COMPUTER-AIDED PARALLELIZATION OF APPLICATIONS

THÈSE N^{O} 2431 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Marc MAZZARIOL

Ingénieur informaticien diplômé EPF
originaire de Genève

Jury de thèse:

Prof. R. D. Hersch, directeur de thèse
Prof. R. Guerraoui, rapporteur
Dr B. Gennart, rapporteur
Dr M. Snir, rapporteur

Lausanne, EPFL
2001

A mon père

“No Brain, No Headache”

Ann O'Nym

Acknowledgements

First and foremost, I am especially grateful to my research director, Professor Roger David Hersch, who offered me a challenging and interesting research area, who gave me guidance and support throughout this research, and spent hours reading and commenting on earlier drafts of this dissertation. His continuous support and guidance were an incomparable stimulation and helped me to focus my attention on the essentials to reach my objectives.

I am particularly indebted to Dr. Benoît Gennart and Dr. Vincent Messerli for their technical expertise and for the enlightening discussions we had, which helped me clarify my ideas and led me to new ideas. Dr. Benoît Gennart and Dr. Vincent Messerli are the main authors of the CAP/PS^2 parallelization framework. The present thesis wouldn't have been possible without their own work.

I would like to thank the members of the Peripheral System Laboratory, among them Oscar Figueiredo, Jean-Christophe Bessaud, Joaquín Táraga, Emin Gabrielyan, Sebastian Gerlach, Olivier Courtois, Itzhak Amidror, Yvette Fishman, Fabienne Allaire, and all the LSP staff for making the LSP such a nice place to work in. They made this research enjoyable on a day-to-day basis providing encouragement and motivation.

I am also thankful to my previous teachers and all the people who helped me to develop my curiosity, and who gave me a taste for research. In particular, I would like the thank Olivier Fischer, Xavier Montet and Laurent Herrmann.

I am especially grateful to all my family, my friends, and all those I met during my sport activities. All of them supported me and helped along the years, sometimes without knowing how much!

Finally I thank the Swiss National Fund and the EPFL for funding this research.

Summary

Within the scope of this thesis, we are interested in running high-performance parallel applications on clusters of commodity components, i.e. PCs or workstations. Creating parallel applications remains a difficult task. Moreover, creating efficient pipelined parallel schedules where communication, computation and I/O are carried out simultaneously is a real challenge.

We use the Computer-Aided Parallelization (CAP) framework developed at EPFL in order to facilitate the design and development of pipelined parallel applications. The CAP based parallelization approach differs from other parallelization approaches by freeing the programmer from low-level issues such as thread management, protocol management and synchronization. The programmer can concentrate his efforts on building efficient parallel schedules. Application programmers express separately the serial program parts and the parallel behavior of the program at a high level of abstraction, i.e. as a parallel schedule. The parallel schedule determines the flow of data and parameters between operations running on the same or on different processors. CAP offers asynchronous processing capabilities allowing to carry out simultaneously I/O operations, communications and computations.

In this thesis, we show how parallel applications from various domains can be developed by taking advantage of CAP: parallel linear algebra algorithms such as matrix multiplication and LU decomposition, parallel image filtering, parallel cellular automata and parallel discrete optimization algorithm such as Branch and Bound. We are also interested in the parallelization of industrial applications, such as the *Radiocontrol* application aiming at computing in parallel the listening quotes of radio stations.

The contributions of this thesis are (1) to validate the CAP C++ language extension by demonstrating its capability of synthesizing parallel programs, and (2) to show that the CAP based parallelization approach yields efficient parallel programs in different application fields.

The reader is introduced to the CAP philosophy and to the formulation of parallel schedules. We discuss flow-control and load balancing issues and propose appropriate CAP constructs. In several parallel applications, we demonstrate the compositionality of CAP and its benefits. In order to demonstrate the performance offered by CAP, we create for most applications a performance model and verify it by experimental measurements. Finally, we describe the perspectives and give ideas for pursuing this research.

Résumé

Dans le cadre de cette thèse, nous avons étudié la réalisation d'applications parallèles sur un ensemble d'ordinateurs communément disponibles sur le marché (PCs ou stations de travail). La création d'applications parallèles reste à ce jour une tâche difficile et fastidieuse. En particulier, la mise en oeuvre d'ordonnancements parallèles performants où les communications, le calcul et les entrées/sorties s'exécutent simultanément est un défi.

L'environnement de parallélisation CAP développé à l'EPFL facilite la réalisation d'applications parallèles complexes. La méthodologie de parallélisation CAP se distingue des autres outils de parallélisation en libérant le programmeur de l'implémentation de tâches de bas niveau tels que la gestion des threads, la gestion de protocol et la synchronisation. Le programmeur peut se concentrer spécifiquement sur la création d'ordonnancements parallèles complexes et efficaces. Le programmeur exprime l'ordonnancement parallèle des tâches indépendamment de la partie sérielle de l'application. Afin d'offrir un maximum de souplesse, CAP permet d'exprimer les ordonnancements parallèles avec un niveau d'abstraction suffisant. De ces ordonnancements sont déduis les dépendances et flux de données entre les tâches s'exécutant sur les différents processeurs. Afin de permettre la réalisation de schéma d'exécution où les communications, le calcul et les entrées/sorties s'effectuent simultanément, CAP offre la possibilité d'exécuter des opérations de façon asynchrone.

Au travers de cette thèse, nous montrons comment des applications parallèles de différents domaines peuvent bénéficier de la méthodologie de parallélisation CAP. En particulier nous étudions le développement d'algorithmes parallèles d'algèbre linéaire (multiplication matricielle, décomposition LU), de filtrage d'image, d'automate cellulaire et d'optimisation combinatoire (Branch and Bound). Nous présentons aussi la parallélisation d'une application industrielle (*Radiocontrol*) qui a pour objectif d'établir l'audimat des stations radio.

La contribution de cette thèse est (1) de valider le langage de parallélisation CAP (extension du C++) en démontrant sa capacité à formuler de façon synthétique des programmes parallèles, et (2) de montrer que la méthodologie de parallélisation CAP permet de paralléliser efficacement des applications de différents domaines.

Au travers de cette thèse, le lecteur est d'abord initié à la philosophie CAP et à la formulation d'ordonnancements parallèles. Nous étudions les problèmes de contrôle de flux de données et d'équilibrage de charge de travail entre processeurs et proposons des constructions adéquates en CAP. Au travers de plusieurs applications, nous montrons l'aspect compositionnel de CAP et ses avantages. Afin de démontrer les performances de CAP, pour la plupart des applications, nous créons un modèle de performances et le vérifions expérimentalement. Nous concluons, en indiquant les perspectives et les suites à donner à cette recherche.

Table of Contents

1 Introduction and Related Work

What is parallel computing? Let us answer this question by drawing an analogy to a real-world scenario. Consider the problem of delivering letters in a village. If the post office hires only a single postman, he cannot accomplish the task faster than a certain rate. This process can be speeded up by employing more than one postman. One simple way to assign the task to the postmen is to divide the letters equally among them. Each postman starts then the delivery of his set of letters. However, this division of work may not be the most efficient way to accomplish the task, since each postman must walk all over the whole village. An alternate way to divide the work is to assign disjoint regions of the village to each postman. As before, each postman is assigned an equal number of letters arbitrarily. If a postman finds a letter that belongs to the region of the village assigned to him, he delivers that letter. Otherwise, he passes it on to the postman responsible for the region of the village it belongs to. The second approach requires less effort from individual postmen.

The preceding example shows how a task can be accomplished faster by dividing it into a set of sub-tasks assigned to multiple postmen. Postmen cooperate, pass the letters to each other when necessary, and accomplish the task in unison. Parallel processing works on precisely the same principles. Dividing a task among postmen by assigning them a set of letters is an instance of *task partitioning*. Passing letters to each other is an example of *communication* between subtasks. Task partitioning and communication are the main issues of parallel processing. *Synchronization* between tasks is also a critical point, but it can be considered as part of the communication.

Problems are parallelizable to different degrees. For some problems, assigning portions to other processors might be more time-consuming than performing the tasks locally. Other problems can be carried out only serially. For example, consider the task of hammering a nail. Although one person can hammer a nail in a certain amount of time, employing more people does not reduce this time. Because it is impossible to partition this task, it is poorly suited to parallel processing. All problems are not equally amenable to parallel processing and moreover, a problem may have different parallel formulations, which result in varying benefits [Kumar94].

Parallelism appears in various domains as a natural way to improve the performance. If you are cooking in your kitchen and you want to accelerate this process, you will ask someone to help you. By communicating together you will be able to cooperate efficiently and reduce cooking time. More generally, the organization of our society could be considered as a huge parallel process with several (hierarchical) communication layers. From the biological point of view, our brain or any multi cellular organism could be seen as a parallel system. The cells collaborate together to coordinate their work. The multi cellular system seems well organized. This collaboration is ensured by several biological mechanisms (hormones, neurotransmitters, etc.).

Figure 1.1 plots the top performance per year of workstations between 1987 and 1997 [Patterson97]. The performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently. The time to perform a basic operation is ultimately limited by the *clock cycle* of the processor, that is, the time required to perform the most primitive operation. However, clock cycle times will not decrease indefinitely due to physical limitations. To circumvent these limitations, the designer may attempt to utilize internal concurrency in a chip, for example, by implementing pipelining, or by operating simultaneously on all 64 bits of two numbers that are to be multiplied. However, a fundamental result in Very Large Scale Integration (VLSI) complexity theory says that this strategy is expensive. Building individual components operating faster is difficult. It may be cheaper and more efficient to connect together slower components [Foster94]. Another important trend that is changing the face of computing is the enormous increase in the capabilities of networks that connect computers. Not long ago, high-speed networks ran at 1.5 Mbits/s; currently 100 Mbits/s and soon 1Gbits/s are commonplace. Considering the evolution of computer technologies, parallel computing appears as a natural way to circumvent the future limitation of computing power.

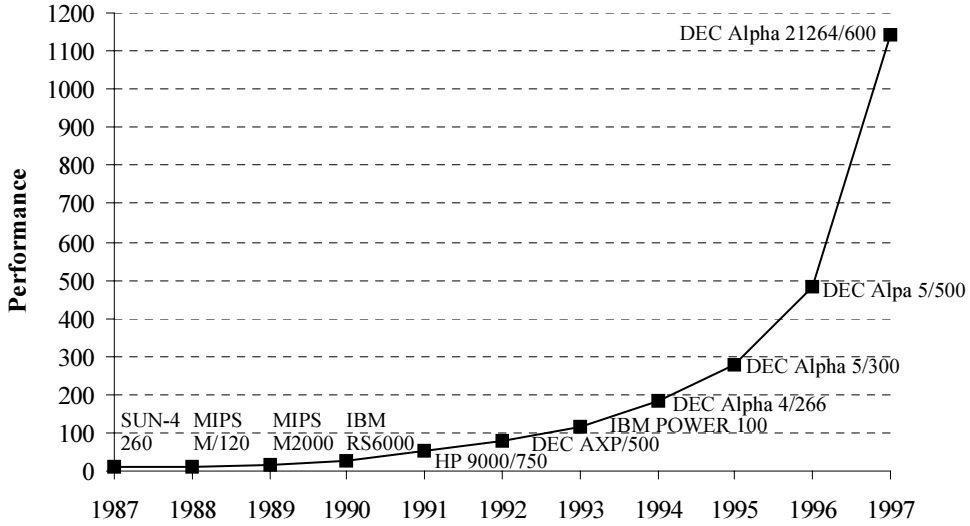


Fig. 1.1 Performance increase of workstations, 1987-1997. Here performance is given as approximately the number of times faster than the VAX-11/780, which was a commonly used yardstick. The rate of performance improvement is about 1.54 per year, or doubling every 1.6 years. These performance numbers are from the integer SPEC92 benchmarks (SPECbase_int92) except the two later machines based on the SPECin95base and multiplied by a factor to estimate SPECbase92 performance.

Figure 1.2 represents typical cost-performance curves of serial computers over the last few decades. Beyond a certain point, each curve starts to saturate, and even small gains in performance come at an exorbitant increase in cost. Furthermore, this transition point has become sharper with the passage of time. By connecting only a few commodity computers together to form a parallel computer, it is possible to obtain raw computing power comparable or even

higher than the fastest serial computers. Typically, the cost of such a parallel computer is considerably lower. From the economical point of view, parallel computing appears also as a natural solution to improve performances with reasonable costs.

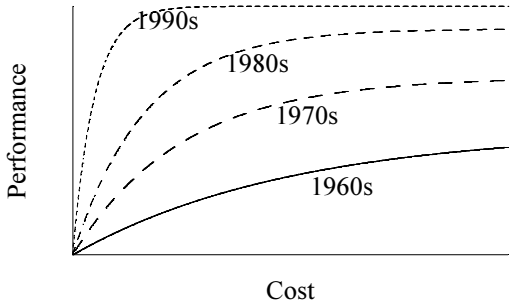


Fig. 1.2 Cost versus performance curve and its evolution over the decades

The previous considerations show us that the idea of parallel processing is a reality imposing itself naturally in the computer sciences. The concept of a computing system consisting of multiple processors working in parallel on different problems or different parts of the same problem is not new. Discussions of parallel computing machines are found in the literature at least as far back as the 1920s [Crichlow97][Denning86]. Throughout the years, there has been a continuing research effort to understand parallel computation. In the 1980s, technical developments (VLSI, large memories, parallel and pipeline ALUs, etc.) reduced the costs of producing computer components while at the same time increasing performance in terms of processing time, quantity of data processed and reliability [DeCegama89]. During the 1980s, the appearance of personal computers enhanced the productivity of individuals, and in turn, the productivity of companies. Since large companies are made up of individuals, the productivity improvement of individuals using stand-alone computers was too compelling to ignore. PCs soon became pervasive [Lewis92]. The 1990 decade is to parallel computing what the 1980 decade was to personal computing. Since the early 1990 there has been an increasing trend to move away from expensive and specialized proprietary parallel supercomputers towards networks of workstations or PCs. Among the driving forces that have enabled this transition has been the rapid improvement in the availability of commodity high performance components for workstations and networks. These technologies are making networks of computers an appealing vehicle for parallel processing, and this is consequently leading to low-cost *commodity supercomputer*.

Let us focus on the effervescence of workstation networks. Why did they become so popular? What are their main advantages compared to a specialized proprietary parallel supercomputer? We already saw that workstation clusters are cheaper and readily available. Their exploitation and maintenance are also less expensive. Workstation clusters are easy to integrate into existing networks. In terms of performance, individual workstations are becoming increasingly powerful. The communication bandwidth between workstations is increasing and latency is decreasing as new networking technologies and protocols are implemented in a LAN (Fast Ethernet, Gigabit Ethernet, Myrinet, FDDI). From an other point of view, the development tools for

workstations are more mature compared to proprietary solutions for parallel computers, mainly due to the nonstandard nature of parallel computers [Buyya99]. Cluster of workstations benefit from the explosion of Internet and the necessity for small businesses to acquire reliable and efficient parallel WWW servers. For many customers and applications, 100 processors provide sufficient computing power. In 1997, most multiprocessor systems were in the range of 8 to 16 processors, with the number moving up slowly. Cluster based solutions correspond to current market necessities.

Traditionally, in science and industry, a workstation referred to a UNIX platform. There has been, however, a rapid convergence in processor performance and kernel-level functionality of UNIX workstations and PC-based machines in the last years. This can be attributed to the introduction of high performance Pentium-based machines and the apparition of operating systems such as Linux, Windows NT and Windows 2000. This convergence has led to an increased level of interest in utilizing PC-based systems as a cost-effective computational resource for parallel computing. This factor coupled with the comparatively low cost of PCs and their widespread availability in both academia and industry has helped to initiate a number of software projects whose primary aim is to harness these resources in some collaborative way [Buyya99].

The next generation of parallel computers is based on clusters of PCs. All the hardware components (processor, network, storage disk) are commodity components. Developing efficient parallel applications on non-dedicated hardware becomes a main research topic. The emergence of distributed memory systems connected through standard high latency networks has a major impact on the conception of parallel applications. Explicit parallelization programming models seem to offer better performance than implicit ones. Implicit parallelization models are based on compiler parallelization which suffers from the difficulties for the compiler to determine the most suitable way to automatically convert sequential programs into efficient parallel ones (compilers for several parallel architectures and languages exists, e.g. Fortran [Koelbel94][Wolfe82]). The difficulties are due to the fact that the compiler must analyze and understand the dependencies in different parts of the sequential code in order to ensure an efficient mapping onto a parallel computer. The explicit parallel programming model requires a parallel algorithm which explicitly specifies how the processors cooperate in order to solve a specific problem. The compiler's task becomes straightforward. However, the programmer's task is quite difficult. Explicit parallelization programming models on distributed memory systems are mostly based on message passing models [Crichlow97], e.g. Concurrent Pascal [Hansen75], Occam [Inmos85][Galletly96]. In order to offer better code portability, architecture independent message passing libraries such as MPI [MPI94] and PVM [Sunderam90] have been developed. However, programs are usually still difficult to understand, debug, and maintain. The direction in language development has been towards making a program more and more a collection of classes, with their private lives separated from their public lives, i.e. object oriented programming [Ghezzi82][Ghezzi85]. The history of programming languages shows a discernible trend towards higher levels of abstraction [Watt90]. We believe that this point is essential in parallel programming languages. This philosophy is shared with many other researchers [Beguelin92][Hatcher91][Shu91]. In order to provide a higher-level language inter-

face simplifying parallel application development new language extensions such as like CAP or MENTAT [Grimshaw93b] have appeared.

The CAP Computer-Aided Parallelization tool has been created by Dr. Benoît Gennart at the Peripheral System Laboratory (LSP) of EPFL. The application programmers express separately the serial program parts and the parallel behavior of the program at a high level of abstraction. This high-level parallel CAP program description specifies a macro-dataflow, i.e. a flow of data and parameters between operations running on the same or on different processors. CAP features also asynchronous processing allowing to handle efficiently I/O bound parallel applications. The CAP framework and its associated parallel file striping services (PS^2) are described in [Gennart98a][Messerli99a].

Within the scope of this thesis we are interested in running high-performance parallel applications on clusters of commodity components, i.e. PCs or workstations. By cluster, we mean a collection from 2 to 50 computers. We are not interested in massively parallel architectures but in small/medium ones. This kind of architectures corresponds to current market necessities. Parallel computing appears from many point of views (computing power, costs, scalability, simplicity, flexibility) as offering potentially efficient solutions. Nevertheless, exploiting this potential remains a difficult task. Building efficient parallel schedules is an art. In this thesis, we present the development of several parallel applications from different domains. We use the CAP framework in order to facilitate the design and development of these parallel applications. The CAP based parallelization approach differs from other parallel frameworks in that it lets the programmer concentrate his efforts on building efficient parallel schedules. However, the fundamental concepts could be transposed to any other parallel language. Within this thesis, we describe our methodology which leads to the development of efficient parallel solutions.

Chapter 2 introduces the reader to the basic parallel notions. We introduce the notion of speedup and analyze theoretical limitations. We present several techniques such as pipelining, load balancing, and asynchronous execution behavior, allowing to improve parallel algorithms. We treat also the problem of reliability of parallel applications. All these considerations are the fundamentals of the CAP based parallelization approach. This chapter presents the underlying philosophy, concepts, and motivations of the CAP framework, without entering into the syntactical details of the CAP language extension. Chapter 3, instead, presents the CAP language from the syntactical point of view. We present the major parallel constructions of CAP. We present how parallel-pipeline execution schemes can be expressed within the CAP language. Chapter 4 presents advanced CAP features such as load balancing and resource handling in order to implement efficient parallel programs. Chapter 5 is dedicated to the CAP runtime system and in particular to the CAP Message-Passing System (MPS) and the serialization tool. Chapters 3 to 5 should allow the reader to familiarize itself with the CAP framework and understand the CAP programs presented in the following chapters. Chapter 6 is dedicated to parallelization of linear algebra algorithms. In particular, we focus on the parallelization of the well-known BLAS routines [Anderson95]. In Chapter 7, we solve the problem of parallel filtering of large images. Chapter 8 treats the parallelization of cellular automata and the inherent dynamic load balancing problem. In Chapter 9, we present the development of the *Radiocontrol* industrial application.

This application has been developed under industrial constraints; in particular we treat the problem of reliability. The last chapter is a didactical chapter dedicated to the parallelization of discrete optimization problems.

This thesis is dedicated to the application of the CAP parallelization methodology, which is based on the formulation of parallel schedules. The contribution of this thesis is (1) to validate the CAP language extension by demonstrating its capability of handling parallel programs and simplifying their development, and (2) to validate the CAP based parallelization approach by presenting efficient parallel programs. Beyond the research work of this thesis, a large collaborative effort with industrial partners has been performed. Currently the CAP framework and its underlying parallelization techniques are used by several companies.

2 Basic notions and parallelization fundamentals

This chapter presents important issues in parallel processing. We focus our interest on the central problem of achieving good performance in terms of scalability. Amdahl's law is discussed in order to explain the inherent difficulties of parallel processing. Then we introduce several concepts such as pipelining, load balancing or asynchronous execution behavior, which need to be considered in order to implement efficient parallel programs. These concepts are the fundamentals of the CAP language extension. Without presenting the syntax of CAP, we explain the fundamental motivation of building a parallel framework such as CAP. This chapter introduces the art of building parallel schedules.

2.1. Performance Measurements

In parallel programming, as in other engineering disciplines, the goal of the design process is to optimize a solution in terms of execution time, memory requirements, implementation costs, maintenance costs, etc. Such a design optimization involves tradeoffs between simplicity, performance, portability, scalability, and other factors. The relative importance of these diverse factors will vary according to the nature of the problem at hand.

In order to evaluate the performance of an algorithm, we need to provide several metrics. A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input. To evaluate parallel algorithms, we must consider, in addition to the execution time, their scalability, the mechanisms by which data is generated, stored, transmitted over networks, moved to and from disks, and passed between different stages of computations. Diverse metrics, such as execution time, parallel efficiency, throughput and latency (network or I/O), should be considered to evaluate the performance of parallel algorithms. Once the costs of these metrics have been determined for a specific parallel algorithm, a *performance model* can be established. These models can be used to compare the efficiency of different algorithms, to evaluate scalability, and to identify *bottlenecks* and other inefficiencies. Performance models can also be used to guide implementation efforts by showing where optimization is needed.

In this section, we introduce some metrics that are commonly used to measure the performance of parallel systems.

2.1.1. Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. *Speedup* is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors. Let us denote t_s the

sequential execution time, and by t_p the parallel execution time. We define speedup by the function $S(p)$:

$$S(p) = \frac{t_s}{t_p} \tag{2-1}$$

For a given problem, several sequential algorithms may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processor[1]. As a consequence, the sequential time t_s used in the speedup definition of equation (2-1) has to be the time taken by the best known serial algorithm.

When looking at the speedup we are in general interested in the evolution of $S(p)$ according to increasing values of p, rather than in a single value of the speedup function. This informs us about the *scalability* of the parallel algorithm. Several terminologies are used to characterize the speedup function. The speedup is *ideal* if $S(p) = p$. The speedup is said *superlinear* (resp. *sublinear*) if, for some p, $S(p) > p$ (resp. $S(p) < p$). We speak about *linear* speedup when $S(p) = \alpha p$ (for some α), i.e. when the parallel program is scalable. In general, all those characterizations are valid on a specific parallel architecture and within a given range of p. Both the parallel architecture and the considered range of p values must be clearly specified when speaking about speedups.

The parallel algorithm spends some execution time for communication and synchronization purposes. This implies that theoretically, the speedup can never exceed the number of processors p (a formal proof is given in [Cosnard95]). In practice, superlinear speedups (an example is described in Chapter 10) are observed. This is usually due either to a nonoptimal sequential algorithm or to hardware characteristics that put the sequential algorithm at a disadvantage. For example, the data for a problem might be too large to fit into the main memory of a single processor, thereby degrading its performance due to the use of secondary storage. But when partitioned among several processors, the individual data-partitions would be small enough to fit into their respective processor's main memories [Kumar94].

A general way to analyze parallel programs is to establish the speedup function and to consider the differences with the ideal speedup. In a second step, these differences should be explained by an appropriate performance model describing the parallel algorithm. The role of the performance model is to identify the bottlenecks, algorithm inefficiencies, and/or parallel architecture limitations. Finally, these explanations could be used to improve the parallel algorithm.

[1] If the sequential algorithm used for the comparison is not the best serial algorithm but the same as the parallel algorithm running on one computer, then we measure the *relative speedup* as opposed to *true speedup* [Patterson97].

2.1.2. Efficiency

An ideal parallel system containing p processors can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processors cannot devote 100% of their time to the computations of the algorithm. The *efficiency* measures the fraction of time in which a processor is usefully employed. The efficiency function $E(p)$ is defined as the ratio of the speedup to the number of processors.

$$E(p) = \frac{S(p)}{p} \tag{2-2}$$

Ideally the efficiency should be equal to one. In practice, the efficiency is between zero and one, depending on the degree of effectiveness with wich the processors are utilized.

2.1.3. Amdahl's law

Various authors attempted to specify the speedup limits. In 1967, Gene Amdahl asserts that the inherent fraction of a program's execution time that must be carried out serially dominates the overall execution time, regardless of the number of processors available.

Amdahl's law can be formulated as follows. For a given serial program let us denote by t_s the total execution time, by t_{ss} the execution time of the serial parts of the program which cannot be parallelized, and by t_{sp} the time spent on a single processor in executing the parallel portions of the programs. We have, therefore:

$$t_s = t_{ss} + t_{sp} \tag{2-3}$$

Let s be the ratio between t_{ss} and t_s:

$$s = \frac{t_{ss}}{t_s} \tag{2-4}$$

The parallel execution time on p processors is:

$$t_p = t_{ss} + \frac{t_{sp}}{p} \tag{2-5}$$

Consequently, the speedup becomes:

$$S = \frac{t_s}{t_p} = \frac{t_{ss} + t_{sp}}{t_{ss} + \frac{t_{sp}}{p}} = \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s} \tag{2-6}$$

This last equation expresses the fact that the speedup is limited by a limit which is independent of the number of processors and the structure of the machine. This result is known as Amdahl's

law [Amdahl67][Amdahl88]; later, this law was generalized by Lee [Lee80]. Since this formula does not take into account any reduction in speedup due to synchronization, it could even be considered as a favorable point of view of parallel processing.

Amdahl's law was used as a powerful argument against the use of parallel processors, especially systems with a large number of processing units. Since Amdahl's predicts that even with an infinite number of processors, a program with $s \geq 0.005$ (which was considered as a realistic lower bound for s) would never achieve a speedup greater than 200. The argument points that it is may be a waste of money to develop (especially massively) parallel processing.

Several research groups (e.g. Sandia National Laboratories) have demonstrated that this last consideration is not valid. In fact, they observed speedups in excess of 1000 on several parallel programs. They achieve such a result by increasing considerably the amount of parallelizable work without increasing the sequential portion of the program, i.e. reducing s. These results do not invalidate Amdahl's law, but point out the importance of understanding the assumption of this law. A fundamental assumption in Amdahl's formula is that the percentage of time spend in executing the parallel sections of the code is independent of the number p of processors. In practice, this is not obvious, since it would amount of taking a fixed-size problem and running it on extremely large numbers of processors. Rather, the problem size tends to scale with the number of processors. If we are interested in developing a parallel application on a very large number of processors, the application is certainly also very large. When the size of a problem is scaled up, frequently the serial portion of the program does not increase proportionately to the problem size. For example, if a finer grid is used or the number of time steps is increased, the serial portion of the program is not affected, but the parallel portion increases [Leiss95]. In order to take into account these considerations, the Sandia National Laboratories developed the *scaled speedup* model [Gustafson88]. This model differs from Amdahl's law in that the serial execution time t_{ss} is considered as independent of the problem size and only the parallel portion scales up with the problem size.

In order to better understand the implication limits of the Amdahl's law, let us consider the following noncomputing problem taken from [Foster94]. Assume that 999 of 1000 workers on an express way construction project are idle while a single worker completes a sequential component of the project. We would not view this as an inherent attribute of the problem to be solved, but as a failure in management. For example, if the time required for a truck to bring material to a single point is a bottleneck, we could argue that the road should be under construction at several points simultaneously. Doing this would undoubtedly introduce some inefficiency - for example, some trucks would have to travel further to get to their point of work - but would allow the entire task to be finished more quickly. Similarly, it appears that almost all computational problems admit parallel solutions. The scalability of some solutions may be limited, the challenge is to find the optimal way to schedule the tasks in parallel. One must not build parallel applications by incrementally parallelizing sequential programs. Parallelism should be the central point that suggests the guiding lines of the algorithm. Finding optimal parallelization strategies is the main challenge.

Amdahl's law suggests how difficult it is to reach ideal speedup. Arranging schedules efficiently in parallel requires a large effort, similar to the management effort necessary to coordinate 1000 workers. This is the fundamental reasons why we believe that programming environments such as MPI do not provide an efficient framework for parallel processing. In such an environment, there is no concrete distinction between the serial work and parallel scheduling. It is like if in real life, there would be no difference between being a worker or a manager. The CAP language extension separates clearly the serial tasks, which are expressed in C++, and the flowgraph describing the parallel schedule. The flowgraph is described with dedicated CAP keywords and constructions. The language offers several parallel constructions and asynchronous execution models which can be combined together to form new parallel schedules. This way, the programmer can focus his attention on building efficient parallel schedule independently from the conception of serial routines.

2.2. Granularity

This section poses the problem of how to make efficient use of parallel systems. Programming parallel systems requires that one starts with a study of the parallelism inherent in the problem to be solved. Thus, the question of where parallelism can occur arises. As pointed out by [Leiss95], several parallelization levels are possible:

1) At the job level
2) At the program, function or thread level
3) At the instruction level
4) At the arithmetic and bit level

These four levels are presented in increasing granularity order. The first level offers the lowest granularity, since there will typically be a few tasks at that level. At the opposite, the last level has a very high granularity. The question is to determine which levels are concerned by high performance parallel computing on cluster based parallel computers. Level one (job management) is rather uninteresting since independent tasks by definition can be executed in parallel. Operating systems or dedicated job management tools such as LSF [Xu01] handle the problem of assigning tasks (eventually dynamically) to processors, but it is not the main topic of high performance parallel computing. The fourth level (parallelization at the bit level) concerns the internal architecture of the processor and some compilers. This level is not relevant for the programmer of parallel applications. The programmer is concerned by levels two and three, i.e. instruction parallelism and thread or function level parallelism.

Instruction level parallelism requires *fine* granularity. Because of the fine granularity, this level of parallelization is mostly used on *vector* computers. Generally, the programmer does not express explicitly the parallelism. Instead, the programmer develops a sequential program (usually in Fortran or C). The application is parallelized by applying a *vectorizer* to the sequential program [Leiss95]. Important in this process is that in doing this translation from sequential to

vector code, the compiler uses only syntactic properties of the program; in other words, there is no need for any understanding of the meaning of the program in order to do the vectorization. In contrast to initial expectations, several scientific and engineering applications have proved to be amenable to automatic vectorization. Furthermore, this approach benefits from the advantage that the programming effort necessary to vectorize the code is in general very low. Nevertheless, this approach requires the use of a specific expensive hardware, and is not well suited for I/O intensive or interactive applications. In addition, from the theoretical point of view, automatic vectorization is unlikely to adhere to any modular design approach. These reasons limits the usage of vectorized applications.

An automatic parallelization tool will never be able to transform a sequential QuickSort into a more appropriated parallel MergeSort. Such a transformation requires the understanding of the meaning of the program which could not be deduced from the syntactic properties of the sequential program. Let us now discuss the second level of parallelization grain, the function or thread based parallelization, which requires *coarse* grain. At this level, the programmer fomulates explicitly the expression of parallelism and focuses his attention on developing an efficient parallel schedule. The CAP language extension is based on these considerations and helps the programmer in its task. CAP facilitates the programmer's work by offering several high-level parallel constructions avoiding the implementation of low-level routines necessary to the execution of the parallel schedules. The high-level of abstraction offered by the CAP framework lets the developer focus his attention on the management of parallel schedules.

2.3. Parallel processing and pipelining[1]

In this section we introduce two fundamental ways of executing tasks in parallel. We distinguish *pipelining* and parallel processing. In order to explain clearly the differences between these two parallelization techniques, let us take an example from real life. Anyone who has done a lot of laundry has intuitively used pipelining or parallel processing. Let us study the several ways to accomplish this task [Patterson97].

The serial approach to laundry would be: (1) Place on dirty load of clothes in the washer, (2) when the washer is finished, place the wet load in the dryer, (3) when the dryer is finished, place the dry load on a table and fold, (4) when folding is finished, put the clothes in the wardrobe. When the clothes are put away, then the next dirty load starts over, and the whole process is repeated. For this example, we suppose that each step of the task takes 30 minutes and that we need to repeat the task four time (tasks A, B, C, D). Figure 2.1 suggests that if the first load is placed in the washer at 6 pm, the last task terminates at 2 am.

If two laundry rooms are available, the process can be improved by using in parallel both laundry rooms. Figure 2.2 illustrates this process. The tasks A and C start both at 6 pm. At 8 pm the

[1] Pipelining refers in general to the internal architectures of processors. In the thesis, we use this terminology in a more general sense.

tasks B and D start and they terminate at 10 pm. By working in parallel with both laundry rooms the eight hours serial work is reduced to four hours.

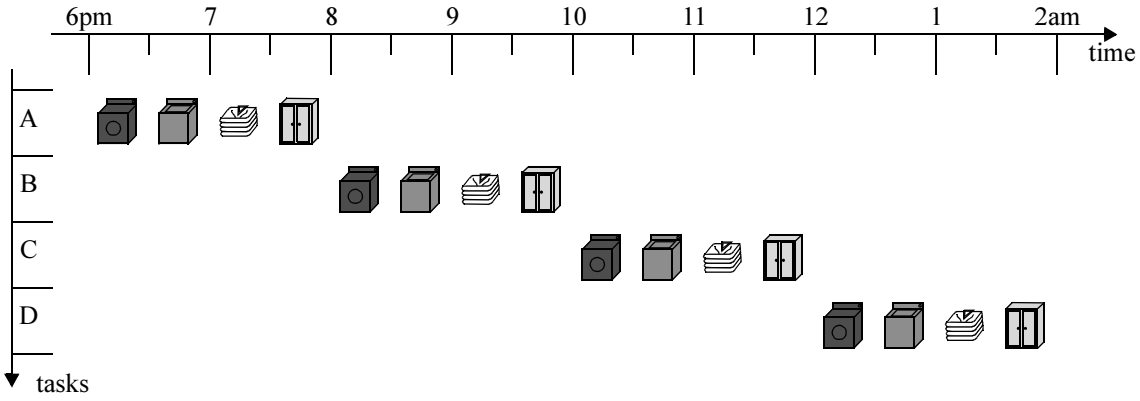


Fig. 2.1 Doing the laundry sequentially

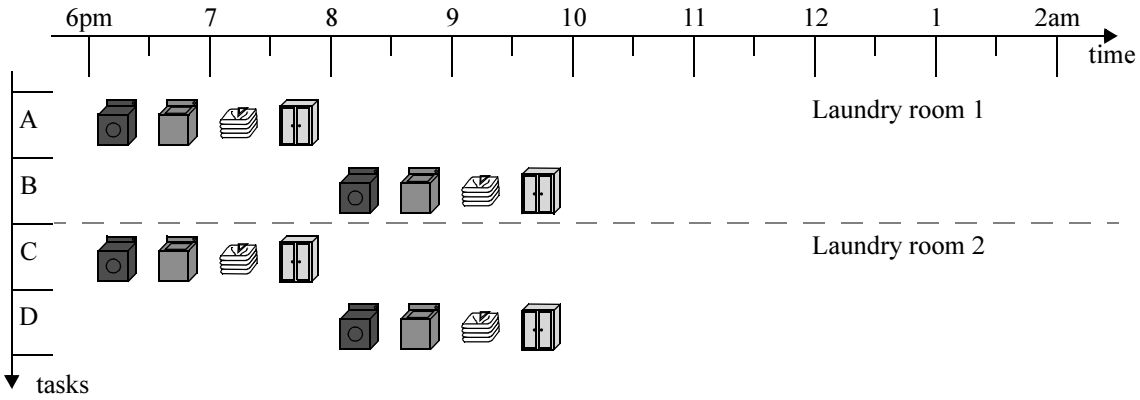


Fig. 2.2 Doing the laundry in parallel

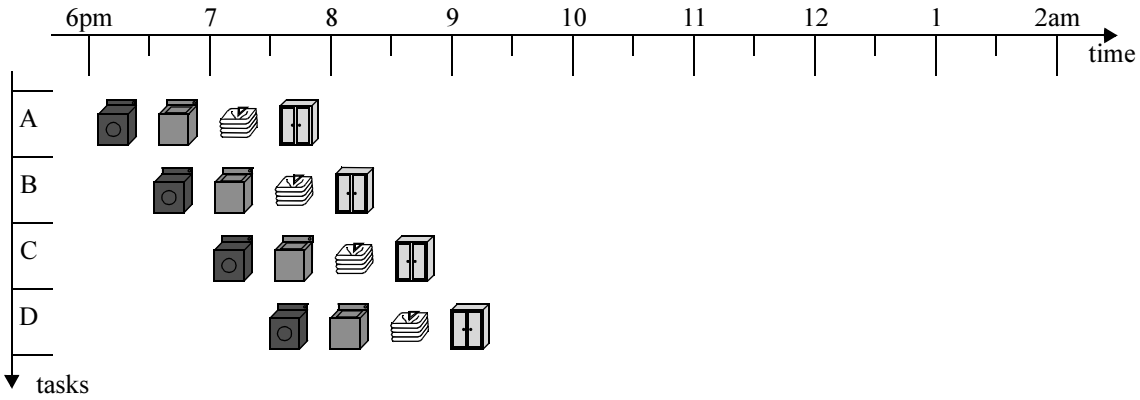


Fig. 2.3 Doing the laundry in pipeline

If only one laundry room is available, doing the laundry in pipeline takes less time than the sequential approach. The pipelined approach is presented by Figure 2.3. As soon as the washer is finished with the first load and placed in the dryer, the washer is loaded with the second dirty load. When the first load is dry, it is placed on the table and the folding starts, at the same time, the wet load is moved to the dryer, and the next dirty load into the washer. Next, the first load A is placed in the wardrobe, the second load B is folded, the dryer receives the third load, and the fourth load D is moved into the washer. At this point (7h30 pm), all steps - called *stages* in pipelining - are operating concurrently. The pipeline reaches the *steady state*. The time until the pipeline reaches the steady state is called the pipeline *starting cost*. Similarly, the pipeline has also an *ending cost*. As long as separate resources are available for each stage, the task can be pipelined.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more work is achieved per hour. If all the stages take about the same amount of time and there is enough work to do, then the speedup due to pipelining is equal to the number of stages in the pipeline. [Patterson97].

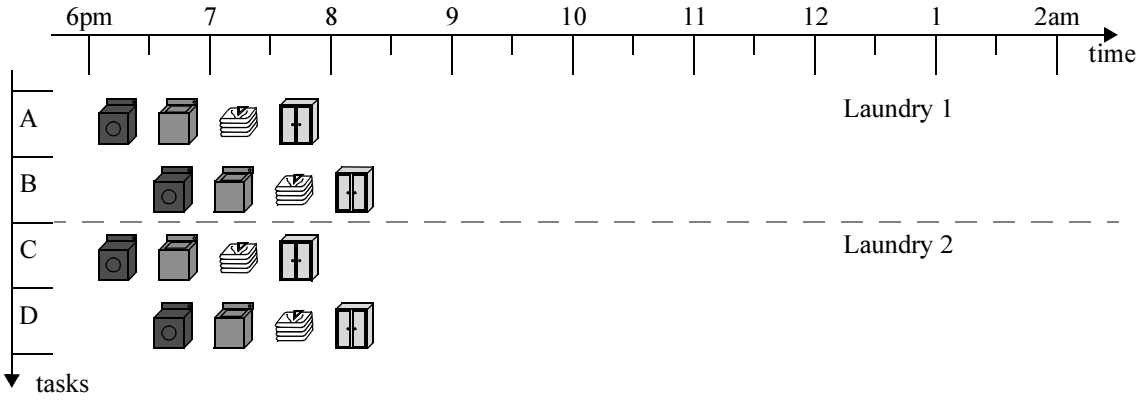


Fig. 2.4 Doing he laundry in a parallel-pipeline fashion

The final improvement consists of combining both parallelization techniques. If two laundry rooms are available, it is possible to accomplish the task in a parallel-pipeline fashion. Figure 2.4 illustrates this process. In this figure the pipeline never reach the steady state (there is no time where all pipeline stages are working in parallel) because there are not enough tasks to perform.

