, excluding any waiting time. (All timings in the table are expressed in ms.)

In Figures 4.10, 4.11 and 4.12, we have plotted the variations of the average execution times for the two sections of the program as they can be found in Table 4.2 (a), (b) and (c), respectively. The results of Figure 4.11 represent strictly the execution time of the critical section, while the timings presented in Figure 4.12 also contain the possible waiting time before entering the critical section.

	$k = 1$	$k = 3$	$k = 6$	$k = 12$
Minimum	1123.85	348.30	239.36	100.07
Maximum	1889.60	1524.13	834.97	502.02
Average	1292.72	534.35	423.04	187.86
Standard dev.	136.51	118.88	84.23	47.10
Coeff. of var.	0.106	0.222	0.199	0.251

(a) Evaluation section

	$k = 1$	$k = 3$	$k = 6$	$k = 12$
Minimum	43.49	16.82	13.59	7.44
Maximum	174.82	186.02	170.96	21.91
Average	47.75	23.96	21.65	11.57
Standard dev.	13.91	11.71	7.67	2.77
Coeff. of var.	0.291	0.488	0.354	0.240

(b) Critical section (without the blocking)

	$k = 1$	$k = 3$	$k = 6$	$k = 12$
Minimum	43.49	16.82	13.59	7.44
Maximum	174.82	199.64	196.97	431.65
Average	47.75	25.63	27.81	177.04
Standard dev.	13.91	13.90	17.67	48.35
Coeff. of var.	0.291	0.542	0.635	0.273

(c) Critical section (including the blocking)

Table 4.2 - Statistics about the two sections of the program

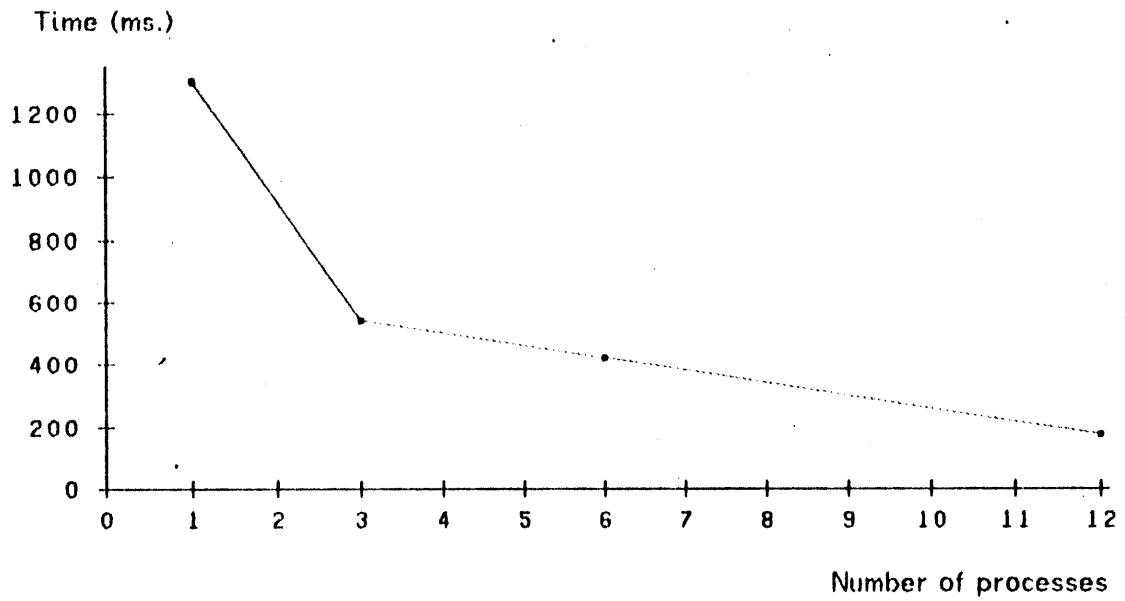


Figure 4.10 - Mean time spent in the evaluation section

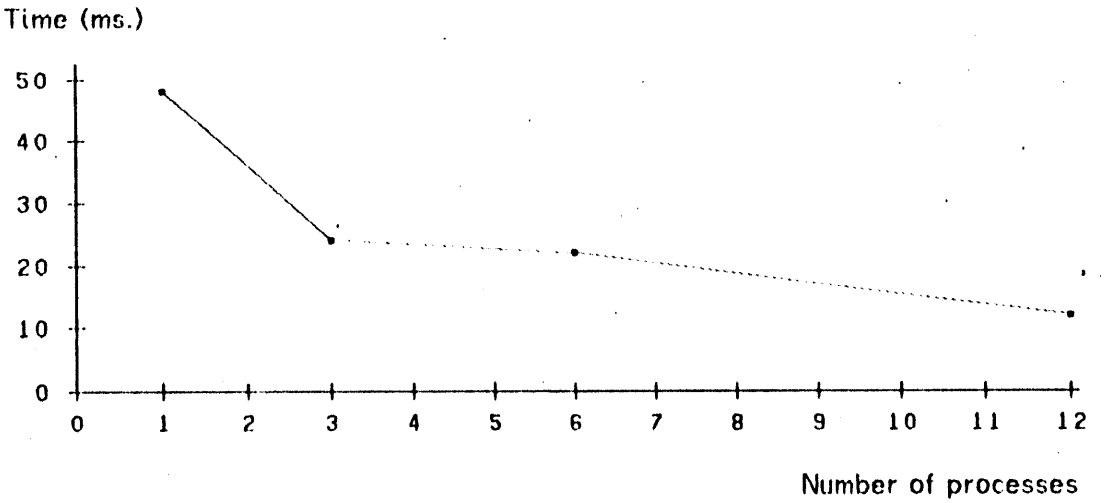


Figure 4.11 - Mean time spent in the critical section (waiting time excluded)

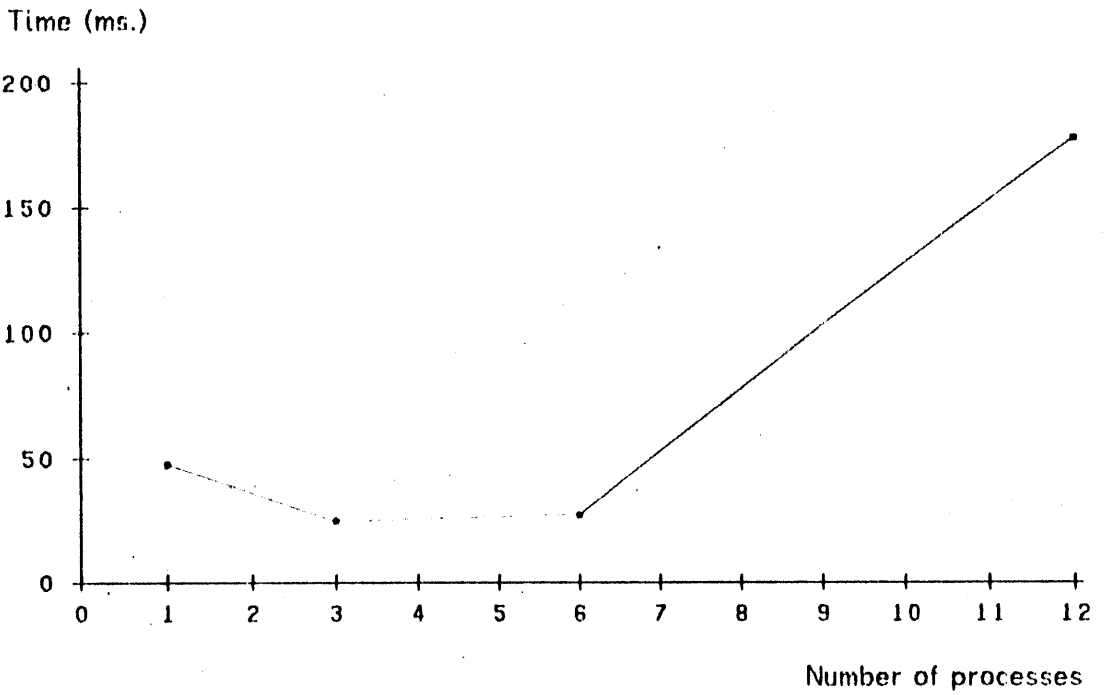


Figure 4.12 - Mean time spent in the critical section (waiting time included)

We note that, while a process does not suffer a very important delay (before the critical section) in the parallel implementation with $k = 3$ and 6 processes, Figure 4.12 shows a very sharp increase in the waiting time for $k = 12$. In fact, further results obtained by tracing the execution of the program showed that, in the parallel implementation with *12* processes, the queue to the critical section contained almost

always 6 or more processes (not counting the process executing the critical section). This means that there has almost always been at least one processor idle among the 7 processors available. The fact that the processes are never competing for a processor can, therefore, explain the steady decrease of the execution times presented in Figures 4.10 and 4.11. In both cases a first approximation can be obtained in the form $a + \frac{1}{k} b$, for some appropriate constants a and b. However, since it will be useful in Section 5, we develop below a closer approximation which takes into account the policy of Hydra to allocate first a PDP-11/40 (i. e., a faster processor).