The previous considerations can be transposed to parallel computing. Parallel processing consists obviously of running the algorithm using several processors, just as using two laundry rooms (Fig. 2.2). Pipelining seems to be more difficult to transpose. One possible way to achieve pipelining is to consider that generally an algorithm is composed of computing and I/O. By I/O we mean mostly disk accesses and network transfers. Modern technologies allow the hardware responsible for I/O to have a Direct Memory Access (DMA), i.e. disk or network

accesses can be performed without loading significantly the processor. Therefore, it is possible to pipeline I/O and computing. If the application is compute-bound, by performing the I/O and computation in pipeline it's possible to *hide* the I/O time. Similarly, if the application is I/O-bound, an efficient pipeline allows to hide the computation time. If a parallel algorithm performs it's I/O serially (as it is commonly), it will consequently fall under the Amdahl's law. By pipelining the I/O and the computation the parallel algorithm can be improved.

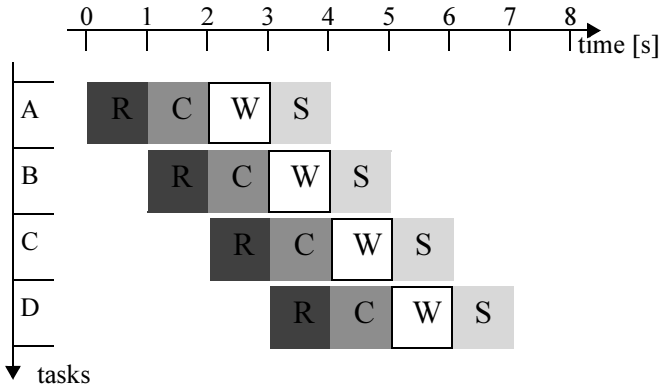


Fig. 2.5 Executing the encryption program in pipeline (1 MB blocks)

Let us give a concrete program example, where parallel-pipeline processing is well suited. The program consists of reading a 4MB file from a hard disk, performing some computation to encrypt it, writing the encrypted file (also 4MB) back to a second hard disk and sending the encrypted file over the network. By stripping the file in four blocks of 1MB it is possible to perform the computation in a pipelined fashion. Figure 2.5 illustrates this process. Each task (A, B, C or D) consists of reading 1MB from the disk (R), encrypting it (C), writing it back to a second hard drive (W) and finally sending it over the network (S). We suppose that each step takes the same amount of time (1 second). Each task is working on a different part of the file: task A on the first MB, task B on the second MB, task C on the third MB and task D on the last MB. The pipeline stages (R, C, W and S) can be performed in parallel since they use different resources. The last figure can be compared with Figure 2.3. If several computers are available the encryption program can be improved by implementing a parallel-pipeline execution model like in Figure 2.4.

Another approach to improve the pipelined encryption program of Figure 2.5 is described by Figure 2.6. Instead of splitting the file into 1MB blocks, we use 0.5MB blocks. Each step of the pipeline takes now 0.5 second instead of 1 second. Since the tasks are smaller there are more tasks to perform. The new version of the encryption program is composed with 8 tasks: a1, a2, b1, b2, c1, c2, d1, d2. Reducing the task size increases the number of tasks (finer grain), reduces the pipeline starting and ending cost, and therefore improve the program execution time. The 1MB block size program takes 7 second to terminates against 5.5 second for the 0.5MB block size encryption program. This indicates - from the pipelining point of view - that the task should be as small as possible. This consideration must be balanced with the fact that smaller tasks have

a relative higher latency. For example, when a disk read access of 100MB is made the latency can be ignored. But when a 1B access is made the latency becomes dominant against the throughput. From the latency point of view, it is more efficient to handle larger tasks. The ideal strategy is a compromise in between taking into account pipelining and latency.

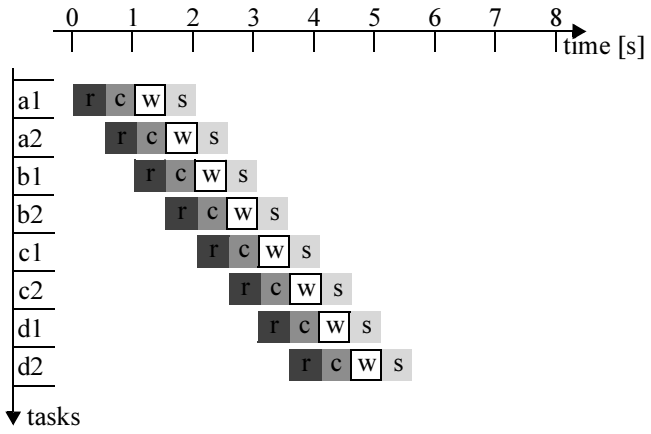


Fig. 2.6 Executing the encryption program in pipeline (0.5 MB blocks)

Parallel and pipeline execution model are critical issues in terms of parallel processing. The CAP language extension allows the programmer to specify parallel and pipeline constructions, and to combine them together to formulate more complex execution schedules. Let us ignore for now the syntax of the CAP language extension. Conceptually the CAP programmer is able to formulate the execution schedule of Figure 2.4 by specifying something like:

Parallel(Pipeline(A,B) , Pipeline(C,D))

Clearly the programmers do not care about any synchronizations between the several pipeline stages and between the different tasks. The programmer formulates at a high-level of abstraction the parallel execution schedule. The CAP kernel is responsible for deducing the dependencies and for managing the required synchronizations. Inside the CAP kernel, everything is performed in an asynchronous manner allowing parallel-pipeline executions. From the CAP programmer point of view, the CAP kernel could be considered as a parallel scheduler. The possibility to formulate at a high-level of abstraction complex parallel-pipeline execution schedules is one of the main issues of the CAP language extension.

2.4. Master-slave or distributed system

There are two fundamental ways of organizing parallel schedules: in a *master-slave* or in a *distributed* fashion. By a distributed system we mean that each processing node, part of the parallel system, is responsible (1) for performing some computation and (2) for cooperating with the other processing node. Cooperating means exchanging some information with other processing nodes. The results of this communication can induce several modifications in the current or further computation activities. Within a parallel algorithm organized in a distributed

fashion, each processing node exchanges information with all or at least with some neighboring processing nodes. Implementing distributed systems can be difficult, since each processing node must take into account the activities of other nodes.

In general, implementing a master-slave parallelization scheme requires less programming efforts. In a master-slave scheme, the slaves are responsible to perform some computation, but the slaves do not communicate with each other. They perform their tasks independently. Instead a master is responsible to coordinate the slaves. This coordination is ensured by several information exchanges between the master and the slaves. The master can be considered as the manager of a team of workers. In a master-slave scheme, the coordination effort is localized at a single place, the master. This explains why implementing a master-slave scheme is simpler than a distributed system. Nevertheless, it's not always possible or efficient to parallelize a program in a master-slave manner.

There is no exact border between the master-slave and distributed parallelization models. Several execution schedules can be categorized in neither of the two execution schemes. For example, let us consider the problem of image filtering (Chapter 7). Such a problem can be parallelized by letting a master divide the input image into tiles. The master then distributes the tiles to the slaves. The slaves filter the tiles received from the master and send the filtered tiles back to master. Once the master has collected all the filtered tiles, the program is terminated. The slaves, in order to perform the filtering step, need to receive some data from the neighboring tiles located on other slaves. Thus the slaves need to exchange information between themselves, like in a distributed system.

The master-slave parallel organization scheme has been criticized by the scientific community. The major argument against the master-slave scheme is that it is not scalable. In fact, the number of slaves cannot increase infinitely, because the master becomes a bottleneck. This argument is clearly true, but we want to formulate three comments on this consideration. First, Amdahl's law suggests that the problem of scalability is not specific to the master-slave scheme, but inherent to any parallelization model. Second, the master-slave scheme could be, if necessary, extended hierarchically, by introducing masters of masters. The hierarchical organization of our society seems to indicate that such a system works pretty well. Finally, the question of knowing if a system is infinitely scalable is not always relevant. Rather, the problem is to perform parallel processing in order to achieve a finite and specified speedup. The corresponding question is to choose the appropriate parallelization scheme (e.g. requiring the least programming effort).

The CAP language allows the programmer to specify either master-slave or distributed parallelization schedules. Nevertheless, CAP facilitates the development of master-slave schedules. This result comes from the fact that we are mostly interested in high-performance parallel computation on a limited number of nodes, e.g. 50. On such a number of processing nodes, master-slave schedules are generally efficient.

2.5. Load balancing

Imperfect partitioning may have a dramatic impact on the overall performance of a parallel program. Suppose a program that takes 1000 seconds on a single processor can be partitioned into 100 task that can be executed without incurring overhead or waiting because of dependences. With a perfect partition, each task would take exactly 10 seconds. However, suppose that all tasks except one take 9.9 seconds and that the remaining task takes 19.9 seconds. If the 100 tasks are distributed among 2 processors, each processor executing 50 tasks, the system achieves a speedup of:

$$S = \frac{1000}{49 \times 9.9 + 19.9} = 1.98 \qquad (2\text{-}7)$$

If we decide to distribute the 100 tasks over 100 processors the speedup becomes:

$$S = \frac{1000}{19.9} = 50.25 \qquad (2\text{-}8)$$

As we see, a difference of 10 seconds between tasks, which is small compared to the 1000 seconds required to accomplish the work sequentially, becomes critical when increasing the number of processors.

A critical issue of parallel algorithms is to balance the load between the contributing processors, i.e. achieving *load balancing*. We distinguish *static* load balancing and *dynamic* load balancing. Static load balancing consists of assigning statically some work to processors. Unfortunately, static partitioning of several algorithms yields poor performance because of substantial variation in the execution time of each partition. Dynamic load balancing is more elaborated, it consists of distributing the load at run time. When a processor runs out of work, it should get more work from another processor. Dynamic load balancing is more flexible and achieves better performance than static load balancing.

Load balancing is another important feature of the CAP language extensions. CAP allows the programmer to specify, at a high-level of abstraction, static or dynamic load balancing directives. The CAP kernel interprets these directives and balances the load of the specified tasks among the different processors.

2.6. Asynchronous behaviour

Asynchronous behavior is opposed to synchronous execution. In a synchronous execution mode all the processors receive computation work at specific time. This may result in some inefficiencies, letting several processors idle. An improvement consists of letting each processor work independently from the others. Instead of receiving tasks synchronously, each processor takes its work from a task-stack. Such an asynchronous execution mode enables each processor to work at its own speed.

The CAP execution model is based on asynchronous behavior. The CAP kernel associates to each thread composing the parallel application a queue. This queue is filled with tasks (produced by other tasks). As soon as a thread terminates a task, it takes a new task from its queue independently from the other threads. The asynchronous behavior of the CAP kernel is hidden to the programmer. The CAP program describes how the tasks are scheduled. The CAP kernel acts at a lower level using asynchronous routines to realize the desired schedule.

2.7. Flexibility

Another important issue of the CAP framework is to allow the programmer to deal with logical processes. Instead of developing architecture dependent programs, the developer programs at a logical level. Within the CAP framework the programmer declares logical processes and develops parallel schedules using these logical processes. At execution time, the logical processes are mapped to OS processes. The mapping is defined by a configuration file. The configuration file name (and path) is given on the command line arguments at execution time. The dissociation of OS processes and logical processes induces some flexibility. The same program can run, without re-compilation, with different numbers of contributing processors. It can also take into account the usage of heterogeneous hardware, like single-processor or bi-processor computers. The developer can at execution time customize the configuration file to adapt and take advantage of the hardware specificities of the cluster on which the program should run.

2.8. Reliability and error handling

When thinking about parallel systems, one must be concerned by reliability. If we run our parallel application on a cluster of 100 PCs, the probability of failure is 100 times the failure probability of a single component. If no adequate error handling exists, the failure of a single component of the cluster can crash the whole parallel application. To avoid such a situation, an effort must be invested for handling errors. Error handling is not costless. Reliability is, in a way, opposed to high-performance computing. The question is not how to built a reliable system, but how to build a reliable system that consumes as less resources as possible for handling errors.

Letting a parallelization tool featuring the ability of recovering from many kinds of errors induces indubitably a considerable error management overhead. A solution consists of letting the parallel program handle failures by implementing a *checkpoint and restart* paradigm [Gray92]. Within the checkpoint and restart paradigm, the application is responsible for saving (on one or several permanent storage devices) its status periodically (checkpoint). This strategy reduces the costs, since the application could decide when it is the most appropriated time to make a checkpoint. When a failure is detected, the parallel application is destroyed and restarted. When the application is restarted, it uses the information saved during the last checkpoint to recover. Using this strategy, only the computation time from the last checkpoint to the system crash is lost.

Within the development of the CAP framework, we have developed a tool which enables the capability of detecting a failure during the execution of a parallel application. Such a tool is commonly called a *watchdog*. If the watchdog detects a failure, it kills the parallel application and eventually reboots the whole system. Then the application is launched again. The application is responsible for using the information from the last checkpoint to recover from the failure.

2.9. Summary

In this chapter we showed that building parallel algorithm cannot be considered as an extension of sequential algorithms. Efficient parallel solutions differ from serial solutions. Parallel scheduling is difficult and requires ingenuity, as suggested by Amdahl's law. We presented several commonly used techniques to improve parallel algorithms, such as pipelining, load balancing, and asynchronous execution behavior. We introduced the CAP language extension from the conceptual point of view. The several aspects discussed in this chapter are the fundamentals (building blocs) of CAP. Explaining the fundamentals of the CAP language extension from the conceptual point of view, rather than from the syntactical point of view, aims at helping the reader to better understand our motivation and philosophy.

3 The CAP Computer-Aided Parallelization Tool

This chapter describes the CAP Computer-Aided Parallelization tool. The CAP language is a general-purpose parallel extension of C++ enabling application programmers to create separately the serial program parts and express the parallel behavior of the program at a high-level of abstraction. This high-level parallel CAP program description specifies a parallel schedule, i.e. a flow of data and parameters between operations running on the same or on different processors. CAP is designed to implement highly pipelined-parallel programs that are short and efficient. With a configuration map, specifying the layout of threads onto different PCs, these pipelined-parallel programs can be executed on distributed memory PCs. In this chapter, we concentrate on the technical aspects of the CAP framework.

3.1. Introduction

This chapter is intended to give readers the necessary background on the CAP methodology and programming skills required to understand the remainder of this dissertation. Most of the current chapter has been taken from [Messerli99a]. Readers who want to have a more in-depth view of CAP can read its reference manual [Gennart98a].

The CAP specification of a parallel program is described in a simple formal language, an extension of C++. This specification is translated into a C++ source program, which, after compilation, runs on multiple processors according to a configuration map specifying the mapping of the threads running the operations onto the set of available processors. The macro-dataflow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [Grimshaw93a][Grimshaw93b].

The control mechanism that selects for execution the primitive units of computation, i.e. sequential operations, is based in CAP on the macro-dataflow MDF model [Grimshaw93a] inspired by the dataflow computational model [Agerwala82][Denning86][Srini86][Veen86]. The CAP macro-dataflow computational model is a coarse grain, *data-driven* model and differs from traditional dataflow in four ways. First, the computation granularity is larger than in traditional dataflow. The basic units of computation are high-level tasks such as multiplying two matrices specified in a high-level language (C/C++ leaf operation in the CAP terminology), not primitive operations such as addition. Second, some operations may maintain state between invocations, i.e. have side-effects such as modifying shared global variables. Third, operations may only depend on the result of their single previous operation, i.e. single data dependency. Finally, in order to be able to have parallel executions of tasks, CAP has introduced two particular operations called *split* and *merge*. A split routine takes as input the token (CAP data structure, see Section 3.2) of its previous operation, and splits it into several sub-tokens sent in a pipelined parallel manner to the next operations. A merge routine collects the results and acts

as a synchronization means terminating its execution and passing its token to the next operation after the arrival of all sub-results.

An algorithm is described in CAP by its macro-dataflow and graphically depicted by a *Directed Acyclic Graph* (DAG). DAGs are completely general, meaning that they can describe any sort of algorithm. Furthermore, they ensure that the generated code will be deadlock free. The CAP language extension enables the programmer to specify DAGs. Figure 3.1 presents a symmetric DAG, i.e. where all the split and merge points in the graph match pairwise. A recent extension to CAP enables the programmer to specify asymmetric DAGs as well, i.e. where split and merge points in the graph do not match pairwise (Fig. 3.2).

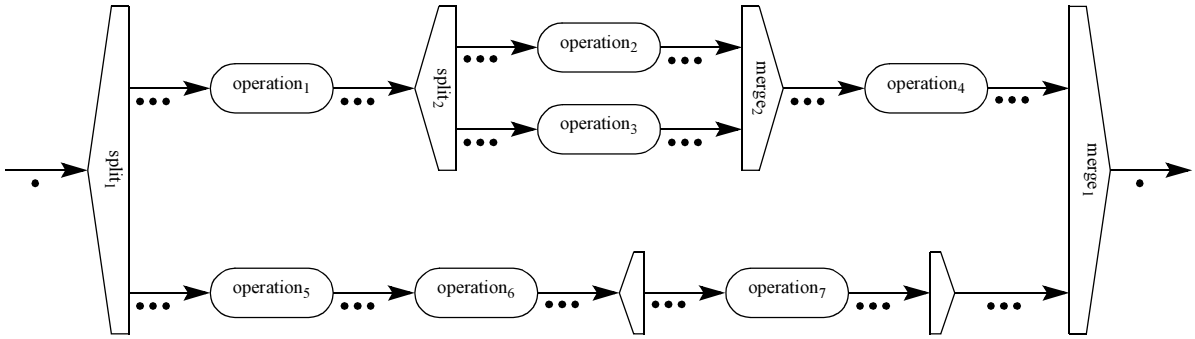


Fig. 3.1 An example of a CAP macro-dataflow depicted by a symmetric directed acyclic graph. Arcs model data dependencies between operations. Tokens carry data along these arcs. Split routines split input tokens into several sub-tokens sent in a pipelined parallel manner. Merge routines merge input tokens into one output token thus acting as synchronization points

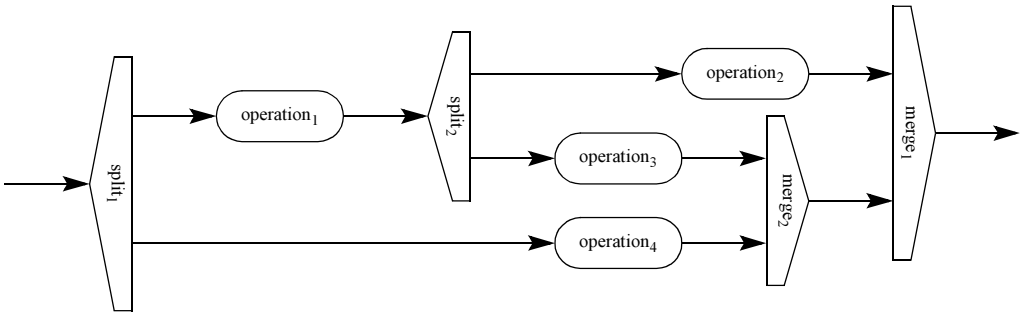


Fig. 3.2 An asymmetric directed acyclic graph that can be described with the recent extension to CAP

Regarding communications, the CAP language does not explicitly provide synchronization tools such as semaphores, barriers, etc. A CAP program is self-synchronized by the data merging operations. Merge routines act as synchronization points returning their output tokens only when all the sub-tokens have been merged.

The CAP computer-aided parallelization tool enables application programmers to specify at a high level of abstraction the set of threads which are present in the application, the processing operations offered by these threads, and the flow of data and parameters between operations. This schedule specification completely defines how operations running on the same or on different PCs are sequenced, and what data and parameters each operation receives as input values and produces as output values.

The term high-level is used to emphasize the fact that the CAP computational model works at a high-level of abstraction compared with other parallel imperative languages such as High-Performance Fortran [Loveman93], Multilisp [Halstead85], Concurrent Pascal [Hansen75], Occam [Miller88][Galletly96][Inmos85] and compared with other parallel programming environments such as Express [Parasoft90], PVM [Beguelin90][Sunderam90], Linda [Carriero89a] [Carriero89b] and MPI [MPI94].

The CAP methodology consists of dividing a complex operation into several suboperations with data dependencies, and to assign each of the suboperations to a thread in the thread hierarchy. The CAP programmer specifies in CAP the data dependencies between the suboperations, and assigns explicitly each suboperation to a thread. The CAP C/C++ preprocessor automatically generates parallel code that implements the required synchronizations and communications to satisfy the data dependencies specified by the user. CAP also handles for a large part memory management and communication protocols, freeing the programmer from low level issues.

CAP operations are defined by a single input, a single output, and the computation that generates the output from the input. Input and output of operations are called *tokens* and are defined as C++ classes with serialization routines that enable the tokens to be packed or serialized, transferred across the network, and unpacked or deserialized. Communication occurs only when the output token of an operation is transferred to the input of another operation. The CAP's runtime system ensures that tokens are transferred from one address space to another in a completely asynchronous manner (socket-based communication over TCP/IP). This ensures that communication takes place at the same time as computation[1].

The CAP language does not explicitly provide data transfer mechanisms such as a *send* or *receive* primitive. Communications, i.e. transfer of data among the address spaces, are automatically deducted by the CAP preprocessor and runtime system based on the CAP specification of the schedule and the mapping of leaf operations onto the threads available for computation. The CAP paradigm ensures that data transfer, i.e. token motion, occurs only at the end of the execution of operations in order to redirect the output token of an operation to the input of the next operation in the macro-dataflow. By managing automatically the communications without programmer intervention, the task of writing parallel programs is simplified.

[1] In the case of a single processor PC communications are only partially hidden, since the TCP/IP protocol stack requires some processing power.

An operation specified in CAP as a schedule of suboperations is called a *parallel operation*. A parallel operation specifies the assignment of suboperations to threads, and the data dependencies between suboperations. When two consecutive operations are assigned to different threads, the tokens are *redirected* from one thread to the other. As a result, parallel operations also specify communications and synchronizations between *leaf operations*. A leaf operation, specified as a C/C++ routine, computes its output based on its input. A leaf operation cannot incorporate any communication, but it may compute variables which are global to its thread.

Each parallel CAP construct consists of a split routine splitting an input request into sub-requests sent in a pipelined parallel manner to the operations of the available threads and of a merging function collecting the results. The merging function also acts as a synchronization means terminating the parallel CAP construct's execution and passing its result to the next operation after the arrival of all sub-results (Fig. 3.3).

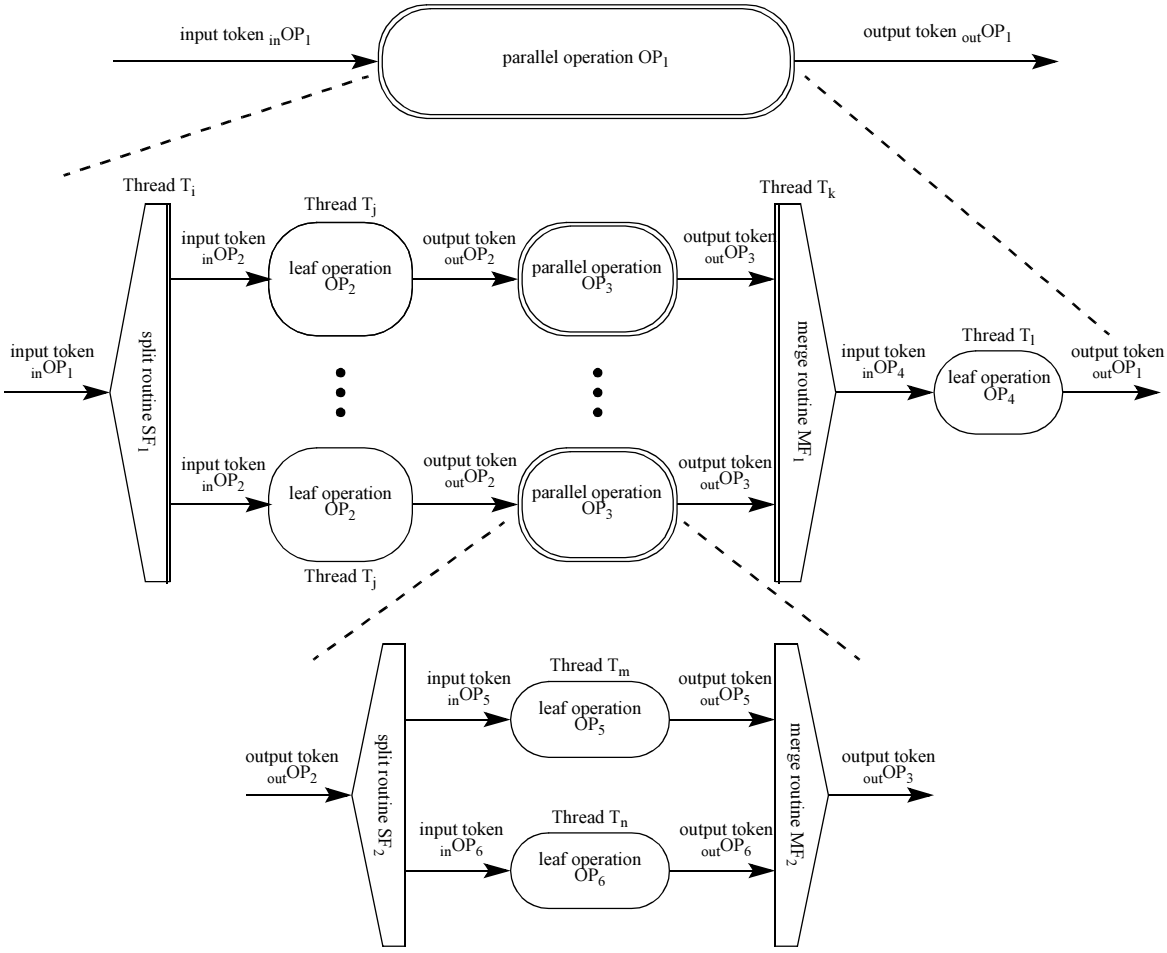


Fig. 3.3 Graphical CAP specification: parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch

3.2. Tokens

In CAP, pieces of data that flow through operations are called tokens since a CAP program is self-synchronized by its data motion. A token declaration is similar to the C/C++ struct/class declaration. Token's members can be any basic or complex (structure or class) type. Tokens can inherit from C++ struct/class. Along with a token declaration, the programmer must provide serialization routines necessary for moving the token from one address space to another, i.e. packing the token in a structure that is easily and efficiently sent through a TCP/IP connection by the CAP's runtime system and unpacking the same structure in the other address space (Section 5.2). Serialization occurs only when a token has to be transferred from one address space to another. Within an address space, CAP's runtime system uses the shared memory to move tokens from one thread to another.

Figure 3.4 shows an example of a token *TokenAT* declaration (lines 4-17). Serialization routines are not shown since Section 5.2 is completely devoted to this issue. The declaration of a token consists of the keyword *token* (lines 4) followed by any C/C++ struct/class field declarations (lines 6-13). As C++ classes, tokens may contain constructors and a destructor (line 15 and 16). Tokens can inherit from any C++ class/struct (line 4). One can even declare MFC objects inside a token (lines 10 and 11) as long as serialization routines are provided. However CAP does not allow pointer declarations since it does not know how to serialize them. That is the reason for lines 1-2 and 12-13.

```
1.  typedef char* PointerToCharT;
2.  typedef double* PointerToDoubleT;
3.
4.  token TokenAT : MyOwnInheritableClassT
5.  {
6.     int IntValue;
7.     float FloatValue;
8.     char CharArray[256];
9.     MyOwnClassT AnObject;
10.    CString StringOfChars;
11.    CArray<float, int> ArrayOfFloat;
12.    PointerToCharT PointerToCharP;
13.    PointerToDoubleT PointerToDoubleP;
14.
15.    TokenAT(int value, float numberOfDegree);
16.    ~TokenAT();
17. };
```

Fig. 3.4 Token declarations in CAP

3.3. Process hierarchy

The fundamental CAP methodology consists of specifying at a high-level of abstraction a process hierarchy, the operations offered by the processes in the hierarchy, and for parallel operations the schedule of suboperations described by a macro-dataflow depicted as a directed acyclic graph (DAG). The CAP language allows the programmer to work at a logical level. The

programmer defines logical processes, grouping hierarchically the processes. Only the lowest process hierarchy level is mapped to operating system threads according to a configuration file.

Figure 3.5 shows a process hierarchy declaration where 5 types of processes are defined, *ProcessAT* (lines 1-10), *ProcessBT* (lines 11-19), *ThreadAT* (lines 20-26), *ThreadBT* (lines 27-33), *ThreadCT* (lines 34-40), *ThreadDT* (lines 41-47) and *ThreadET* (lines 48-54). Processes are declared as C++ classes with the *process* keyword (lines 1, 11, 20, etc.). Note that there are processes with subprocess declarations, *ProcessAT* and *ProcessBT*, and processes without subprocess declarations, *ThreadAT*, *ThreadBT*, *ThreadCT*, *ThreadDT* and *ThreadET*. CAP makes a significant distinction between these two types of declaration. Therefore in the continuation of this dissertation the term *hierarchical process* refers to processes with subprocesses defined in a *subprocesses* section, the term *leaf process* or *thread* refers to processes without a *subprocesses* section, and the term *process* refers to either hierarchical or leaf processes. In a *subprocess* section programmers instantiate processes, e.g. the hierarchical process *ProcessAT* has three subprocesses: a *ProcessB* hierarchical process of type *ProcessBT* (line 4), a *ThreadA* leaf process of type *ThreadAT* (line 5) and a *ThreadB* leaf process of type *ThreadBT* (line 6). In the same manner *ProcessB* is hierarchically defined.

```
1.  process ProcessAT
2.  {
3.  subprocesses:
4.     ProcessBT ProcessB;
5.     ThreadAT ThreadA;
6.     ThreadBT ThreadB;
7.  operations:
8.     Op1 in TokenAT* InP out TokenDT* OutP;
9.     Op2 in TokenAT* InP out TokenDT* OutP;
10. };
11. process ProcessBT
12. {
13. subprocesses:
14.    ThreadCT ThreadC;
15.    ThreadDT ThreadD;
16.    ThreadET ThreadE;
17. operations:
18.    Op1 in TokenCT* InP out TokenDT* OutP;
19. };
20. process ThreadAT
21. {
22. variables:
23.    int ThreadLocalStorage;
24. operations:
25.    Op1 in TokenAT* InP out TokenBT* OutP;
26. };
27. process ThreadBT
28. {
29. variables:
30.    int ThreadLocalStorage;
31. operations:
32.    Op1 in TokenBT* InP out TokenCT* OutP;
33. };
34. process ThreadCT
35. {
36. variables:
37.    int ThreadLocalStorage;
38. operations:
39.    Op1 in TokenCT* InP out TokenCT* OutP;
40. };
41. process ThreadDT
42. {
43. variables:
44.    int ThreadLocalStorage;
45. operations:
46.    Op1 in TokenCT* InP out TokenCT* OutP;
47. };
48. process ThreadET
49. {
50. variables:
51.    int ThreadLocalStorage;
52. operations:
53.    Op1 in TokenCT* InP out TokenDT* OutP;
54. };
55.
56. ProcessAT MyHierarchy;
```

Fig. 3.5 CAP specification of a process hierarchy

In a hierarchical process, *high-level operations* also called *parallel operations* are defined as a schedule of suboperations (either parallel or sequential suboperations) offered by its subprocesses and/or offered by the process itself (lines 8, 9 and 18). The flow of tokens between

operations, i.e. macro-dataflow, is programmed by combining the high-level CAP constructs described in Section 3.5.

In leaf processes, operations are defined as standard sequential C/C++ subprograms (lines 25, 32, 39, 46 and 53), henceforth called *leaf operations* or *sequential operations*. The term *operation* is used to refer to either parallel or leaf operations.

Both parallel and leaf operations take a single input token (*InP*) and produce a single output token (*OutP*). The input token is in the parallel case redirected to the first operation met when flowing through the macro-dataflow, and in the sequential case it is the input parameter of the function. Alternatively, the output token is in the first case the output of the last executed operation in the dataflow and in the second case the result of the serial execution of the C/C++ function.

At line 56 the process hierarchy is instantiated. At run time and with a configuration file (Section 3.3.1), the *MyHierarchy* process hierarchy is created on a multi-PC environment[1], i.e. only the leaf processes are spawn since they are the threads executing the leaf operations they offer. Hierarchical processes are merely entities for grouping operations in a hierarchical manner. They do not participate as threads during execution since the parallel operations represent exclusively schedules used by leaf processes at the end of a leaf operation to locate the next leaf operation, i.e. the *successor*. In other words, tokens flow from leaf operations to leaf operations guided by parallel operations[2]. Therefore at run time behind the *MyHierarchy* process hierarchy, there are five threads of execution, i.e. *ThreadA*, *ThreadB*, *ThreadC*, *ThreadD*, *ThreadE*. These threads are distributed among the PCs according to a configuration file.

In leaf processes it is possible to declare thread local variables[3] in the *variables* sections (lines 22, 29, 36, 43 and 50), i.e. variables that are distinct across different thread instantiations. Hierarchical processes may also contain a *variables* section but care must be taken if the leaf subprocesses are mapped by the configuration file onto different address spaces. In that case a local copy of the variables present in the *variables* section is available in each address space. CAP's runtime system does not ensure any coherence between address spaces. Each of them gets a local copy of all the variables, i.e. the global variables, the thread local variables and the hierarchical process local variables. Also the CAP variables are supposed to be accessed only by the thread within which they are declared. It's possible for a thread to access a variable within another thread, but the CAP runtime system does not ensure any exclusive access to that variable. It is the programmer's responsibility to protect that variable (*mutex*) against concurrent accesses.

The execution model of *MyHierarchy* CAP process hierarchy is shown in Figure 3.6. Each thread in the process hierarchy executes a loop consisting of (1) removing a token from its input

[1] In the case where the configuration file is omitted, all the threads are spawned in a single address space.

[2] In addition, split and merge sequential functions enable scattering and gathering tokens.

[3] This is equivalent to the thread local storage (TLS) used in Win32 programs [Cohen98].

queue; (2) selecting the leaf operation to execute based on the current parallel operation; (3) running the leaf operation to produce an output token; (4) finding out the successor, i.e. the next leaf operation to be executed by a thread, and sending asynchronously the output token to that thread using the CAP Message Passing System (Chapter 5).

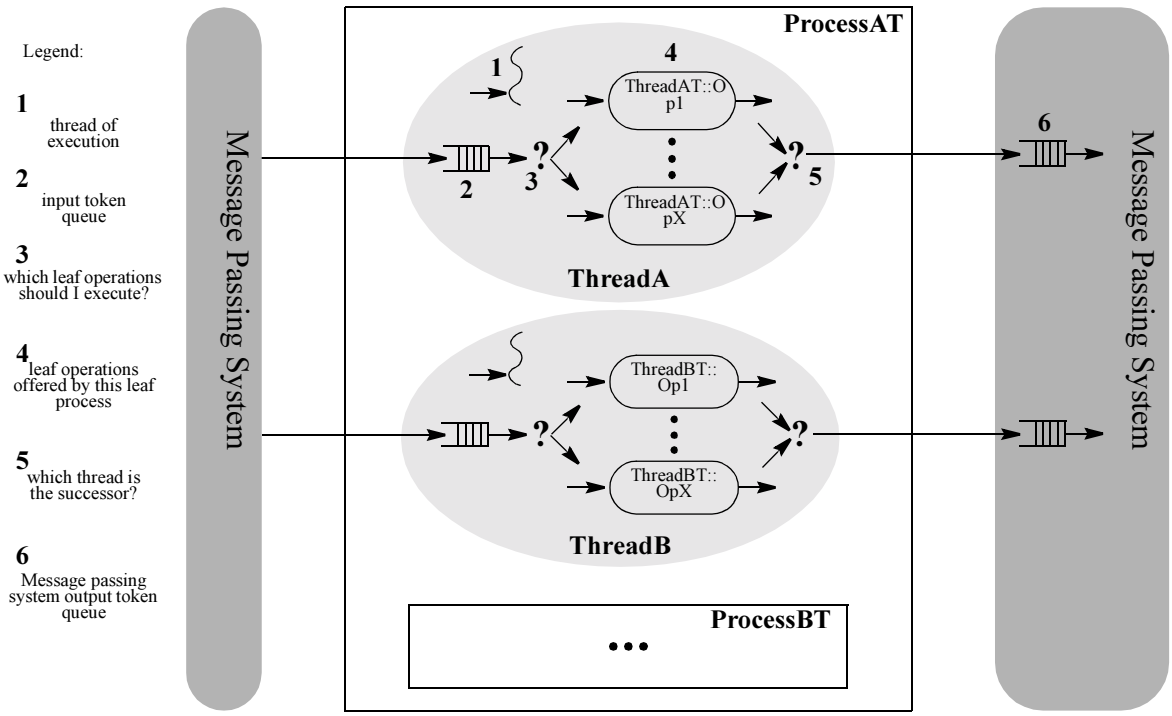


Fig. 3.6 Graphical representation of *MyHierarchy* CAP process hierarchy

3.3.1. Configuration file

In order to be able to run a CAP program on multiple address spaces, i.e. with several Windows NT processes distributed on a multi-PC environment, CAP's runtime system needs a configuration file. A configuration file is a text file specifying: (1) the number of address spaces (or Windows NT processes) participating in the parallel computation; (2) the PC's IP addresses on which these Windows NT processes run; (3) the Windows NT process executable filenames; (4) the mapping of threads to address spaces.

Figure 3.7 gives an example of a configuration file for the *MyHierarchy* process hierarchy shown in Figure 3.5. A configuration file always contains two sections:

1) A *processes* section (lines 2-6) where all the address spaces or Windows NT processes participating in the parallel computation are listed. For each of them a PC's IP address and an executable filename are given in order to allow the runtime system to spawn the Windows NT process on that PC with the specified executable file (lines 4-6). Note the special keyword *user* (line 3) used to refer the Windows NT process launched by the

user as a console command for starting the CAP program (Fig. 3.8). In that case, the PC's IP address and the executable file name is obviously not mentioned since the user launches that process.

2) A *threads* section (lines 7-12) specifying for each thread in the *MyHierarchy* process hierarchy the Windows NT process where the thread runs.

```
1.  configuration {
2.  processes :
3.     A ( "user" );
4.     B ( "128.178.75.65", "\\FileServer\SharedFiles\CapExecutable.exe" );
5.     C ( "128.178.75.66", "\\FileServer\SharedFiles\CapExecutable.exe" );
6.     D ( "128.178.75.67", "\\FileServer\SharedFiles\CapExecutable.exe" );
7.  threads :
8.     "ThreadA" (C);
9.     "ThreadB" (B);
10.    "ProcessB.ThreadC" (D);
11.    "ProcessB.ThreadD" (A);
12.    "ProcessB.ThreadE" (C);
13. };
```

Fig. 3.7 Configuration file declaring four address spaces, the PC's addresses where these four Windows NT processes run, the three executable filenames and the mapping of the five threads to the four address spaces

Figure 3.8 shows how to start a CAP program with a configuration file. At the beginning of the execution the CAP's runtime system parses the configuration file and thanks to the CAP Message Passing System (Chapter 5) spawns all the Windows NT processes (except the *user* one, Fig. 3.7, line 3) on the mentioned PCs. Then the four Windows NT processes (*A*, *B*, *C*, and *D*) parse the configuration file so as to spawn the five threads in their respective address space, e.g. the Windows NT process *C* spawns the *ThreadA* thread and the *ProcessB.ThreadE* thread. The CAP configuration file enables the possibility of running the CAP parallel program in a variety of configurations without recompiling the program. Any combination of multi-process or multi thread configuration can be specified within the CAP configuration file.

```
128.178.75.67> CapExecutable.exe -cnf \\FileServer\SharedFiles\ConfigurationFile.txt ...↵
```

Fig. 3.8 Starting a CAP program with a configuration file at DOS prompt

If the configuration file is omitted when starting a CAP program, then CAP's runtime system spawns all the threads in the current Windows NT process.