Let p_1 and p_2 be the number of PDP-11/20 and PDP-11/40 available, respectively; and let $p = p_1 + p_2$. We denote by ρ the relative speeds of the two types of processors; experimental evidence, from the results of Section 4.2.1, showed that $\rho \sim 1.4$ corresponds to a reasonable estimate in the particular case of our problem. Consider a program which requires an average time x when it is executed on a PDP-11/40, and let x_k be the average execution time of the same program when it is executed in an environment with k processes (each process is assumed to receive its fair share of computing power). Firstly, when $k \leq p_2$, a PDP-11/40 is allocated to the process, and its actual execution time is, therefore, simply given by:

$$x_k = x \quad \text{if } k \leq p_2 . \tag{4.2}$$

Next, assume that $p_2 < k \leq p = p_1 + p_2$. In this case, the process is allocated a PDP-11/40 the fraction $\frac{p_2}{k}$ of the time, and it is allocated a PDP-11/20 the fraction $\frac{k-p_2}{k}$ of the time. This means that 1 unit of actual execution time contributes to $\frac{1}{\rho} \frac{k-p_2}{k} + \frac{p_2}{k}$ units of (PDP-11/40) time toward the total time x. We then have:

$$x_k = \frac{\rho . k}{k - p_2 + \rho . p_2} x \quad \text{if } p_2 < k \leq p = p_1 + p_2 . \tag{4.3}$$

Lastly, if $k > p = p_1 + p_2$, let us assume, as it is evidenced in the experiments, that the processes are not in competition for a processor (i. e., at least $k-p$ processes are always waiting for entering the critical section). With the same argument as above, we find, in this case, that:

$$x_k = \frac{\rho . p}{p_1 + \rho . p_2} x \quad \text{if } k > p = p_1 + p_2 . \tag{4.4}$$

This shows that, in each of the three cases, the average execution time x_k can be expressed as:

$$x_k = x.\varphi_k ,$$

where the factor φ_k is deduced from equations (4.2), (4.3) and (4.4).

We can now find an approximation in the form $(a + b \frac{1}{k}) \varphi_k$ for the average execution times of the evaluation section and of the critical section in the implementation with k processes (denoted by τ_k and c_k, respectively). We determine the values a and b using a least square approximation to the values in Table 4.2 (a) and (b). We find that:

$$\tau_k = (82.89 + 1207.73 \frac{1}{k}) \varphi_k , \quad (4.5)$$

$$c_k = (7.972 + 39.907 \frac{1}{k}) \varphi_k . \quad (4.6)$$

Using $p_1 = 5$ and $p_2 = 5$ (and $\rho = 1.4$) in the evaluation of the factor φ_k, we find that, for $k = 1, 3, 6$ and 12, the values obtained from equations (4.5) and (4.6) are consistently within 15% of the experimental results. In addition, these two equations provide us with some estimates for τ_k and c_k which are a useful complement to the values of Table 4.2, for other values of k.

4.3 - Global results

In this section, we report the global measurements of the parallel implementations with k processes for the iterative methods that we have presented in Section 3. Jacobi's, the AJ and the AGS methods have been implemented on C.mmp with a configuration of $p = 6$ processors (4 PDP-11/20 and 2 PDP-11/40), and all the experiments have been run with $k = 1, 2, 3, 4, 6, 7, 8, 9, 12$ and 14 processes. The PA method has only been implemented later, by Raskin [48], on Cm* [59] (along with the first three methods), and the results we present below for this method are the results of his measurements. A comparison between the results of C.mmp and of Cm* for the three other methods showed a complete agreement, and we have normalized the timings of the PA method so that it coincides with those of the AGS method for the implementation with 1 process (since, in

this case, both methods reduce to Gauss-Seidel's method). The configuration of Cm* included *8* processors (LSI-11) at the time of the experiments, and the PA method has been implemented with $k = 1, 2, 3, 4, 6, 7$ and 8 processes. (The results corresponding to *7* and *8* processes cannot be compared with the results obtained on C.mmp, and they are indicated with dashed lines in all the figures.)

In Figure 4.13, we present the total running times for the various methods as a function of the number of processes used in the parallel implementation.

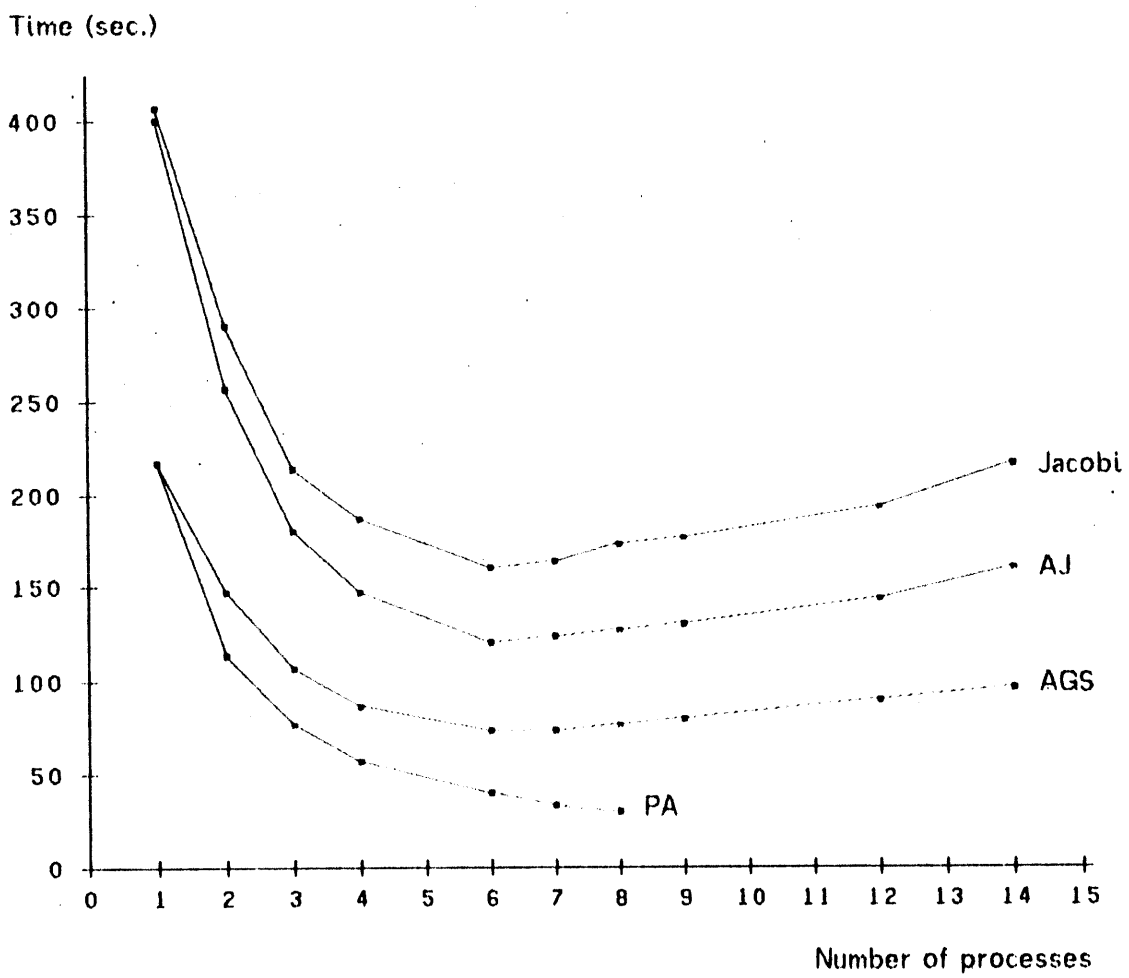


Figure 4.13 - Total execution times with Jacobi's, the AJ, the AGS and the PA methods

This direct comparison is somewhat "unfair" vis à vis Jacobi's and the AJ methods since we know that, for the particular problem we are considering, Gauss-Seidel's method is already twice as fast as Jacobi's method. In Figure 4.14, we have reported the relative

variation of the running time (i. e., t_1/t_k where t_k is the running time when k processes are used). This is also a measure of the speed-up achieved in using k processes.