3.4. Operations

After having shown how to declare a process hierarchy and the operations offered by the processes in the hierarchy, this section looks at how to implement leaf operations and parallel operations.

CAP enables the programmer to declare an operation, either a parallel operation or a leaf operation, outside its process interface. This feature is extremely useful for extending the functionality of existing CAP programs. Instead of declaring the operation inside a given process, the programmer merely declares its interface globally, using the keyword *operation* (Fig. 3.9).

```
operation ProcessBT::Op2      // Additional parallel operation declaration
   in TokenDT* InP
   out TokenBT* OutP;

leaf operation ThreadET::Op2  // Additional leaf operation declaration
   in TokenBT* InP
   out TokenAT* OutP;
```

Fig. 3.9 Additional operation declarations

```
1.  ProcessAT MyHierarchy;
2.
3.  int main(int argc, char* argv[])
4.  {
5.     TokenAT* InP = new TokenAT(2, 4.562);
6.     TokenDT* OutP;
7.     call MyHierarchy.Op1 in InP out OutP;
8.     printf("Result = %s\n"), OutP->APointerToCharP);
9.     delete OutP;
10.    return 0;
11. }
```

Fig. 3.10 Synchronous call of a CAP operation from a sequential C/C++ program

```
1.  ProcessAT MyHierarchy;
2.
3.  int main(int argc, char* argv[])
4.  {
5.     capCallRequestT* CallRequestP;
6.     TokenAT* InP = new TokenAT(2, 4.562);
7.     TokenDT* OutP;
8.     start MyHierarchy.Op1 in InP out OutP return CallRequestP;
9.     ...
10.    OutP = capWaitTerminate(CallRequestP);
11.    printf("Result = %s\n"), OutP->APointerToCharP);
12.    delete OutP;
13.    return 0;
14. }
```

Fig. 3.11 Asynchronous call of a CAP operation from a sequential C/C++ program

CAP allows the programmer to call a CAP operation within a C/C++ program or library using the *call* keyword (Fig. 3.10, line 7). It is the programmer's responsibility to create the input token (line 5) and to delete the output token (line 9). The *call* instruction is synchronous, i.e. the thread that performs the *call* instruction is blocked until the called operation completes.

CAP also provides an asynchronous *start* instruction where the thread that performs the *start* instruction is not blocked and may synchronize itself with the *capWaitTerminate* CAP-library function (Fig. 3.11).

3.4.1. Leaf operations

As mentioned previously, CAP's primitive units of computation are leaf operations or sequential operations, i.e. C++ subprograms usable as building blocks for concurrent programming. A leaf operation is defined by a single input token, a single output token, and the C/C++ function body that generates the output from the input.

In our directed acyclic graph formalism (Fig. 3.3), we depict leaf operations as a single rounded rectangle with an input arrow with the input token's type, an output arrow with the output token's type, and the thread which performs this leaf operation (Fig. 3.12).

Figure 3.13 shows the implementation of the *ThreadAT::Op1* sequential operation using the *leaf operation* CAP construct. It is the responsibility of the leaf operation to create the output token (line 6) using one of the defined constructors. Once the leaf operation is completed, by default, the CAP's runtime system deletes the input token. A call to the *capDoNotDeleteInputToken* CAP-library function inside a leaf operation tells the CAP runtime system not to delete the input token of the leaf operation.

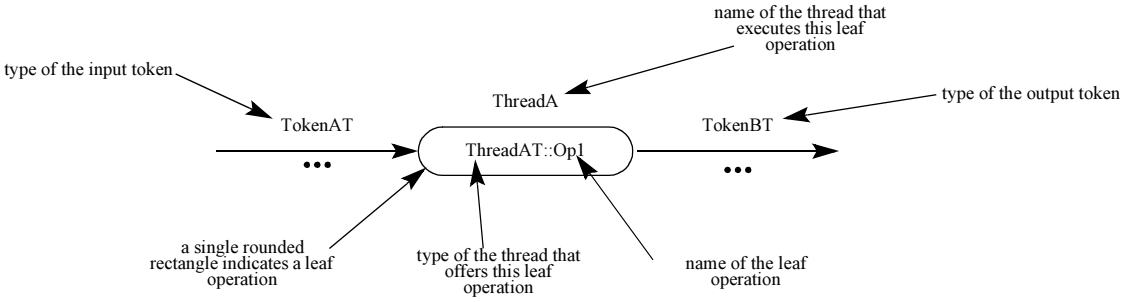


Fig. 3.12 A leaf operation with its input and output token. Single rounded rectangles depict leaf operations

A leaf operation may have side-effects, i.e. modifying shared global variables or thread local variables, so as to exchange information between threads in a same address space. It is the programmer's responsibility to ensure the coherence of the shared data by using appropriate synchronization mechanisms, e.g. *mutexes*, *semaphores*, *barriers*, provided by the CAP runtime library. Care must be taken when using these synchronization tools in order to avoid deadlocks. Indeed, CAP ensures that parallel programs are deadlock free by specifying macro-dataflows as directed acyclic graphs. However, if additional synchronizations outside CAP are used, deadlock free behavior cannot be guaranteed any more.

```
1.  leaf operation ThreadAT::Op1
2.     in TokenAT* InpP
3.     out TokenBT* OutP
4.  {
5.     ... // Any C/C++ statements
6.     OutP = new TokenBT("Switzerland");
7.     ... // Any C/C++ statements
8.  }
```

Fig. 3.13 CAP specification of a leaf operation. Note the *leaf* keyword

By default, when a leaf operation terminates, the CAP runtime system calls the successor, i.e. the next leaf operation specified by the DAG. CAP enables the programmer to prevent the CAP runtime system to call the successor, by calling the *capDoNotCallSuccessor* CAP-library function in the body of the leaf operation. The effect of the *capDoNotCallSuccessor* CAP-library function is to suspend the execution of the schedule of this particular token. To resume the execution of a suspended token, CAP supplies the *capCallSuccessor* CAP-library function. It is the programmer's responsibility to keep track of the suspended tokens, e.g. by having a global list of suspended tokens. A typical place to use this feature is when the leaf operation uses asynchronous system calls, e.g. the *ReadFile* Win32 system call. When the leaf operation finishes, the thread is able to execute other leaf operations while the OS is asynchronously doing the system call. When the system call completes, the callback routine resumes the schedule of the suspended token by calling the *capCallSuccessor* CAP-library function.

3.4.2. Parallel operations

As said in Section 3.3, a parallel operation, i.e. a hierarchically higher-level operation, is defined by an input token, an output token, and a schedule of suboperations that generates the output from the input. Parallel operations are possibly executed in parallel in the case of a parallel hardware environment, i.e. a cluster of PC.

In a directed acyclic graph (Fig. 3.3), parallel operations are depicted as a double rounded rectangle having an input arrow with the input token's type, and an output arrow with the output token's type (Fig. 3.14).

Figure 3.15 shows the implementation of the *ProcessAT::Op1* parallel operation using the *operation* CAP construct. In order to specify the content of the parallel operation, i.e. to build the schedule of suboperations (line 5), the programmer may use one or several parallel CAP constructs described in the following section. C/C++ statements are strictly forbidden since a parallel operation only describes the order of execution of suboperations.

3.5. Parallel CAP constructs

This section looks at the parallel CAP constructs. CAP provides several parallel constructs: *pipeline*, *if* and *ifelse*, *for*, *while*, *parallel*, *parallel while*, and *indexed parallel*. In this section, we will focus our interest on the *pipeline* and the *indexed parallel* constructs, the

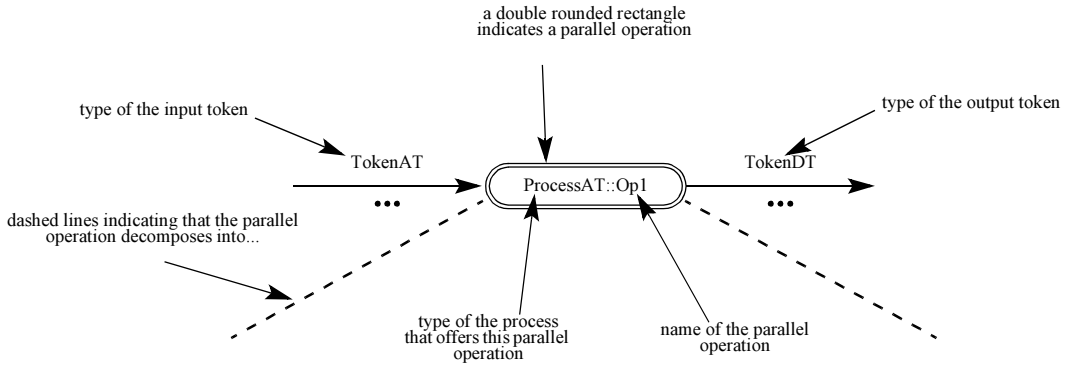


Fig. 3.14 A parallel operation with its input and output token. Note double rounded rectangle depict parallel operations.

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     // One or several parallel CAP constructs
6.  }
```

Fig. 3.15 CAP specification of a parallel operation

other parallel constructions are described in [Messerli99a]. CAP parallel constructs are used as building blocks for specifying the macro-dataflow of parallel operations, i.e. the schedule of the underlying leaf operations. These high-level parallel CAP constructs are automatically translated into a C/C++ source program, which, after compilation, runs on a multi-PC environment according to a configuration file specifying the mapping of the threads running the leaf operations onto the set of available Windows NT processes. For each of the parallel CAP constructs, its graphical specification, i.e its DAG, and its CAP specification are shown.

A question that may arise when reading this section is who executes parallel operations, i.e. who evaluates the boolean expression in *if*, *ifelse* and *while* CAP constructs, who executes the three expressions (init expression, boolean expression, and increment expression) in a *for* CAP construct, who executes the split functions in a *parallel* and *parallel while* CAP construct, and who executes the three expressions (init expression, boolean expression, and increment expression) and the split function in an *indexed parallel* CAP construct. The question of who executes a leaf operation is simple: the thread specified by the programmer executes the leaf operation. The question of who executes parallel operation is much more subtle [Gennart98a]. The job of a parallel operation is to redirect tokens from their producing sequential suboperation, i.e. the suboperation that generates it, to the consuming sequential suboperation, i.e. the suboperation that consumes it. The producing suboperation is not necessarily executed by the same thread as the consuming suboperation. If the producing thread is not in the same address space as the consuming thread, the token must be transferred from one address space to the other, a costly operation that should be performed only when explicitly required by the programmer. Therefore in the current implementation of the CAP runtime system, the producing thread performs the

parallel operation in order to decide itself where to redirect the token it produced. For example, the split functions in the *parallel*, *parallel while* and *indexed parallel* CAP constructs are always executed by the thread that produced the input token of the parallel construct.

3.5.1. The *pipeline* CAP construct

The *pipeline* CAP construct enables the output of one operation to be redirected to the input of another. It is the basic CAP construct for combining two operations. Figure 3.16 shows the DAG of the *pipeline* construct where three operations are connected in pipeline, i.e. *ThreadAT::Op1*, *ThreadBT::Op1* and *ProcessBT::Op1* operations. The output of the *ThreadAT::Op1* leaf operation is redirected to the input of the *ThreadBT::Op1* leaf operation whose output is redirected to the first leaf operation met when flowing through the DAG of the *ProcessBT::Op1* parallel operation.

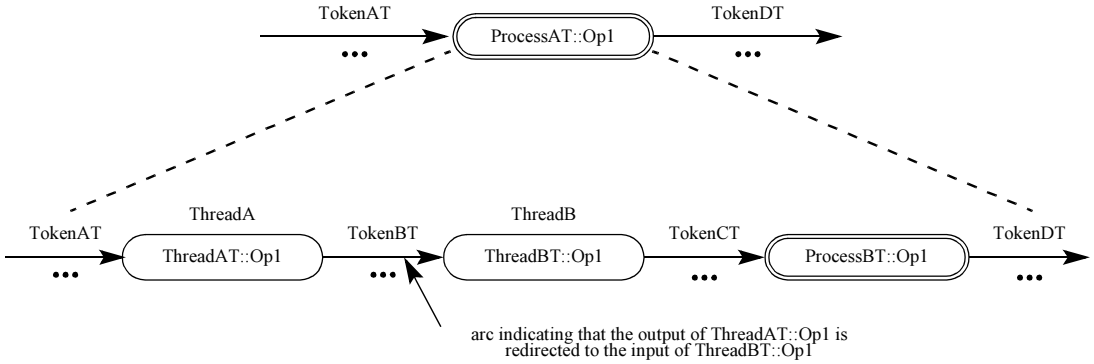


Fig. 3.16 Graphical CAP specification of the *pipeline* construct

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA.Op1
6.     >->
7.     ThreadB.Op1
8.     >->
9.     ProcessB.Op1;
10. }
```

Fig. 3.17 CAP specification of the *pipeline* construct

The CAP specification of the DAG in Figure 3.16 is shown in Figure 3.17. The output of one operation is redirected to the input of another operation using the >-> CAP construct (lines 6 and 8).

The execution of the program presented in Figure 3.17 strongly depends on the configuration file, i.e. whether the threads are mapped onto different processors or not. If the *ThreadA* thread, the *ThreadB* thread and the threads running the *ProcessB::Op1* parallel operations are mapped

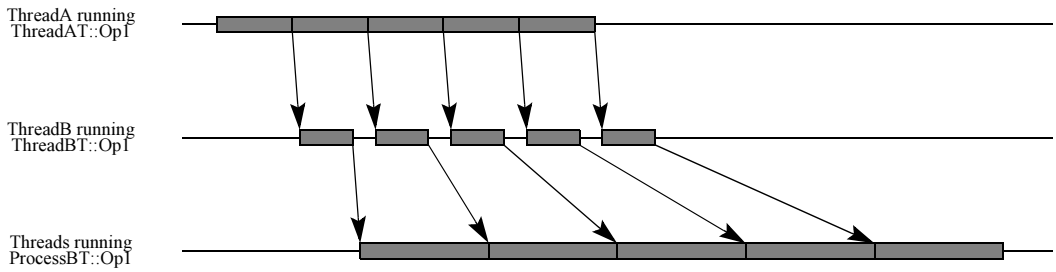


Fig. 3.18 Timing diagram of the execution of the 3 operations in a pipelined manner

```
1.  process ProcessAT
2.  {
3.  subprocesses:
4.     ProcessBT ProcessB;
5.     ThreadAT ThreadA[];
6.     ThreadBT ThreadB[];
7.     ...
```

Fig. 3.19 Declaring two pools of threads within a hierarchical process

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA[thisTokenP->ThreadAIndex].Op1
6.     >->
7.     ThreadB[thisTokenP->ThreadBIndex].Op1
8.     >->
9.     ProcessB.Op1;
10. }
```

Fig. 3.20 Selecting a thread within a pool

onto different processors and if the first operation in the horizontal branch, i.e. *ThreadAT::Op1* leaf operation, is fed with several tokens, then the 3 operations are executed in a pipelined manner like an assembly line (Fig. 3.18).

Supposing now that the *ProcessAT* hierarchical process declaration contains a pool of *ThreadAT* threads (Fig. 3.19, line 5) and a pool of *ThreadBT* threads (Fig. 3.19, line 6) instead of *ThreadA* and *ThreadB* threads (Fig. 3.5, line 5 and 6), then in the CAP specification of *ProcessAT::Op1* parallel operation the programmer must select within the 2 pools which threads are running *ThreadAT::Op1* and *ThreadBT::Op1* leaf operations (Fig. 3.20, lines 5 and 7).

The CAP runtime system provides programmers with the *thisTokenP* variable pointing to the token about to enter a CAP construct, e.g. line 5 *thisTokenP* refers to *ProcessAT::Op1* input token and line 7 refers to *ThreadAT::Op1* output token. The *thisTokenP* variable enables tokens

to be dynamically redirected according to their values. If the configuration file maps all the threads to different processors, then program of Figure 3.20 is executed in a pipelined parallel manner, i.e. *ThreadAT::Op1*, *ThreadBT::Op1* and *ProcessBT::Op1* operations are executed in pipeline while *ThreadAT::Op1* and *ThreadBT::Op1* operations are executed in parallel by the threads in the two pools. Figure 3.21 illustrates this schedule; in the example it is supposed that the *ThreadAT[]* thread pool is composed with 4 threads, i.e. *ThreadAT[0], ThreadAT[1], ThreadAT[2], ThreadAT[3],* and that the *ThreadBT[]* thread pool is composed with 2 threads, i.e. *ThreadBT[0], ThreadBT[1]*.

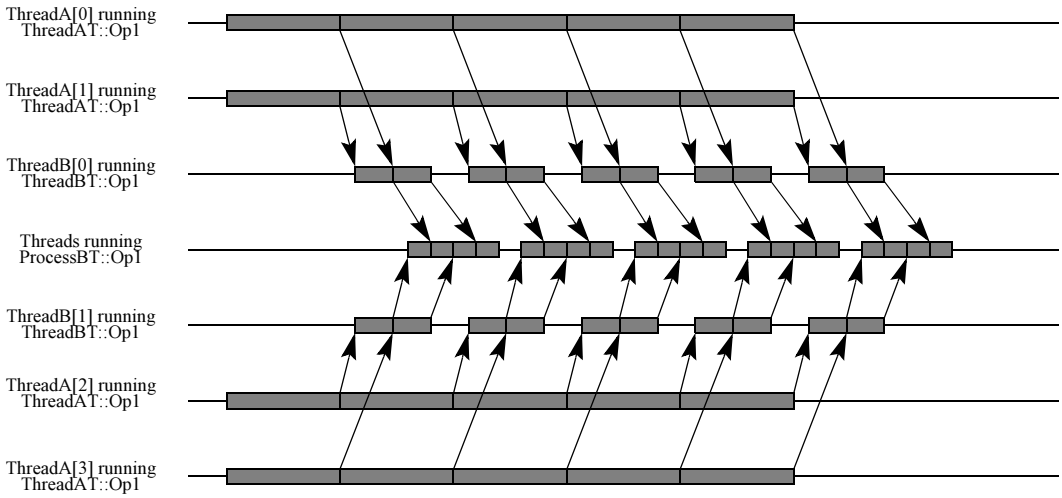


Fig. 3.21 Timing diagram of a pipelined parallel execution

In order to express this pipelined parallel execution, the DAG of *ProcessAT::Op1* hierarchical operation is modified so as to include two parallel branches (Fig. 3.22).

Selecting a thread within a pool raises the issue of load balancing in CAP (Fig. 3.20, lines 5 and 7). Supposing that the execution time of *ThreadAT::Op1* depends on the values in its input token and that *ThreadA[]* threads are selected in a round-robin fashion (Fig. 3.20, line 5), then the execution flow may be unbalanced, i.e. the loads of the different *ThreadA[]* threads may differ strongly one from another and therefore decrease overall performances. This issue is further discussed in Chapter 4.

3.5.2. The *indexed parallel* CAP construct

The *indexed parallel* CAP construct is a split-merge construct. It iteratively divides a token into several subtokens, performs in pipeline similar operations on each of the subtokens, and merges the results of the last operation in the pipeline. The iteration is based on a C/C++ *for* loop.

Figure 3.23 shows the DAG of the *indexed parallel* CAP construct. The *indexed parallel* construct input token (*TokenAT*) is iteratively divided into several subtokens by the *Split* split

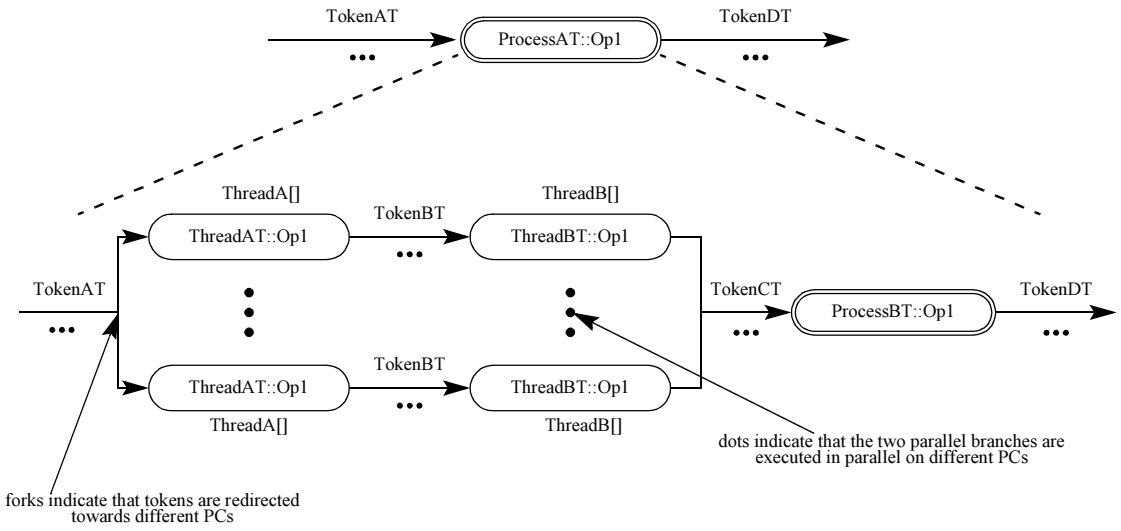


Fig. 3.22 Graphical representation of a pipelined parallel execution

function. Once a subtoken is generated, it is redirected to the *indexed parallel* body subconstruct which performs the operations in a pipelined manner (*ThreadBT::Op1* and *ThreadCT::Op1* operations). The output tokens of the *indexed parallel* body subconstruct are merged into the *indexed parallel* construct output token using *Merge* merge function by *ThreadA* thread. When all the subtokens are merged, the *indexed parallel* construct output token (*TokenDT*) is redirected to its successor. The operations contained in the *indexed parallel* body subconstruct (*ThreadBT::Op1* and *ThreadCT::Op1* operations) may execute in a pipelined manner (Fig. 3.17 and 3.18) or in a pipelined parallel manner (Fig. 3.20 and 3.21) depending whether different threads on different processors are selected for computation or not.

The CAP specification of the DAG in Figure 3.23 is shown in Figure 3.24. The *indexed parallel* CAP construct consists of the keyword *indexed* (line 15) followed by standard C/C++ *for* expressions (line 16), and of the keyword *parallel* (line 17) followed by the four construct initialization parameters (line 18) and an *indexed parallel* body subconstruct (lines 19-23). The first two initialization parameters are the split function and the merge function. A split function is a sequential C/C++ routine that creates a new subtoken from the *indexed parallel* construct input token and the current index values (lines 1-6). The split function is called for all the index values specified in the standard C/C++ *for* expressions (line 16) and may return a null subtoken to skip an iteration. A merge function is a sequential C/C++ routine that merges *indexed parallel* body subconstruct output tokens into the *indexed parallel* construct output token (lines 7-10). The last two initialization parameters are the name of the thread (*ThreadA*) that merges the output of the *indexed parallel* body subconstructs into the *indexed parallel* construct output token (*Out1*) and the output token declaration (line 18). Note the keyword *remote* which indicates that the *indexed parallel* construct input token (*thisTokenP*) is sent to *ThreadA* thread in order to initialize the *indexed parallel* construct output token in *ThreadA* address space. If, instead, the keyword *local* would have been used, the *indexed parallel* construct output token would have

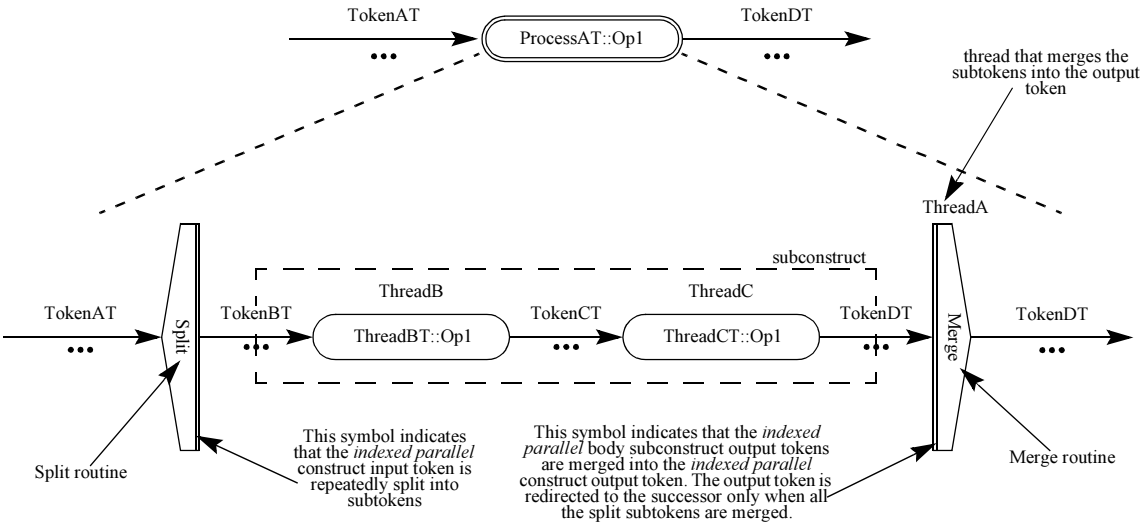


Fig. 3.23 Graphical CAP specification of the *indexed parallel* construct

```
1.  void Split(TokenAT* InP, TokenBT* &subtokenP, int index)
2.  {
3.     ... // Any C/C++ statements
4.     subtokenP = new TokenBT(...);
5.     ... // Any C/C++ statements
6.  }
7.  void Merge(TokenDT* OutP, TokenDT* subtokenP, int index)
8.  {
9.     ... // Any C/C++ statements
10. }
11. operation ProcessAT::Op1
12.    in TokenAT* InP
13.    out TokenDT* OutP
14. {
15.    indexed
16.    (int Index = 0; Index < 100; Index++)
17.    parallel
18.    (Split, Merge, ThreadA, remote TokenDT Out1(thisTokenP))
19.    (
20.       ThreadB.Op1
21.       >->
22.       ThreadC.Op1
23.    );
24. }
```

Fig. 3.24 CAP specification of the *indexed parallel* construct

been initialized in the current address space, i.e. the address space of the thread executing the *indexed parallel* CAP construct, and sent in the *ThreadA* address space. If the programmer doesn't specify anything, the CAP default behavior corresponds to the case denoted by the *remote* keyword. Depending on their sizes, the programmer can choose to transfer either the *thisTokenP* or the *Out1* token from the current address space to *ThreadA* address space.

3.6. Summary

The CAP computer-aided parallelization tool simplifies the creation of pipelined parallel distributed memory applications. Application programmers create separately the serial program parts and express the parallel behavior of the program with CAP constructs. Thanks to the automatic compilation of parallel applications, application programmers do not need to explicitly program the protocols to exchange data between parallel threads and to ensure their synchronizations. The predefined parallel CAP structures ensure that the resulting parallel program executes as an acyclic directed graph and is therefore deadlock free. Furthermore, due to its macro-dataflow nature, it generates highly pipelined applications where communication operations run in parallel with processing operations.

CAP parallel programs can be easily modified by changing the schedule of operations or by building hierarchical CAP structures. CAP facilitates the maintenance of parallel programs by enforcing a clean separation between the serial and the parallel program parts. Moreover, thanks to a configuration text file, generated CAP applications can run on different hardware configurations without recompilation.

The CAP approach works at a higher abstraction level than the commonly used parallel programming systems based on message passing (for example MPI [MPI94] and MPI-2 [MPI97]). CAP enables programmers to express explicitly the desired high-level parallel constructs. Due to the clean separation between parallel construct specification and the remaining sequential program parts, CAP programs are easier to debug and to maintain than programs which mix sequential instructions and message-passing function calls. Wilson [Wilson96] provides a thorough overview of existing approaches using object oriented programming for supporting parallel program development.

This chapter presented the fundamental aspects of the CAP language. The reader should understand the basic CAP specifications such as tokens, process hierarchies, leaf operations, parallel operations, configuration files and parallel constructions (pipeline, indexed parallel). Every parallel schedule can be expressed as a macro-dataflow; in this chapter, we have shown how these macro-dataflows can be defined and executed within the CAP framework.

One contribution of the present thesis is the develoment of several programs for testing and validating the CAP language. A timing analysis tool has been developed. This tool enables the programmer to analyze (once the execution is finished) the parallel program behavior by indicating which operation is executed by which thread on which computer at which time. Several programs were analyzed with this tool demonstrating the validity of the CAP approach.

Another contribution is to suggest the development of advanced CAP features such as the CAP token suspension (3.4.1). This feature allows the CAP programmer to control the CAP scheduling mechanism and therefore to develop more elaborated parallel programs.

4 CAP flow-control and load balancing issues

In this chapter we are concerned about the creation of efficient parallel programs. Writing efficient parallel programs also requires the ability of managing resources such as the amount of available memory. We explain in which situations resource management is required and how it can be expressed within the CAP language. Another important aspect of efficient parallel programming is the ability of balancing the load between several computing nodes. The CAP language features a mechanism allowing the programmer to control the flow of tokens between several operations. Resource management and load balancing can both be handled with the CAP flow-control mechanism.

4.1. Introduction[1]

Who regulates the number of tokens generated by the split routine? Who prevents the split routine from generating all its subtokens without knowing if the *indexed parallel* (or the *parallel while*) body subconstruct is consuming them at the same rate as they are produced? The answer is nobody. The split routine actually generates all the subtokens without stopping. If the split routine creates few subtokens this is probably acceptable, but if the split routine generates thousands of large subtokens then the problem is different. The processor will be overloaded executing continuously the split routine, memory will overflow, and the network interface will saturate sending all these subtokens. This will result in a noticeable performance degradation and later in a program crash with a *no more memory left* error message.

The problem with the *indexed parallel* (or the *parallel while*) CAP constructs is that the split routine generates input tokens faster than the merge routine consumes *indexed parallel* body subconstruct output tokens. This happens either because the *indexed parallel* body subconstruct output token rate is too slow or because the merging time of a token is too long. Therefore tokens accumulate somewhere in the pipeline between the split and the merge routine in the token queue located in front of the most loaded PCs, thus becoming the bottleneck of the application (Fig. 4.1).

In a multi-PC environment potential bottlenecks are: processors, memory interfaces, disks, network interfaces comprising the message passing system interface, the PCI network card adapter and the 100 Mbits/s Fast Ethernet network. If the processor, the memory interface or the disks are the bottleneck, then tokens are accumulated in the input token queue (Fig. 4.1) of the thread that executes the leaf operation that uses this offending resource, i.e. the resource that forms the bottleneck. Alternately, if the message passing system interface, the PCI network card adapter or the Fast Ethernet network is the bottleneck, then tokens are accumulated in the message passing system output queue (Fig. 4.1) of the offending Windows NT process, i.e. the

[1] Part of this chapter has been taken from [Messerli99a].

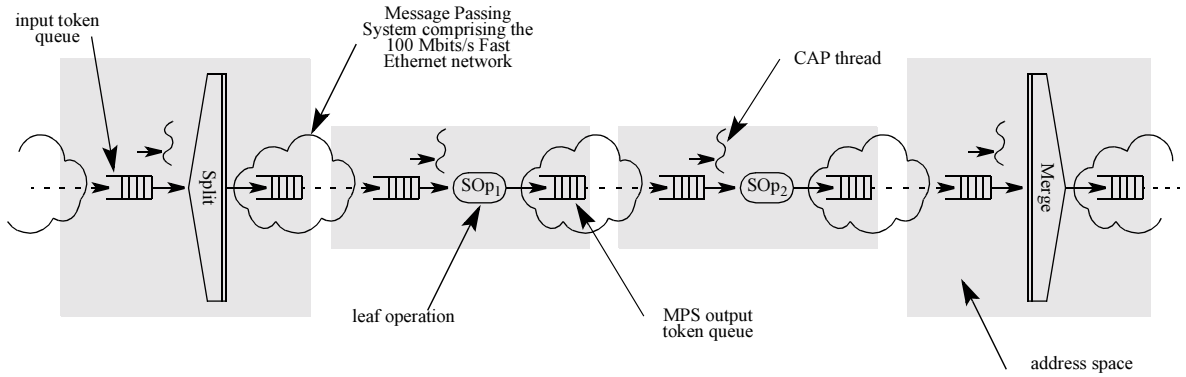


Fig. 4.1 Example of a 4-stage pipeline composed of a split routine, two intermediate leaf operations and a merge routine. Note the input token queues in front of each CAP threads and the MPS output token queues at the border of each address spaces.

Windows NT process that sends too many tokens. The consequence of such a congestion point is that computing resources are monopolized for handling such a peak in the flow of tokens, e.g. memory space for storing tokens and computing power for sending/receiving tokens, leading to a rapid degradation of performances (virtual memory thrashing).

Best performances are achieved when the split routine generates tokens at the same rate as the merge routine consumes the *indexed parallel* body subconstruct output tokens. In that situation the most loaded component in the pipeline becomes the bottleneck, i.e. it is active 100% of the time and no further improvement is possible.

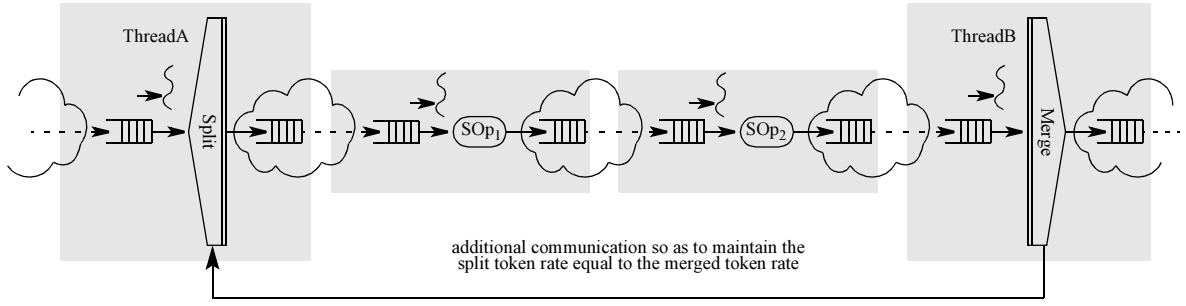


Fig. 4.2 High-level flow-control mechanism requiring additional communication between the thread that runs the merge routine (the ThreadB thread) and the thread that runs the split routine (the ThreadA thread) so as to maintain the split token rate equal to the merged token rate

4.2. CAP flow-control issues

The adopted flow-control approach consists in applying a high-level flow-control mechanism. The offending *indexed parallel* (or *parallel while*) CAP constructs are replaced by a combination of *indexed parallel* and *for* CAP constructs [Messerli99a] in order to maintain the difference between the number of split tokens and the number of merged tokens constant by

having additional communication between the thread that runs the merge routine and the thread which runs the split routine (Fig. 4.2). By doing this, the program regulates by itself the split token rate according to the merged token rate, thus preventing tokens from being accumulated somewhere in the pipeline causing a possible performance degradation.

Figure 4.3 shows a CAP program with an *indexed parallel* CAP construct at lines 7-13. The inline operation at line 5 constraints the C/C++ *indexed parallel* indexing expression (line 8) and the *Split* routine (line 10) to be executed by the *ThreadA* thread. The *Merge* routine is executed by the *ThreadB* thread (line 10). Without an adequate flow-control mechanism this parallel CAP construct generates 10'000 *TokenCT* tokens that will certainly fill up the memory and cause a memory overflow condition.

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA.{}
6.     >->
7.     indexed
8.     (int Index = 0; Index < 10000; Index++)
9.     parallel
10.    (Split, Merge, ThreadB, remote TokenDT Out(thisTokenP))
11.    (
12.       ProcessB.Op1
13.    );
14. }
```

Fig. 4.3 Example of a CAP program requiring a flow-control preventing the split routine from generating 10'000 tokens

In order to maintain the difference between the number of split tokens and the number of merged tokens constant, e.g. 20 tokens, the *indexed parallel* CAP construct is rewritten. Figure 4.4 shows the first high-level flow-controlled split-merge construct replacing the original *indexed parallel* CAP construct in Figure 4.3. The idea consists of generating a small amount of tokens, e.g. 20, with a first *indexed parallel* CAP construct (lines 7-10) and to redirect 500 times, with a second *for* CAP construct (line 12), each of these 20 tokens, through the pipeline formed by the split routine (line 14), the parallel *ProcessBT::Op1* operation (line 16) and the merge routine (line 18). In other words, the *indexed parallel* CAP construct (lines 7-10) generates 20 tokens (or 20 parallel flows) and each of these 20 tokens circulates, in parallel, 500 times through the pipeline (lines 14-18) thanks to the *for* CAP construct (line 12). The combination of the *indexed parallel* and the *for* CAP constructs ensures that exactly 20 tokens are always flowing in the pipeline, thus preventing a large memory consumption and a possible memory overflow.

This first high-level flow-controlled split-merge construct (Fig. 4.4) requires one additional communication between the thread which runs the merge routine and the thread that runs the split routine. For example, on a 6-stage pipeline (split routine, 4 intermediate pipe stages, and

merge routine) this correspond to a communication increase of 20%, i.e. 6 token transfers instead of 5 token transfers.

The number of tokens that are simultaneously flowing in the pipeline, i.e. within the *for* CAP construct (lines 14-18) is henceforth named the *filling factor*. The performance of a flow-controlled split-merge construct (Fig. 4.4) highly depends on its filling factor. A too large filling factor creates a congestion point that consumes lots of memory degrading performance and possibly crashing the program. A too small filling factor does not provide enough tokens to saturate one of the PC's components, i.e. the most loaded, on which the pipeline executes. This decreases the parallelism between the different pipeline stages and performance suffers. At the extreme case, a filling factor of 1 corresponds to a serial execution of the split-merge construct. Best performances are obtained when the filling factor is equal to the minimum factor that saturates the most loaded components on which the pipeline executes.