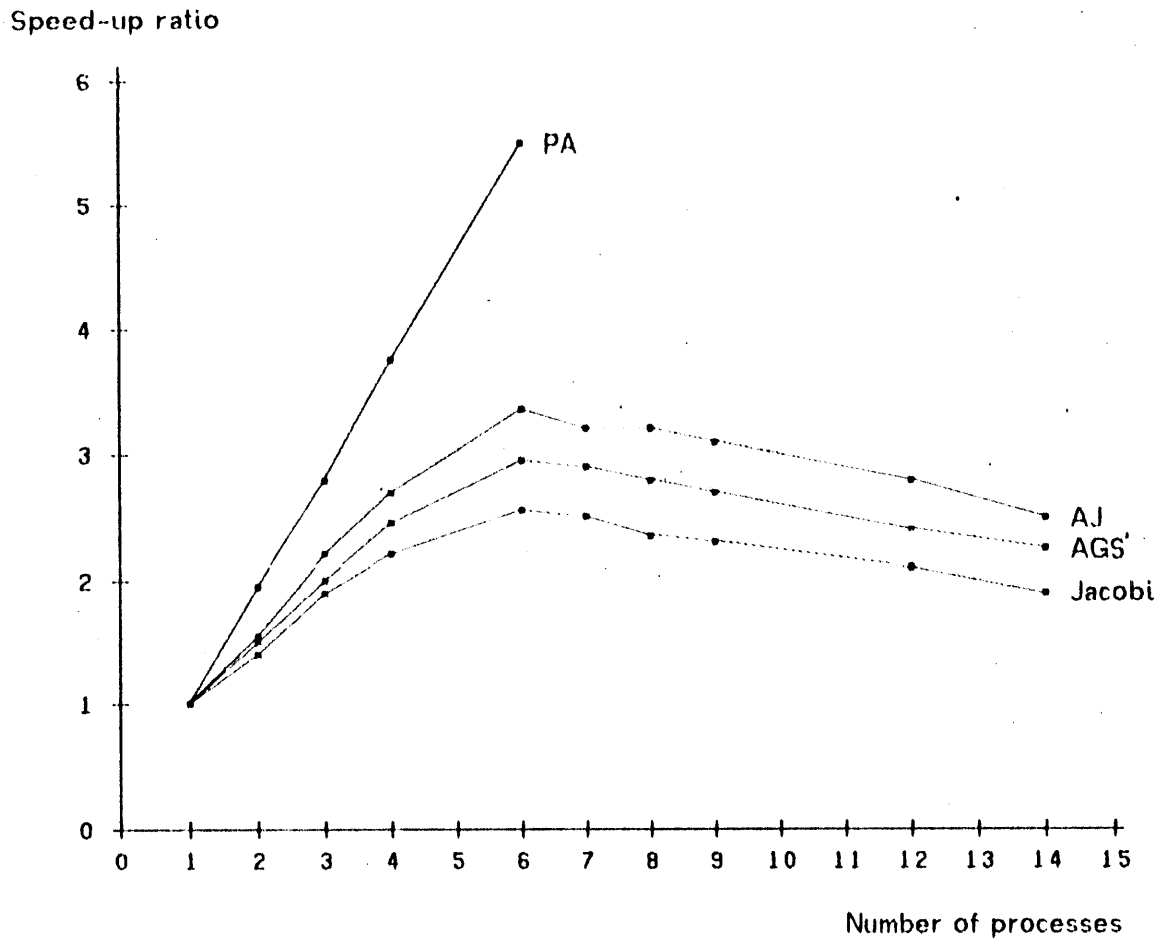


Figure 4.14 - Relative improvements with Jacobi's, the AJ, the AGS and the PA methods

Figure 4.14 shows clearly the effects of using the different forms of synchronization in a parallel algorithm. Due to the full synchronization of all processes at each step of the iteration, Jacobi's method exhibits the worst behavior of all four methods, while the PA method, which uses no synchronization at all, achieves an almost optimal speed-up.

Although the AJ and AGS methods are very similar in nature, Figure 4.14 shows that the speed-up ratios achieved by the two methods differ substantially. This difference is mainly due to the fact that the total number of iterations increases only slightly with the number of processes for the AJ method, while the increase is more important for the AGS

method. This is illustrated in Figure 4.15 where we have plotted the number, $N(k)$, of iterations required to solve our system using k processes as a function of k.

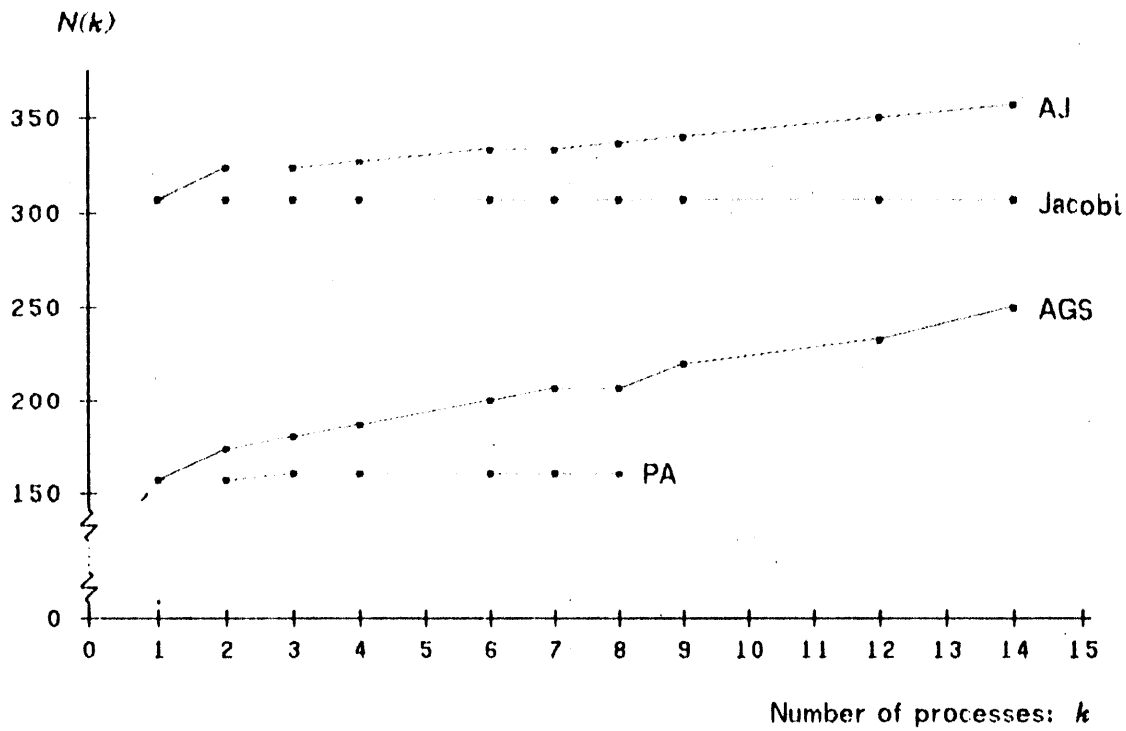


Figure 4.15 - Number of iterations required to solve the system

Figure 4.15 shows that for the AJ, AGS and PA methods $N(k)$ increases regularly (and almost linearly) with k. This difference with respect to the sequential method (Jacobi's or Gauss-Seidel's method) is one of the factors that determine the total running time of the various methods, but, obviously, the presence (or absence) of synchronization is another important factor. When the number of processes increases, a critical section, for instance, acts as a bottleneck, which tends to decrease the parallelism and increase the total execution time. In the next section, we proceed to the evaluation of this factor.

5 - On the analysis of algorithms for asynchronous multiprocessors

We want to illustrate in this section that the analysis of parallel algorithms for asynchronous multiprocessors can benefit from techniques developed in the framework of other general theories. We show that some simple results of order statistics (see, for

example, [14]) and of queueing theory (see, for example, [33]) can be used effectively in the analysis of algorithms for multiprocessors.

As examples of multiprocessors algorithms, we use in this section some of the asynchronous iterative methods described in Section 3. We use the parallel implementation of Jacobi's method (Section 3.1) as a typical example of a *synchronized algorithm*, and we use the AJ and AGS methods (Section 3.2 and 3.3) as typical examples of *asynchronous algorithms* in which communication takes place through the use of a critical section.

The evaluation of the performance of an asynchronous iteration depends principally on two main factors. The number of iteration steps required to solve the system of equations within some given admissible error ϵ is one of the important factors which determine the global running time of an iterative method. This number can be derived through the tools of numerical analysis, and we will not be concerned with its evaluation in this section We will simply use the empirical results observed in the experiments themselves. (Upper bounds on the number of iteration steps for various asynchronous iterative methods have been derived in Section 6 of Chapter III. In the case of Jacobi's method, the exact number of iterations can, in fact, be derived from the theory.) The (average) time for each process to execute a complete cycle (i. e., from the instant it starts an evaluation to the instant it starts the next evaluation) is another important factor contributing to the global running time. This factor is evaluated in the present section.

We assume throughout that the execution times for the evaluation section by all k processes are independent identically distributed random variables distributed according to the probability distribution F_k, associated with the density function f_k. Let τ_k and σ_k denote their mean and variance, respectively. Similarly, we assume that the execution times for the critical section by all k processes are independent identically distributed random variables distributed according to the probability distribution G_k, associated with the density function g_k. Let c_k denote their mean. Estimates for the quantities τ_k and c_k

are given in equations (4.5) and (4.6); an estimate for the quantity σ_k can be derived similarly.

In Section 5.1, we consider Jacobi's method and, in Section 5.2, the AJ and AGS methods. The results derived in these two sections are compared, in Section 5.3, with the experimental results.

5.1 - Synchronized algorithms

It follows from our parallel implementation of Jacobi's method that each process cooperating in the evaluation of an iterate has the cyclic behavior depicted in the diagram of Figure 5.1.

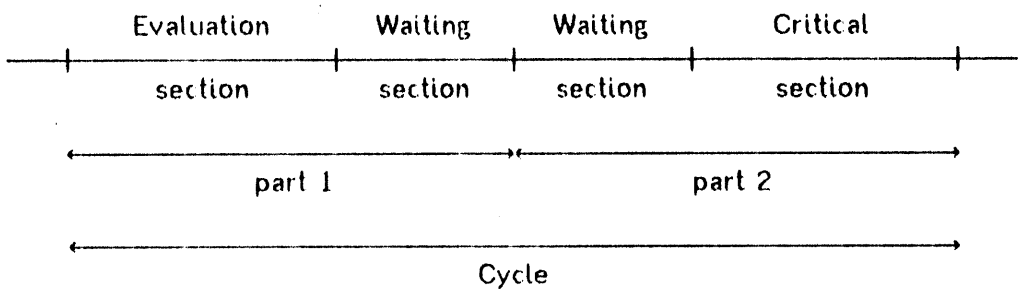


Figure 5.1 - Cyclic pattern of a process with Jacobi's method

The first waiting section is due to the full synchronization of all processes at the end of the evaluation of an iterate and before the evaluation of the next iterate. The second waiting section is simply due to the presence of the critical section used for updating and reading the values of the components of the current iterate. (A process might have to wait if another process is already executing the critical section.) The average time t_k to execute a complete cycle in the parallel implementation with k processes can, therefore, be decomposed as:

$$t_k = a_k + b_k , \tag{5.1}$$

where a_k and b_k are the average execution times for the first and second parts of the cycle respectively.

Let us first consider the quantity a_k. It corresponds to the largest finishing time of the evaluation section by the k processes. When $k \leq p$, therefore, a_k is simply given by the average of the maximum of k independent random variables distributed according to the same probability distribution F_k, and we have (see, for example, [14, p. 46]):

$$a_k = \int_0^\infty t.d[F^k(t)] = \int_0^\infty [1 - F^k(t)].dt \,, \tag{5.2}$$

where, for clarity, the index k has been dropped from F_k. Let us examine some probability distributions F_k for which analytical results can be derived from equation (5.2).

(i) Exponential distribution with parameter $\mu = \frac{1}{\tau_k}$. Using simple integral calculus, equation (5.2) yields:

$$a_k = \int_0^\infty [1 - (1-e^{-\mu t})^k].dt = \frac{1}{\mu} \int_0^1 \frac{1 - u^k}{1 - u}\, du$$

$$= \frac{1}{\mu} \int_0^1 \sum_{1 \leq i \leq k} u^{i-1}.du = \frac{1}{\mu} \sum_{1 \leq i \leq k} \frac{1}{i} \,,$$

$$= \frac{1}{\mu} H_k = H_k.\tau_k \,, \tag{5.3}$$

where H_k is the k-th harmonic number.

(ii) Uniform distribution over the interval $[\tau_k - \sigma_k\sqrt{3}, \tau_k + \sigma_k\sqrt{3}]$ (i. e., with mean τ_k and standard variation σ_k). Integration of equation (5.2) yields, in this case (see, for example, [14, p. 27]):

$$a_k = \tau_k + \frac{k-1}{k+1} \sigma_k \sqrt{3} \,. \tag{5.4}$$

Similar results can be obtained for other probability distributions F_k, but unfortunately they usually cannot be expressed so easily. For most common probability distributions F_k, however, a_k is shown to be in the form $a_k = \tau_k + \alpha_k.\sigma_k$ (as is the case in equation (5.4), for example), where the coefficient α_k (which depends on F_k) can be found in many numerical tables. (See, for example, [14, p.50] for a short table listing α_k in the case of the normal and the uniform distributions.)

When $k > p$, the quantity a_k cannot be obtained directly from equation (5.2) since, as long as i processes, with $p < i \leq k$, have *not* completed their evaluation sections, they are in competition for the p processors available, and they are, therefore, slowed down by the

factor $\frac{p}{i}$. Let x_i, for $1 \le i \le k$, be the i-th smallest execution time required by the k processes. The first process to complete its evaluation section has to share the p processors with the remaining $k-1$ processes during its entire execution. It finishes therefore after a time $y_1 = \frac{k}{p} x_1$. Similarly, the second process to complete its evaluation section, finishes after a time $y_2 = y_1 + \frac{k-1}{p}(x_1 - x_2)$. The last process to complete its evaluation section finishes after a time:

$$a_k = \frac{k}{p} x_1 + \frac{k-1}{p}(x_2 - x_1) + \ldots + \frac{p+1}{p}(x_{k-p} - x_{k-p-1}) + (x_k - x_{k-p}) . \quad (5.5)$$

The quantities x_i, for $1 \le i \le k$, can be evaluated directly from the distribution function F_k, and we have (see, for example, [14, p. 25]):

$$x_i = k \binom{k-1}{i-1} \int_0^{+\infty} t\, F^{i-1}(t)\, [1-F(t)]^{k-i}\, dF(t) , \quad (5.6)$$

where, for clarity, the index k has been dropped from F_k. Again, x_i can be evaluated explicitly for some distribution functions F_k. In particular, we have the following results.

(i) Exponential distribution with parameter $\mu = \frac{1}{\tau_k}$. Integrating equation (5.6) by parts and solving a recurrence relation, we find that:

$$x_i = \frac{1}{\mu} \sum_{k-i+1 \le r \le k} \frac{1}{r} = [\, H_k - H_{k-i} \,]\, \tau_k ,$$

where H_0 is defined to be 0. We deduce immediately from equation (5.5) that:

$$a_k = [\, \frac{k-p}{p} + H_p \,]\, \tau_k . \quad (5.7)$$

(ii) Uniform distribution over the interval $[\tau_k - \sigma_k\sqrt{3}, \tau_k + \sigma_k\sqrt{3}]$. From [14, p. 27]), we obtain:

$$x_i = \tau_k - \frac{k-2i+1}{k+1} \sigma_k \sqrt{3} .$$

We deduce immediately from equation (5.5) that, in this case:

$$a_k = \frac{k}{p} \tau_k + \frac{p-1}{k+1} \sigma_k \sqrt{3} . \quad (5.8)$$

Again, for other probability distributions F_k, equation (5.6) can always be integrated numerically, and, for most probability distributions, numerical tables are available (see, for example, [60] for the normal distribution).

Let us now consider the quantity b_k of equation (5.1). Since all processes will try to access the critical section at the same time (when the last process completes its evaluation), b_k is simply given by:

$$b_k = \frac{1}{k}(c_k + 2c_k + \ldots + kc_k) = \frac{k+1}{2} c_k$$

Table 5.1 summarizes the results of this section and presents, for $k = 1, 3, 6$ and 12, the average time t_k for a complete cycle when the distribution F_k is exponential, normal and uniform. In these three cases, the parameters τ_k and c_k are taken directly from the estimates derived in Section 4.2.2; σ_k has been estimated in the same way. These results are compared to the results derived from the experiments presented in Section 4.3. (All timings in the table are given in ms.)

	$k = 1$	$k = 3$	$k = 6$	$k = 12$
Exponential:	1338.50	1087.99	923.27	872.88
Normal:	1338.50	694.90	513.73	604.39
Uniform:	1338.50	696.74	511.37	589.70
Experimental:	1327.47	700.20	515.96	629.42

Table 5.1 - The average execution time for a complete cycle with Jacobi's method

We notice that the exponential distribution certainly does not predict adequately the experimental results. A reason for this discrepancy is that the exponential distribution does not take into account the standard deviation σ_k, which is a direct measure of the fluctuations in the execution times of the evaluation section. These fluctuations have an important role in the case of Jacobi's method since the processes (in the first part of their cycles) synchronize themselves on the largest execution time. The results obtained with the normal and uniform distribution, on the other hand, show a fair agreement with the experimental results; the difference, in this case, is partly due to the fact that the experiments have not always been run in a consistent manner (for instance, the results presented in Section 4.2 and 4.3 have not been obtained with the same number of processors).