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA.{ }                                          filling factor
6.     >->                                                  ↙
7.     indexed
8.     (int IdxFlwCtrl1 = 0; IdxFlwCtrl1 < 20; IdxFlwCtrl1++)
9.     parallel
10.    (SplitFlwCtrl1, MergeFlwCtrl1, ThreadB, remote FlwCtrlT Out1(thisTokenP))
11.    (
12.       for (int IdxFlwCtrl2 = 0; IdxFlwCtrl2 < 10000/20; IdxFlwCtrl2++)
13.       (
14.          ThreadA.SplitFlwCtrl2
15.          >->
16.          ProcessB.Op1
17.          >->
18.          ThreadB.MergeFlwCtrl2
19.       )
20.    )
21.    >->
22.    ThreadB.ReturnOutputToken;
23. }
```

Fig. 4.4 First high-level flow-controlled split-merge construct that requires one communication between the thread that runs the merge routine, i.e. the *ThreadB* thread, and the thread that runs the split routine, i.e. the *ThreadA* thread, for each merged tokens

Our first attempt of a high-level flow-controlled split-merge construct (Fig. 4.4) requires one communication between the thread that runs the last leaf operation of the pipeline (or merge routine, line 18) and the thread that runs the first leaf operation of the pipeline (or split routine, line 14) for each iterations (provided that *ThreadA* thread and *ThreadB* thread are mapped onto two different PCs). Remember that the two original split-merge CAP constructs do not require any additional communications but they do not offer any flow-control mechanism. Although communications are mostly overlapped by computations, sending and receiving many small tokens such as those required for the high-level flow-control add an additional load on processors and

network interfaces. Often, network interfaces and the processing power required for the TCP/IP protocol represent the bottleneck. Therefore, even if flow-control tokens are small (between 50 and 100 Bytes), it is essential to reduce communications so as to alleviate these two scarce resources.

The second high-level flow-controlled split-merge construct in Figure 4.5 requires much less communication between the thread that runs the merge routine, i.e. the *ThreadB* thread, and the thread that runs the split routine, i.e. the *ThreadA* thread. This is achieved by having a second *indexed parallel* CAP construct (lines 16-19) within the *for* CAP construct (line 12). Consequently only every each 10 tokens merged by the *ThreadB* thread using the *MergeFlwCtrl2* routine a *FlwCtrlT* token (line 19, *Out2* token) is sent back to the *ThreadA* thread (line 14) which uses the *SplitFlwCtrl2* routine to split the next 10 tokens feeding the pipeline. The empty inline leaf operation at line 14 forces the *ThreadB* thread to send the flow-control *Out2* token to the *ThreadA* thread. Otherwise the *SplitFlwCtrl2* routine would have been executed by the *ThreadB* thread which is different than the original program (Fig. 4.3, line 10) where the *Split* routine is executed by the *ThreadA* thread. The outer *indexed parallel* CAP construct (lines 7-10) with an appropriate filling factor, allows to saturate the most loaded PC's components in the pipeline. With an outer filling factor of 1, the pipeline is empty while the flow-control *Out2* token is being sent from the *ThreadB* thread to the *ThreadA* thread thus decreasing performance.

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA.{ }
6.     >->
7.     indexed
8.     (int IdxFlowCtrl1 = 0; IdxFlowCtrl1 < 2; IdxFlowCtrl1++)
9.     parallel
10.    (SplitFlowCtrl1, MergeFlowCtrl1, ThreadB, remote FlowCtrlT Out1(thisTokenP))
11.    (
12.       for (int IdxFlowCtrl2 = 0; IdxFlowCtrl2 < 10000/(2*10); IdxFlowCtrl2++)
13.       (
14.          ThreadA.{ }
15.          >->
16.          indexed
17.          (int IdxFlowCtrl3 = 0; IdxFlowCtrl3 < 10; IdxFlowCtrl3++)
18.          parallel
19.          (SplitFlwCtrl2, MergeFlwCtrl2, ThreadB, remote FlwCtrlT Out2(thisTokenP))
20.          (
21.             ProcessB.Op1
22.          )
23.       )
24.    )
25.    >->
26.    ThreadB.ReturnOutputToken;
27. }
```


Fig. 4.5 High-level flow-controlled split-merge construct requiring much less communication between the thread that runs the merge routine, i.e. the *ThreadB* thread, and the thread that runs the split routine, i.e. the *ThreadA* thread

In contrast to the first high-level flow-controlled split-merge construct (Fig. 4.4), in the improved construct (Fig. 4.5) the filling factor is composed of 2 values. A first value named *filling factor*$_1$ specifies the number of bunches of tokens (line 8), and a second value named *filling factor*$_2$ specifies the number of tokens per bunch (line 17). Additional flow-control communications occur only every each *filling factor*$_2$ merged tokens. The maximum number of tokens that may simultaneously be in the pipeline is *filling factor*$_1$ multiplied by *filling factor*$_2$ tokens. With a *filling factor*$_2$ of 1, the improved high-level flow-controlled split-merge construct (Fig. 4.5) is equivalent to the first construct (Fig. 4.4).

The CAP preprocessor is able to generate by itself the two high-level flow-controlled split-merge constructs of Figures 4.4 and 4.5 from the original *indexed parallel* (or *parallel while*) CAP constructs thus simplifying the task of writing parallel CAP programs. Figure 4.6 shows how the programmer can specify with the keyword *flow_control* a split-merge CAP construct with a high-level flow-control mechanism. The first flow-controlled construct in Figure 4.4 is generated if the *flow_control* keyword contains a single argument, i.e. the filling factor. The second flow-controlled construct in Figure 4.5 is generated if the *flow_control* keyword contains two arguments, i.e. the *filling factor*$_1$ and *filling factor*$_2$. Programmers must be aware that the two high-level flow-controlled *parallel while* and *indexed parallel* CAP constructs (Fig. 4.6) require additional communications. Performance may suffer if the pipeline empties, either due to the additional communication cost or due to too small filling factors.

```
1.  operation ProcessAT::Op1
2.     in TokenAT* InP
3.     out TokenDT* OutP
4.  {
5.     ThreadA.{ }
6.     >->
7.     flow_control(20) // or flow_control(2, 10)
8.     indexed
9.     (int Index = 0; Index < 1000; Index++)
10.    parallel
11.    (Split, Merge, ThreadB, remote TokenDT Out1(thisTokenP))
12.    (
13.       ProcessB.Op1
14.    );
15. }
```

Fig. 4.6 Automatic generation of the 2 flow-controlled split-merge constructs using the CAP preprocessor

The high-level flow-control mechanism CAP constructs prevent a program from crashing but has several disadvantages. It requires additional communication that may slightly reduce performances. Also, experience has shown that it is quite difficult to adjust the filling factors so as to make the most loaded PC the bottleneck thus giving the best performance. Finally, the high-level flow-control (i.e. the filling factor) lacks some dynamicity in order to take into account changing conditions at run time, for example when processor, memory, disk or network utilizations change.

4.3. Issues of load balancing in a pipelined parallel execution

Figure 4.7 shows a pipelined parallel execution on a multi-PC environment. The master PC divides a large task into many small jobs which are executed in parallel on *M* different pipelines of execution. In this example, the pipeline of execution is composed of *N* different PCs each performing a particular stage. After a job has performed the *N* stages of a pipeline, the result is sent back to the master PC where all the job results are merged into the result buffer. When all the jobs making up the task are completed, the result of the execution of the task is sent back to the client PC which requested the execution of the task. The master PC is responsible for distributing the jobs making up a task and for balancing the load amongst the *M* pipelines.

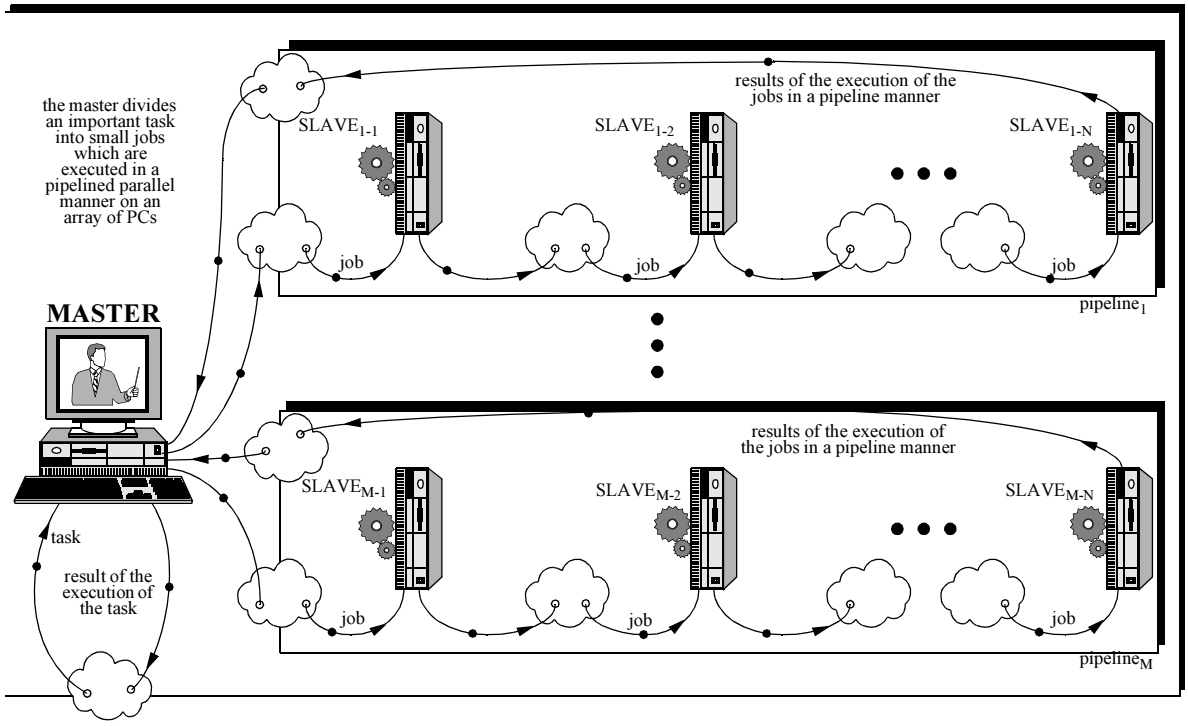


Fig. 4.7 Execution in a pipelined parallel manner on a multi-PC environment

The graphical representation (DAG) of the pipelined parallel schedule depicted in Figure 4.7 is shown in Figure 4.8. The input of the parallel *ParallelProcessingServerT::PerformTask* operation is a task request, i.e. a *TaskT* token. Using a split-merge CAP construct, this task request is divided into many small jobs by the *Master* thread using the *SplitTaskIntoJobs* split routine. Then, each job is routed through one of the *M* pipelines. At the end of the *M* pipelines, the job results, i.e. the *JobResultT* tokens, are merged into a *TaskResultT* token using the *MergeJobResults* merge routine. When all the job results are merged, the result of the execution of the task, i.e. the *TaskResultT* token, is passed to the next operation.

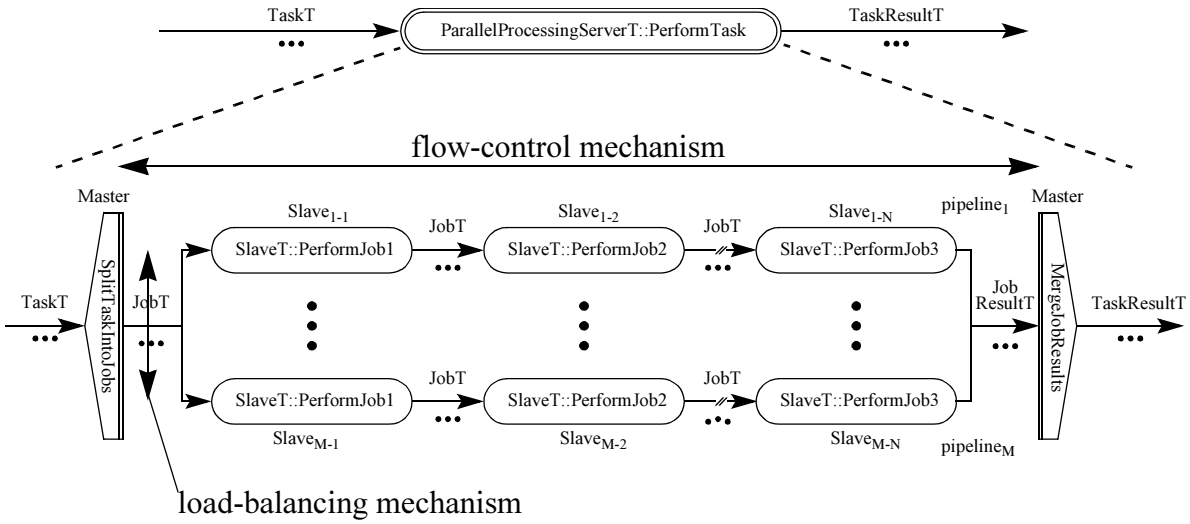


Fig. 4.8 Graphical representation of a pipelined parallel execution. The horizontal arrow represents the flow-control mechanism and the vertical arrow represents the load-balancing mechanism

A pipelined parallel execution such as the one depicted in the schedule of Figure 4.8 raises two orthogonal problems, i.e. flow-control and load-balancing. The mechanism of flow-control, presented in the previous section, attempts to regulate the flow of tokens through a single pipeline so that the split token rate is both high enough for maintaining the most loaded PC active 100% of the time and low enough for preventing tokens from accumulating in front of the most loaded PC and causing problems to the application.

```
1.  const int NUMBER_OF_SLAVES = 5;
2.  const int FILLING_FACTOR_PER_SLAVE = 3;
3.
4.  operation ParallelProcessingServerT::PerformTask
5.     in TaskT* InP
6.     out TaskResultT* OutP
7.  {
8.     Master.{ }
9.     >->
10.    flow_control(NUMBER_OF_SLAVES * FILLING_FACTOR_PER_SLAVE)
11.    indexed
12.    (int TaskIndex = 0; TaskIndex < NumberOfTasks; TaskIndex++)
13.    parallel
14.    (Split, Merge, Master, local TaskResultT Out(thisTokenP))
15.    (
16.       Slave[TaskIndex%NUMBER_OF_SLAVES].PerformJob
17.    );
18. }
```

Fig. 4.9 Example of a CAP program where the jobs are statically distributed in round-robin manner among the slaves

A mechanism of load-balancing balances the load among the parallel pipelines (*pipeline*$_1$ to *pipeline*$_M$ in Figure 4.7 or 4.8), i.e. prevents the most loaded thread in each of the parallel pipe-

lines from being more loaded[1] than the most loaded threads in other pipelines. Such a mechanism is necessary when the utilization of the computing resources, e.g. processors, memory, disks, network, etc., is unequal between the PCs on which the parallel pipelines execute, i.e. when:

- The execution time of a pipeline stage is data dependent, e.g. the number of iterations in a Mandelbrot computation depends on the initial value of the free variable (complex number).
- The multi-PC environment is made of different machines, e.g. Pentium Pro 200 MHz PCs, Pentium II 333MHz PCs, Pentium 90 MHz PCs, etc.
- The multi-PC environment is shared among several users, e.g. on a network of multi-user workstations.

```
1.  const int NUMBER_OF_SLAVES = 5;
2.  const int FILLING_FACTOR_PER_SLAVE = 3;
3.
4.  operation ParallelProcessingServerT::PerformTask
5.     in TaskT* InP
6.     out TaskResultT* OutP
7.  {
8.     Master.{}
9.     >->
10.    indexed
11.    (int SlaveIndex = 0; SlaveIndex < NUMBER_OF_SLAVES; SlaveIndex++)
12.    parallel
13.    (LBSplit, LBMerge, Master, local TaskResultT Out1(thisTokenP))
14.    (
15.       flow_control(FILLING_FACTOR_PER_SLAVE)
16.       indexed
17.       (int TaskIndex = 0; IsTaskAvailable(); TaskIndex++)
18.       parallel
19.       (Split, Merge, Master, local TaskResultT Out2(thisTokenP))
20.       (
21.          Slave[SlaveIndex].PerformJob
22.       )
23.    );
24. }
```

Fig. 4.10 Example of a CAP program where the jobs are dynamically distributed amongst the slaves according to their loads, i.e. the master balances the load amongst the slaves

Figure 4.9 shows a CAP program where the master PC divides a large task into small jobs and statically distributes them in a round-robin fashion (line 16) amongst the worker slaves. The *Split* routine splits the input *TaskT* task into jobs. The *Merge* routine merges the results of the jobs into a single *TaskResultT* token. The parallel *ParallelProcessingServerT::PerformTask* operation consists of iteratively getting a job (*Split* function), performing the job by a slave and merging its result into the output *TaskResultT* token using the high-level flow-controlled

[1] The load of a thread is defined as the percentage of elapsed time that this thread is executing a pipe stage.

indexed parallel CAP construct (lines 11-17). Note that the selection of the slave (line 16) is done statically without balancing the loads of the slaves.

The principle of a load-balancing mechanism consists of having a *distinct* high-level flow-controlled split-merge construct for each of the parallel pipelines. Each flow-controlled split-merge construct is dedicated to a specific slave. There is exactly one flow-controlled split-merge construct per slave. As soon as a slave completes a job, a new job is redirected to it. In that way the load is automatically balanced, since jobs are dynamically redirected to slaves completing their computation.

Figure 4.10 shows the CAP program equivalent to Figure 4.9 with a load-balancing mechanism. The first *indexed parallel* CAP construct (lines 10-13) generates *NUMBER_OF_SLAVES* parallel flows of execution consisting each of a high-level flow-controlled *indexed parallel* CAP construct (lines 15-19) distributing jobs to its slave. As soon as a slave completes a job, a new job is redirected to it (as long as tasks are available, line 17) thanks to the flow-controlled *indexed parallel* CAP construct (lines 15-19). The filling factor (line 15) ensures that slaves are active 100% of the time and that computations overlap communications.

```
1.  const int NUMBER_OF_SLAVES = 5;
2.  const int FILLING_FACTOR_PER_SLAVE = 3;
3.
4.  operation ParallelProcessingServerT::PerformTask
5.     in TaskT* InP
6.     out TaskResultT* OutP
7.  {
8.     Master.{}
9.     >->
10.    flow_control(NUMBER_OF_SLAVES * FILLING_FACTOR_PER_SLAVE)
11.    indexed
12.    (int TaskIndex = 0; TaskIndex < NumberOfTasks; TaskIndex++)
13.    parallel
14.    (Split, Merge, Master, local TaskResultT Out2(thisTokenP))
15.    (
16.       Slave[cap_fcindex0%NUMBER_OF_SLAVES].PerformJob
17.    );
18. }
```

Fig. 4.11 Automatic generation of a dynamic load balanced program using the CAP preprocessor, i.e. the load is automatically balanced by the master among the slaves.

The CAP preprocessor is able to generate itself the load balancing mechanism. CAP constructs thus simplifies the task of writing parallel CAP programs. Figure 4.11 shows how the programmer can specify with the keyword *flow_control* and *cap_fcindex0* a load balanced split-merge CAP construct. The *cap_fcindex0%SlaveIndex* represents the *SlaveIndex* of the example in Figure 4.10. The programs of Figures 4.10 and 4.11 are equivalent.

To summarize, CAP provides an automatic mechanism for balancing the load between worker threads. This construction is based on the combination of the *flow_control* and *cap_fcindex0*

keywords. This provides to the application programmers a simple and efficient load-balancing mechanism.

4.4. Summary

In the previous chapters, we have already shown the capability of the CAP language to specify parallel schedules. In this chapter, we explained how the CAP programmer controls the flow of tokens within these parallel schedules. This flow-control CAP mechanism leads to the specification of efficient parallel programs in terms of resource management and load balancing. The flow-control CAP mechanism has been used in the CAP application examples described in chapters 6 to 10.

The Visible Human application [Hersch00][VisibleHuman98] developed at the Peripheral Systems Laboratory (LSP) of EPFL is based on the CAP technology. In order to avoid memory overflows and to regulate the disk accesses, the Visible Human application uses several stages of flow-control CAP constructions. The Visible Human application is running for 3 years and 675,000 slice extractions have been carried out. The server, based on Window NT, is rebooted only a few times a year. This example tends to show that the CAP flow-control mechanism fulfills its requirements and is reliable.

Within the context of this thesis, I contributed in defining the flow-control feature and proposed to use it for load balancing.

5 CAP Message passing and Serialization

In this chapter, we describe the main issues of the CAP runtime system. We introduce the CAP Message-Passing System (MPS), i.e. the inter-process communication mechanism. MPS is an important part of the CAP runtime system. We explain the CAP token serialization mechanism. We present also an automatic serialization tool reducing the programming effort to serialize the data structures. We conclude this chapter by studying the CAP integration capabilities.

5.1. The CAP token-oriented Message-Passing System (MPS)

This section describes the token-oriented message-passing system named MPS developed within the context of the CAP computer aided parallelization tool. The development of the CAP Message-Passing System is an important part of Vincent Messerli's thesis [Messerli99a]. We present here only the main aspects which are necessary to understand our work on automatic serialization.

The *sockets* abstraction was first introduced in 1983 in the 4.2 Berkeley Software Distribution (BSD) Unix [Wright95] to provide a generic and uniform application programming interface (API) to interprocess and network communication protocols such as the Internet protocols [Stevens94]. Since then, all Unix versions, e.g. Solaris from Sun Microsystems, have adopted the sockets abstraction as their standard API for network computing. Microsoft has also adopted and adapted the sockets paradigm as a standard on its various operating systems, i.e. Windows 3.11, Windows 95/98 and Windows NT (Windows Sockets 1.1 and 2 API). All the programming details for sockets are available in [Stevens90].

A socket is a communication endpoint, i.e. an object through which a sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Sockets exchange data with other communication endpoints, e.g. other socket, in the same *communication domain* which uses for example the Internet protocol suite.

To implement a portable message passing system for transmitting CAP tokens from one address space to another different possibilities are available. Since CAP needs a reliable means of communication, we choose an implementation based on *stream sockets*. Stream sockets provide sequenced[1], reliable, flow-controlled, two-way, connection-based data flows without record boundaries, i.e. byte streams. Stream sockets use the underlying TCP/IP protocol stack and may have a name composed of a 32-bit IP address and a 16-bit TCP port number. Stream sockets are bi-directional: they are data flows that can be communicated in both directions simultaneously, i.e. full-duplex.

[1] Sequenced means that packets are delivered in the order sent

Contrary to the UDP protocol (datagram sockets), the TCP protocol (stream sockets) has no notion of packets with boundaries. A TCP/IP connection is a two-way reliable flow-controlled stream of bytes. The receiver has no information about the number of bytes that the sender is transmitting and how its *WSABUF* buffers are scattered, i.e. the number of *WSABUF* buffers and their sizes. Figure 5.1 depicts a TCP/IP transmission example where the sender with a single call to the *WSASend* routine transmits a whole group of scattered *WSABUF* buffers, while the receiver has to implement a mechanism that repeatedly calls the *WSARecv* routine in order to fill the destination scattered *WSABUF* buffers with the incoming data. Remember that a read completion may occur regardless of whether or not the incoming data fills all the destination *WSABUF* buffers.

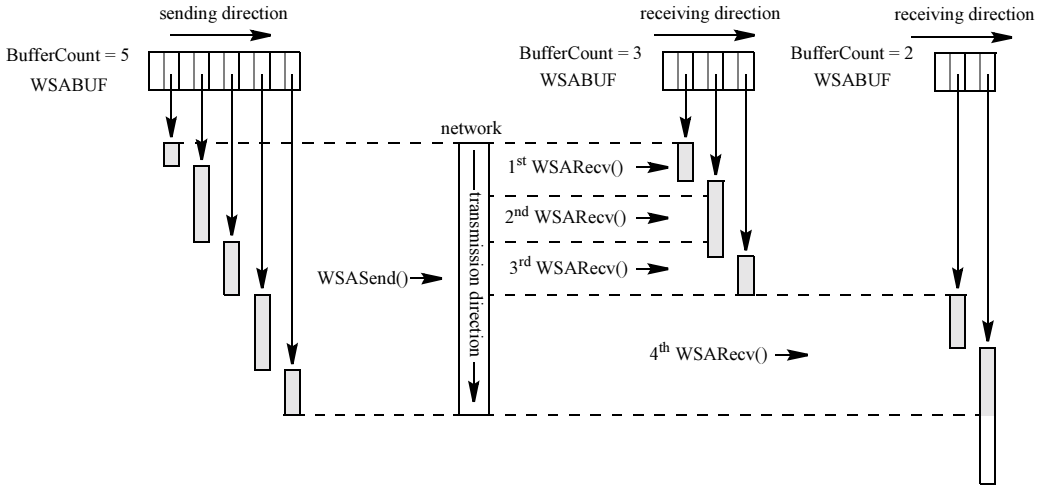


Fig. 5.1 The TCP protocol has no notion of packets. There is no correspondence between the *WSASend* and *WSARecv* calls.

To transfer CAP tokens through a TCP/IP connection, a three steps mechanism indicating to the receiver the number of scattered buffers and their sizes has been implemented (Fig. 5.2). For the remainder of this section, the term *packet* (or socket packet) is used to refer to the array of *WSABUF* buffers sent with a single *WSASend* call and received using the appropriate three step mechanism that repeatedly calls the *WSARecv* routine.

The CAP MPS provides the CAP runtime system with a portable communication environment. The problem of portability has been addressed by isolating platform-dependent code within a few number of files. A high-level platform-independent stream socket kernel has been devised providing simple robust efficient functions for creating passive and active sockets [Microsoft96], for asynchronously sending C/C++ structures to active sockets and asynchronously receiving C/C++ structures from active sockets. The interest and the efficiency of the socket kernel resides in the fact that all network events, i.e. new incoming connection, lost connection, connection established, data received, data sent, etc., are asynchronously handled by a single thread, the message-passing system thread, called the *MPSThread*. All low-level tedious error-prone platform-dependent communication mechanisms such as the packet transfer mech-

anism, the C/C++ structure serialization, the coalescence of C/C++ structures into a same socket packet for improving performance, are located within a same file enabling to optimize the code for a particular platform. At the present time, MPS runs on Sun Solaris, Microsoft Windows NT and Digital Unix OSF 1. However, the most advanced features such as the zero-copy serialization (address-pack serialization) are only available on Windows NT platforms.

The CAP MPS must provide a means for a thread to asynchronously receive tokens from any threads independently whether they are in the same address space, in a different address space but on the same PC, or located on another PC, i.e. communication of any threads to one thread. In the MPS terminology such a communication endpoint is called an *input port* comprising a FIFO queue (Chapter 3, Fig. 3.6, input token queue) where received tokens are inserted. Before a thread can receive tokens from an input port, it must be registered to the message-passing system. That is, the CAP runtime system must name the input port using a string of characters and a 32-bit instance number uniquely identifying the connection endpoint so that any other threads in the MPS network can use this logical name for sending tokens to this input port. The creation and the destruction of the input ports are completely dynamic. A Windows NT process can create and delete input ports at any time during execution.

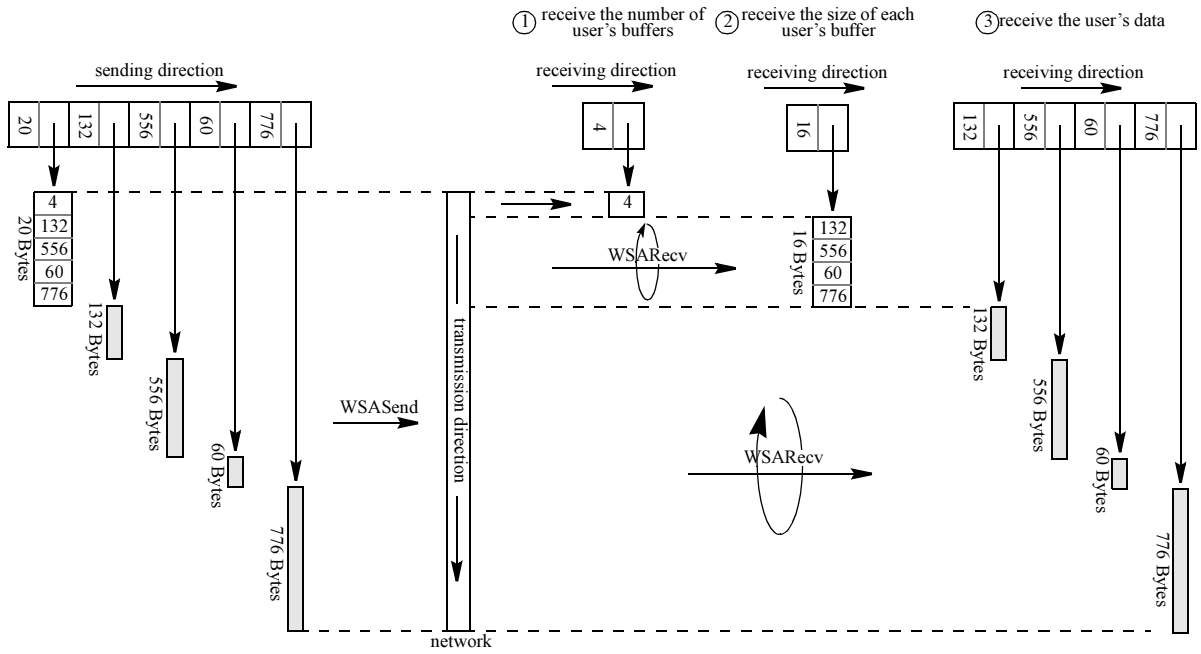


Fig. 5.2 The three steps mechanism that enables a scattered packet to be transmitted from one address space to another using a stream socket (TCP/IP protocol) and the overlapped gather/scatter *WSASend* and *WSARecv* routines

The CAP MPS provides also a means for a thread to asynchronously send tokens to any threads independently whether they are in the same address space, in a different address space but on the same PC, or located on another PC. In the MPS terminology such a communication endpoint is called an *output port*. Before a thread can send tokens through an output port, it must

be connected to an input port. That is, the CAP runtime system must open the output port by providing the name of the input port previously registered. At opening time, the message-passing system resolves the input port name into an IP address and a 16-bit TCP port number, and opens a TCP/IP connection between the output port and the input port. If the input port is located in the same address space as the output port, then instead of using a TCP/IP connection, the shared memory along with an inter-thread synchronization mechanism is used avoiding to serialize the tokens, i.e. only pointers are copied. Any number of output ports can be connected to a same input port. The creation and the destruction of the output ports are completely dynamic. A Windows NT process can create and delete output ports whenever it wants during execution.

The CAP MPS is a critical actor during the initialization phase of a CAP parallel program. The CAP processes are spawned using the standard Remote SHell Daemon (RSHD) daemon [BSD93]. The process which spawns all other processes is called the *master* process, the spawned processes are the *slave* processes. The master process opens a passive socket and informs the spawned slave processes (through the command line argument) how to connect to his passive socket. The slave processes also open a passive socket and inform the master process (by connecting themselves to the master passive socket) about it. The master collects all the information and then broadcasts it to all the slaves. At this point, all the slave processes are able to communicate with each other. Since one input port is created per thread in each process, a similar broadcast process is repeated until each process knows all input ports of the other threads. The initialization phase is then finished. The output ports are created dynamically (the first time they are needed) during the parallel program execution.

The CAP MPS has been developed in order offer good performance in terms of throughput and latency, i.e. the performance are near to the physical limits. A complete performance analysis is given in [Messerli99a].

5.2. Serialization of CAP tokens

To move a C/C++ structure, i.e. a CAP token, from one address space to another, the structure's data should be prepared for transfer, sent over a communication channel linking the two address spaces, received in the destination address space, and finally restored the data into a C/C++ structure identical to the original one. This process of preparing a data structure for transfer between two address spaces is called *serialization*.

The state of the art in respect to serialization (or marshaling) can be resumed by two major approaches. Some languages are based on a type library (the term of library is used here in a general sense). Each of these types features serialization capabilities. The program data structures must be based on these types or on a combination of these types in order to benefit from serialization capabilities. Several languages or libraries use this concept such as Java [Cornell97] or MFC [MSDN00]. The other approach consists of using data *description* statements in order to describe the content of the structures to serialize. Such an approach is used by Corba, COM, DCOM (through IDL) [MSDN00] or MPI (derived datatype) [Snir98]

[Pachero97]. The CAP approach is similar to the second approach. In our view, this approach offers improved integration capabilities for existing programs. The CAP approach is not based on a standard description language (e.g. IDL, XDR) but is custom made in order to control the whole serialization process and to be more efficient.

The CAP serialization process consists of four steps:

- The *packing* step prepares the data structure for the transfer.
- The *sending* step sends the prepared data structure over a communication channel, i.e. a TCP/IP stream socket for MPS, linking two address spaces.
- The *receiving* step receives the data in the destination address space.
- The *unpacking* step restores the received data into the original C/C++ structure in the destination address space.

The token-oriented message-passing system implements two types of serialization operations, the *copy-pack* serialization and the *address-pack* serialization. The copy-pack mechanism uses a temporary buffer for transmitting the structure's data thus enabling two computers with different data encoding (little endian, big endian, etc.), i.e. heterogeneous environment, to communicate. However, the use of a temporary buffer involves two memory-to-memory copies, one at the sending side and one at the receiving side. In the case where the two computers use the same data encoding, i.e. homogeneous environment, the address-pack serialization avoids these two copies by copying the addresses to the structure's data into a list of pointers to memory blocks. Efficient serialization (address-pack) becomes critical for data intensive applications (such as imaging applications) or for continuous media applications.

When packing a token, the CAP's runtime system adds its *type index*, which is a 32-bit value identifying the type of the transmitted token so that in the destination address space, the corresponding unpack function is called. The unpack function creates a token of the original type and copies the received data into it.

In Figure 5.3, the copy-pack packing mechanism copies all the token's fields into a single memory buffer called the *transfer-buffer*. The CAP tool automatically generates instructions for packing the predefined C/C++ types, e.g. *int*, *float*, *double*, *char*, etc. For user defined C/C++ structures and pointers, the CAP's runtime system calls, thanks to C++ function overloading, the appropriate user defined pack routine that recursively copies the structure's data fields into the same memory block. At the receiving address space, the token is first created based on the received type index and then data is copied from the transfer-buffer into the token's fields using the corresponding unpack routines.

In Figure 5.4, the address-pack packing mechanism creates a list of pointers to memory blocks (*WSABUF* buffers), and the asynchronous send handles the transfer of these scattered memory blocks. The CAP tool automatically generates instructions for packing the token's memory block. For user's defined C/C++ structures and pointers, the CAP's runtime system calls (thanks

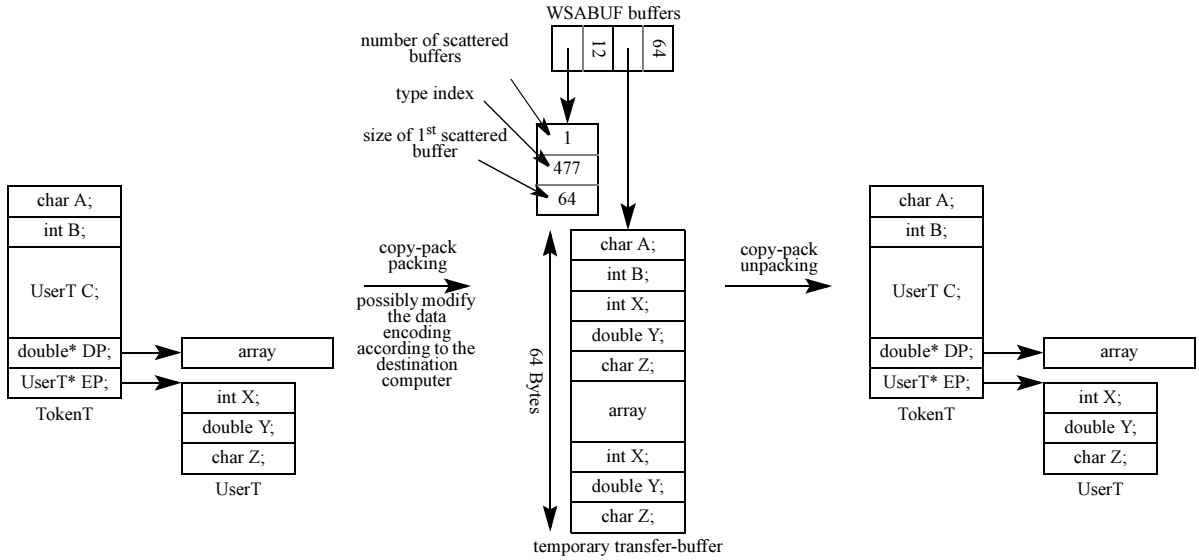


Fig. 5.3 The copy-pack serialization uses a temporary transfer buffer to copy the structure's data so that the data encoding can be modified

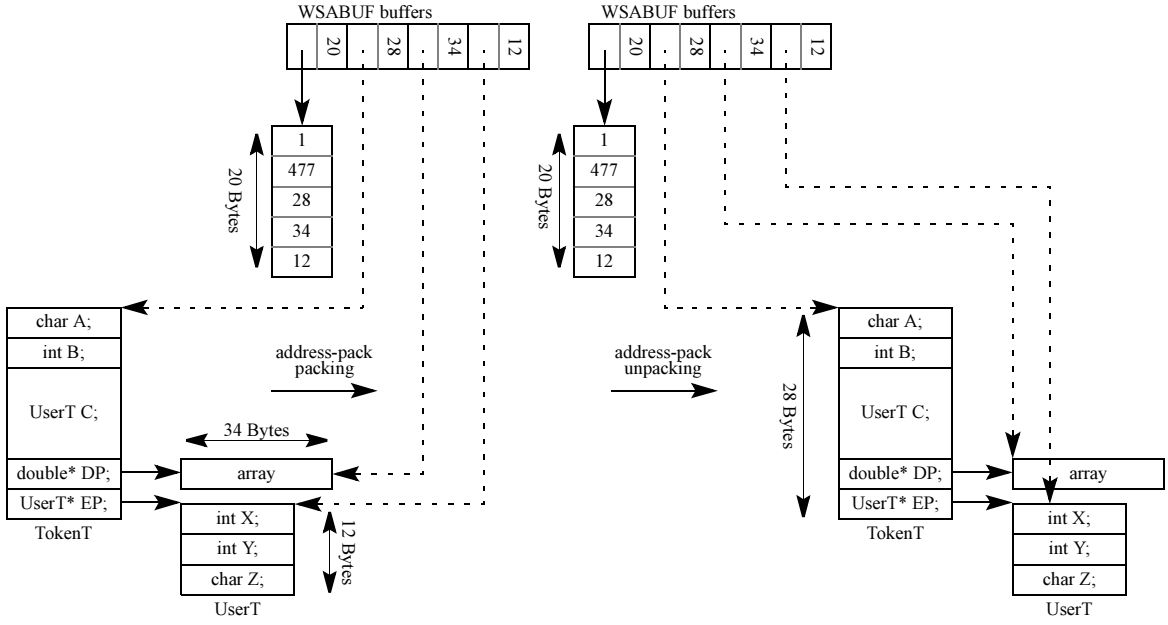


Fig. 5.4 The address-pack serialization uses a list of scattered buffers for sending and receiving the structure's data with no memory-to-memory copy

to C++ function overloading) the appropriate user's defined pack routine that recursively copies all the pointers to the structure's memory blocks into the list of *WSABUF* buffers. At the receiving address space, the token is first created based on the received type index, then a list of pointers to memory blocks is created using the corresponding unpack routines, and the overlapped scatter receive handles the transfer. Since the memory block-oriented transfer process

also copies pointers, e.g. in Figure 5.4 the *DP* and *EP* pointers, an additional *address-restore* stage restoring the clobbered pointers is required. Note that it is necessary to restore the pointer to the virtual function table in the case of a C++ class containing at least one virtual function.

```
1.  class ArrayOfIntsT
2.  {
3.  public:
4.     ArrayOfIntsT();
5.     ~ArrayOfIntsT();
6.     int Size;
7.     int* ArrayP;
8.  };
9.  ArrayOfIntsT::ArrayOfIntsT() :
10.    Size(0), ArrayP(0)
11. {
12. }
13. ArrayOfIntsT::~ArrayOfIntsT()
14. {
15.    delete ArrayP;
16. }
17. int capAddressListSize(
18.    ArrayOfIntsT* udP)
19. {
20.   return 1;
21. }
22. void capAddressPack(int& listSize,
23.    WSABUF* &bufferP, ArrayOfIntsT* udP)
24. {
25.    bufferP->Len = udP->Size;
26.    bufferP->BufP = (char*) udP->ArrayP;
27.    bufferP++;
28. }
29. void capAddressUnpack(int& listSize,
30.    WSABUF* &bufferP, ArrayOfIntsT* udP)
31. {
32.    if( bufferP->Len ) {
33.       udP->ArrayP =
34.          new int[bufferP->Len];
35.    }
36.    else {
37.       udP->ArrayP = 0;
38.    }
39.    bufferP->BufP = (char*) udP->ArrayP;
40.    thrAddressSave(listSize,
41.       bufferP, (void*) &(udP->ArrayP));
42.    bufferP++;
43. }
44. void capAddressRestore(int& listSize,
45.    WSABUF* &bufferP, ArrayOfIntsT* udP)
46. {
47.    thrAddressRestore(listSize,
48.       bufferP);
49.    bufferP++;
50. }
51. token PrimeNumbersT
52. {
53.    ArrayOfIntsT PrimeNumbers;
54.    int NumberOfPrimeNumbers;
55. };
```

Fig. 5.5 To serialize a user's defined C/C++ structure, the CAP's runtime system needs 4 routines: an address-list-size, an address-pack, an address-unpack and an address-restore routine

Figure 5.5 shows a *PrimeNumbersT* token comprising a user's defined *ArrayOfIntsT* object (line 53). To serialize such a C/C++ structure, the CAP's runtime system needs 4 routines. An address-list-size routine (lines 17-21) calculating the number of scattered memory blocks necessary for transmitting the whole structure's data. An address-pack routine (lines 22-28) copying the pointers to the structure's memory blocks (line 26) and their sizes (line 25) into the list of memory buffers to send. An address-unpack routine (lines 29-43) allocating the internal structure's memory blocks (lines 33-34) and copying their pointers into the list of memory buffers where the incoming network data will be stored. And finally, an address-restore routine (lines 44-50) restoring the clobbered pointers (lines 47-48) previously saved in the unpack-routine (lines 40-41). For transmitting *PrimeNumbersT* tokens, the CAP tool automatically generates appropriate calls to the 4 user defined serialization routines and assigns a unique token type index.