5.2 - Asynchronous algorithms

In the parallel implementations of the AJ and AGS methods, the processes

cooperating in the evaluation of an iterate have the cyclic behavior depicted in Figure 5.2. In this case, the waiting section is only due to the presence of the critical section.

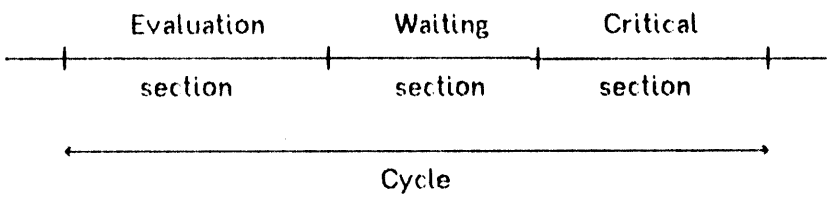
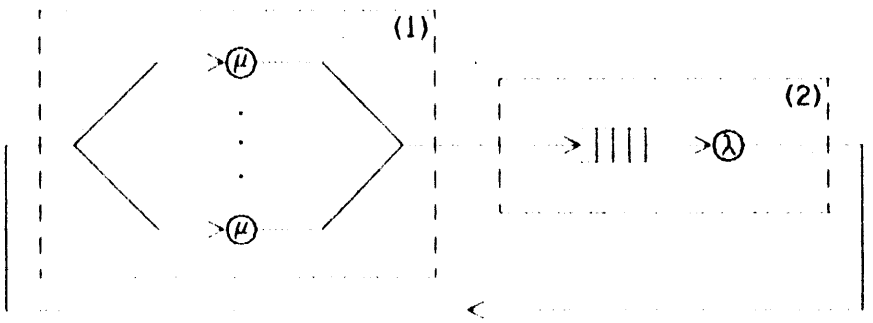


Figure 5.2 - Cyclic pattern of a process with the AJ and AGS methods

The parallel implementation with k processes on p processors can be modeled by the queueing system of Figure 5.3.



(a) k customers in the whole system: our processes;

(b) p servers in system (1): the evaluation section;

(c) 1 server in system (2): the critical section;

(d) with the restriction that at most p servers are active at the same time in the entire system.

Figure 5.3 - A queueing system for asynchronous algorithms

This queueing system has been extensively studied in the case $k = p$ as a model of time-shared processor [55], [33], when the two probability distributions F_k and G_k are exponential. We show that the results can be extended to the case $k \geq p$.

Let us assume that F_k and G_k are exponential distributions with parameter $\mu = 1/\tau_k$

and $\lambda = 1/c_k$, respectively. For $i = 0, 1, ..., k$, let q_i be the steady state probability that i customers be in system (1) of Figure 5.3 (i. e., i processes are executing their evaluation sections, while $k-i$ processes are ready to execute the critical section). Let π_0 denote the probability that no process be executing the critical section, either because all processes are within their evaluation sections or, possibly, because *no processor is allocated to a process ready to execute the critical section.*

We assume throughout that, if there exists at any time in the entire system i processes, with $i > p$, which are not blocked (waiting for another process to complete the critical section), each of the i processes receives the same fraction $\frac{p}{i}$ of the computing power. It follows directly that the probability π_0 is given by:

$$\pi_0 = q_k + \sum_{p \le i \le k-1} \frac{i - p + 1}{i + 1} q_i . \tag{5.9}$$

Theorem 5.1:.

Assume that $k \ge p$. The average time t_k required to execute a complete cycle is given by:

$$t_k = k\, c_k / (1-\pi_0) , \tag{5.10}$$

where π_0 is the probability that the server of system (2) be idle (i. e., no process is executing the critical section, although some may be blocked because no processors are available). If we assume that each process which is not blocked receives an equal share of the computing power, the probabilities q_i, for $i = 0, 1, ..., k$, satisfy:

$$q_i = \begin{cases} q_i & \text{if } i \le k , \\ (i+1)\, \dfrac{(k-1)!}{i!}\, \beta^{k-i}\, q_k & \text{if } p \le i \le k-1 , \\ p\, \dfrac{(k-1)!}{i!}\, \beta^{k-i}\, q_k & \text{if } 0 \le i \le p-1 . \end{cases} \tag{5.11}$$

Proof:

Equations (5.10) and (5.11) are immediate consequences of simple results of queueing theory. Equation (5.10) follows directly from Little's formula (see, for example, [33, p.17]) by considering the throughput of system (2). Equation (5.11) also follows directly from the fact that (under the exponential assumption for both F_k and G_k) the

system of Figure 5.3 corresonds to a *pure birth-death process* (see, for example, [33, p.89]). ■

The average execution time, t_k, for a complete cycle can now be evaluated from the results of Theorem 5.1 using equation (5.9) and the fact that:

$$q_0 + q_1 + \ldots + q_k = 1 .$$

5.3 - A comparison with the experimental results

The results of Sections 5.1 and 5.2 provide us with an estimate of the average time t_k required to execute a complete cycle in the parallel implementation with k processes of Jacobi's method and of the AJ and AGS methods. In order to evaluate the total running time T_k for the three methods, we also need some estimate of the number of iterations N_k required by each of the methods in the parallel implementation with k processes. In the case of Jacobi's method, N_k does not depend on k and can be computed analytically from the spectral radius, $\rho(B)$, of the Jacobi matrix. In the case of the AJ and AGS methods, we have simply chosen to take directly the number of iterations observed in the experiments themselves.

The total running time $T_k = N_k \cdot t_k$ now follows immediately. The resulting values are plotted in Figure 3.3, along with the values observed from the experiments. (In the case of Jacobi's method, t_k is evaluated using for F_k a uniform distribution.)

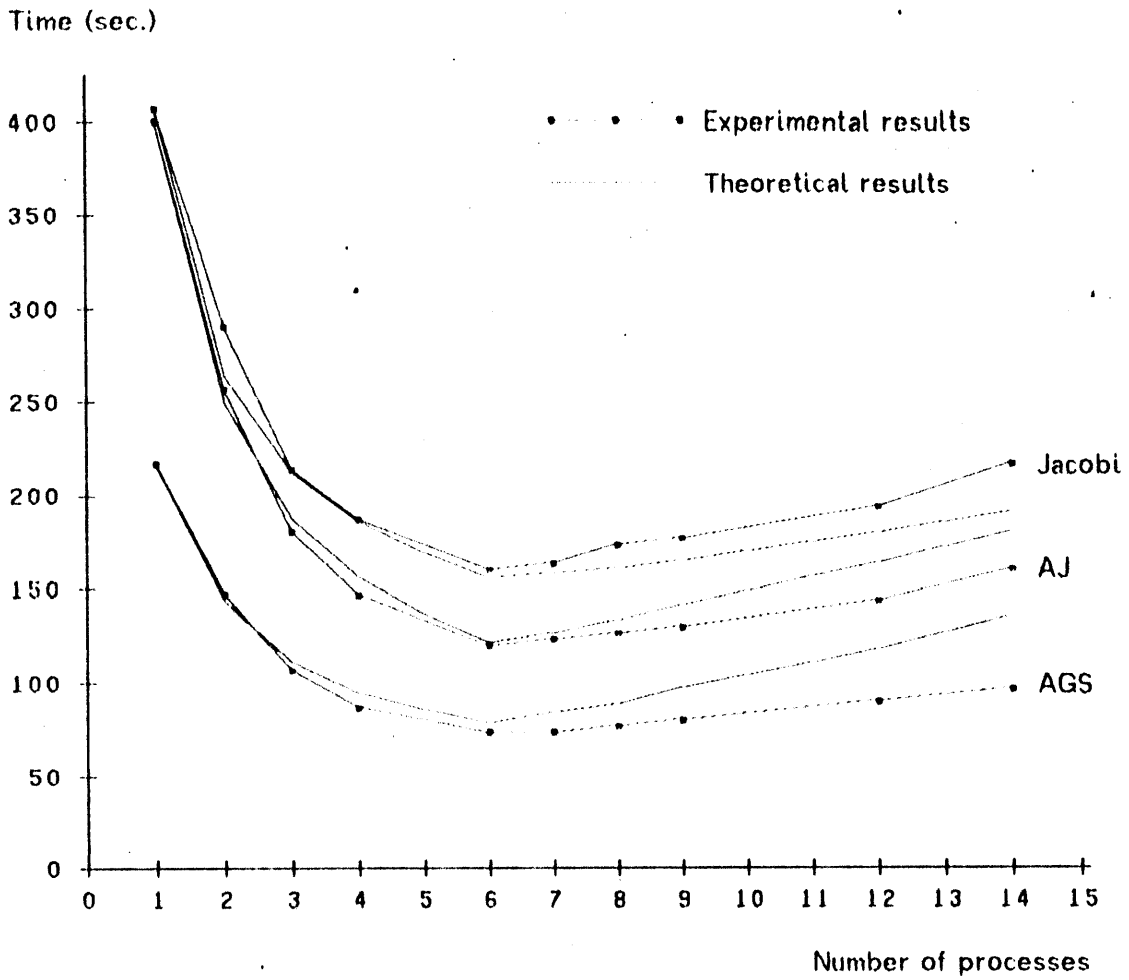


Figure 5.4 - Experimental and theoretical running times

We see that the "theory" matches fairly well the actual measurements especially in the case of most interest, i. e., when $k \leq p$ (clearly we cannot expect any gain from using more processes than processors). In particular, if we rely on our model, at least for $k \leq p$, we can compute the optimum value for k (beyond which no gain is obtained), and we find, in particular, that