5.3. Automatic serialization of CAP tokens[1]

Writing the four serialization routines is often tricky and error prone. Moreover, bugs in these routines are hard to handle, since the program will not crash immediately but run with corrupted data. The objective of this section is to present an automatic serialization tool. This tool should alleviate the task of the programmer and reduce the programming errors, making the data serialization as simple as possible. Currently this tool is not completely integrated into the CAP runtime, but can be used as an external preprocessor. Therefore, we limit the development of this section to the fundamentals ideas.

The automatic serialization tool should be able to work within both homogeneous and heterogeneous environment. In heterogeneous environment, the data can not be copied directly from one address space to another since the data representation could be different on both architectures (e.g. big/little indian). In an heterogeneous environment the data should be transmitted using a standard data encoding system such as XDR [XDR95]. XDR is useful for transferring data between different computer architectures, and has been used to communicate data between diverse machines. Transmitting data in an heterogeneous environment requires two memory-to-memory copies (copy-pack) in order to perform the data encoding/decoding. In an homogeneous environment the serialization should use the efficient address-pack serialization mechanism described in the last section. The automatic serialization tool should handle this two different situations transparently from the programmer's point of view.

One solution consists of developing a CAP specific type library. This library should cover all the usual types (char, integer, float, etc.), but also complex types such as lists, arrays, hash-tables, etc. Such a solution has not been considered for two major reasons. Firstly, it is not a suitable solution in terms of integration, i.e. all the classes and structures of a sequential program must be redefined in order to be parallelized. Secondly, in some situations it could introduce an unacceptable overhead, decreasing considerably the performance. For these reasons we adopted a more flexible and efficient solution.

The adopted solution consists of letting the programmer describe, in a dedicated part of the program, all the members (one by one) of the classes/structures he wants to serialize. CAP must therefore provide a mechanism allowing the programmer to specify such a statement. This has been done by adding the *serialize* CAP keyword and by providing a *type description library*. This library contains the routines needed to serialize all the usual types (char, integer, float, etc.) and also complex types (lists, arrays, hash-tables, etc.). Since the provided library is a *type description library* and not a *type library*, it's not needed to redefine all the classes/structures of a sequential program in order to parallelize it. A serialization kernel responsible to collapse and interpret the programmer's serialization instructions has been added to the CAP runtime system.

[1] This section is an original contribution of the present thesis.

One of the critical issue of the automatic serialization tool is its ability to deal with pointers. Indeed, a pointer is an incomplete C/C++ type, i.e. it's impossible to deduce from the pointer itself if the pointers refers to a single element, to an array of elements or even if it is an *alias* (it refers to an object that has already been serialized). Another issue of the automatic serialization tool is its capability to deal with hierarchical classes/structures. The serialization kernel is responsible to traverse recursively through all the inherited (simple inheritance, multiple inheritance, virtual inheritance, and any combination of them) or included classes/structures.

Figure 5.6 presents the serialization of structure A and it's inherited structure B. Structure A contains one integer *n* and an array *dP* containing *n* doubles. Structure B contains a list *bP*. It must be notated that this list can contain either elements of type B either elements of type A (polymorphism) or both. The serialization information written by the programmer to serialize structure A are given by the lines 17-22. The serialization information are interpreted line by line. Line 19 specify that *n* is an integer type, using the description library type *BasicType*. Lines 20-21 are dedicated to the serialization of the *dP* array. Line 21 specifies that *dP* is an array, using the description library type *Array*. The line above indicates the size the *dP* array, using the keyword *ArraySize*. The *this* keyword refers to the object currently serialized. Therefore the expression *this→n* refers to the current size of the array. The serialization of structure B is given by the lines 12-16. Line 14 indicates that *bP* points on a list, using the defined type *ListPointer* from the description library. *ListPointer* is defined so that all the list is serialized (the last pointer is supposed to null). Line 15 specifies that *aP* is an alias and that it should be initialized to the *this* pointer value. The inheritance of structure B must also be specified in order to serialize correctly structure A, this is done at line 17.

The presented serialization concept is able to handle inheritance, basic and complex type serialization. The complex type serialization is based on a description library of complex types which can be extended. Since the serialization tool is able to serialize any type, the original non-parallel program structures and classes do not need to be modified in order to be parallelized. The serialization supports either copy-pack (heterogeneous environment) or efficient (no memory copy) address-pack (homogeneous environment) serialization. Theses last issues are transparent from the programmer's point of view. One inconvenience of this serialization mechanism is that the programmer's needs to provide, for each structure/class member, an indication about how to serialize it. This represents some amount of work. The mechanism could be improved by letting the CAP preprocessor determine, for each structure/class member, a default serialization, e.g. line 19 (serialization of the integer *n*) could be determined automatically by the preprocessor. The CAP preprocessor default choice may then be overridden by a programmer instruction.

As mentioned before, at the present time the automatic serialization tool is not completely integrated into the CAP framework. The difficulties arises mainly from the parsing of the C++ language and the capability of handling all the cases of type serialization. Nevertheless we exposed here the main principle of an automatic serialization tool for a future version of the CAP framework.

```
1.  struct A;
2.  struct B
3.  {
4.    B* bP;    // a list
5.    A* aP;    // an alias
6.  };
7.  struct A : B
8.  {
9.    int n;
10.   double* dP;
11. };
12. serialize B
13. {
14.   bP : ListPointer<B>;
15.   aP = this;
16. };
17. serialize A : B
18. {
19.   n  : BasicType <int>;
20.   ArraySize = this->n;
21.   dP : Array <double>;
22. };
```

Fig. 5.6 Automatic serialization of a user's defined C/C++ structure, the CAP's runtime system needs serialization information for each member of the C/C++ structures.

5.4. Integration

The serialization mechanism has been developed in order to facilitate the parallelization of an existent application with CAP. In general, the integration capability offered by the CAP framework is one of its major feature. Since the CAP language is preprocessed and translated into pure C++ code, the CAP parallelization facilities are easily integrated into an existent project. Also CAP works at a coarse grain level, i.e. task or procedural level; therefore the transformation of a sequential program into a parallel program can be done smoothly without major modifications of the original sequential program.

The evolution of a sequential program into a parallel solution requires a general competence in parallelism and some experience with the CAP framework. Once the parallel solution is working, no particular knowledge is required to maintain or add small modifications to the system. In fact, for real projects, the ratio of the CAP code compared to the rest of the cost is not more than a few percents. The CAP part of the program source makes the parallel structure of the program explicit.

The main step involved in turning a sequential program into a parallel solution with the help of the CAP framework can be formulated as follows:

1. Define the global parallel schedule and the data dependencies.
2. Adapt and modify the sequential program in order to respect the constraints imposed by the parallel schedule

3. Define the original data structures that need to be transmitted across the processes, i.e the tokens.
4. Write a few CAP lines to describe the schedule represented by the macro-dataflow.

5.5. Summary

The CAP environment has been ported to a number of operating systems, including Microsoft Windows NT, Sun Solaris and Digital OSF Unix. Its underlying MPS communication library is portable but requires a socket interface for providing asynchronous *SendToken* and *ReceiveToken* routines.

This chapter has shown how MPS asynchronously transmits a packet, i.e. a serialized CAP token, through a TCP/IP connection using sockets. Two mechanisms used in MPS for serializing CAP tokens have been presented, i.e. the copy-pack and the address-pack serialization mechanisms. Since the copy-pack mechanism requires the use of a temporary buffer involving additional memory-to-memory copies, it is only used in heterogeneous environments where the transmitted data format has to be adapted to the destination machine (big/little endian, 32/64 bits). The address-pack mechanism, only used in homogeneous environments (e.g. in a multi-PC environment), enables CAP tokens to be serialized and transferred with no superfluous memory-to-memory copy providing optimal performance.

The original contribution of this thesis was to provide an automatic serialization tool (Section 5.3) to the CAP framework. At the present time, this tool is not completely integrated in the CAP runtime system. Nevertheless, we presented an efficient (support address-pack) and flexible serialization tool able to serialize any types without major modification of the original code.

The serialization tool and the CAP framework in general have been developed in order to allow easy integration into existent applications. Turning a sequential program into a parallel solution with the help of the CAP framework does not require major modifications of the original sequential program. This integration aspect is one of the most interesting feature of the CAP framework.

6 Parallel linear algebra algorithm

The traditional approach to the parallelization of linear algebra algorithms such as matrix multiplication and LU factorization is based on the static allocation of matrix blocks to processing elements (PEs). Such algorithms suffer from two drawbacks: they are very sensitive to load imbalances between PEs and they make it difficult to take advantage of pipelining. This chapter describes dynamic versions of linear algebra algorithms, where subtasks (matrix block multiplication, matrix block LU factorization) are dynamically allocated to PEs. It analyzes the performance of the dynamic algorithms. We show that the dynamic-pipelined linear-algebra algorithms can be specified compactly in CAP and yet achieve good performance. The content of this chapter is published in [Mazzariol97].

6.1. Introduction

This chapter is dedicated to the parallelization of parallel linear algebra algorithms [Quinn87]. Section 6.2 explains the static and dynamic algorithms for parallel matrix multiplications, analyzes theoretically their performance, and lists the CAP specification for the dynamic parallel matrix-multiplication algorithm. Section 6.3 describes the dynamic algorithm for parallel LU factorization and analyzes theoretically its performance. Section 6.4 lists experimental performance results. In the experimental setup, the hardware consists of a network of biprocessor SPARC 20 workstations connected through FDDI. Low-level linear algebra routines are carried out by the BLAS software package [Anderson95].

6.2. Matrix Multiplication

6.2.1. Notations and problem formulation

Lowercase letters represent matrix terms. Uppercase letters represent matrices. Subscripted uppercase letters represent matrix blocks. The matrix size is N^2. The number of blocks in the matrix is p^2, and the size of the blocks is $n^2 = (N/p)^2$. A row of blocks is called a horizontal matrix slice. A column of blocks is called a vertical matrix slice. For simplicity we assume that N *modulo* $p = 0$. Using these conventions, the matrix multiplication is written as:

$$C = A \cdot B \tag{6-1}$$

Since all matrice sizes are $N \times N$, equation (6-1) can be written as:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj} \tag{6-2}$$

If we divide all the matrices into *p x p* blocks of size *n x n* (where $n = N/p$), the block matrix multiplication is written as:

$$C_{mn} = \sum_{l=1}^{p} C_{mn}^{l} = \sum_{l=1}^{p} A_{ml} \cdot B_{ln} \quad (6\text{-}3)$$

We consider a physical machine consisting of *P* processing elements (PEs) connected to a single client requesting the computation. We assume that the hardware supports direct memory access, i.e. that it is possible to perform data transfers between PEs without interrupting the ongoing computations in which the PEs are involved.

6.2.2. Dynamic parallel algorithm

Traditionally, matrix multiplication is performed statically. Matrix blocks are assigned according to a static mapping to all the participating PEs. The PEs exchange matrix blocks at fixed synchronization points. Between two synchronization events a part of the matrix multiplication is performed. Static matrix multiplication parallel algorithms tend to minimize the communications between the PEs. Nevertheless, the multiple synchronizations limit their efficiency.

In order to improve these limitations, we developed a dynamic matrix multiplication parallel algorithm. Here the objective is not to minimize the communications, but to propose a flexible algorithm allowing to balance the load between the participating PEs and to reduce the synchronization costs. The parallelization is based on a master-slave parallelization scheme. The dynamic version of the algorithm assumes that initially both input matrices are located in the client address space. The client divides both matrices (*A*, *B*) into p^2 matrix blocks. Then, according to equation (6-3), the client creates all p^3 matrix block pairs needed for performing matrix multiplication:

$$(A_{ml}, B_{ln}), \{l, m, n\} \in \{1, ..., p\} \quad (6\text{-}4)$$

and sends each matrix block pair to the PEs for partial matrix computation:

$$C_{mn}^{l} = A_{ml} \cdot B_{ln}, \{l, m, n\} \in \{1, ..., p\} \quad (6\text{-}5)$$

The partial results are returned to the client for merging into the final resulting matrix:

$$C_{mn} = \sum_{l=1}^{p} C^{l}_{mn} \quad (6\text{-}6)$$

The dynamic algorithm requires, according to equation (6-4) and (6-5), p^3 times the transfer of *n x n* matrix blocks for both input matrices and also for the output matrix. Since $n = N/p$, the total transfer requirement is:

$$\text{Total transfer size} = 3p^3n^2 = 3p^3\left(\frac{N}{p}\right)^2 = 3pN^2 \quad (6\text{-}7)$$

The transfers are well distributed among the PEs, but the client, receiving and transferring all messages is clearly a potential bottleneck. No synchronization is required between the PEs and it is possible to keep the PEs busy during the execution of the algorithm, assuming several matrix block pairs are queued waiting to be executed by each PEs. It is easy to perform load balancing, so as to make the algorithm insensitive to PEs load imbalances. Besides being a communication bottleneck, the client is also a memory bottleneck as it is required to store both input matrices and the output matrix in the client memory. The client bottleneck problem may be solved by out-of-core programming, where matrix blocks are stored on disks and prefetched when required, or by splitting the client node into several nodes, each owning part of the matrix data. This chapter analyses theoretically and experimentally the performance of the dynamic algorithm with a single client, and shows the capabilities and limitations of the dynamic algorithm.

6.2.3. Dynamic parallel algorithm: theoretical analysis

The timing diagram for the dynamic matrix multiplication is presented in Figure 6.1. There are 5 PEs, one for the client (master) and 4 for the computation threads (slaves). Each PE incorporates one additional thread for communication. The time line flows from left to right. The thick arrows represent thread activity, i.e. time during which the communication or computation occurs. We have assumed DMA, i.e. that communication can overlap computation on a given PE. The shaded boxes indicate the data transfers. The thin arrows represent data dependencies. The final timing diagram section (rightmost part of Figure 6.1) represents a minimal length section for 4 matrix-block-pair transfers (client to computation threads), 4 block by block matrix multiplications, and 4 matrix-block transfers (computation threads to client). Taking into account that communications are hidden by computation[1], the critical path through the graph consists of one matrix-block-pair transfer to each PE when initializing the pipeline, i.e. P matrix-block-pair transfers, p^3 matrix-block multiplications distributed over P PEs, i.e. p^3/P matrix-block multiplications, and one matrix-block transfer to the client for the last partial result.

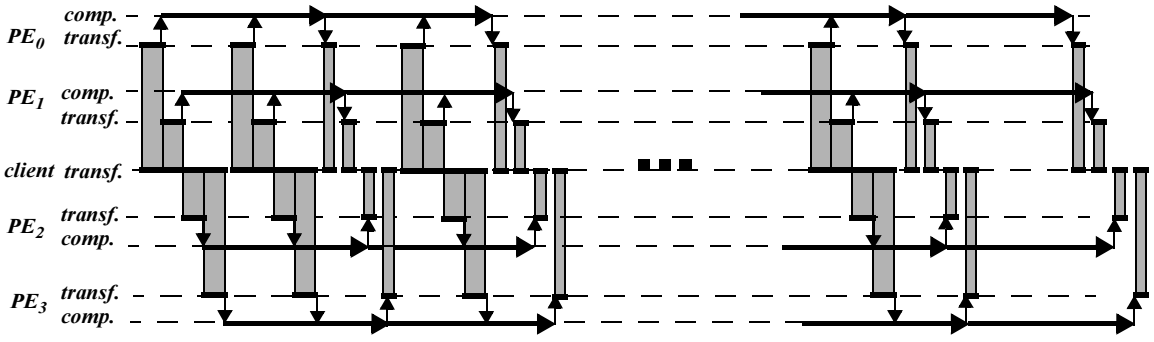


Fig. 6.1 Dynamic matrix multiplication: timing diagram

[1] Complexity of matrix multiplications is $O(n^3)$, whereas matrix size is $O(n^2)$. Therefore, for large matrices, it's possible to choose a matrix block size which ensures that computation hides communications.

Assuming a network latency of l_t, a network throughput of $8/\tau_t$ (τ_t is the transfer time for 1 matrix element, and 8 is the number of bytes per (double) floating point number), and a computation throughput of $1/\tau_c$ (τ_c is the nominal computation time for 1 element of the resulting matrix), we obtain:

$$\text{Communication time for pipeline initialization} = P \cdot (l_t + 2\tau_t n^2) \tag{6-8}$$

$$\text{Computation time} = \frac{\tau_c N^3}{P} \tag{6-9}$$

$$\text{Communication time for pipeline termination} = (l_t + \tau_t n^2) \tag{6-10}$$

$$\text{Total time} = P(l_t + 2\tau_t n^2) + \frac{\tau_c N^3}{P} + (l_t + \tau_t n^2) \tag{6-11}$$

The condition on the network bandwidth is that the transfer time during one section of the timing diagram is less than the computation time during the same time interval:

$$P((l_t + 2\tau_t n^2) + (l_t + \tau_t n^2)) < \tau_c n^3 \tag{6-12}$$

The speedup formula is given by:

$$S(P) = \frac{\tau_c N^3}{P(l_t + 2\tau_t n^2) + \frac{\tau_c N^3}{P} + (l_t + \tau_t n^2)} \tag{6-13}$$

Let us assume reasonable values for the parameters: l_t = 1ms, τ_t = 1.6µs/elem (5MB/s throughput), τ_c=875ns/elem (all numbers resulting from experimental measurements on the FDDI network and the BLAS routine *dgemm*). We find according to equation (6-13) for matrices of 1000x1000 elements (*N = 1000*), for a block of 125x125 elements (*n = 125*, *p = 8*) and for a number of processors *P* ranging from 1 to 20, theoretical speedups ranging from 1 to 19.50, showing that the dynamic approach is definitely valid. The left and right part of equation (6-12) in the worst case (*P* = 20) are 1.61s and 1.83s, ensuring that the condition holds. The experimental results are presented in Section 6.4.1.

6.2.4. CAP specification of the matrix multiplication

Figure 6.2 is the textual specification of the dynamic matrix multiplication in CAP. The indexed parallel construct in Figure 6.2 (line 4 to 8) features 3 range indices. The CAP runtime iteratively calls the *SplitInput* routine (line 8) on the *TwoMatricesT Input* token, and generates matrix block pairs. As soon as a matrix-block-pair is generated it is sent to the appropriate thread (located on another computer) for a sequential matrix computation (line 10). When all matrix

block pairs are sent, the CAP run time initializes in the client address space (called *Main*, line 8) the output matrix (*MatrixT Output*, line 8). When a computation thread completes a block by block submatrix multiplication, it returns the partial result immediately to the client thread, and gets a new matrix block pair from its input queue. The client thread, as soon as it receives a partial result from a computation thread, merges the partial result into the output matrix result.

Figure 6.2 specifies all communication and synchronization requirements of the parallel matrix multiplication program. The rest is sequential C++ code specifying how to split the input matrices *TwoMatricesT* token into matrix block pairs, how to merge partial results into the output matrix *MatrixT* token, and specifying the sequential matrix multiplication.

The issue of whether matrix blocks are copied by the *SplitInput* routine is left to the implementation of the *TwoMatricesT* token and the *SplitInput* routine. A simple implementation will implement *TwoMatricesT* tokens as actual matrices and implement the *SplitInput* routine as memory copies. A sophisticated implementation will implement *TwoMatricesT* tokens as matrix references and make sure that copies occur only when data is transferred from one address space to the other. Our implementation uses the sophisticated BLAS array data referencing mechanism which avoids unnecessary data copies. We ensure that data is copied only when data is transferred between address spaces. The actual *ParallelMatrixMultiplication* operation we used implements load balancing. As we already know (Section 4.3, flow-control), the implementation of load balancing requires only a few modification in the program of Figure 6.2.

```
1.  operation CompositeThreadT::ParallelMatrixMultiplaction (int p)
2.     in TwoMatricesT Input out MatrixT Output
3.  {
4.     indexed
5.     ( int i = 0; i < p; i++ ) // first construct range
6.     ( int j = 0; j < p; j++ ) // first construct range
7.     ( int k = 0; k < p; k++ ) // third construct range
8.     parallel (SplitInput(p,i,j,k), MergeOutput(p,i,j,k), Main, MatrixT Output)
9.     (
10.       Thread[((i*p+j)*p+k)%4].SequentialMatrixMultiplication
11.    );
12. }
```

Fig. 6.2 CAP specification of the ParallelMatrixMultiplication operation

6.3. LU factorization

The LU factorization is interesting in two respects. It features more data dependencies than the matrix multiplication, and uses the matrix multiplication. This example is ideally suited to show two important features of CAP, its compositionality and its support for pipelining. We reuse without modification the CAP parallel matrix multiplication specification in the LU factorization algorithm. We pipeline triangular system resolution with matrix multiplication to achieve higher speedups. The complex data dependencies of the algorithm also show the generality of the CAP approach.

6.3.1. Problem description

LU factorization consists of decomposing a matrix A into a unit lower triangular matrix L and an upper triangular matrix U so that[1]:

$$A = L \cdot U \tag{6-14}$$

For example:

$$\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix} \tag{6-15}$$

Golub describes in [Golub96] (p. 100) a block-based LU factorization. We summarize it here. Consider a $N \times N$ matrix A divided in 4 blocks: A_{11} $n \times n$ matrix, A_{12} $n \times (N-n)$ matrix, A_{21} $(N-n) \times n$ matrix and B $(N-n) \times (N-n)$ matrix:

$$A = \begin{matrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} & \begin{matrix} n \\ N-n \end{matrix} \\ \begin{matrix} n & N-n \end{matrix} & \end{matrix} \tag{6-16}$$

Consider the following decomposition:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ T_{21} & X \end{bmatrix} \cdot \begin{bmatrix} U_{11} & T_{12} \\ 0 & Y \end{bmatrix} \tag{6-17}$$

According to this equation, a block-based LU factorization can be realized by a 3 steps procedure, let us call these steps LU_1, T_1 and M_1:

LU_1: This first step consists of solving a smaller LU factorization problem described by equation (6-18). This can be done by using the *dgetf2* routine in LaPack [Anderson95].

$$A_{11} = L_{11} \cdot U_{11} \tag{6-18}$$

T_1: The second step consists of solving two triangular systems. This can be performed by the *trsm* routine in BLAS.

$$A_{12} = L_{11} \cdot T_{12} \tag{6-19}$$

$$A_{21} = T_{21} \cdot U_{11} \tag{6-20}$$

[1] The LU factorization exists and is unique, if and only if A is a nonsingular matrix.

M_1: The last step is related to the last condition extracted from (6-17). This condition is specified by equation (6-21). Clearly if X is a unit lower triangular matrix and Y an upper triangular matrix we are done. In general it's not the case. Let us rename $X \cdot Y$ as A' and define equation (6-22). The matrix multiplication of equation (6-22) can be performed by the *dgemm* BLAS routine. If we repeat recursively the 3 steps algorithm on the new defined A' matrix, we will finally obtain the LU factorization.

$$B = T_{21} \cdot T_{12} + X \cdot Y \tag{6-21}$$

$$A' = X \cdot Y = B - T_{21} \cdot T_{12} \tag{6-22}$$

The complete block-based LU factorization algorithm consists of repeating $p = N/n$ times the above 3 steps procedure. We need to perform the following steps: LU_1, T_1, M_1, LU_2, T_2, M_2, ..., LU_{p-1}, T_{p-1}, M_{p-1}, LU_p.

Let us analyze the complexity of each step of the proposed algorithm. Step LU_1 consists of performing the LU factorization on a *n x n* matrix. The complexity is given by $O(n^3)$. Since this step is repeated *p* times the total complexity is given by:

$$\sum_{i=1}^{p} n^3 = pn^3 \approx O(Nn^2) \tag{6-23}$$

Step T_1 consists of solving two triangular system. The two matrices involved in the system are of size *n x n* and *n x (N-in)*, where *i* corresponds to the *i*-th step T_i. The complexity is given by $O(n^2(N-in))$. Since this step is repeated *(p-1)* times, the total complexity is given by:

$$\sum_{i=1}^{p-1} n^2(N-in) = n^3 \sum_{i=1}^{p-1} (p-i) = n^3 \sum_{i=1}^{p-1} i \approx n^3 p^2 \approx O(N^2 n) \tag{6-24}$$

Step M_1 consists of multiplying two matrices. The size of the two matrices involved in the multiplication are respectively *(N-in) x n* and *n x (N-in)*, where *i* corresponds to the *i*-th step M_i. The complexity is given by $O(n(N-in)^2)$. Since this step is repeated *(p-1)* times the total complexity is given by:

$$\sum_{i=1}^{p-1} n(N-in)^2 = n^3 \sum_{i=1}^{p-1} (p-i)^2 = n^3 \sum_{i=1}^{p-1} i^2 \approx n^3 p^3 \approx O(N^3) \tag{6-25}$$

If we assume that the matrix block size *n* is small enough compared with the total matrix size *N*, we see, according to equations (6-23)(6-24)(6-25), that the dominant operation is the multi-

plication step M_i. The complexities of the two first steps are below $O(N^2)$, only the multiplication step requires $O(N^3)$ operations.

6.3.2. Parallelization

The block-based matrix multiplication step M_i is easy to parallelize because the size of each transfer is $O(n^2)$ and the number of scalar multiplications per matrix block multiplication is $O(n^3)$. It is therefore possible to select a matrix block size n ensuring that communication time is small compared with computation time. We use our previously developed parallel matrix multiplication algorithm in order to solve in parallel the LU factorization problem.

However we cannot afford to perform at each step the block LU factorization followed by the block triangular system resolutions sequentially, before performing in parallel the matrix multiplication. The sequential part of the algorithm would be much too long. This situation is presented in Figure 6-3. In this figure, we show the activity of the client and of the PEs. In order to simplify the diagram the communication costs are not shown.

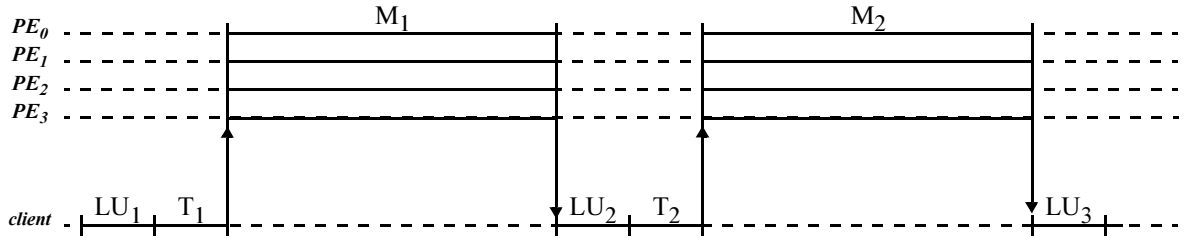


Fig. 6.3 LU factorization without pipelining

We apply pipelining to improve the parallel algorithm. Instead of performing the triangular system resolution step T_i with a single matrix operation, we perform it block by block (each block having $n \times n$ elements). Similarly, we perform the parallel matrix multiplication step M_i block by block. This block decomposition allows the parallel matrix multiplication M_i to start on the PEs as soon as the first blocks of the triangular system have been computed. Figure 6-4 shows this situation.

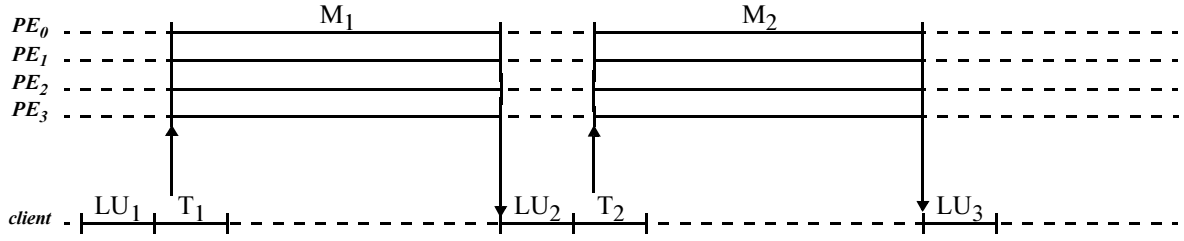


Fig. 6.4 LU factorization with pipelined matrix multiplication

Again we can improve the performance by resorting to pipelining. The LU factorization LU_2 can start before the end of the parallel matrix multiplication. As soon as the required blocks are available, the LU factorization can start. Figure 6.5 presents the full pipelined parallel solution for solving the LU factorization.

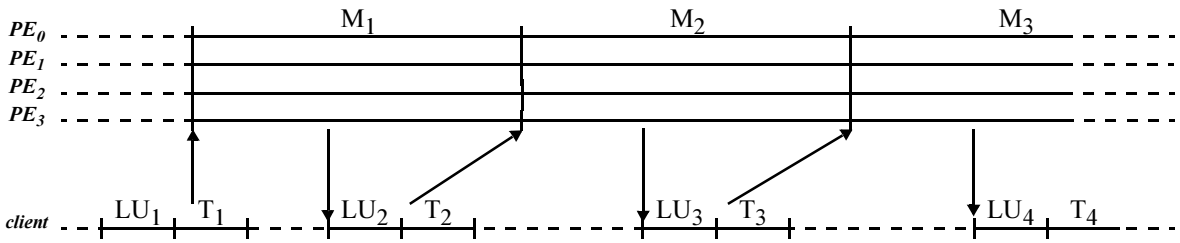


Fig. 6.5 LU factorization with full pipelining

The parallel schedule presented in the last figure cannot be directly expressed with the basic CAP statements (*indexed parallel*,...). To implement this parallel schedule we use the advanced *capCallSuccessor/capDoNotCallSuccessor* CAP features (Section 3.4.1) allowing the programmer to have a more detailed control of the CAP scheduling mechanism.

The dominant part of the algorithm is the matrix multiplication which is performed in parallel by the PEs. The size of the two matrices involved in the parallel matrix multiplication M_i at the *i*-th step are *(N-in) x n* and *n x (N-in)*. Since we decompose the matrices into *n x n* matrix blocks, step M_i requires $(p-i)^2$ matrix block by block multiplications (where $p = N/n$). According to equation (6-9) the duration of step M_i is given by $(p-i)^2(\tau_c n^3/P)$. Since this step is performed *(p-1)* times, the duration for all M_i step is given by:

$$\text{Parallel matrix multiplication time} = \sum_{i=1}^{p-1}(p-i)^2\frac{\tau_c n^3}{P} = \frac{\tau_c n^3}{P}\sum_{i=1}^{p-1} i^2 \qquad (6\text{-}26)$$
$$= \frac{\tau_c n^3}{P}\frac{p(p-1)(2p-1)}{6} \approx \frac{\tau_c N^3}{3P}$$

The only part of the algorithm during which the PEs cannot work is at least the first LU factorization LU_1, the last LU factorization LU_p and the two first triangular system resolution of steps T_1 and T_2 (Fig. 6.5). Equation (6-27) defines this time:

$$\text{Sequential time} = 2\tau_{lu}n^3 + 2\tau_{\Delta}n^3 \qquad (6\text{-}27)$$

To compute the total parallel computation time we need to add the communication costs required to initiate and terminate the pipeline. This costs have already been defined in equations

(6-8) and (6-10). The total parallel computation time to perform the LU factorization is given by:

$$\text{Total parallel time} = P(l_t + 2\tau_t n^2) + 2(\tau_{lu} + \tau_\Delta)n^3 + \frac{\tau_c N^3}{3P} + (l_t + \tau_t n^2) \qquad (6\text{-}28)$$

The approximated formula of equation (6-26) is true provided the matrix multiplication M_i takes longer than the LU factorization LU_{i+1} and the triangular system resolution T_{i+1} together (Fig. 6.5). This condition is usually true for the early steps of the parallel LU factorization, but becomes false at the end of the algorithm. When the expression becomes false the PEs are not fully loaded and becomes partially idle, waiting for new matrix blocks multiplication jobs. We express in equation (6-29) the condition that the client thread computation time is less that the PEs multiplication time for step *i* and check that it is true for most of the steps of the algorithm. To establish equation (6-29) we consider that at step *i*, $(p\text{-}i)^2$ matrix block multiplications are performed on the PEs. At the same time on the client thread, one LU factorization LU_{i+1} and *2(p-i-1)* block triangular system resolution T_{i+1} are performed.

$$(\tau_{lu} + 2(p - i - 1)\tau_\Delta)n^3 < (p - i)^2 \frac{\tau_c n^3}{P} \qquad (6\text{-}29)$$

Let us assume that the computation time at the end of the algorithm, where equation (6-29) is not valid, is small relative to the total computation time. Then, the speedup is given by

$$S(P) = \frac{\tau_c N^3 / 3}{P(l_t + 2\tau_t n^2) + (\tau_{lu} + \tau_\Delta)n^3 + \frac{\tau_c N^3}{3P} + (l_t + \tau_t n^2)} \qquad (6\text{-}30)$$

We analyze the values of equation (6-30) for a 2000x2000 matrix ($N = 2000$), block size of 125x125 ($n = 125, p = 16$), and for unitary values of the LU factorization τ_{lu} = 260ns, triangular system resolution τ_Δ = 700ns and matrix multiplication τ_c = 875ns. The communication latency and throughput are respectively l_t = 1ms and τ_t = 1.6µs/elem (5MB/s throughput) respectively. For a number of processors *P* ranging from 1 to 10, theoretical speedups ranging from 1 to 8.75 are obtained. The experimental results are presented in Section 6.4.2.

6.4. Performance measurements

We run our performance measurements on a network of Sun Sparc20 computers, each with two processors, connected by an FDDI network. We run a single computation thread and a single communication thread per workstation, thus allowing for overlapped communications and computations. No single thread can exceed 100% utilization per processor.

We integrated LaPack into CAP. Sequential routines are BLAS routines. The implementation effort consisted of wrapping BLAS routines in CAP tokens, and providing serialization routines for the tokens, to allow BLAS structures to be transferred from one address space to the other. The communication and synchronization between sequential operations are specified in CAP.

6.4.1. Dynamic matrix multiplication

Figure 6-6 displays the speedup results of the matrix multiplication on 1000x1000 matrices. We compare the speed of the parallel program with 1 to 20 slave computers (i.e. 21 workstations involved) to the sequential program performance (single LaPack call). Two threads run on each slave computer (workstation), one for computation and one for communication.

The left part of Figure 6.6 shows a near linear speedup as a function of the number of slave computers. The single thread matrix multiplication comprising of a single call to LaPack is performed in 875s. The 20-computation-PEs matrix multiplication with a 125x125 block size is performed in 46.6s, yielding an 18.8 speedup, close to the theoretically predicted speedup. The slight difference between the predicted theoretical speedup and the measured speedup comes essentially from the fact that not all computations threads terminates exactly at the same time (small load unbalance). The experiments show that better results are obtained with 125x125 matrix blocks than with 100x100 blocks. This is in contradiction with equation (6-13). The reason is that in equation (6-13) τ_c is supposed to be constant, in practice this value tends to decrease for larger matrix blocks because of some internal processor optimizations (caching, instruction prefetching,...). Larger matrix sizes may improve the overall speedup, but would require out-of-core programming, i.e. storing matrix data on disks, and prefetching the data into memory in advance.

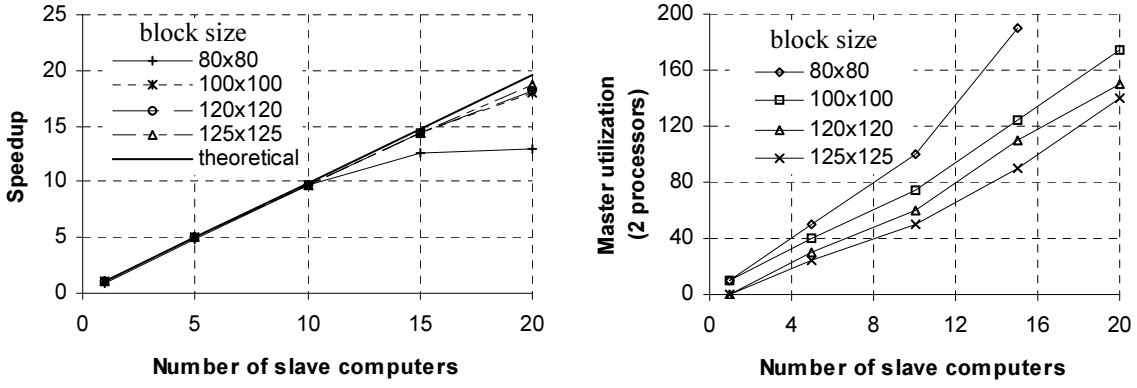


Fig. 6.6 Matrix multiplication speedup and client computer utilization (1000x1000 matrices)

The right part of Figure 6-6 shows the processor utilization on the client computer (master). The client computer utilization is affected by the block size. As is expected, the smaller the block size, the more data chunks need to be transferred between client and computation nodes. The utilization ranges between 0 and 200%, showing that both computation and communication

threads are fully utilized. For 80x80 blocks matrices we see that the client becomes rapidly a bottleneck, decreasing the overall performance. For the matrix multiplication, the client computation thread handles matrix-block-pair serialization and partial result accumulation. The client communication thread handles the matrix deserialization, as well as serialized data emission and reception. The computation threads have a 100% utilization for the complete duration of the algorithm, except at pipeline startup and termination.

6.4.2. LU factorization

Figure 6.7 shows the performance results of the LU factorization for a 2000-by-2000 matrix, for a number of slave computers varying from 1 to 10. The single process sequential computation time for the LU factorization of the matrix is 2095s. The fastest parallel time we achieved with 10 computers is 250s, yielding an 8.38 speedup. The difference with the theoretical model is due to the fact that for the later steps of the algorithm, condition (6-29) becomes false.

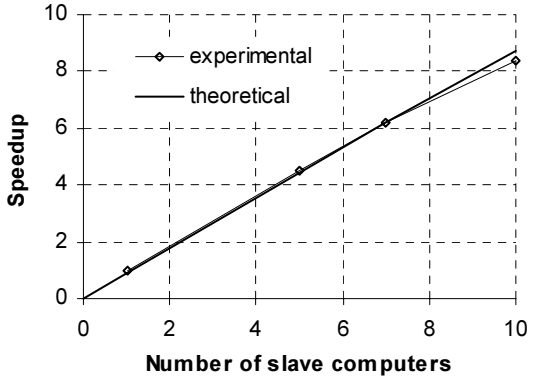


Fig. 6.7 .LU factorization performance

6.4.3. Analysis of results

The client load is much higher than what the theoretical analysis suggests. This is due to the fact that data structure serialization is a time consuming process requiring several data copies, in particular in the version of the LSP library running on Sun Sparc20 workstations. Moreover, the use of the TCP/IP protocol between workstations consumes an important amount of CPU resources. Typically a 5MB/s throughput at the client leads to a 100% CPU utilization.

It is not possible to handle matrices larger than 1000x1000 without having the client thread starting to swap. The three matrices required for a 1000x1000 multiplication represent 24 MBs of data. Out of core programming is required, where the client thread prefetches the matrix blocks from the disks as needed. As pipelining is a CAP feature, disk prefetching is not difficult to integrate in the CAP implementation of the LaPack routines. The disk prefetching mechanism has been successfully used in the CAP implementation of an out of core 3-D visualization package [Hersch00][VisibleHuman98].