$$k_{opt} = \begin{cases} 14 & \text{for Jacobi's method,} \\ 15 & \text{for the AJ method,} \\ 12 & \text{for the AGS method.} \end{cases}$$

6 - Concluding remarks

The actual implementation of parallel algorithms on an asynchronous multiprocessor has proved to be an invaluable help for providing us with a better understanding of parallel algorithms, for illustrating some of the notions and concepts associated with these algorithms, and for supporting some of the assumptions that we have introduced in their analysis. In particular, the figures of Section 4.2.1 show clearly that the execution time of a program can hardly be regarded as a constant, and that it is more accurate to consider this execution time as a random variable distributed according to some probability distribution. In view of the histograms presented in Figures 4.2 through 4.9, an Erlang or a normal distribution seems to be a reasonable approximation, in our case, to account for the fluctuations in the execution times of the programs that we have implemented on C.mmp.

These experiments also constitute a clear illustration of the advantage of purely asynchronous algorithms over synchronized algorithms. To give a quantitative evaluation of the effects of synchronization, assume that it takes *1* unit of time for a process to perform one step of the iteration (excluding any overhead). Then, it follows from the results we have presented that, in a parallel implementation with 6 processes, it will take each process an average of about *1.05*, *1.62* and *2.34* units of time with the PA, the AJ and Jacobi's methods, respectively, to perform the same step of the iteration (for that matter, both the AJ and the AGS methods have the same behavior). While the overhead in the PA method (about 5%) is mainly due to memory contention, the overheads in the AJ and Jacobi's methods measure almost directly the effects of using critical sections and of using full synchronization between the processes, respectively.

In addition to the experiments reported in this chapter, we have also run some other experiments to consider the effect of the introduction of a relaxation factor in the different iterative schemes. These results confirmed exactly the simulation results obtained by Rosenfeld and presented in [52]. In particular, while we are guaranteed of

the convergence of any asynchronous iterations when we use a relaxation factor ω in the range $0 < \omega < 2/[1+\rho(B)]$, this is not so when $\omega \geq 2/[1+\rho(B)]$, and divergence was, indeed, often observed (for the problem that we have considered, $\rho(B) \sim 0.991$, thus $2/[1+\rho(B)] \sim 1.005$). It seems to be very useful to obtain more (experimental or analytical) results on the effects of using relaxation factors, since our experiments show that (when convergence is achieved) it is a very promising way to accelerate the iteration.

The results presented in Section 5 are also an interesting aspect of this chapter. We have shown how simple techniques from order statistics and queueing theory could be adapted to the analysis of algorithms for asynchronous multiprocessors. The analysis that we have developed gives a fair account of the experimental results. This is very useful in practice since it can be used to predict the optimal decomposition of a problem (i. e., the optimal number of processes to create in order to, for example, minimize the overall execution time).

Chapter VI

Conclusion

1 - A summary of the results and their implications

An evident advantage of using asynchronous multiprocessors, and parallel computers in general, rather than conventional uni-processors, is to be able to substantially reduce the execution time required for solving a problem. Given a particular parallel computer, therefore, one of the first goals in designing a parallel algorithm for solving a problem is to try to minimize the required execution time on the given machine. This leads us naturally to consider the execution time of a parallel algorithm as one of the primary measures of the performance of the algorithm.

When we consider a *sequential* algorithm for solving a given problem, say, sorting or matrix multiplication, the number of comparisons or the number of scalar multiplications performed by the algorithm is usually used as the measure of complexity of the algorithm. In this respect, parallel algorithms for SIMD machines are very similar to sequential algorithms, in the sense that, in this case, the number of *parallel instructions* (e. g., *parallel comparisons* or *parallel multiplications*) is the usual complexity measure of an algorithm. The intuitive reason for this cost measure with both sequential algorithms and parallel algorithms for SIMD machines is that the execution time in these two types of algorithms is directly related to the number of instructions executed, and that, therefore, it is realistic to only count those instructions for performance evaluation purposes.

When we are dealing with a parallel algorithm for asynchronous multiprocessors,

however, its non-deterministic behavior contributes to making its analysis drastically different from the analysis of a sequential algorithm. In particular, there usually does not seem to exist a direct relation between the (average) execution time of a parallel algorithm for multiprocessor and the number of instructions executed by each of the processes. As an illustration, let us examine again Jacobi's method for solving a linear system of n equations, and consider a parallel implementation with k processes in which each process evaluates $q = n/k$ components. Let us first choose, as a measure of performance for this implementation, the number of *parallel evaluations* of a component (or, within a factor of n, the number of *parallel multiplications*). The immediate conclusion, in this case, is that, in order to decrease the cost of the algorithm, we should always increase the number of processors. Let us now consider directly the total average time T_k required to perform one step of the iteration with the parallel implementation with k processes. Assume, as before, that the execution times for the evaluation of q components by all k processes are independent identically random variables distributed according to an exponential distribution with mean τ_k. Then, due to the synchronization between the processes, the total average time for one iteration step is given by $T_k = H_k.\tau_k$, where H_k is the k-th harmonic number. Let us further assume that τ_k is of the form $\tau_k = a + \frac{1}{k} b$ (which is natural in view of our decomposition). Then, it follows that for large k, the total average time grows with k like $a.\ln(k)$ and, thus, increases as the number of processes increases. Therefore, we conclude, in this case, that there exists a (finite) number k of processes which minimizes the total average time T_k. This is in contradiction with the conclusion derived from using the other cost measure.

This example shows that the analysis of the efficiency of a parallel algorithm for asynchronous multiprocessors usually requires techniques very different from those previously developed in the analysis of sequential algorithms or parallel algorithms for SIMD machines. We think that one of the main contributions of this thesis is to have presented and used very diverse techniques applicable in the analysis of parallel algorithms for asynchronous multiprocessors. These techniques are used in various

applications areas. The analyses developed in Chapter II Section 5 and in Chapter IV Section 7.3.1, for instance, are related to some analyses commonly found in Operations Research, while the treatment of Section 6 of Chapter II applies some techniques typical of renewal theory. In Chapter III Sections 6 and 8, the complexity of asynchronous iterative methods is derived using the tools of numerical analysis (this is obviously due to the nature of the problem treated in this chapter).

We also have presented in Chapter V Section 5 some of the techniques which seem to be most typical of the analysis of parallel algorithms for multiprocessors, namely techniques drawn from order statistics and from queueing theory. An important advantage of this approach is that a large number of results are available from well developed theories. Most of these results are directly applicable to the analysis of parallel algorithms for asynchronous multiprocessors, and we have shown, in particular, that a very simple queueing model (initially intended to represent a time-shared uni-processor) accounts appropriately for the behavior of an asynchronous parallel algorithm in which the processes communicate among themselves through the use of a critical section. These results can be used to predict the optimal decomposition of a problem (i. e., the optimal number of processes cooperating in the solution of the problem). Some other examples of the use of queueing theory to the analysis of parallel algorithms for multiprocessors are also presented in [51] with various applications to sorting algorithms.

A deficiency common to several of the analyses that we have presented is that, in some cases, strong assumptions must be made in order to be able to carry out the analysis of an algorithm. In Chapter II Section 5 and in Chapter V Section 5.2, for instance, our results are based on the assumption that the various execution times are exponentially distributed. We have observed, however, that whenever we were also able to derive an analysis of an asynchronous algorithm based on other (more realistic) probability distributions (see Chapter II Section 6, for instance), the results did not show any substantial differences with the results derived from the exponential distribution.