The good performance of CAP generated programs can be attributed to the following factors: (1) threads are allocated statically to processing elements; (2) operation overhead is small. The input data of each operation is tagged with a 20-byte header, which typically represents a overhead of less than 1%; (3) data can be passed by reference between threads sharing a common address space; (4) data elements (tokens) are routed dynamically to threads, allowing to adapt the load of each processing element; (5) pipelining is a part of the CAP semantics, allowing to remove unnecessary synchronization barriers. Current difficulties in the CAP implementation of the LaPack routines are: (1) the Solaris version of the MPS communication library performs 3 copies for each transfer (copy-pack mechanism, see Chapter 5); (2) serialization of sophisticated data structures is a time-consuming effort and (3) some data dependencies are impossible to specify in the current version of the language.

6.5. Summary

This chapter has presented pipelined-parallel algorithms for matrix multiplication and LU decomposition. It has analyzed the theoretical performance of the algorithms, shown their CAP implementation, and presented performance results. The results show that the dynamic pipelined approach is viable for matrix multiplication and LU decomposition for a number of computers varying between 10 to 20 computers. Large matrix sizes require out of core programming. CAP support for pipelining is effective to overcome the important network latencies. CAP specifications are short, and yet the parallel programs achieve good performance, demonstrating the validity of the CAP model.

7 Parallel Imaging

Imaging applications such as filtering, image transforms and compression/decompression require vast amounts of computing power when applied to large data sets. These applications would potentially benefit from the use of parallel processing. However, dedicated parallel computers are expensive and their processing power per node lags behind that of the most recent commodity components. Furthermore, developing parallel applications remains a difficult task: writing and debugging the application is difficult (deadlocks), programs may not be portable from one parallel architecture to the other, and performance often comes short of expectations. In this chapter, we will use the Computer-Aided Parallelization (CAP) framework for creating a typical pipelined parallel image processing application. This chapter shows how processing and I/O intensive imaging applications must be implemented to take advantage of parallelism and of pipelining between data access and processing. This chapter's contribution is (1) to show how such implementations can be compactly specified in CAP, and (2) to demonstrate that CAP specified applications achieve the performance of custom parallel code. The chapter analyzes theoretically the performance of CAP specified applications and demonstrates the accuracy of the theoretical analysis through experimental measurements. The content of this chapter is published in [Mazzariol98].

7.1. Introduction

Imaging computations such as filtering, image transforms, compression/decompression and image content indexing [Equitz95] require, when applied to large data sets (such as 3-D medical images, satellite images and aerial photographs), vast amounts of computing power. In order to facilitate the development of parallel applications, we propose the CAP computer-aided parallelization tool which enables application programmers to specify at a high-level of abstraction the flow of data between pipelined-parallel operations. The CAP environment supports the programmer in developing parallel imaging applications. The CAP environment features (1) support for the parallel storage of large data sets; (2) an image library supporting 1-bit, 8-bit, 16-bit, 24-bit images, as well as the division of images in tiles of user-defined size; (3) the CAP language extension to C++ which allows to write deadlock-free portable pipelined-parallel applications, and combine parallel storage access routines and image processing operations.

This chapter shows how processing and I/O-intensive imaging applications can be implemented to take advantage of parallelism and pipelining between data access and processing operations. This chapter's contribution is (1) to show how such implementations can be compactly specified using the CAP set of flow control instructions, and (2) to demonstrate that CAP specified applications achieve the performance of custom code. The chapter analyzes theoretically the performance of CAP specified applications and demonstrates the accuracy of the theoretical analysis through experimental measurements. To implement I/O intensive applications, large 2D (resp. 3D) images are divided into square (resp. cubic) subsets with good locality called *tiles*. Two kinds of applications are considered in this chapter: *neighborhood-indepen-*

dent operations, and *neighborhood-dependent* operations. Neighborhood-independent operations are operations where no data must be exchanged between tiles to compute the resulting final image.

Section 7.2 shows how large images are divided in tiles for storage and processing purposes. Section 7.3 describes in general terms the *process-and-gather* operation, i.e. the problem of applying a neighborhood-independent operation to selected tiles stored on disk(s) and gathering the processed tiles in a single address space. It shows the ideal execution schedule for performing a process-and-gather operation and analyzes theoretically the performance of such a schedule. Section 7.3.5 shows how process-and-gather operation is specified in CAP. The process-and-gather operation is limited to linear filters. Section 7.4 describes the more general *exchange-process-and-store* operation, i.e. the problem of applying a neighborhood-dependent operation to tiles stored on disk(s) and storing the result back to disk(s). Section 7.5 lists performance results for the exchange-process-and-store parallel operations.

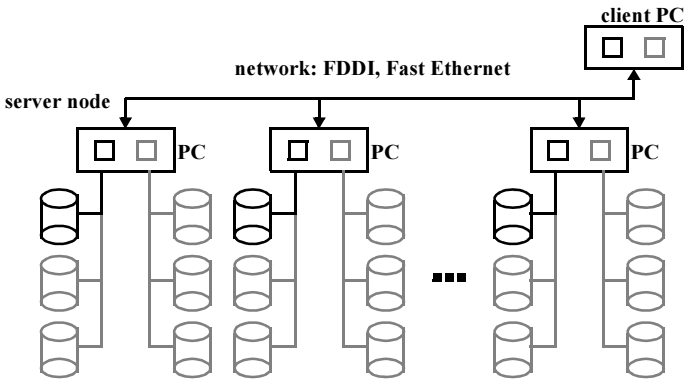


Fig. 7.1 Commodity component based parallel architecture composed of one client node and several server nodes

7.2. System support for managing large images

7.2.1. Hardware architecture

The hardware we consider consists of multiple PentiumPro PCs connected through a commodity 100Mb/s network such as Fast Ethernet (Fig. 7.1). The PCs run the WindowsNT operating system and communicate using the TCP/IP-based MPS message-passing system (Chapter 3). Each PC represents a storage/processing node (server node) consisting of one or two processors connected through its PCI internal bus to one or more disks. The client requesting some image processing operation is also located on a PC. This platform scales from a single PC architecture with one processor and one disk, to a multiple-PC multiple-disk architecture. We assume that both the disk and the network can access memory through DMA (Direct Memory Access). While this hypothesis is accurate for disks[1], network interfaces based on the TCP/IP protocol consume an important amount of processing power[2].

7.2.2. Software architecture

Large images are divided in rectangular or squared tiles which are stored independently, possibly on multiple disks. Pixmap image tiling is often used for the internal representation of images in software packages such as PhotoShop. Square tiles enable accessing image windows efficiently, with a good data locality. In addition parallel access to individual tiles distributed on different disks and computers is possible [Hersch93]. The CAP imaging library provides data types and functions for splitting images in tiles, and allocating tiles to disks. Figure 7.2 shows an image divided in tiles, as well as a visualization window covering part of the image. Figure 7.2 also shows a possible allocation of tiles to disks, assuming an image striped over 8 disks. The allocation index consists of the disk index, as well as the local tile index on the disk. For example, the bottom right tile in Figure 7.2 is allocated on disk 3, and is the 6^{th} tile on that disk. The distribution of tiles to disks is made so as to ensure that direct tile neighbors reside on different disks. We achieve such a distribution by introducing, between two successive rows of tiles (and between two successive planes of tiles in the case of 3-D images), offsets which are prime to the number of disks.

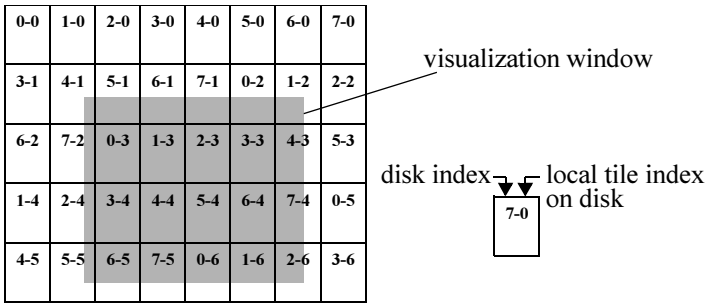


Fig. 7.2 Distribution of tiles on disks

The data types required in this example are the *WindowT* and the *TileT* classes, provided in the CAP imaging library (Fig. 7.3). The *WindowT* class fields are a file name, the window position within an image, the window size, the window data, and a pixel descriptor (pixel size in bit, color scheme (gray level, RGB,...). The *WindowT* class is used both to specify a window request parameters (in which case the data field is empty), as well as the window itself. The *TileT* class fields give its position, its size, the tile data, a pixel descriptor, as well as the index of the disk where the tile is stored, and the local index of the tile on the disk.

Figure 7.4 lists a simple sequential program which performs a process-and-gather operation. It assumes that the whole window to be processed is in memory. The while loop (lines 10 to 14) repeatedly calls the *GetNextTileInWindow* routine, until it returns 0. At each iteration, the *Get-*

[1] Our experience shows that for disk throughputs of up to 45 MB/s (throughput achieved by connecting 15 disks to 5 SCSI boards, the processor utilization remains below 15%.

[2] Our experience shows that a single PentiumPro processor reaches over than 50% utilization rate when Fast Ethernet reaches its maximal throughput of ~8.5MB/s [Messerli99a].

```
1.  class WindowT {
2.     char* FileName;
3.     int PositionX, PositionY;
4.     int SizeX, SizeY;
5.     char* DataP;
6.     PixelDescriptorT Descr;
7.  };
```

```
1.  class TileT {
2.     int DiskIndex
3.     int LocaTileIndex;
4.     int PositionX, PositionY;
5.     int SizeX, SizeY;
6.     char* DataP;
7.     PixelDescriptorT Descr;
8.  };
```

Fig. 7.3 CAP imaging-package data-types

NextTileInWindow routine returns the next window tile (*nextP* parameter), based on the window description (*windowP* parameter) and the previous window tile (*prevP* parameter). The first time the *GetNextTileInWindow* is called, the *prevP* parameter is 0. In the body of the while loop (lines 11 to 13), the new tile is processed using the user-defined *ProcessTile* routine. In this simple program, all tiles are processed independently. When the tile is processed, the result is merged into the final window using the *MergeAndAddTile* routine. This simple routine handles correctly neighborhood-independent and linear filtering operations (see Section 7.3.1) The *GetNextTileInWindow* and the *MergeAndAddTile* are provided by the imaging library. The *ProcessTile* routine is user-defined.

The next sections show more sophisticated imaging programs which in a pipelined-parallel manner access tiles stored on disks and perform on it processing operations.

```
1.  int GetNextTileInWindow (WindowT* windowP, TileT* prevP, TileT* &nextP);
2.  void MergeAndAddTile (WindowT* windowP, TileT* tileP);
3.  void ProcessTile (TileT* inputP, TileT* &outputP);
4.  void ProcessWindowByTile (WindowT* inputP, WindowT* &outputP)
5.  {
6.     TileT* prevP = 0;
7.     TileT* nextP;
8.     TileT* processedTileP;
9.     outputP = new WindowT (...);
10.    while (GetNextTileInWindow (inputP, prevP, nextP)) {
11.       ProcessTile (nextP, processedTileP);
12.       MergeAndAddTile (outputP, processTileP);
13.       prevP = nextP;
14.    }
15. }
```

Fig. 7.4 Sequential processing of a window, tile by tile

7.3. The parallel process-and-gather operation

7.3.1. Problem description

A process-and-gather operation consists of reading the tiles composing the image from the disks (on the several server nodes), performing a neighborhood-independent operation on the tiles, gathering the processed tiles in a single address space, and merging the tiles to form the final visualization window. The server nodes perform the neighborhood-independent operation,

and send the processed tiles to the client PC, where processed tiles are merged to form the final image. Linear filtering can be carried out according to a process-and-gather scheme (Fig. 7.5). The linear operation is performed on each tile, assuming that elements beyond the tile border are set to 0. The linear filtering operation generates enlarged tiles. After filtering, tiles overlap. When merging the tiles to form the final image, the overlapping part of the tiles are added together, leading to the correct final result. As an example, we filter a 1-D gray-level discrete signal, by averaging elements with a 3-by-1 convolution kernel (Fig. 7.5). The 3-by-1 convolution kernel is applied to an 8-element vector, with and without tiling. Both situations assume that elements outside the range of the vector are 0. With tiling, the filter is applied to two 4-element vector slices, and both vector slices grow to 6 elements after filtering. The overlapping parts of the vector slices are then added at tile-merging time to recover the correct 8-element vector.

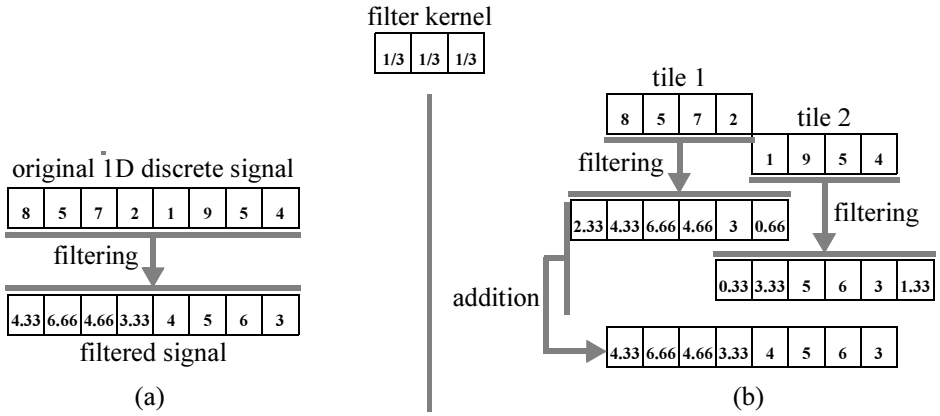


Fig. 7.5 Applying a linear filtering operation to a 1-D discrete signal.
(a) standard filtering (b) filtering tiles separately

7.3.2. Modelled single-PC execution schedule

This hardware configuration consists of a single PC reading data from the disks, performing the neighborhood-independent operations on all tiles, and merging the processed tiles. We assume that the disks can read the tiles faster than the processor can process them (Fig. 7.6).

The schedule described in Figure 7.6 guarantees that the PC processor is busy at all times, after the first tile has been fetched. In this model, all disk accesses but the first one are performed while the processor carries out computations.

7.3.3. Modelled multiple-PC execution schedule

Figure 7.7 shows the ideal execution schedule for a multiple server node situation. We assume that the disks read tiles faster than the processors can process them, that the network transfers processed tiles faster than the processor can produce them, and that the client PC merges processed tiles faster than the network can transfer them.

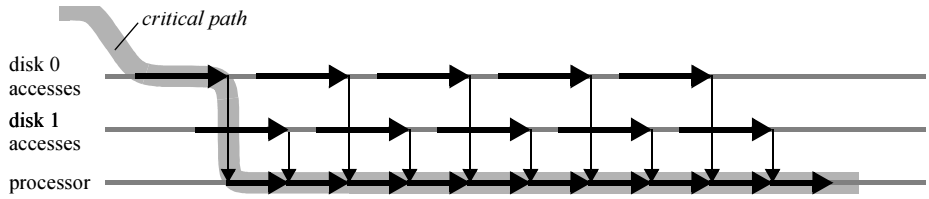


Fig. 7.6 Execution schedule with a single PC and two disks (processor bottleneck)

In Figure 7.7, horizontal arrows represent disk access, processing operations and network transfer time. Vertical arrows represent ordering between operations. The critical path is represented as a smooth light-gray line. As in Section 7.3.2, this schedule ensures that all disk transfers but the first one, all network transfers but the last P (where P is the number of server nodes) and all merging operations but the last one are performed while the server node processors perform the neighborhood-independent tile computations.

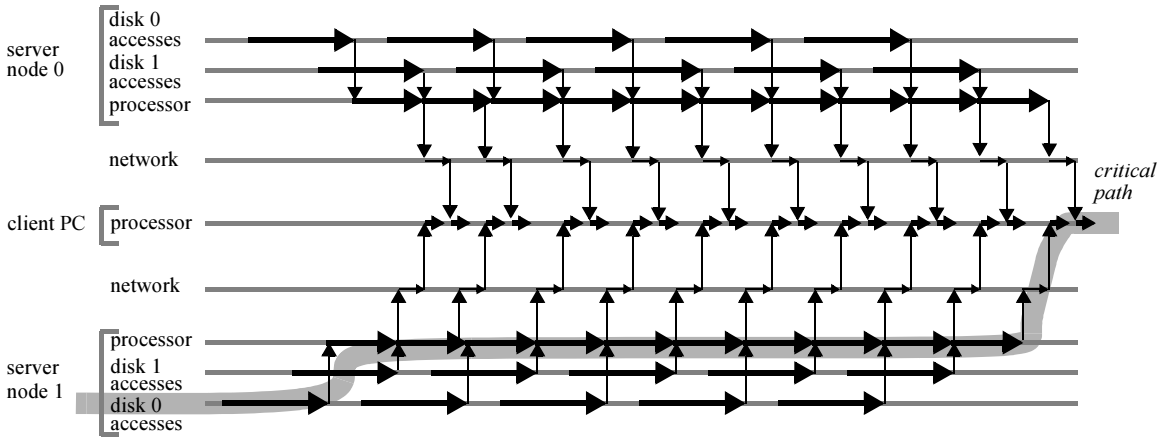


Fig. 7.7 Execution schedule (multiple single-processor PCs with two disks, slave processor bottleneck)

7.3.4. Theoretical performance analysis

According to Figure 7.7, the critical path in the pipelined-parallel process-and-gather operation consists of one disk access, $\lceil N/P \rceil$ tile processing steps, P network transfers and one *MergeAndAddTile* operation, where N is the number of tiles in the window, and P the number of server nodes in the architecture. We assume that the size of each tile is $TileSize \times TileSize$ (in bytes) and that this size does not change significantly regarding the neighborhood-independent computation. The time required to read a tile from a disk is written as $t_d = l_d + \tau_d \cdot TileSize^2$ where l_d is the disk latency and $1/\tau_d$ is the disk throughput. The time required to transfer data over the network is written as $t_n = l_n + \tau_n \cdot TileSize^2$ where l_n is the network latency and $1/\tau_n$ is the network throughput. The time required to process a tile is written as $t_p = \tau_p \cdot f(TileSize)$ where τ_p is the computation time per byte and f gives the complexity

of the algorithm as a function of the tile size. The time required to merge a tile into the visualization window is $t_m = \tau_m \cdot TileSize^2$. The duration of the process-and-gather operation is:

$$T = t_d + \left\lceil \frac{N}{P} \right\rceil \cdot t_p + P \cdot t_n + t_m \qquad (7\text{-}1)$$

The assumptions behind the execution schedule of Figure 7.7 are that tile accesses are faster than tile processing steps ($t_d < D \cdot t_p$), where D is the number of disks per server node, that P network transfer times are faster than a single tile processing step ($P \cdot t_n < t_p$), and that merging a tile into a window is faster than a network transfer step ($t_m < t_n$).

7.3.5. CAP specification of the process-and-gather operation

CAP threads are grouped hierarchically. In the context of this chapter, the *CapServerT* thread hierarchy (Fig. 7.8) consists of a client thread running on the client node (line 3) and two sets of threads running on the server nodes (lines 4 and 5). The *TileServer* threads perform I/O operations (*ReadTile* and *WriteTile*, lines 16 to 19) and the *ComputeServer* threads perform computations on the tiles extracted from the disks (e.g. filtering, lines 25 and 26). Each server node comprises one *ComputeServer* thread and as many *TileServer* threads as disks. The *CapServerT* thread hierarchy can perform two parallel operations: the *process-and-gather* and the *exchange-process-and-store* operations. The current and the next section specify the behavior of these two operations.

```
1.  process CapServerT {
2.  subprocesses:
3.     ClientProcessT Client;
4.     TileServerT TileServer[NDISK];        // NDISK:total nb of disks
5.     ComputeServerT ComputeServer[NSTOR];  // NSTOR:total nb of
6.  operations:                              // storage/processing nodes
7.     ProcessAndGather (FilterT filter)
8.        in WindowT Input out WindowT Output;
9.     ExchangeProcessAndStore (FilterT filter)
10.       in WindowT Input out void Output;
11.    ...
12. };
13.
14. process TileServerT {
15. operations:
16.    ReadTile
17.       in TileReadingRequestT Input out TileT Output;
18.    Writetile
19.       in TileWritingRequestT Input out void Output;
20.    ...
21. };
22.
23. process ComputeServerT {
24. operations:
25.    Filtering (FilterT filter)
26.       in TileT Input out TileT Output;
27. };
```

Fig. 7.8 Parallel storage and processing system threads

Figure 7.9 is the CAP specification of the process-and-gather operation declared in Figure 7.8 (lines 7-8). This program applies in a pipeline-parallel manner a linear filter to all tiles within a window specified by the *WindowT Input* class instance. The *WindowT* class consists of a window position, and a file name (see Section 7.2.2). The CAP *parallel while* expression semantics (Fig. 7.9, lines 4 to 9) is to perform in parallel the body of the pipeline (lines 6 to 8). The *parallel while* expression iteratively calls the *GetNextTileInWindow* split function (line 4, first *parallel while* parameter) until it returns 0. Each token generated by the call is immediately (before the next token is generated) sent to the appropriate *TileServer* thread which reads a tile (line 6). The tile is then processed by the *ComputeServer* thread (line 8). Tiles sent to different server nodes are processed in parallel. Tiles sent to the same server node are processed in pipeline: the *TileServer* thread fetches the next tile, while the *ComputeServer* is processing the previous tile. When a tile has been processed, it is returned to the *Client* PC (line 4, third initialization parameter) where it is merged using the *MergeTile* routine (line 4, second initialization parameter) into the final window (line 4, fourth initialization parameter).

The program given in Figure 7.9 performs in a pipeline-parallel manner the operations performed sequentially by the program of Figure 7.4. It specifies all communications and synchronizations required to implement the execution schedule of Figures 7.6 and 7.7. If a single PC with two disks is available, the *process-and-gather* operation (Fig. 7.9) follows the schedule described Figure 7.6. If two server nodes with a total number of 4 disks are available, it follows the schedule given in Figure 7.7.

```
1.  operation CapServerT::ProcessAndGather (FilterT filter)
2.     in WindowT Input out WindowT Output
3.  {
4.     parallel while (GetNextTileInWindow, MergeTile, Client, WindowT Window)
5.     (
6.        TileServer[thisTokenP->DiskIndex].ReadTile
7.        >->
8.        ComputeServer[thisTokenP->DiskIndex].Filtering (filter)
9.     )
10. }
```

Fig. 7.9 CAP process-and-gather operation

7.4. The exchange-process-and-store operation

7.4.1. Problem description

We consider the situation where the source image is read from disks, and the target image is written to disk(s). Both, input and output images are divided into tiles stripped over the disks on the several server nodes. Before filtering can be performed on a tile, tile sides must be exchanged: a tile must receive pixels from its 8 neighboring tiles. The width of the border exchanged between tiles is defined as w and depends on the filtering operation (Fig. 7.10). Neighboring dependent operations may for example consist of non-linear kernel operation such as erosions or dilations.

We assume that there are enough disks to ensure that disk(s) throughput is superior to the processor(s) throughput, i.e. our algorithm is always compute bound. The CAP runtime library incorporates a tile cache keeping loaded tiles in memory. The tile cache works according to a least-recently used scheme. Provided that the cache can store at least 3 tile rows, most of the required tiles will be in memory during a given computation step. Considering a tile size of 256-by-256 pixels or 64KB, in a 4096-by-4096 image, 3 rows represent 24 tiles, or 1.5MB, well below the typical memory size of current PCs. In the theoretical analysis and experimental measurement sections, we consider two situations: tile cache disabled, tile cache enabled.

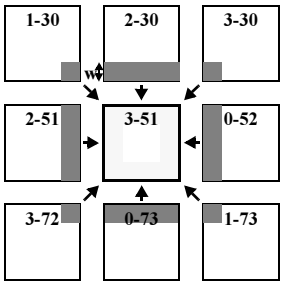


Fig. 7.10 Borders exchanged during a neighborhood dependent operation

We select an execution schedule where the processors are always busy. We must ensure that the required data to compute a given tile (or part of it) is in memory when the computation starts, i.e. the required data is read from the disks, and exchanged between the neighboring server nodes before the computation is started.

To achieve this result, all server nodes work in parallel, and each server node runs a four-step pipeline. The first step consists of reading tiles from disk (possibly cached in memory). During the second pipeline step, the server node computes the *central part* of the tile and in parallel reads the neighboring tiles borders from the other server nodes. During the third pipeline step, the server node computes the *border* of the tiles after having received the neighboring tile borders. During the fourth pipeline step, the server node writes the computed tile back to disk. The tile central part is defined as the part of the tile that is not affected by the neighboring tile sides. The tile border is defined as the part of the tile that is affected by the neighboring tile sides. As opposed to hardware pipelining, the pipeline steps are not performed synchronously: the only guarantee is that a given tile will undergo the four pipeline steps in the specified order.

Tiles are allocated to server nodes so as to ensure proper load-balancing. Assuming P server nodes, the tile disk index n is processed by the server node $n \bmod P$. For example, in Figure 7.11, tile (4,6) is stored as local tile 12 on disk 2. On a 2 server node machine, tile (4,6) will be processed by server node 0. In the present allocation of tiles to disks, adjacent tiles on the same row are processed by different server nodes.

Figure 7.11 shows the four stages pipeline activity pattern during a computation step, with 2 server nodes. During the algorithm step shown in Figure 7.11, server node 0 (resp. 1) reads tiles (3,9) (resp. (3,8)) from disk, processes the central part of tile (2,6) (resp. (2,5)), computes the

border of tile (2,4) (resp. (2,3)) after having exchanged the neighboring tile sides, and writes tile (2,2) (resp. (2,1)) to disk. For the next computation step, the activity pattern is shifted two tiles along the arrow. The tiles are scanned in serpentine order so as to benefit from the tile cache.

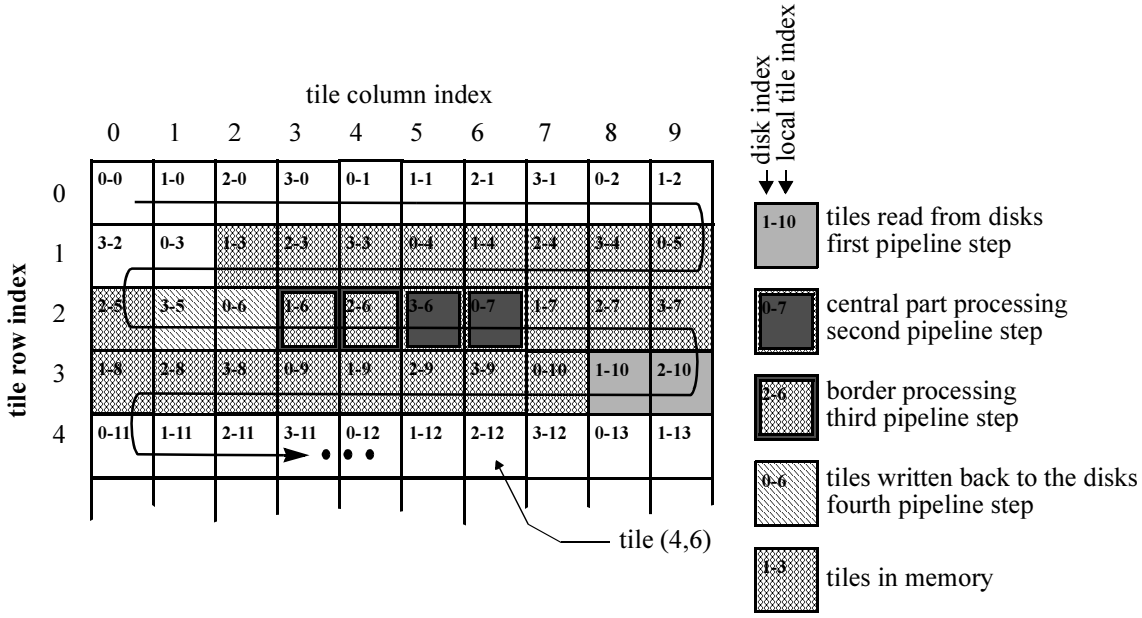


Fig. 7.11 Activity pattern among image tiles during a computation step

A number of trade-offs must be taken into considerations:

- The basic activity pattern can be adapted so that each server node processes more than one tile during each computation step. This reduces the number of synchronizations during the course of the algorithm, but increases the pipeline startup cost.
- For the same reasons, an increase in tile size reduces the number of synchronizations during the execution of the algorithm, but increases the pipeline startup cost.
- As explained in Section 7.2.2, the tile allocation scheme selected in this chapter ensures that neighboring tiles are allocated on different server nodes in order to improve load balancing. This in turn increases the number of communications required during each computation step. An alternative tile allocation scheme could optimize communications at the expense of load balancing, and would be easy to specify in CAP.

7.4.2. Theoretical performance analysis

The theoretical performance analysis is done under two separate assumptions: disabled tile cache and enabled tile cache.

We first assume that the tile cache is disabled. During the algorithm execution, each server node receives requests from the other server nodes. To serve these requests each server node

must in the worst case read nine tiles from disks (since the cache is disabled). During each computation step, four activities are carried out simultaneously. Each server node reads nine tiles from disks, communicates with the other server nodes to obtain at most 4 tile sides and 4 tile corners, computes one tile, and writes one tile to the disk. Assuming that the tile processing operation is a computation intensive operation (as opposed to data-intensive), we try to keep the server node processors busy at all times by reading in advance data from the disks.

Assuming that the size of each tile is $TileSize \times TileSize$ (in bytes), the time required to read a tile from the disk is written as $t_{dr} = l_{dr} + \tau_{dr} \cdot TileSize^2$ where l_{dr} is the read disk latency and $1/\tau_{dr}$ is the read disk throughput. The time required to write a tile to the disk is written as $t_{dw} = l_{dw} + \tau_{dw} \cdot TileSize^2$ where l_{dw} is the write disk latency and $1/\tau_{dw}$ is the write disk throughput. During one algorithm step, each server node reads in the worst case nine tiles from the disk, and writes one tile to the disk. This yields equation (7-2). The time required to transfer tile borders (resp. corners) over the network is written as $t_n = l_n + \tau_n \cdot DataSize$ where l_n is the network latency, $1/\tau_n$ is the network throughput. Assuming that the width of a tile border to be exchanged is defined as w, we determine that the size of a tile border is $DataSize = TileSize \cdot w$, and that the size of a tile corner is $DataSize = w^2$. This leads to equation (7-3). The time required to process a tile is written as $t_p = \tau_p \cdot f(TileSize)$ where τ_p is the computation time per byte and f gives the complexity of the algorithm as a function of the tile size. When the pipeline reaches the steady state, equations (7-2), (7-3) and (7-4) determine respectively the disk access, the network transfer and the processing times during each step of the algorithm:

$$\text{disk access time: } T_d = 9(l_{dr} + \tau_{dr} \cdot TileSize^2) + (l_{dw} + \tau_{dw} \cdot TileSize^2) \qquad (7\text{-}2)$$

$$\text{network transfer time: } T_n = 4(l_n + \tau_n \cdot TileSize \cdot w) + 4(l_n + \tau_n \cdot w^2) \qquad (7\text{-}3)$$

$$\text{processing time: } T_p = \tau_p \cdot f(TileSize) \qquad (7\text{-}4)$$

In order to establish the total parallel execution time, we need to compute the pipeline startup and termination time. The pipeline startup cost is the time of preloading one tile on each server node, i.e t_{dr}. The pipeline termination cost is the time of writing back one tile on each server node, i.e t_{dw}. Here, we ignore the small communication time between the client and the server nodes when the program starts or terminates. We suppose that all server nodes start and terminate at the same time. The total parallel execution time consists of the pipeline startup cost, the computation time, and the pipeline termination time. We define the number of server nodes as P and the number of tiles to be processed as N. Equation (7-5) establishes the total parallel execution time.

$$T = t_{dr} + \left\lceil \frac{N}{P} \right\rceil \cdot max(T_d, T_n, T_p) + t_{dw} = t_{dr} + \left\lceil \frac{N}{P} \right\rceil \cdot T_p + t_{dw} \qquad (7\text{-}5)$$

The last equality is true only if the algorithm is compute-bound, i.e. processing-intensive enough to hide the disk access and the network transfer time. Provided that the number of tiles

is large, the relative startup cost can become very small. The trade-off in the tile size is that (1) the larger the tile size, the smaller the overhead due to synchronization and communications; (2) the smaller the tile size, the smaller the overhead due to pipeline startup cost.

We now assume that the tile cache is enabled. When the pipeline reaches the steady state the number of read tiles per server node is reduced to one (vs. nine). Equations (7-3) and (7-4) remain unchanged. Equation (7-2) becomes:

$$\text{disk access time: } T_d = (l_{dr} + \tau_{dr} \cdot TileSize^2) + (l_{dw} + \tau_{dw} \cdot TileSize^2) \qquad (7\text{-}6)$$

Assuming that the time until the pipeline reaches the steady state (pipeline startup and first tile row computation) is not significant and that the algorithm is compute-bound, then the total computation time with or without tile cache is similar, equation (7-5) remains unchanged. The only difference between the two situations is the number of disks required to keep the algorithm compute-bound.

The speedup function is given by equation (7-7). It gives a bound on the number of server nodes that can effectively contribute to the parallelization of the application.

$$S(P) = \frac{NT_p}{t_{dr} + \left\lceil \frac{N}{P} \right\rceil \cdot T_p + t_{dw}} \qquad (7\text{-}7)$$

7.4.3. CAP specification

The CAP specification of the exchange-process-and-store operation is given in Figure 7.12 (lines 28 to 41). The input token to the exchange-process-and-store operation is a window description (image name, size, position, but no data) and the output token is *void* because the filtered image is directly stored on the disk without producing any output. The indexed parallel construct semantics (lines 32 to 35) consists of: one or more iteration expressions (lines 33 and 34), a *split-function* name (line 35, *DuplicateWindow*), a *merge-function* name (line 35, second initialization parameter, *void*), an output token (line 35, fourth initialization parameter, *Result*) initialized in the specified address space (line 35, third initialization parameter, *Client*) and a CAP expression as body of the loop (lines 36 to 40). The split-function generates tokens which are forwarded to the *TileFiltering* operations. The merge-function indicates how to merge the parallel-construct body-output token into the parallel-construct output-token. The program consists of a double parallel iteration. The first loop (lines 33) iterates on successive window tile rows and the second loop (line 34) iterates on all tiles of one input window tile row. As explained in Figure 7.11, the second loop iterates from left to right on even rows and from right to left on odd rows. At line 35, the split-function *DuplicateWindow* duplicates the window parameters; the merge-function is *void*, indicating that the parallel-construct body-output is used for synchronization purposes only. The indexed parallel expression executes in parallel instances of its body (lines 36 to 40), as many times as expressed in the index specification (lines 33 and 34). The body consists of two parts, performed in parallel: filter the tile by calling the

```
1.  enum NeighbourhoodT
2.  {  Center, North, North_East, East, South_East,
3.     South, South_West, West, North_West };
4.
5.  operation capServerT::TileFiltering (int Row, int Col)
6.     in windowsT Input out TileT Output
7.  {
8.     indexed
9.     ( NeighbourhoodT n = Center; n <= North_West; n++ )
10.    parallel
11.    (  SplitToTiles(Row, Col), MergeToTileWithBorders(Row, Col),
12.       ComputeServer[ComputeIndex(Row,Col)], TileWithBordersT Result )
13.    (
14.       ifelse ( n == Center)
15.       (
16.          TileServer[thisTokenP->DiskIndex].ReadTile
17.          >->
18.          ComputeServer[ComputeIndex(Row, Col)].CenterFiltering
19.       )
20.       ( // else: n != Center
21.          TileServer[thisTokenP->DiskIndex].ReadBorder
22.       )
23.    )
24.    >->
25.    ComputeServer[ComputeIndex(Row, Col)].BorderFiltering;
26. }
27.
28. operation capServerT::ExchangeProcessAndStore (int P)
29.    in windowsT Input out void Output
30. {
31.    flow_control (2*P)
32.    indexed
33.    ( int Row = 1; Row <= Rmax; Row++)
34.    ( int Col = 1; Col => 1 && Col <= Cmax; Col = NextCol(Row, Col) )
35.    parallel (DuplicateWindow, void, Client, void Result)
36.    (
37.       TileFiltering (Row, Col)
38.       >->
39.       TileServer[thisTokenP->DiskIndex].WriteTile
40.    );
41. }
```

Fig. 7.12 CAP specification of the exchange-process-and-store operation

TileFiltering operation (line 37) and then save the filtered tile (line 39). The *flow_control* specification (line 31) indicates that only *2*P* instances of the body should be executed simultaneously; i.e. each processor receives two tile processing requests. This ensures pipelining while avoiding memory overflow (see Chapter 4).

The *TileFiltering* parallel operation (lines 5-26) is called in pipeline by the *ExchangeProcessAndStore* operation. The tile filtering operation filters a specific tile depending on the value of the *TileFiltering* parallel operation parameters (*Row*, *Col*). Let us call this tile the center tile. The filtering of the center tile requires data from its neighboring tiles (called North, North_East, East, etc., lines 1-3). The *TileFiltering* operation is divided into two parts. The first part (lines 8-23) reads the center tile from the local disk, filters its central part, and ask the neighboring server nodes for the required data. All this work is launched in parallel by the *indexed parallel*

loop (lines 8-10) at the beginning of the operation. The loop iterates over all the tiles, i.e. the center tile and the eight neighboring tiles. The center tile and neighboring tiles are differentiated at line 14 by the *ifelse* statement. The center tile is processed at lines 15-19; at line 16 it is read from the local disk, and then at line 18, its central part is filtered. The request for neighboring data is performed in the second part of the *ifelse* statement at lines 20-22. Within these lines, a request is send to each neighboring server node (selected by the *thisTokenP->DiskIndex* expression). The results of these requests are collected in the *Result* variable by the *MergeToTileWithBorder* merge function executed by the local *ComputeServer*. Once all the results are collected, the *Result* variable goes into the second part of the operation. The second part of the *TileFiltering* operation consists of filtering the remaining part of the tile, i.e. the borders (lines 24-25). This is performed by the *BorderFiltering* operation executed by the local *ComputeServer*.

The CAP *thisTokenP* variable refers to the input token of the operation about to be executed. For example, at lines 16, 21 and 39, the expression *thisTokenP->DiskIndex* selects, based on the *DiskIndex* field of the current token, the appropriate *TileServer* thread (see definition in Figure 7.8, line 14) to execute the next operation. The *DiskIndex* value is set at run time by the split functions (*DuplicateWindow*, or *SplitToTiles*) according to the mapping of the tiles to the disks (Fig. 7.2).

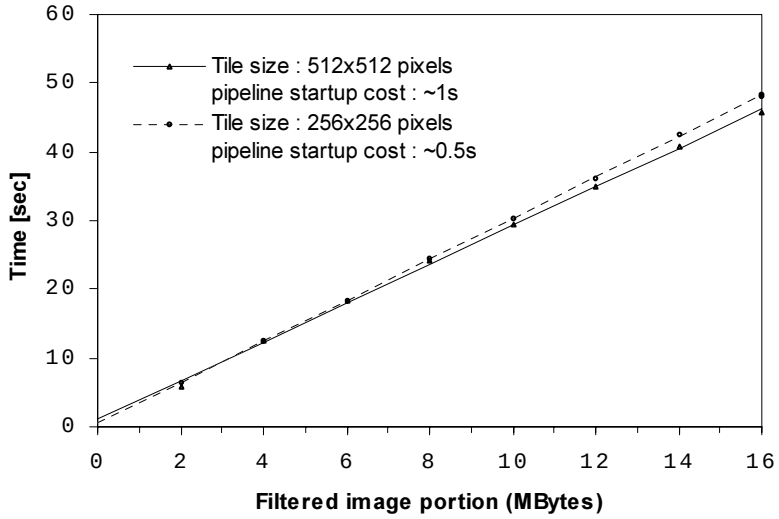


Fig. 7.13 Single processor computation time according to the filtered data size (MBytes)

7.5. Performance results

In this section we present and discuss performance results, and compare them to the theoretical models of Section 7.4.2.

We run the performance measurements on a network of Bi-PentiumPro (200MHz), connected through Fast Ethernet (100 Mbits/s). We filter a 4096x4096 pixel graylevel image (16 MBytes) split in 256 (resp. 64 tiles) of size 256x256 pixels (resp. 512x512 pixels). The tiles are stored on several disks (from one to nine per computer). The filtering operation consists of applying a 5x5 median filter mask [Weeks96] to the whole image.

Figure 7.13 displays the computation time as a function of the image size (MBytes). These results are computed locally on a single PentiumPro PC and image tiles are stored on two local disks. The cache is disabled. The diagram shows that a smaller tile size reduces the pipeline startup cost but increases the synchronization overhead, since more image tiles need to be processed. For reference, performing locally and sequentially the same tile-based algorithm with the whole image preloaded in memory (16 MBytes) takes about 44s. This shows that disk accesses are almost completely hidden during the computation. It also illustrates the ability of CAP to properly handle pipelining: programs reading data from disk have the same performance as programs working directly from main memory.

Figure 7.14 presents the speedup results when filtering in parallel a 16MByte image. With the cache disabled, two disks are required for each processor. With the cache enabled, a single disk per processor is sufficient. To produce speedup figures, we compare the speed of the parallel program (tile-based program) running on multiple processors (up to ten processors located on five Bi-PentiumPro PC) with the sequential program performance. In Figure 7.14, the continuous thick line represents the ideal speedup, and the dashed thick line represents the theoretical speedup according to equation (7-7). The dip in the theoretical speedup curve for 6 processors is due to an imperfect load balancing, expressed by the ceiling operator.

Performance results are similar with and without cache. In both situations the algorithm is compute bound, and the total execution time is mostly computation time. In contradiction with the theoretical previsions, we achieve better results for larger tiles. The reason is that the theoretical analysis assumes that the unitary processing throughput τ_p of equation (7-4) is constant independently of the tile size. In practice this value tends to increase for smaller tile size, because of the overhead due to the larger number of tiles.

The theoretical previsions for four processors are close to the experimental results. With $l_{dr} = l_{dw} = 18.5$ms (disk latency), $1/\tau_{dr} = 1/\tau_{dw} = 3.3$MBytes/s (disk throughput), *TileSize* = 512x512 pixels, $N = 64$ (number of tiles), $P = 4$ (number of server nodes), $T_p = 0.69$s (time needed to filter one tile) we obtain a theoretical total time of $T = 11.24$s (see Section 7.4.2) against 11.9s measured. The difference is explained by two factors: (1) the theoretical pipeline startup cost underestimates the actual pipeline startup cost; (2) the load is not perfectly balanced between the processors. In fact, with 64 tiles and 4 processors ($N = 64$, $P = 4$) equation (7-5) (see Section 7.4.2) predicts an ideal balanced load. Practically, the processors (PCs under WindowsNT) work at a slightly different pace and terminate with a time difference of at most 10% of the total processing time.

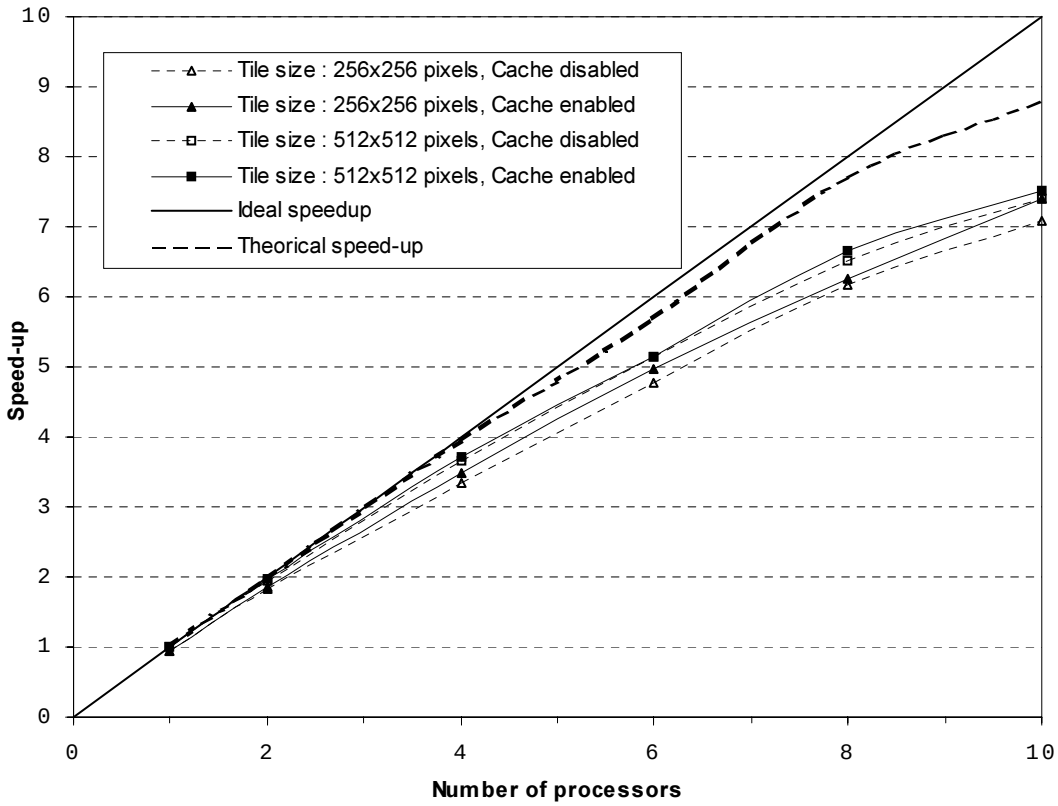


Fig. 7.14 Speedup as function of the number of processors

With ten processors, the efficiency falls down to 75% against 88% theoretically (equation (7-7), Section 7.4.2). Nevertheless, the processors are kept busy during the whole program execution. The reason is that with current PC hardware, processing time is spent handling the TCP/IP stack protocol. Therefore the computation part of the algorithm does not benefit from the full processor activity as it does on a single-processor execution (i.e. without communication). We measured the TCP/IP protocol overhead by running the program without any disk accesses and computations, and found that it is about 1s. Subtracting it from the experimental measurements, the resulting time is in accordance with the theoretical predictions.

7.6. Summary

This chapter shows that CAP enables the compact specification of pipelined-parallel imaging applications. The CAP environment is not restricted to the process-and-gather or exchange-process-and-store operations described in this chapter. It may be applied to various imaging algorithm, including *non-oblivious* algorithms[1]. Once the imaging library is available, the implementation and test effort for the parallelization of the two applications described in this

[1] Oblivious algorithms are algorithms whose execution flow is independent of the content of the data being processed.

chapter requires only a few days. The generated programs run on PCs under WindowsNT and on Sun workstations under Solaris. With a limited effort, reusable and customizable parallel code can be produced.

The CAP imaging library supports the subdivision of images in tiles. The CAP language handles the communication and synchronization of messages in a parallel program. These two features of the CAP environment free the programmer to concentrate on the algorithm(s) to be applied. Once the algorithm has been designed, the programmer can either reuse existing CAP programs and modify the processing operation performed on each tile, or create new CAP programs to handle new parallel execution schedules.

Performance measurements on a slave computers comprising two disks per processor show that we obtain similar results by reading the image directly from disks or from memory. Therefore, the CAP-specified algorithm achieves excellent pipelining between disk accesses and filtering operation. In the multi-processor configuration, the speed-ups achieved with the 4-processor and 10-processor configurations are 3.71 and 7.5 respectively. Disk accesses and network transmissions are hidden during the computation. Except at the beginning and the end of the algorithm, the processor utilization is 100%.

8 Parallel Cellular Automata

We are interested in running in parallel cellular automata. We present an algorithm which explores the dynamic remapping of cells in order to balance the load between the processing nodes. The parallel application runs on a cluster of PCs connected by Fast Ethernet. As a typical example of a cellular automaton we consider the image skeletonization problem. Skeletonization requires spatial filtering to be repetitively applied to the image. Each step erodes a thin part of the original image. After the last step, only the image skeleton remains. Skeletonization algorithms require vast amounts of computing power, especially when applied to large images. Therefore, skeletonization application can potentially benefit from the use of parallel processing. Two different parallel algorithms are proposed, one with a static load distribution consisting in splitting the cells over several processing nodes and the other with a dynamic load balancing scheme capable of remapping cells during the program execution. The content of this chapter is published in [Mazzariol00a].

8.1. Introduction

We are interested in running in parallel cellular automata. We present an algorithm which explores the dynamic remapping of cells in order to balance the load between the processing nodes. The parallel application runs on a cluster of PCs (Windows NT) connected by Fast Ethernet (100 Mbits/sec.).

A general cellular automaton [Sipper97][Toffoli87] can be described as a set of cells where each cell is a state machine. To compute the next cell state, each cell needs some information from neighboring cells. There are no limitations on the kind of information exchanged nor on the computation itself. Only the automaton topology defining the neighbors of each cell remains unchanged during the automaton's life.

Let us describe a simple solution for the parallel execution of a cellular automaton. The cells are distributed over several threads running on different computers. Each thread is responsible for running several automaton cells. Every thread applies successively to all its cells a 3 steps algorithm: (1)(2) exchange (send and receive) neighboring information, (3) compute the next cell state. If communications are based on synchronous message passing, the whole system is synchronized at exchange time because of the neighborhood dependencies. Due to the serial execution of communications, computations and multiple synchronizations, some processors remain partly idle and the achievable speedup does not scale when increasing the number of processors.

Improved performance can be obtained by running communications asynchronously. One can then overlap data exchange with computation. Neighboring information is received during the computation of the previous step and sent during the computation of the next step. This solution offers improved performances, but still does not achieve a linear speedup. Like in the skeleton-

ization problem, discussed in Section 8.2, the computation load may be highly data dependent and may considerably vary from cell to cell. Furthermore, the parallel application may run on heterogeneous processors inducing a severe load balancing problem. Due to the neighboring dependencies, cells consuming more computation time slow down the whole system. To reach an optimal solution we need a flexible load balancing scheme.

One solution is to allow each cell to be dynamically remapped during program execution. One or more cells may be displaced from overloaded threads to partly idle threads. Cell remapping requires 3 steps after terminating the computation of the cell to be remapped: (1) notify every thread about the decision to remap a given cell, (2) wait for acknowledgement from all threads and (3) remap the cell. Step (2) ensures that the neighborhood information for the remapped cell is redirected towards the target thread. In the applications we consider, the overhead for remapping a cell is insignificant compared with the computation time. For the sake of load balancing, we will present in Section 8.4 a strategy for cell remapping.

As a typical example of a cellular automaton, we consider the image skeletonization problem [Schalkoff89][Jain89][Manzanera99]. Skeletonization requires spatial filtering to be repetitively applied to the image. Each step erodes a thin part of the original image. After the last step, only the image skeleton remains. Skeletonization algorithms require vast amounts of computing power, especially when applied to large images. Therefore, skeletonization application can potentially benefit from the use of parallel processing.

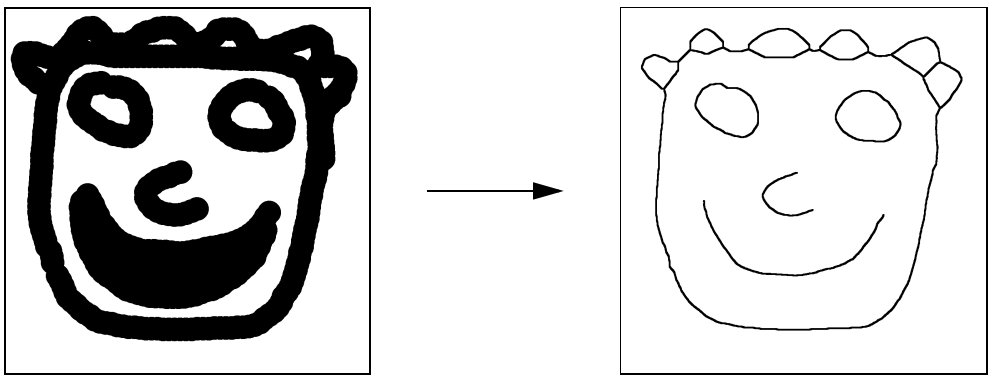
To parallelize image skeletonization, we divide the original image into tiles. These tiles are distributed across several threads. Each thread applies successively the skeletonization algorithm to all its tiles. Threads are mapped onto several processors according to a configuration file. Tiles cannot be processed independently from their neighboring tiles. Before each computation step, neighboring tiles need to exchange their borders. In addition, each computation step depends on the preceding step.

Section 8.2 presents the image skeletonization algorithm. Section 8.3 develops a parallelization scheme. Section 8.4 shows how to load balance the application by cell remapping. The performance analysis is presented in Section 8.6.

8.2. Image skeletonization algorithm

Image skeletonization consists of extracting the skeleton from an input black and white image. The algorithm erodes repeatedly the image until only the skeleton remains. The erosion is performed by applying a *5x5* thinning filter to the whole image. The thinning filter is applied repeatedly, thinning the input image pixel by pixel. The algorithm ends once the thinning process leaves the image unchanged. Figure 8.1 shows a skeletonized image.

Since several skeletonization algorithms exist, let us describe the one providing excellent result*s* [Jain89]. Let $TR(P_1)$ be the number of white to black ($0 \rightarrow 1$) transitions in the



Original image **Skeletonized image**

Fig. 8.1 Skeletonization algorithm example

ordered set of pixels $P_2, P_3, P_4, ..., P_9, P_2$ describing the neighborhood of pixel P_1 (Fig. 8.2). Let $NZ(P_1)$ be the number of black neighbors of P_1 (black = 1).

P_3	P_2	P_9
P_4	P_1	P_8
P_5	P_6	P_7

Fig. 8.2 Neighborhood of pixel P_1

P_1 is deleted, i.e. set to background (white = 0) if:

$$\left.\begin{array}{ll} & 2 \le NZ(P_1) \le 6 \\ \text{and} & TR(P_1) = 1 \\ \text{and} & P_2 \cdot P_4 \cdot P_8 = 0 \text{ or } TR(P_2) \ne 1 \\ \text{and} & P_2 \cdot P_4 \cdot P_6 = 0 \text{ or } TR(P_4) \ne 1 \end{array}\right\} \quad (8\text{-}1)$$

The process is repeated as long as changes occur. This algorithm is highly data dependent. One thinning filter step modifies only small parts of the input image and leaves the major part unchanged. In the next section we take advantage of this fact to improve the algorithm.

8.2.1. Improvement of the image skeletonization algorithm

To improve the image skeletonization algorithm, we divide the input image into cells (or tiles). The program maintains a list of living and dead cells. A cell is dead if further applications of the thinning filter leave the cell unchanged. A cell is alive if it is not dead. The algorithm

applies the thinning filter to each living cell, decides if the cell is still alive and if necessary updates the list of living and dead cells.

This algorithm improves considerably the performance of the skeletonization since a significant part of the image is removed from the computation. What should be the cell size? A small cell size ensures a fine grain selection of the living and dead parts of the image, but increases the cell management overhead. The cell size should be chosen so as to keep the management overhead time small compared with the average computation time.

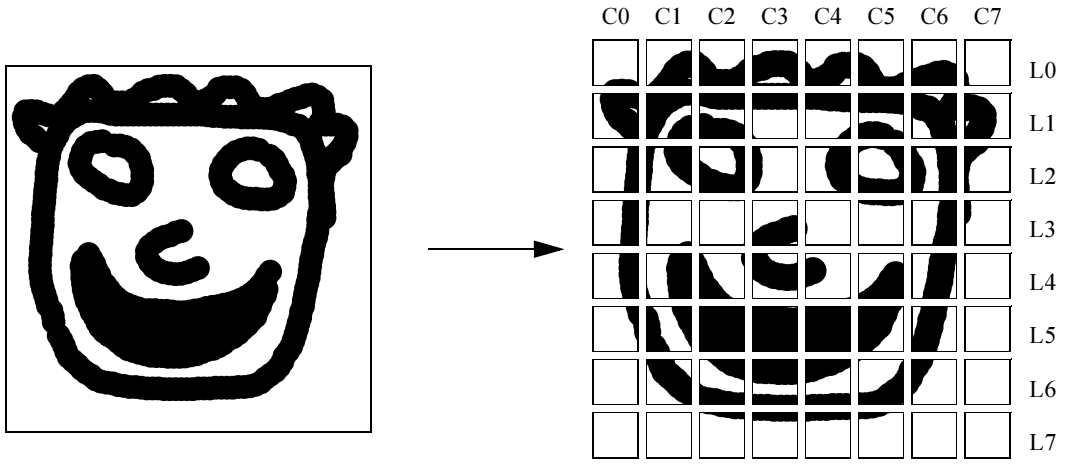


Fig. 8.3 Division of the input image into cells

8.3. Parallel skeletonization with static load distribution

To parallelize the skeletonization algorithm, the cells are uniformly distributed over *N* processing nodes. Each processing node applies repeatedly the thinning filter to all its living cells. Since the cells can not be processed independently from their neighbors, the processing nodes may need to exchange neighboring information before applying the thinning filter to a given cell. The program ends once all the cells of every processing node are dead. This parallelization scheme ensures that all processing nodes are performing the same task.

Initially the cells are distributed in a round robin fashion over the *N* processing nodes. For example if there are 4 processing units, the cells (Fig. 8.3) are distributed in row major order modulo the number of nodes (*(L0, C0)->P0*, *(L0, C1)->P1*,...). More generally, if there are *LMax* lines, *CMax* columns and *N* processing nodes, the distribution of the cells over the processing nodes in function of the line and column numbers *(L, C)* is given by:

$$NodeIndex(L, C) = (L \cdot CMax + C) \text{ Modulo } N \qquad (8\text{-}2)$$

In the parallel algorithm, the overhead for the exchange of information between neighboring cells increases since communication and synchronization is needed between processing nodes responsible for adjacent cells. The parallel program requires therefore larger cell sizes.

To develop the parallel application, we use the Computer-Aided Parallelization (CAP) framework, which allows to manage the neighborhood dependencies and the data flow synchronization. The CAP Computer-Aided Parallelization framework is specially well suited for the parallelization of applications having significant communication and I/O bandwidth requirements. Application programmers specify at a high level of abstraction the set of threads present in the application, the processing operations offered by these threads, and the flow of data and parameters between operations. Such a specification is precompiled into a C++ source program which can be compiled and run on a cluster of distributed memory PCs. A configuration file specifies the mapping between CAP threads and operating system processes possibly located on different PCs. The compiled application is executed in a completely asynchronous manner: each thread has a queue of input tokens containing the operation execution requests and their parameters. Network I/O operations are executed asynchronously, i.e. while data is being transferred to or from the network, other operations can be executed concurrently by the corresponding processing node. If the application is compute bound, in a pipeline of network communication and processing operations, CAP allows to hide the time taken by the network communications. After initialization of the pipeline, only the processing time, i.e. the cell state computation, determines the overall processing time.

Each processing node contains two threads (*IOThread*, *ComputeThread)* running in one shared address space. The address space stores the data relative to all its cells. The *IOThread* runs the asynchronously called *ReceiveNeighbouringInfo* and *SendNeighbouringInfo* functions. The *ComputeThread* runs the *ComputeNextCellStep* function. The *ReceiveNeighbouringInfo* function receives the neighboring information from the other cells and puts it in the local processing node memory. The *SendNeighbouringInfo* function sends the neighboring information from the local cells to the neighboring cells. The *ComputeNextCellStep* applies the skeletonization filter to the given cell. Before starting the computation, the *ComputeNextCellStep* waits for the required neighboring information and until the current neighboring information has been sent. Once the filter is applied, the *ComputeNextCellStep* determines if the cell is still alive, and if necessary updates the list of living and dead cells.

Each processing node runs the same schedule. The schedule comprises a main loop executing for all the living cells of the local address space one running step. A running step consists of executing asynchronously the *SendNeighbouringInfo*, *ReceiveNeighbouringInfo* functions and in parallel the *ComputeNextCellStep* function for the given cell. Figure 8.4 shows a schematic view of the schedule. The CAP tool allows the programmer to specify this schedule by appropriate high-level language constructs. The CAP specification is similar to the *exchange and process* operation described in the previous chapter (Section 7.4.3, Fig. 7.12).

When the program starts, all the cells are at step zero. Each time the thinning filter is applied to a cell, the cell step is incremented by one. Because of the neighboring dependencies, the dif-

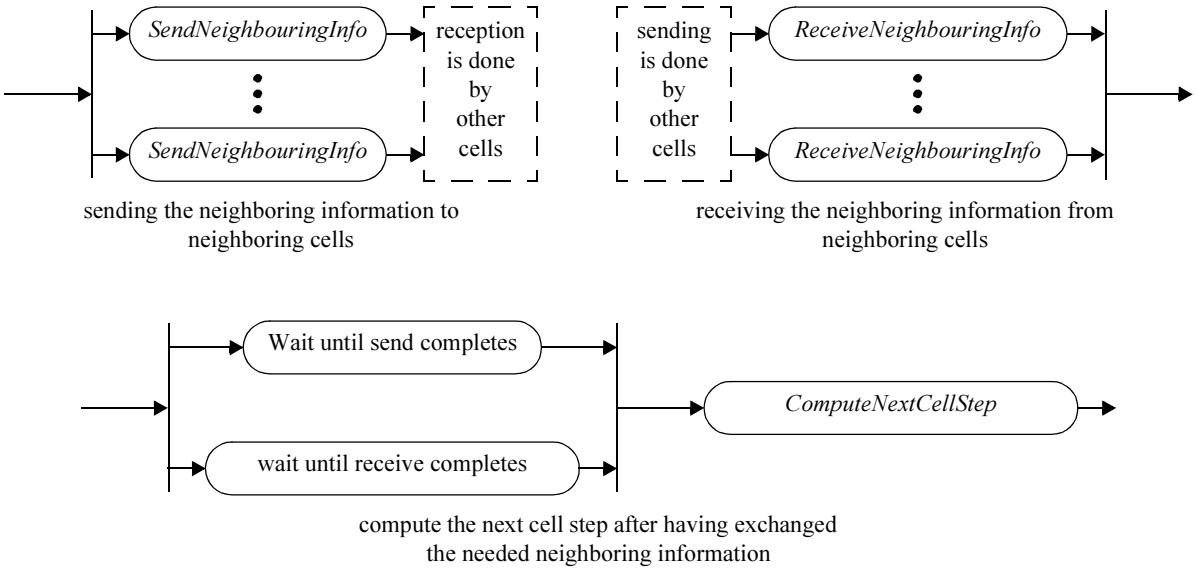


Fig. 8.4 Parallel skeletonization algorithm schedule for one cell

ferences between the step of a given cell and of its neighbors is at most one. Therefore, during the program execution, some cells are waiting for their neighbors to perform the computation of the next step. If all the cells of a processing node are waiting, the processor becomes idle, reducing the overall performance. To avoid as much as possible such a situation, the parallel algorithm is improved by computing first the cell with the smallest step value on each processing node.

While some cells are sending their neighboring information or waiting for the reception of neighboring information, other cells could potentially keep the processor busy. This argument fails if there is just one cell per processing node or if the cellular automaton topology implies that every cell is depending on all other cells. In order to run computations in parallel with communications, one may partially compute a cell without knowing the neighboring information. Cell computation may start while receiving the neighboring information from other cells.

If the total computation load is evenly distributed over the processing nodes, the parallel algorithm can potentially keep all the processors busy. However, in the case of the skeletonization algorithm, the computation time is highly data dependent. To keep all processors busy, we need to balance dynamically the computation load.

8.4. Dynamic load balanced parallel scheme

For load balancing, we need to remap the cells during program execution. In order to migrate a cell from one address space to another, we need to maintain the load (or the inverse of the load,

i.e. an idle factor) for each processing node. A simple way of computing the load is presented here. Let A be the average step value of all cells

$$A = \frac{1}{NumberOfCells} \sum_{AllCells} CellStepValue \tag{8-3}$$

For each processing node, we compute an *IdleFactor* by adding the signed differences between the processing node cell step values and the average step value A. When the processor is idle, the *IdleFactor* is set to a *MaxIdleFactorValue* minus the number of cell in the processing node[1].

$$IdleFactor = \begin{cases} \sum_{AllCells} (CellStepValue - A), & \text{if the processor is not idle} \\ MaxIdleFactorValue - NumberOfCells, & \text{if the processor is idle} \end{cases} \tag{8-4}$$

A negative *IdleFactor* indicates that the cell step values of the corresponding processing node are behind the other processing nodes. A processing node with a strongly negative *IdleFactor* is overloaded and slows other processing nodes which run its neighboring cells. A positive *IdleFactor* indicates that the corresponding processing node is ahead of the others. The processor of such a processing node may soon become idle since the neighboring dependencies with the cells of other processing nodes will put it in a wait state. To balance the load, a cell from the processing node having the most negative *IdleFactor* should be remapped to the processing node having the largest positive *IdleFactor*. The *IdleFactor* is evaluated periodically, every time a new cell migration is performed. Between two cell migrations, a specific *IntegrationTime* allows the system to take advantage of the previous cell migration.

In order to compute the *IdleFactor*, one thread, called *MigrationThread*, is added in each processing node. Periodically, a new token is generated and traverses all the *MigrationThreads* of every processing nodes. The token is generated in processing node *P0*, then it visits all processing nodes in the order: *P1, P2*,..., *PN* and back to *P0*. The migration token makes three full traversals in order to allow the parallel system to decide which cell to remap. During the first traversal, the migration token collects the number of living cells of each processing node and the sum of their step values. This information is distributed to all the processing nodes during the second traversal. During this same traversal, every node computes its *IdleFactor*. This *IdleFactor* is collected by the migration token and distributed over all processing nodes during the third and last traversal. Then every node decides in a distributed manner which processing nodes are involved in the migration.

[1] A processing node having no cell to process should have a higher *IdleFactor* than processing nodes with cells waiting for neighbouring information.

The processing node from which the migration starts, migrates the cell with the smallest step value. In order to perform the migration, the *IOThread* broadcasts to every processing node the migration cell destination and waits for acknowledgment. Once the *IOThread* receives acknowledgments from every processing node, no further information for the migrating cell will be received on the current processing node. The *IOThread* sends the cell data and all the previously received neighboring information to the destination processing node. The migration is done. The time period between each migration cycle is set by the *IntegrationTime* parameter. If the *IntegrationTime* is too short, the processing nodes will waste time for performing useless cell migrations. In the worst case, a too short *IntegrationTime* results in migrating all the cells of a processing node leaving it without any cell. If the *IntegrationTime* is too large, then the processors may become idle before receiving a migrated cell.

Experiments show that it is difficult to find an a good *IntegrationTime*. In order to improve the cell remapping strategy, let us introduce the notion of *stability*. A processing node is *stable* if the difference between the *CellStep* values within a processing node is at most one:

$$\text{Processing node is } stable \Leftrightarrow Max(CellStepValue) - Min(CellStepValue) \leq 1 \quad (8\text{-}5)$$

A processing node is *unstable*, if it is not *stable*. Since the *ComputeNextCellStep* function processes first the cells with the smallest *CellStep* value, the *stable* state is a permanent state if no cell migration occurs. Without cell migration, each *unstable* processing node will sooner or later reach the *stable* state. The migration cell emission and receiving processing nodes are determined by the *IdleFactor*. In order to improve the cell migration strategy, we take into account two migration rules avoiding in some special cases the migration of cells. We do not migrate the cell if the cell receiving processing node is in an unstable state. This rule avoids to carry out consecutive migrations to the same cell receiving processing node. We also do not migrate if the migrated cell will leave the receiving processing node in a stable state. This rule avoids migration if the receiving processing node has no major advance compared with the emission processing node. These two rules are not applied if the receiving processing node is detected to be idle. The *stability* information is exchanged in the same way as the *IdleFactor*.

```
1.  process SlaveProcessT
2.  {
3.  subprocesses:
4.    IOThreadT IOThread;
5.    ComputeThreadT ComputeThread;
6.    MigrationThreadT MigrationThread;
7.  operations:
8.    // ...
9.  };
10.
11.
12.
13. process GlobalProcessT
14. {
15. subprocesses:
16.   SlaveProcessT SlaveProcess[];
17.   MasterThreadT MasterThread;
18. operations:
19.   Run
20.    in void* inputP out void* outputP;
21.   ExchangeMigInfo
22.    in void* inputP out void* outputP;
23.   // ...
24. }
```

Fig. 8.5 CAP thread hierarchy

```
1.  operation GlobalProcessT::ExchangeMigInfo
2.     in void* inputP out void* outputP
3.  {
4.     SlaveProcess[0].MigrationThread.{} >->
5.     for(int SlaveID = 0; SlaveID < NBSLAVEPROCESS; SlaveID++)
6.     (
7.        SlaveProcess[SlaveID].
8.        MigrationThread.UpdateAndSendMigrationInfo >->
9.        SlaveProcess[(SlaveID+1)%NBSLAVEPROCESS].MigrationThread.ReceiveMigInfo
10.    );
11. }
12. operation GlobalProcessT::Run
13.    in void* inputP out void* outputP
14. {
15.    parallel(MasterThread, void result)
16.    (
17.       ( // here is the exchange migration info part
18.          void,
19.          while(!Finished())
20.          (
21.             ExchangeMigInfo
22.          ),
23.          void
24.       )
25.       ( // here is the computation part
26.          void,
27.          indexed
28.          (int SlaveID = 0; SlaveID < NBSLAVEPROCESS; SlaveID++)
29.          parallel(void, void, MasterThread, void result)
30.          (
31.             SlaveProcess[SlaveID].ComputeThread.{}
32.             indexed
33.             (int CellID = 0; CellID < GetNbCells(); CellID++)
34.             parallel(void, void, MasterThread, void result)
35.             (
36.                if(SlaveID == GetCellSlaveID(CellID))
37.                (
38.                   while(IsCellAive(CellID))
39.                   (
40.                      SlaveProcess[SlaveID].Run(CellID) >->
41.                      SlaveProcess[SlaveID].Migrate(SlaveID, CellID)
42.                   )
43.                )
44.             )
45.          ),
46.          void
47.       )
48.    );
```

Fig. 8.6 CAP specification of the parallel cellular automata

8.5. CAP specification

Figure 8.5 present the CAP thread hierarchy. The main process *GlobalProcessT* contains two subprocesses: the master thread *MasterThread* (line 17) and the hierarchical process pool *SlaveProcess* (line 16). The slave processes contain three subprocesses: *IOThread* (line 4), *ComputeThread* (line 5) and *MigrationThread* (line 6). The *IOThread* is responsible for sending

(resp. receiving) the data to (resp. from) the neighbouring cells. The *ComputeThread* is responsible for performing the computation on each cell located on the current slave. The *MigrationThread* exchanges the migration information (*IdleFactor*,...) between the slaves. These processes contain all operations required to run the parallel cellular automaton. Figure 8.5, shows only two particular operations: the *Run* (lines 19-20) and the *ExchangeMigInfo* (lines 21-22) operations.

The CAP specification of the parallel cellular automaton is given in Figure 8.6. This figure presents the CAP code for the *Run* (lines 12-48) and for the *ExchangeMigInfo* (lines 1-11) parallel operations. The *Run* operation begins with the CAP *parallel* statement, which divide the execution flow into two parts executed independently in parallel. The first part (lines 17-24) is responsible for exchanging the migration information during the program execution. This part is composed with the CAP *while* (line 19) statement and a call to the *ExchangeMigInfo* operation. The *ExchangeMigInfo* operation is based on a CAP *for* (lines 5-10) loop responsible for forwarding the migration information token (*IdleFactor*, *stability*,...) over all the slave processes. At lines 7-8, the migration information token is updated and sent to the next slave. The token is received by the next slave at line 9. Then, the CAP *for* loop is repeated until the token is forwarded over all slaves. The second part of the *Run* operation (lines 25-47) runs the cellular automaton itself. This part is composed with two CAP *indexed parallel* statement traversing respectively all the slaves and all the cells. Then, if the cell belongs to the current slave (line 36) and as long this cell is alive (line 38), the main part of the program is executed (lines 40-41). The main part is divided into two part executed in pipeline. The first part (line 40) consists of performing one computation step on the current cell including the communication with the neighbouring cells. This part is similar to the *exchange and process* operation described in the previous chapter (Section 7.4.3, Fig. 7.12). The second part (line 41) checks if a cell needs to be migrated and if necessary, migrates the current cell to another slave. If the cell is migrated, the *SlaveID* variable (which specify on which slave the current cell is located) is modified so that the next iteration of *while* loop is performed on the new slave.

This program shows the advantages of the CAP language. When the parallel schedule is complicated, expressing the parallel schedule independently from the serial code becomes an advantage. In this example, two parallel schedules, i.e. the computation and the migration information exchange, are combined using the CAP *parallel* (line 15) statement. This shows the compositional capability of CAP. Previously developed schedules can be combined to form a new parallel program. CAP allows to combine parallel schedules without modifying the serial part of the program and without dealing with low level routines such as synchronization or thread creation.

8.6. Performance measurement

The performance measurements were carried out on three input images: a balanced input image, a highly unbalanced input image and a slightly unbalanced input image. The balanced input image (Fig. 8.8) consists of a repetitive pattern ensuring an evenly distributed computation

load. In the highly unbalanced input image (Fig. 8.9), according to the cell distribution scheme defined in equation (8-2), the non-empty cells are distributed unevenly across the processing nodes. One of two processing nodes receives empty cells which require only one computation step. The slightly unbalanced input image (Fig. 8.10) is an intermediate case between the balanced and the highly unbalanced input images. The balanced and unbalanced input images are of size 2048x2048 pixels (8 bits/pixel) and splitted into 16x16 cells, incorporating 128x128 pixels. The slightly unbalanced input image is of size 1024x1536 pixels (8 bits/pixel) and splitted into 8x12 cells, incorporating 128x128 pixels.

The *N* processing nodes are all Bi-Pentium Pro 200 MHz PCs running under WindowsNT 4.0. They are connected through a Fast Ethernet switch.

Figure 8.7 shows the speedup for the balanced input image, the highly unbalanced input image and the slightly unbalanced input image. The measurements are done for 1 to 10 processors. The performances of the algorithms with and without the cell migration scheme are compared. The measurements for the dynamically load balanced algorithm are done with an *IntegrationTime* of 0.5 sec. between each cell migration..

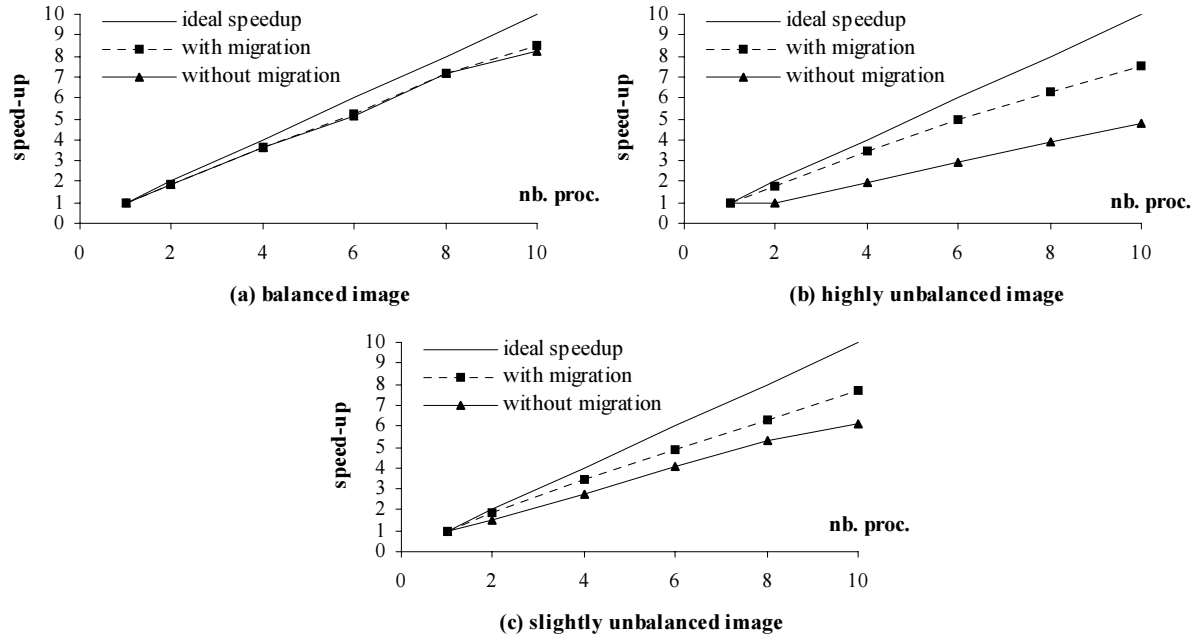


Fig. 8.7 Speedup for (a) the balanced input image, (b) the highly unbalanced input image and (c) the slightly unbalanced image

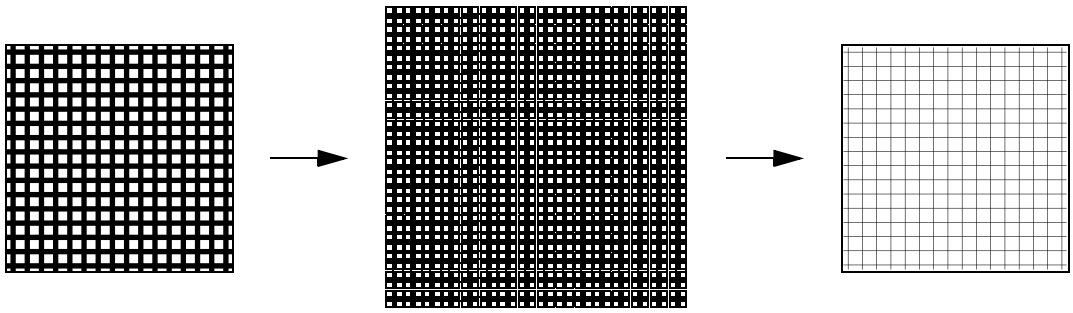


Fig. 8.8 Example (a) of a balanced input image, (b) after segmentation into cells and (c) after skeletonization

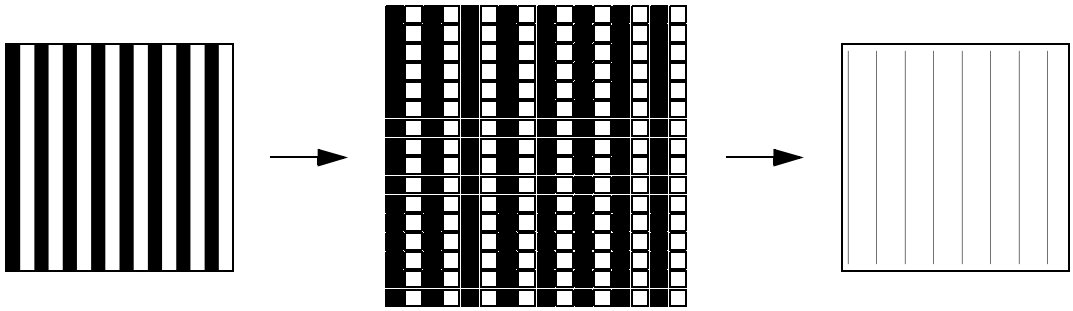


Fig. 8.9 Example (a) of a highly unbalanced input image, (b) after segmentation into cells and (c) after skeletonization

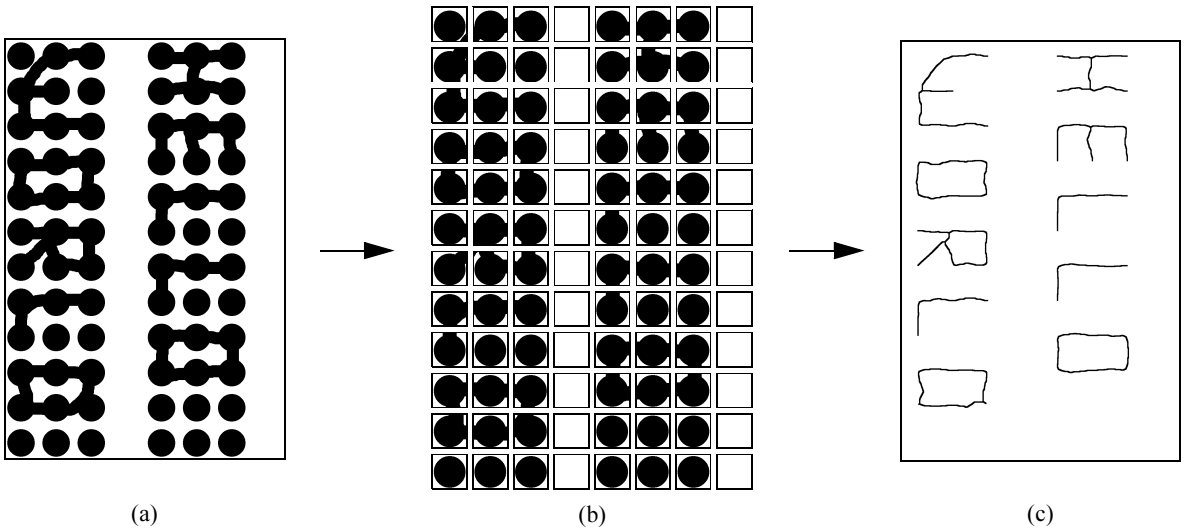


Fig. 8.10 Example (a) of a slightly unbalanced input image, (b) after segmentation into cells and (c) after skeletonization

In the case of the balanced input image, there is no significant performance difference between the two algorithms. For such an input image, the cell migration is useless since the load is perfectly balanced between the processing nodes. The results show that the overhead induced by the management of the cell migration is low. The parallelization does not provide a linear speedup because the neighbouring information exchange consumes processing resources (CPU power for the TCP/IP communication protocol).