Moreover, the analytical results derived in Chapter V Section 5.3 are in excellent agreement with the experimental results that we have presented in Chapter V. Therefore, it seems that, although the exponential distribution is not necessarily a very realistic assumption for the distribution of the execution times, it still provides us with useful results for asynchronous algorithms. In the case of synchronized algorithms (see Chapter V Section 5.1), however, analytical results obtained with the exponential distribution do not show an excellent agreement with the experimental results, whereas a closer approximation is achieved with the normal and the uniform distributions. A reason for this discrepancy is that the fluctuations are measured directly by the standard deviation of the probability distribution and this cannot be captured by the exponential distribution (for which the standard deviation is the same as the mean).

Another very important aspect of the thesis is to have presented and illustrated some of the notions and concepts unique in the design of parallel algorithms for asynchronous multiprocessors. The algorithm proposed in Chapter II, for example, illustrates an a priori very counter-intuitive idea that the execution of a purely sequential program can be sped-up on an asynchronous multiprocessor without introducing any parallelism within the program itself. The acceleration is achieved by decomposing the program into a succession of tasks (executed serially), and by taking advantage of the *fluctuations* in the execution times of the tasks. These fluctuations in computing times represent a dimension unique in the design of parallel algorithms for asynchronous multiprocessors. Their consequences are twofold. A negative aspect is evidenced with the example of Jacobi's method presented in the introductory chapter; the net effect, in this case, is to create a substantial overhead due to the use of a full synchronization of the processes. The algorithm of Chapter II, on the other hand, demonstrates that the fluctuations in the computing times can actually be used to accelerate the execution of a program. Although we do not feel that the algorithm in this chapter should be used directly as it is presented, we think that the idea embedded into the algorithm can be used together with other considerations, such as reliability, in the construction of asynchronous

algorithms. Probably the most important aspect of the algorithm presented in Chapter II is that it illustrates the fact that innovations are required for the design of parallel algorithms for asynchronous multiprocessors.

The experimental results presented in Chapter V are fundamental in the thesis. They lend us insight into the behavior of parallel programs executed on an asynchronous multiprocessor; and, with a better understanding of their behavior, we can expect to be able to design better parallel algorithms for multiprocessors. In addition, they have been particularly useful in validating some of the assumptions that we have made in our analyses. These experimental results are important in another practical aspect, namely, they provide us with a quantitative comparison of the different uses of synchronization.

The results that we have mentioned so far contribute directly toward the general goal of the thesis: design and analysis of parallel algorithms for asynchronous multiprocessors. Some of the results of the thesis seem to be of theoretical and practical importance in their own rights.

In Chapter III, for instance, we have introduced the class of *asynchronous iterative methods* to remove the need for synchronization in the implementation of iterative methods on a multiprocessor. We think that the results presented in this chapter are a contribution to the area of iterative methods, and, in particular, they provide some extensions and generalizations of previously published results [11], [41], [42], [43], [50]. Theorem 4.1, for example, extends the convergence results obtained by Chazan and Miranker for chaotic iterations [11], by relaxing a technical condition that they had introduced; furthermore, our results also provide a generalization to non-linear operators. The results of Section 5, on the class of *asynchronous iterative methods with memory*, also generalizes some of the results obtained by Miellou [42].

Chapter IV contains some important results concerning the α-β pruning algorithm. We have shown in the first part of this chapter that the branching factor of the

α-β pruning algorithm in a uniform game tree of degree n is $\Theta(n/\ln n)$, when all bottom values are assigned independent identically distributed random variables. This confirms a claim by Knuth and Moore [35] that deep cut-offs only have a second order effect on the behavior of the algorithm. The results of the second part constitute the main contribution of Chapter IV. We have proposed in this part an asynchronous parallel implementation of the α-β pruning algorithm. Our analysis of the parallel implementation with k processes shows, rather surprisingly, that the speed-up is larger than k. This implies that the (sequential) α-β pruning algorithm is not optimal and can be substantially improved upon. This particular result, which has been obtained very indirectly in the thesis, might find applications in the area of Artificial Intelligence.

2 - Some topics for future research

We certainly do not believe that we have covered in this thesis every possible aspect of the design and the analysis of algorithms for asynchronous multiprocessors. Clearly, much research remains to be done in this area, and this section mentions several topics for future research.

We think that the thesis has clearly illustrated an important characteristic of algorithms for multiprocessors, namely, the a priori unpredictable behavior in their execution. This characteristic, therefore, makes it an absolute requirement to consider very carefully the correctness of parallel algorithms for multiprocessors, and research in this area would certainly be very useful. We are (personally) convinced that every algorithm proposed in this thesis performs correctly, and we have also given (we hope) convincing arguments for their correctness. However, in each case, the *proof* of correctness is based on techniques which are, usually, only adequate to the problem at hand. A formal (and general) theory would certainly be a very useful tool for the design of algorithms for multiprocessors.

Probably, the greatest emphasis of the thesis has been placed on the analysis of

parallel algorithms for asynchronous multiprocessors, and we have presented (and used) diverse techniques which appear to be applicable to numerous problems. , Those techniques have proved to be effective to the algorithms presented, but we think that most of them could still be improved upon, in particular with regard to the generality of their applications. Possible generalizations in this area would include, for instance, the relaxation of some of the assumptions used in the various analyses that we have presented. The execution time of an algorithm has been regarded in most of the thesis as the primary measure of complexity of the algorithm. While this measure is, in fact, of primary importance in real time applications, other complexity measures should also be considered. Processor utilization, for example, would be another meaningful measure of performance, particularly if an asynchronous multiprocessor is used in a multi-user environment. In this case, it would also be of interest to consider the possibility of increasing the processor utilization by multiprogramming several programs (for example, several instances of the same parallel algorithm).

The experiments presented in Chapter V have proved to be an invaluable tool. In general, direct experimentation on an asynchronous multiprocessor can be very useful especially when it is difficult to derive any analytical results. In particular, it would be very interesting to perform more experiments with asynchronous iterations, for example, to consider the effects of using a relaxation factor. Other experiments could also be performed to evaluate some of the *adaptative asynchronous iterations* described in Section 3.4.2 of Chapter V.

The parallel implementation that we have proposed for the α-β pruning algorithm appears to be very efficient when few processes are used, but the maximum speed-up achievable with this method is typically limited to 5 or 6 even with an infinity of processes. It does not seem that a direct adaptation of the α-β pruning algorithm into a parallel algorithm is the best approach to follow, particularly because it is based on a depth first search, which is inherently sequential. A better approach would probably be

to consider a game tree searching algorithm based on a best first search along with a preliminary evaluation of the internal nodes.

Lastly, we view this thesis as a first step towards a systematic study of the issues raised by the design and the analysis of algorithms for asynchronous multiprocessors.

Bibliography

[1] Anderson, J. P., Hoffman, S. A., Shifman, J., and Williams, R. J., D825 - A multiple computer system for command and control, *Proceedings of the AFIPS 1962 Fall Joint Computer Conference*, Vol. 22, 1962, pp. 86-96.

[2] Andler, S., Synchronization primitives and the verification of concurrent programs, Carnegie-Mellon University, Computer Science Department Report, May 1977.

[3] Barak, A. B., and Downey, P. J., Asynchronous parallel execution of a chain of tasks with interrupts, The Pennsylvania State University, Computer Science Department Report, December 1977.

[4] Barak, A. B., and Downey, P. J., Using task duplication to reduce finishing time, The Pennsylvania State University, Computer Science Department Report, February 1978.

[5] Barnes, G. H., Richard, M. B., Kato, M., Kuck, D. J., Slotnick, D. L., and Stokes, R. A., The ILLIAC IV computer, *IEEE Transactions on Computers*, Vol. C-17, No. 8, August 1968, pp. 746-757.

[6] Baudet, G. M., Asynchronous iterative methods for multiprocessors, *Journal of the ACM*, Vol. 25, No. 2, April 1978, pp. 226-244.

[7] Baudet, G. M., On the branching factor of the Alpha-Beta pruning algorithm, Carnegie-Mellon University, Computer Science Department Report, September 1977. (To appear in *Artificial Intelligence*.)

[8] Baudet, G. M., Brent, R. P., and Kung, H. T., Parallel execution of a sequence of tasks on an asynchronous multiprocessor, Carnegie-Mellon University, Computer Science Department Report, June 1977.

[9] Baudet, G., and Stevenson, D., Optimal sorting algorithms for parallel computers, *IEEE Transactions on Computers*, Vol. C-27, No. 1, January 1978, pp. 84-87.

[10] Charnay, M., Itérations chaotiques sur un produits d'espaces métriques, Thèse de 3ème cycle, Université Claude Bernard, Lyon, 1975.

[11] Chazan, D., and Miranker, W., Chaotic relaxation, *Linear Algebra and Its Applications*, Vol. 2, 1969, pp. 199-222.

[12] Chen, T. C., Overlap and pipeline processing, in *Introduction to Computer Architecture*, ed. by H. S. Stone, Science Research Associates, Chicago, 1975, pp. 375-431.

[13] Courtois, P. J., Heymans, F., and Parnas, D. L., Concurrent control with 'readers' and 'writers', *Communications of the ACM*, Vol. 14, No. 10, October 1971, pp. 667-668.

[14] David, H. A., *Order Statistics*, John Wiley and Sons, New York, 1970.

[15] Digital Equipment Corporation, BLISS-11 programmer's manuel, DEC, Maynard, 1972.

[16] Dijkstra, E. W., Co-operating sequential processes, in *Programming Languages*, ed. by F. Genuys, Academic Press, New York, 1966, pp. 43-112.

[17] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[18] Donnelly, J. D. P., Periodic chaotic relaxation, *Linear Algebra and Its Applications*, Vol. 4, 1971, pp. 117-128.

[19] Enslow, P. H., Multiprocessor organization - A survey, *Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 103-129.

[20] Flon, L., On the design and verification of operating systems, Ph.D. dissertation, Carnegie-Mellon University, May 1977.

[21] Flynn, M. J., Very high-speed computing systems, *Proceedings of the IEEE*, Vol. 54, No. 12, December 1966, pp. 1901-1909.

[22] Forsythe, G. E., and Wasow, W. R., *Finite-Difference Methods for Partial Differential Equations*, John Wiley and Sons, New York, 1960.

[23] Fuller, S. H., Gaschnig, J. G., and Gillogly, J. J., Analysis of the alpha-beta pruning algorithm, Carnegie-Mellon University, Computer Science Department Report, July 1973.

[24] Gillogly, J. J., The Technology chess program, *Artificial Intelligence*, Vol. 3, No. 3, Fall 1972, pp. 145-163.

[25] Gillogly, J. J., Performance analysis of the Technology Chess Program, Ph.D. dissertation, Carnegie-Mellon University, March 1978.

[26] Habermann, A. N., Synchronization of communicating processes, *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 171-176.

[27] Heller, D., A survey of parallel algorithms in numerical linear algebra, Carnegie-Mellon University, Computer Science Department Report, February 1976. (To appear in *SIAM Review*.)

[28] Hibbard, P., Hisgen, A., and Rodeheffer, T., A language implementation design for a multiprocessor computer system, *Proceedings of the Fifth Annual Symposium on Computer Architecture*, Palo Alto, California, April 3-5, 1978.

[29] Hintz, R. G., and Tate, D. P., Control Data STAR-100 processor design, *Proceedings of Compcon 72, IEEE Computer Society Conference*, IEEE, New York, 1972, pp. 1-4.

[30] Jones, A. K., Chansler, R. J., Durham, I., Feiler, P. H., Scelza, D. A., Schwans, K., and Vegdahl, S. R., Programming issues raised by a multiprocessor, *Proceedings of the IEEE*, Vol. 66, No. 2, February 1978, pp. 229-237.

[31] Kantorovitch, L. V., Vulich, B. Z., and Pinsker, A. G., *Functional Analysis in Partially Ordered Spaces* (Russian), Gostekhizdat, Moscow, 1950.

[32] Kleinrock, L., Certain analytic results for time-shared processors, *Information Processing 68*, North-Holland, Amsterdam, 1969, pp. 838-845.

[33] Kleinrock, L., *Queueing Systems, Volume I: Theory*, John Wiley and Sons, New York, 1975.

[34] Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 2nd edition, 1973.

[35] Knuth, D. E., and Moore, R. W., An analysis of alpha-beta pruning, *Artificial Intelligence*, Vol. 6, No. 4, Winter 1975, pp. 293-326.

[36] Kuck, D. J., A Survey of parallel machine organization and programming, *Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 29-59.

[37] Kung, H. T., Synchronized and asynchronous parallel algorithms for multiprocessors, in *Algorithms and Complexity: New Directions and Recent Results*, ed. by J. F. Traub, Academic Press, New York, 1976, pp. 153-200.

[38] Kung, H. T., The complexity of coordinating parallel asynchronous processes, *Proceedings of the Fifteenth Annual Allerton Conference on Communication, Control, and Computing*, University of Illinois at Urbana-Champaign, 1977, pp. 34-43.

[39] Kung, H. T., and Lehman, P. L., A concurrent database manipulation problem: binary search trees, Carnegie-Mellon University, Computer Science Department Report, to appear.

[40] Kung, H. T., and Song, S. W., A parallel garbage collection algorithm and its correctness proof, *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, October 1977, pp. 120-131.

[41] Miellou, J.-C., Itérations chaotiques à retards, *Comptes Rendus de l'Académie des Sciences de Paris*, Series A, Vol. 278, April 1974, pp. 957-960.

[42] Miellou, J.-C., Itérations chaotiques à retards; études de la convergence dans le cas d'espaces partiellement ordonnés, *Comptes Rendus de l'Académie des Sciences de Paris*, Series A, Vol. 280, January 1975, pp. 233-236.

[43] Miellou, J.-C., Algorithmes de relaxation à retards, *R. A. I. R. O.*, Vol. 9, R-1, April 1975, pp. 55-82.

[44] Miranker, W. L., Parallel methods for solving equations, IBM T. J. Watson Research Center, Research Report RC 6545 (No. 28250), May 1977.

[45] Newborn, M. M., The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores, *Artificial Intelligence*, Vol. 8, No. 2, April 1977, pp. 137-153.

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

[47] Owicki, S., and Gries, D., Verifying properties of parallel programs: an axiomatic approach, *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 279-285.

[48] Raskin, L., Performance of a stand alone Cm* system, in Cm* review, ed. by S. H. Fuller, A. K. Jones, and I. Durham, Carnegie-Mellon University, Computer Science Department Report, June 1977, pp. 26-56.

[49] Robert, F., Contractions en norme vectorielle, *Linear Algebra and Its Applications*, Vol. 13, 1976, pp. 19-35.

[50] Robert, F., Charnay, M., and Musy, F., Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe, *Aplikace Matematicky*, Vol. 20, 1975, pp. 1-38.

[51] Robinson, J. T., Analysis of asynchronous multiprocessor algorithms with application to sorting, *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977, pp. 128-135. (A revised version is to appear in *IEEE Transactions on Software Engineering*.)

[52] Rosenfeld, J. L., A case study in programming for parallel processors, *Communications of the ACM*, Vol. 12, No. 12, December 1969, pp. 645-655.

[53] Rosenfeld, J. L., and Driscoll, G. C., Solution of the Dirichlet problem on a simulated parallel processing system, *Information Processing 68*, North-Holland, Amsterdam, 1969, pp. 499-507.

[54] Russel, R. M., The CRAY-1 computer system, *Communications of the ACM*, Vol. 21, No. 1, January 1978, pp. 63-72.

[55] Scherr, A. L., *An Analysis of Time-Shared Computer Systems*, MIT Press, 1960.

[56] Slagle, J. R., and Dixon, J. K., Experiments with some programs that search game trees, *Journal of the ACM*, Vol. 16, 1969, pp. 189-207.

[57] Stone, H. S., Parallel computers, in *Introduction to Computer Architecture*, ed. by H. S. Stone, Science Research Associates, Chicago, 1975, pp. 318-374.

[58] Stone, H. S., Sorting on STAR, *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2, March 1978, pp. 138-146.

[59] Swan, R. J., Fuller, S. H., and Siewiorek, D. P., Cm*: a modular multi-microprocessor, *Proceedings of the AFIPS 1977 National Computer Conference*, Vol. 46, 1977, pp. 637-644.

[60] Teichroew, D., Tables of expected values of order statistics and products of order statistics for samples of size twenty and less from the normal distribution, *Annals of Mathematical Statistics*, Vol. 27, 1956, pp. 410-426.

[61] Thompson, C. D., and Kung, H. T., Sorting on a mesh-connected parallel computer, *Communications of the ACM*, Vol. 20, No. 4, April 1977, pp. 263-271.

[62] Varga, R., *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, 1962.

[63] Wulf, W. A., and Bell, C. G., C.mmp - A multi-mini-processor, *Proceedings of the AFIPS 1972 Fall Joint Computer Conference*, Vol. 41, December 1972, pp. 765-777.

[64] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., "Hydra: the kernel of a multiprocessor operating system," *Communications of the ACM*, Vol. 17, No. 6, June 1974, pp. 337-345.

[65] Yau, S. S., and Fung, H. S., Associative processor architecture - A survey, *Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 3-27.