In the case of the highly unbalanced input image, the performances are considerably improved by dynamic load balancing. Without cell migration the efficiency (speedup/N) is approximately 50% since one processor of two becomes idle. Implementing the cell migration allows the parallel program to reach approximately the same speedup with a balanced or an unbalanced input image.

In the case of the slightly unbalanced input image, the performances are improved by using the dynamically load balanced algorithm. Since in the input image, about one out of four cells is empty, the theoretically maximal performance improvement factor obtained by the dynamically load balanced algorithm is $4/3 = 1.33$. Our algorithm reaches a performance improvement factor of 1.22. The difference is due to the fact that some time is needed until the system reaches a load balanced state.

Globally, performances are improved by dynamic cell remapping. No improvement is expected in the case of a well balanced input image. A major improvement is achieved in the case of an unbalanced input image. The presented results closely match the expected results.

8.7. Summary

We are interested in the parallelization of cellular automata. Our experiment is based on a particular image skeletonization method. We have developed a parallellization algorithm which can be easily applied to other cellular automata. We explore two parallelization methods, one with a static load distribution consisting in splitting the cells over several processing nodes and the other with a dynamic load balancing scheme capable of remapping cells during the program execution. In order to maintain a load information across the processors, CAP statements allow to propagate a global state in parallel with the main computation. The dynamic approach leads to an efficient parallel solution despite the possible load unbalance of parallel cellular automata. Performance measurements show that the cell migration doesn't reduce the speedup if the application is already load balanced. It improves the performance if the parallel application is not well balanced.

Cellular automata have a wide range of applications: matrix computation, state machines, Von Neumann automata, etc. Many problems can be expressed as cellular automata. Developing from the scratch a custom parallel application requires a large effort. This chapter shows the possibility of developing first a generic parallel cellular automaton and, on top of it, parallel

applications making use of the cellular automaton program interface. This approach reduces the programming effort without loosing efficiency.

9 Parallel Computation of Radio Listening Rates

Obtaining the listening rates of radio stations in function of time is an important instrument for determining the impact of publicity. Since many radio stations are financed by publicity, the exact determination of radio listening rates is vital to their existence and to their further development. Special watches were created which incorporate a custom integrated circuit sampling the ambient sound during a few seconds every minute. Each watch accumulates these compressed sound samples during one full week. Watches are then sent to an evaluation center, where the sound samples are matched with the sound samples recorded from candidate radio stations. The present chapter describes the processing steps necessary for computing the radio listening rates, and shows how this application was parallelized on a cluster of PCs using the CAP Computer-aided parallelization framework. Since the application must run in a production environment, the chapter describes also the support provided for graceful degradation in case of transient or permanent failure of one of the system's components. The content of this chapter is published in [Mazzariol00b].

9.1. Introduction

Obtaining the listening rates of radio stations in function of time is an important instrument for determining the impact of publicity. Since many radio stations are financed by publicity, the exact determination of radio listening rates is vital to their existence and to their further development. Existing methods of determining radio listening rates are based on face to face interviews or telephonic interviews made with a sample population. These traditional methods however require the cooperation and compliance of the participants.

In order to significantly improve the determination of radio listening rates, special watches [Sun99] were created which incorporate a custom integrated circuit sampling the ambient sound during a few seconds every minute. The sound samples are split into sub-bands and converted into energy signals. Their envelopes are extracted, low-pass filtered and resampled at a much lower rate. Sound compression allows to store a full week of one minute records within the limited memory space of a watch. Watches are then sent to an evaluation center, where transformed sound samples are matched with the transformed sound samples recorded from candidate radio stations. Based on the matching performance, reliable listening rate statistics are established. Figure 9.1 shows the complete data flow from the radio station emitter to the correlation center.

The evaluation center should be able to determine in *real time* the listening rates for a configuration of approximately 1000 watches and 100 radio stations. The computation of the listening rates for a single day (24h) should therefore take less then one day.

The present chapter describes the processing steps necessary for computing the radio listening rates, and shows how this application was parallelized on a cluster of PCs using the CAP Computer-aided parallelization framework. Since the application must run in a production

environment, the chapter also describes the support for graceful degradation in case of transient or permanent failure of one of the system's components.

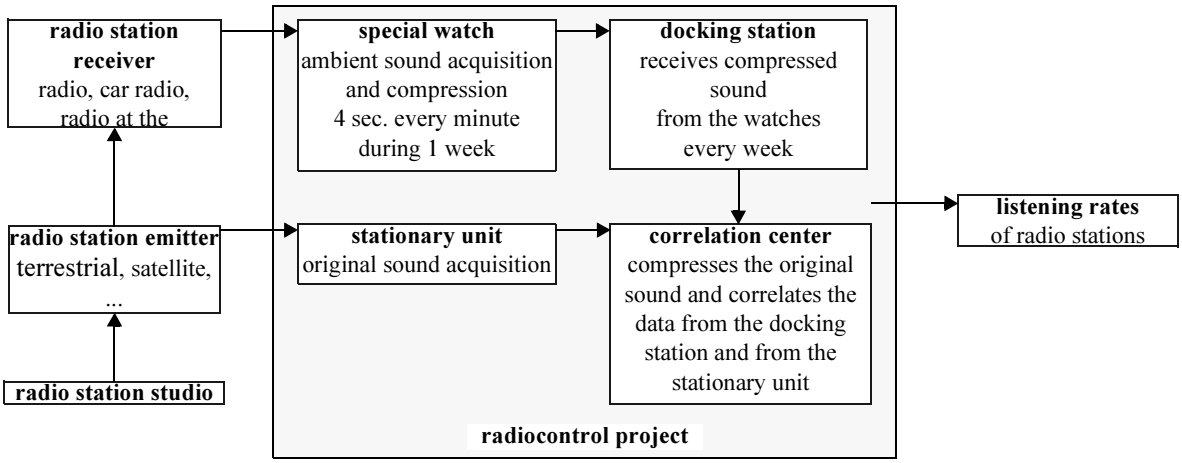


Fig. 9.1 Complete data flow from the radio emitter to the correlation center

9.2. The matching problem

In order to obtain the listening rates of radio stations for a given hour within a given day, one must match the transformed sound of the 1000 watches and of the 100 candidate radio stations. Each watch records the ambient sound during 4 seconds every minute. Each radio station is recorded during 10 seconds every minute. In both cases, sound is sampled at 3KHz. A small time shift may appear between sound acquisition in the watch and in the stationary unit, due to different transmission modes (e.g. terrestrial, cable, satellite). In addition, the quartz inside the watch has a limited precision and the frequency may vary due to temperature variations. To reduce the time shift, the watches are synchronized every week (when they are sent back to the docking station) and a temperature sensor is incorporated in the watch to correct the clock deviation. In order to take into account the time shift, the discrete correlation between the compressed sound from the watch and from the stationary unit is done at successive 1 ms intervals, i.e. the matching is established by varying, millisecond per millisecond, the time position of the 4 seconds of sound acquired by each watch within a time window of 10 seconds. The largest match (correlation maximum) between watch and radio station is recorded for further evaluation, i.e. to determine for the considered minute and for a given watch whether the sound stems from one of the candidate radio stations. Figure 9.3 shows a global scheme of the matching procedure.

9.2.1. Storage

The data required to perform the matching between ambient sound samples and radio station sound is organized in files. There is one radio station file per hour, which contains 60 minute records incorporating each 10 seconds of uncompressed sound. Each minute record takes

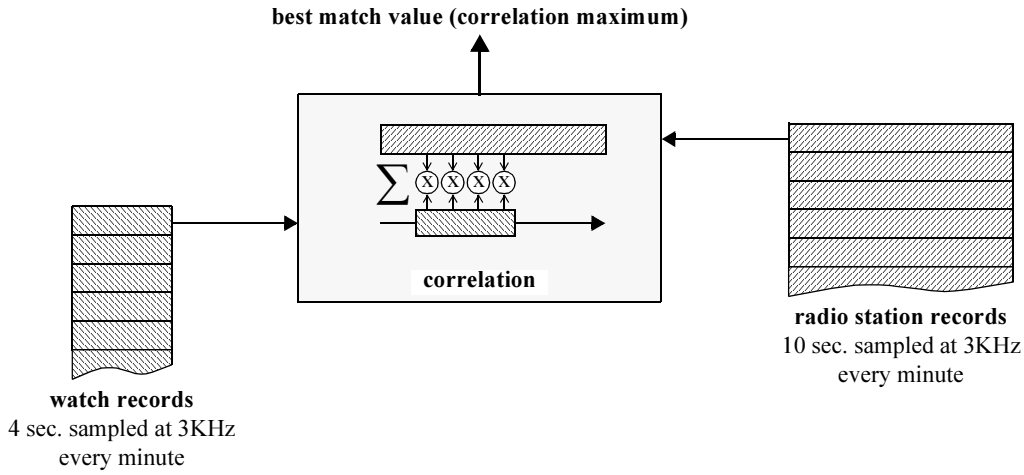


Fig. 9.2 Global scheme describing the matching process

60 KB. Therefore each hourly file has a size of 3.5 MB. Since we want to establish the listening rate for a full day (24 hours) and 100 radio station, we need 8.2 GB.

$$\text{one radio station minute record size} = 10\ \text{sec} \times 3\ \text{KHz} \times 2\ \text{Bytes} = 60\ \text{KB} \tag{9-1}$$

$$\text{one hourly radio station file size} = 60 \times 60 \cong 3.5\ \text{MB} \tag{9-2}$$

$$\text{total radio station file size} = 3.5 \times 24 \times 100 \cong 8.2\ \text{GB} \tag{9-3}$$

The watch file contains all minute records comprising 4 seconds of highly compressed sound for a full day, i.e. 24x60 records of 95 Bytes. Therefore each daily watch file takes 134 KB. Since we have 1000 watches, the total space required for the watch files is 134 MB.

$$\text{one watch minute record size} = 95\ \text{Bytes} \tag{9-4}$$

$$\text{one dayly watch file size} = 95 \times 24 \times 60 \cong 134\ \text{KB} \tag{9-5}$$

$$\text{total watch file size} = 134 \times 1000 \cong 134\ \text{MB} \tag{9-6}$$

Correlating one watch record with one radio station record produces a new result record of 8 Bytes (correlation maximum). Each result file contains all the result records for one hour, one radio station and all the 1000 watches. Therefore one result files has a size of 470 KB. Since there are 24x100 result files, the total required space for the result files is 1.1 GB.

$$\text{one result record size} = 8\ \text{Bytes} \tag{9-7}$$

$$\text{one hourly result files size} = 8 \times 60 \times 1000 \cong 470\ \text{KB} \tag{9-8}$$

$$\text{total result file size} = 470 \times 24 \times 100 \cong 1.1\ \text{GB} \tag{9-9}$$

By considering the total radio station file size (9-3), the total watch file size (9-6) and the total result file size (9-9), the total amount of data used to establish the listening rate for a full day (24 hours), 100 radio stations and 1000 watches is about 10 GB.

$$\text{total size} = 8.2 + \frac{134}{1024} + 1.1 \cong 9.4 \text{ GB} \qquad (9\text{-}10)$$

The radio station files, the watch files and the result file are all stored on a single PC file server. The correlation processes can freely access (read and write) this file server. Remote access to the file is achieved thanks to the NTFS distributed file system, which allows to share the physical hard drives of the file server over the local network. The full matching process for one day needs to transfer from the file server over the network a total amount of data of about 10 GB. The file server and the correlation process computer run under WindowsNT 4.0 and are connected through a Fast Ethernet network (100Mbits/sec.) using the TCP/IP protocol.

9.2.2. Serial correlation algorithm

In serial processing mode, in order to obtain the result for one hour, 100 radio stations, and 1000 watches, the matching program incorporates the following steps:

1. Open and seek within each of the 1000 watch files to get the required target single hour records, i.e. 60 consecutive minute records of 95 Bytes (9-4) from each watch file. This requires 1000 disk accesses, each one for reading $95 \times 60 \cong 5.5$ KB.
2. For all the 100 radio stations:
   2a. Load the hourly radio station file for the current hour. This step consists of a single disk access of 3.5 MB (9-2).
   2b. Correlate the corresponding records from the hourly radio station file and the 1000 watch files. Since there are 60 records in the radio station file (one per minute) and each must be correlated with the corresponding minute record of each of the 1000 watches, i.e. 60x1000 = 60000 correlations need to be carried out. All the resulting match values (correlation maxima) are stored in local memory.
   2c. Flush the resulting match values from the local memory to the remote result file on the file server. This step consists of a single write disk access of 470 KB (9-8).

To establish the listening rate for a full day (24 hours), we need to repeat the steps listed above 24 times.

9.2.3. Serial performance analysis/measurements

The following serial performance measurements have been done with one Pentium-II PC 400 MHz file server and one Pentium-II PC 400 MHz correlation computer, both running under WindowsNT 4.0 and connected through a Fast Ethernet network (100 Mbits/sec.). Let us ana-

lyze the time taken by each step described in the previous section. Step 1 requires 1000 disk accesses of 5.5 KB each. Since the data size is small and seek time is predominant, we obtain an effective throughput of 550 KB/sec., i.e. 10 sec. are needed to complete step (1).

$$\text{total time for step 1} = \frac{1000 \times 5.5}{550} = 10 \text{ sec} \tag{9-11}$$

Step 2a is just a single disk access of 3.5 MB. At an effective throughput of 2 MB/sec., it takes about 1.75 sec. Since this step is repeated for all the radio station, we get the following total time:

$$\text{total time for step 2a} = 100 \times \frac{3.5}{2} = 175 \text{ sec} \tag{9-12}$$

The match time of two records is highly data dependent due to optimizations in the correlation procedure. Nevertheless, the mean time to correlate two records is 1.85 milliseconds. This value is correct for normal ambient sound. Step 2b requires 60000 correlations and therefore needs 111 sec. to complete. Since this step is repeated 100 times, we get:

$$\text{total time for step 2b} = 100 \times 60000 \times 0.00185 = 11100 \text{ sec} \tag{9-13}$$

Step 2c is just a single disk write access of 470 KB. At an effective write throughput of 2 MB/sec. it takes 0.23 sec. Since this step is repeated 100 times:

$$\text{total time for step 2c} = 100 \times \frac{470}{2 \times 1024} \cong 23 \text{ sec} \tag{9-14}$$

The total time to establish the listening rate for 24 hours, 100 radio stations and 1000 watches is obtained by adding the previous results and multiplying them by 24:

$$\text{total serial time} = 24 \times (10 + 175 + 11100 + 23) \cong 75.3 \text{ hours} \tag{9-15}$$

Since the I/O time represents less than 2% of the total time, the process is compute bound and no significant gain could be achieved by pipelining step 2a, step 2b and step 2c (i.e. making asynchronous accesses to the file server) in the serial implementation. Therefore the total processing time is approximately proportional to the number of watches multiplied by the number of radio stations, i.e. it can be deduced from equation (9-13).

$$\begin{aligned}\text{tot ser time} &\cong 24 \times 60 \times 0.00185 \times nbWatches \times nbRadioStations = \\ &2.664 \times 1000 \times 100 = 74 \text{ hours}\end{aligned} \tag{9-16}$$

The practical measurements are consistent with the previous analysis. The total computation time to produce the listening rate for one day exceeds the 24 hours limit. Therefore parallelization is needed.

9.3. Parallelization

9.3.1. The Computer-Aided Parallelization (CAP) framework

The CAP Computer-Aided Parallelization framework is specially well suited for the parallelization of applications having significant I/O bandwidth requirements. Application programmers specify at a high level of abstraction the set of threads present in the application, the processing operations offered by these threads, and the flow of data and parameters between operations. Such a specification is precompiled into a C++ source program which can be compiled and run on a cluster of distributed memory PCs. The compiled application is executed in a completely asynchronous manner: each thread has a queue of input tokens containing the operation execution requests and its parameters. Disk I/O operations are executed asynchronously, i.e. while data is being transferred to or from disk, other operations can be executed concurrently by the corresponding processor. In a pipeline of disk access and process operations, CAP allows therefore to hide the time taken by the disk access operations if the process is compute bound. A configuration file (see Chapter 3, Section 3.3.1) specifies the mapping between CAP threads and operating system processes possibly located on different PCs. This configuration file is used by the run-time system to launch the parallel application. A Remote Shell Daemon (rshd) is located on each target computer to launch processes remotely and carrying out I/O redirection.

9.3.2. Parallel correlation algorithm

The correlation application has been parallelized in a master-slave fashion (Fig. 9.3). The master only distributes jobs to the slaves, which are the effective workers. Since the master is idle most of the time, he resides on the file server computer. The *n* slaves are connected to the master through a Fast Ethernet network (100 Mbits/sec.) using the TCP/IP protocols. All the computers are Pentium-II 400 MHz PCs running under WindowsNT 4.0. The radio station files, watch files and result files (total 10 GB) are stored in an array of Ultra Wide SCSI-II disks connected to the file server. The disks offer an effective throughput of 2 MB/sec. each, the SCSI bus has a maximal throughput of 40 MB/sec. and the Fast Ethernet network does not support more than 8 MB/sec.

The serial processing operations described in the previous section are parallelized by having different processors (*n* processors) working on different radio station files. The only change resides in step 2. Instead of making a simple loop over the 100 radio stations, each slave receives and computes a different iteration of the loop, i.e. works on a different hourly radio station file. Each slave reads a new radio station file and computes a new result file. This implies that the radio station and result files are read/written once during the whole program execution. When a slave starts to compute a new hour, he loads the corresponding watch minute records within the 1000 watch files. This correspond to step 1 and is done on every *n* slaves each time a new hour computation starts, i.e. each watch minute record is accessed *n* times, once by each slave. Therefore the watch files are loaded *n* times during the program execution. Equations (9-3) and (9-9) remain unchanged, but equation (9-6) must be multiplied by *n*. If *n* is small (near 10) the

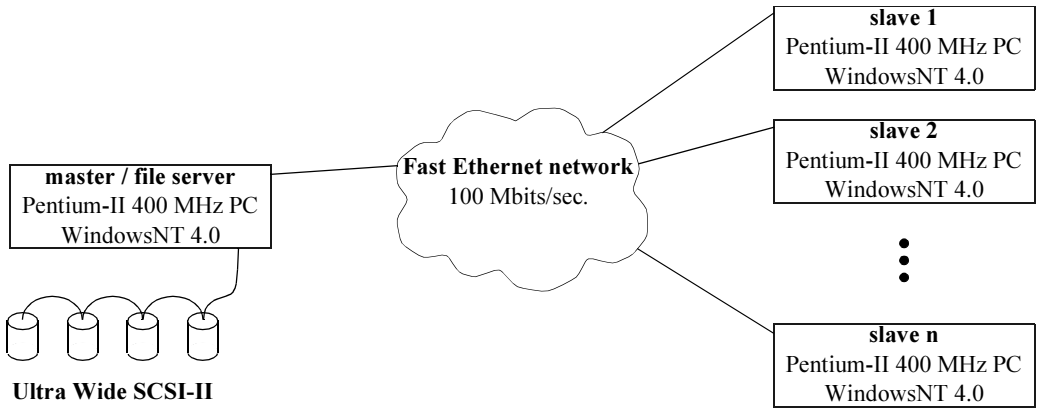


Fig. 9.3 Parallel architecture

total amount of data transferred from the disks over the network will not increase more than 20%, which is acceptable. Equation (9-6) and (9-10) become respectively for $n = 10$:

$$\text{total watch file transfered size} = n \times 134 \text{ MB} = 10 \times 134 \cong 1.31 \text{ MB} \quad (9\text{-}17)$$

$$\text{total transfered size} = 8.2 + 1.31 + 1.1 \cong 10.6 \text{ GB} \quad (9\text{-}18)$$

Since there are 100 radio stations and 24 hours, the master distributes 2400 jobs to the slaves representing a total computation time of about 75.3 hours (9-15). The CAP framework helps the programmer to distribute these jobs intelligently. When the computation starts, each slave receives a job from the master and starts computing it. Immediately after this first job distribution, i.e. while the slaves are still busy with their first job, the master sends a second job to all the slaves. This new job is not executed immediately but inserted in a job queue in each slave. As soon as a slave finishes computing his first job, he starts processing the second job (no idle time). During the execution of the second job, the master sends a third job to the slave and the whole process is repeated. This job distribution mechanism ensures that the slaves are never waiting for a new job, i.e. are always busy, and that the total computation time is load balanced between the slaves. Since each slave receives a new job when he finishes computing the previous one, each slave works at its own speed; a faster slave will receive more jobs than a slower slave and since the jobs do not exactly require the same computation time, a slave can (for example) receive 2 large jobs while another receives 3 small jobs during the same time interval.

Since the radio station, watch and result files reside on a single computer, the file server treats the slaves read/write requests sequentially. In the previous section, equations (9-11), (9-12) and (9-14) shows that the total access time to the file server is:

$$\text{total transfered size} = 8.2 + 1.31 + 1.1 \cong 10.6 \text{ GB} \quad (9\text{-}19)$$

This time is insignificant when compared with the 75.5 hours (9-15) required to establish the listening rates in serial execution mode, but it may become significant in parallel execution mode, as stipulated by Amdahl's law. Therefore it is important to access asynchronously to the file server, i.e. read/write accesses should be hidden by the slaves computation time. The CAP framework helps the programmers in this task. CAP allows to assign the steps 2a, 2b and 2c to three different threads. This enables the execution of the three steps in a pipeline. While a job is requiring computation power and keeping the slave processor busy at step 2b, another job can make accesses to the file server (step 2a or 2c). In a totally transparent manner for the programmer, CAP ensures the correct data flow through the operations of the pipeline. Figure 9.4 shows the slave activity and resource occupation during the correlation process.

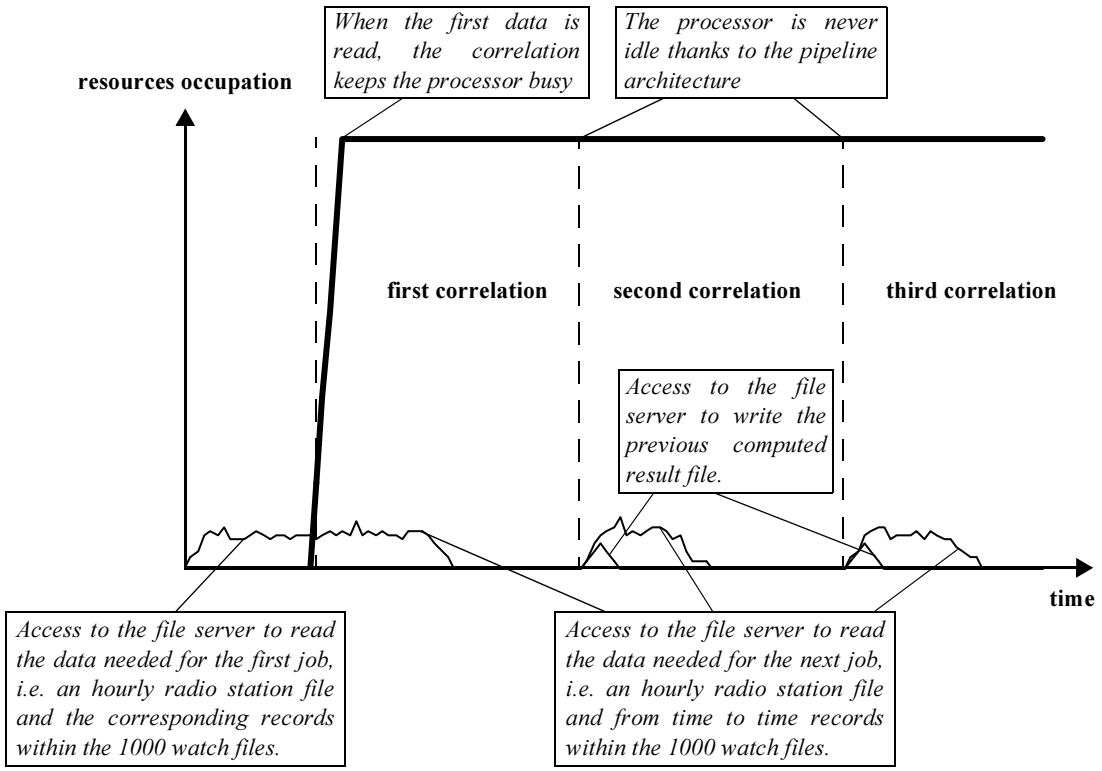


Fig. 9.4 Slave activity during the correlation process: the thick line represents the processor occupation on the slave, and the thin lines represent the file server accesses

9.3.3. CAP program specification

In this section, we present the CAP parallel program specification. Figure 9.5 shows the complete CAP code used in this project. The rest of the code (not presented in Figure 9.5) is the original sequential C/C++ code.

The *GlobalProcess* (lines 31-40 and 64) is the top level process. This hierarchical (see Chapter 3, Section 3.3) process defines the top level parallel operation *ProcessOneDay*. *GlobalProcess*

```
1.  process IOProcessT
2.  {
3.  operations:
4.     ReadData
5.        in ReadDataInT* inputP
6.        out ReadDataOutT* outputP;
7.     WriteResult
8.        in WriteResultInT* inputP
9.        out WriteResultOutT* outputP;
10. };
11. process ComputeProcessT
12. {
13. operations:
14.    Correlate
15.       in CorrelateInT* inputP
16.       out CorrelateOutT* outputP;
17. };
18. process SlaveProcessT
19. {
20. subprocesses:
21.    IOProcessT IOProcess[2];
22.    ComputeProcessT ComputeProcess;
23. operations:
24.    ProcessOneHour
25.       in ProcessOneHourInT* inputP
26.       out ProcessOneHourOutT* outputP;
27. };
28. process MasterProcessT
29. {
30. };
31. process GlobalProcessT
32. {
33. subprocesses:
34.    SlaveProcessT SlaveProcess[];
35.    MasterProcessT MasterProcess;
36. operations:
37.    ProcessOneDay
38.       in ProcessOneDayInT* inputP
39.       out ProcessOneDayOutT* outputP;
40. };
41. operation SlaveProcessT::ProcessOneHour
42.    in ProcessOneHourInT* inputP
43.    out ProcessOneHourOutT* outputP
44. {
45.    IOProcess[0].ReadData >->
46.    ComputeProcess.Correlate >->
47.    IOProcess[1].WriteResult;
48. }
49. operation GlobalProcessT::ProcessOneDay
50.    in ProcessOneDayInT* inputP
51.    out ProcessOneDayOutT* outputP
52. {
53.    MasterProcess.{} >->
54.    flow_control(PipelineDepth*NbSlaves)
55.    indexed
56.    (int i = 0; !SplitIsFinished(); i++)
57.    parallel (SplitHour, MergeHour,
58.       MasterProcess, MergeHourOutT res())
59.    (
60.       SlaveProcess[cap_fcindex0%NbSlaves]
61.          .ProcessOneHour
62.    );
63. }
64. GlobalProcessT GlobalProcess;
```

Fig. 9.5 CAP program specification for the radiocontrol parallel application

contains a leaf process *MasterProcess* (lines 28-30 and 35) and a pool of slave processes *SlaveProcess[]* (lines 18-27 and 34). The *MasterProcess* process is a leaf process, it is only used to specify where the *SplitHour* split function and the *MergeHour* merge function (line 57) must be executed. The *SlaveProcess* process is a hierarchical process responsible for scheduling (*ProcessOneHour* CAP operation) the work on the hourly radio station files (read, correlate and write). *SlaveProcess* contains two *IOProcess* (lines 1-10 and 21) leaf processes and one *ComputeProcess* (lines 11-17 and 22) leaf process. *IOProcess[1]* is responsible for the step 2a (see 9.2.2) of the serial algorithm (*ReadData* CAP leaf operation). *IOProcess[2]* is responsible for the step 2c of the serial algorithm (*WriteResult* CAP leaf operation). The *ComputeProcess* is responsible for the correlation (*Correlate* CAP leaf operation), i.e step 2b of the serial algorithm. The *IOProcess[1]*, *IOProcess[2]* and *ComputeProcess* are the only leaf processes (except *MasterProcess*) of the application. Typically, these three leaf processes directly map (configuration file, see Chapter 3, Section 3.3.1) to three operating system threads within the same operating system process. The three CAP processes are logically grouped into the CAP *SlaveProcess* abstraction. Since we have a pool of *SlaveProcess* processes, the subprocess

(*IOProcess[1]*, *IOProcess[2]* and *ComputeProcess*) of the two different *SlaveProcess* processes are typically mapped to different computing nodes.

Such a process decomposition ensures a perfect pipelining between the leaf operations (read, correlate and write). Such a pipelining is only possible if the underlying hardware supports it. Clearly, it is assumed that the network interface allows to carry out network transfer without interrupting the processing activities and offers full duplex connection (simultaneous in and out network transfers).

The implementation of the *ProcessOneDay* and *ProcessOneHour* CAP operation is straightforward. At lines 54 and 60 we note the usage of the CAP keyword *flow_control* in combination with the CAP keyword *cap_fcindex0* in order to ensure dynamic load balancing between the slaves (see Section 4.3).

Compared with a solution based on scripts launching parallel programs on a farm of processors, the CAP program shown in Figure 9.5 offers a complete parallel solution combining efficiency (see next section), reliability (see Section 9.4), scalability, asynchronous data access, debugging capabilities, and simplicity.

9.3.4. Parallel performance analysis/measurements

Within a single processor, disk accesses are asynchronous and executed in parallel with processing operations. After initialization of the pipeline, only the processing time, i.e. the correlation between radio station files and watch records determines the overall processing time. Therefore the time to perform the steps 1, 2a and 2c described by equations (9-11), (9-12) and (9-14) is hidden behind the processing time. Only the processing time of step 2b described by equation (9-13) remains. The total parallel time on *n* slaves and when doing asynchronous I/O accesses is given by:

$$\text{total parallel time} = \frac{24 \times 11100}{n} = \frac{74}{n} \text{ hours} \qquad (9\text{-}20)$$

Figure 9.6 shows the computation time when increasing the number of slaves. The measurements have been done for 24 hours, 100 radio stations and 1000 watches on Pentium-II 400 MHz PCs. The performance results show a linear speedup, i.e. processing time which is 74 hours on a single computer decreases to 24.7 hours with 3 PCs, 18.5 hours with 4 PCs (i.e. under 24 hours limit) and 14.8 hours with 5 PCs which is under the 16 hours time constraint imposed by the industrial specification. When the number of slaves becomes larger (above 20) the speedup will not remain linear since the shared file server will become the bottleneck.

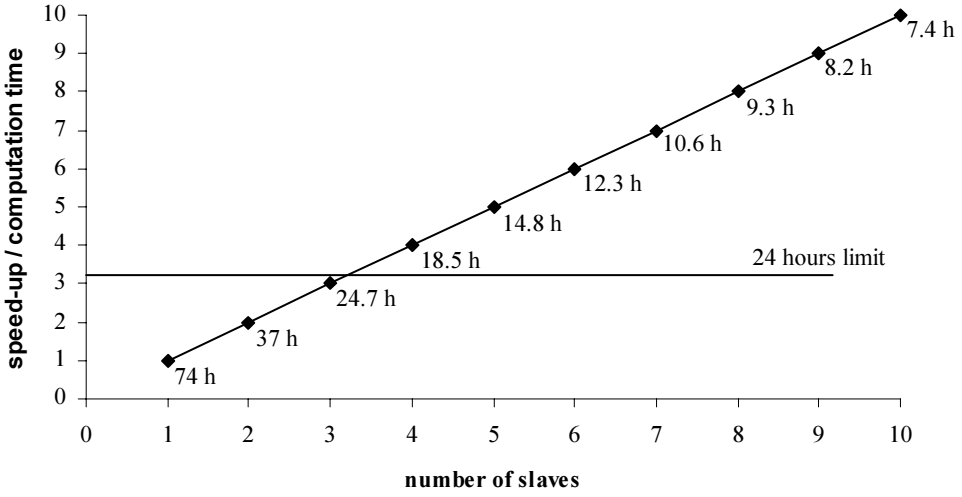


Fig. 9.6 speedup diagram

9.4. Graceful degradation in case of failure

Since the presented application should work in a production environment, it is highly important to provide support for graceful degradation in case of transient or permanent failure of one of the system's components (failure of a slave PC for example). This support is based on the *checkpoint and restart* paradigm [Gray92]. A watchdog process (independent from the CAP program) running in the master computer checks that at least once every 20 minutes, new results are written to the result file. If this is not the case, the watchdog program kills the radio listening rate computation process on every participating compute nodes. Since the current version of Windows NT does not allow to kill properly some processes (releasing DLLs), the watchdog can optionally also reboot the participating compute nodes. Then, the watchdog checks the network to verify which PCs are working, and generates a new configuration file is generated incorporating only those PCs which are currently living. The application is relaunched on the new configuration. The application first determines if the result file exists and which were the last consistent records written on file. The application then resumes the computation in order to create the next matching records.

Figure 9.7 shows a typical configuration file for the radiocontrol parallel application. This configuration file runs the application (*Radiocontrol.exe*) on one master operating system process (*M*) and two slave operating system processes (*S1*, *S2*). In the present example, we have one thread in the master (*MasterThread*) and three threads on each slave (on *S1*: *Slave[0].ComputeThread*, *Slave[0].IOThread[0]*, *Slave[0].IOThread[1]*). The master is launched at the IP-address given by *MasterLocation*. The slaves are launched on any of the IP-addresses in the *SlavePool*. If both computers from the *SlavePool* are living, *S1* is launched on the first and *S2* on the second computer. If one computer from the *SlavePool* is down then *S1* and *S2* are launched on the other living computer. This configuration allows the system to work properly

```
configuration {
pools :
   MasterLocation {123.456.78.00};
   SlavePool      {123.456.78.01, 123.456.78.02};
processes :
   M  (MasterLocation,  Radiocontrol.exe);
   S1 (SlavePool,       Radiocontrol.exe);
   S2 (SlavePool,       Radiocontrol.exe);
threads :
   MasterThread            (M);
   Slave[0].ComputeThread  (S1);
   Slave[0].IOThread[0]    (S1);
   Slave[0].IOThread[1]    (S1);
   Slave[1].ComputeThread  (S2);
   Slave[1].IOThread[0]    (S2);
   Slave[1].IOThread[1]    (S2);
};
```

Fig. 9.7 Typical configuration file for the radiocontrol parallel application

in the case of a single slave computing node failure. The master remains a critical resource (since it's the file server). Other configurations with more slaves (e.g. 10 slaves) and multiple masters are possible.

9.5. Summary

In order to compute the listening rates of radio stations, we developed a parallel sound correlation program running on a cluster of PCs interconnected by Fast Ethernet. The parallel sound matching server offers a linear speedup up to a large number of PCs thanks to the fact that disk access operations across the network are done in parallel with computations. Support is provided for graceful degradation of the sound matching server. In the case of failure of slave computation PCs, the current computation is stopped, contributing processes are killed, the network is checked for living PCs and the application is automatically restarted on the new configuration.

The Swiss Radio company decided to adopt this system to measure the listening rates of radio stations in Switzerland [SSR99]. The system listens to 120 radio stations and 30 televisions in 17 different studio locations. There are 800 test persons equipped with the special watch. The correlation is done with 9 slave PCs and 1 master PC, each one a Pentium-III 500 MHz computer. Commercial contacts with other countries have been established. A test panel has already been installed in Paris.

The project was a large collaborative effort with the contribution of many partners: the Institute of Microtechnics, Univ. of Neuchatel, for developing the custom integrated circuits (very low power A/D-converter) for sound acquisition and the compression in the watch [IMT99], EPFL for parallelizing the sound matching algorithms, IBW AG for integrating all the software into the evaluation system [IBW99], Liechti AG for creating the general concept and building

the watches [Liechti99][Wüthrich94] and University of Zurich, Martin Bichsel, for creating the first version of the sound acquisition and compression software [Bichsel98].

The parallel application is currently working for about 2 years in an industrial environment. No particular difficulties concerning the CAP framework have been encountered at this time. This project proves the ability of CAP to handle and solve industrial problems.

10 Discrete Optimization Problems

This chapter present a didactical CAP parallel program solving the famous traveling salesman problem. The basic notions of discrete optimization problems are introduced. We enter in the world of NP-Hard problems. Several sequential search techniques are presented. Their parallelization and inherent main difficulties are discussed.

10.1. Parallelization of hard nonnumeric problems

Parallel computation of scientific problems has been the main focus of supercomputing. Numerous scientific problems have been successfully implemented on various parallel computers. Regular runtime behavior and deterministic resource usage are among the reasons which have contributed to their successful parallel implementations. Due to this type of deterministic behavior, the resource allocation becomes rather straightforward for this type of numeric problems. Nonnumeric problems on the other hand have received little attention in terms of parallel implementation on multiprocessors. The main reasons that parallel nonnumeric computation have been less successful than numeric computations are the large amount of computational resource usage, their irregular runtime behavior, and often the unknown number of iterations. Due to these unknown patterns, these types of problems are difficult to parallelize and deploy on parallel machines. Even if nonnumeric problems are successfully implemented, the resulting performance is often poor compared to that of numeric problems [Sohn95].

10.2. Discrete optimization problems[1]

A Discrete Optimization Problem (DOP) can be expressed as a tuple (S, f). The set S is a finite or countably infinite set of all solutions that satisfy specified constraints. This set is called the set of *feasible solutions*. The function f is the *cost function* that maps each element in set S onto the set of real numbers $\mathbb{R}$.

$$f : S \to \mathbb{R}$$

The objective of a DOP is to find a feasible solution x_{opt} such that $f(x_{opt}) \leq f(x)$ for all $x \in S$.

Problems from various domains can be formulated as DOPs. Many examples could be found in the literature: traveling salesman problem, postman's problem, knapsack problem, parallel machine scheduling, vertex coloring, spanning tree, shortest path, etc. Some are extremely well solved, while others continue to frustrate researchers after two centuries of attention. More prac-

[1] Definitions and examples are taken from [Kumar94]

tical examples are: planning and scheduling, finding optimal layouts of VLSI chips, robot motion planning, test pattern generation for digital circuits, and logistics and control.

Example 10-1. The 0/1 integer-linear-programming

Give an $m \times m$ matrix A, an $m \times 1$ vector b, and an $m \times 1$ vector c. The objective is to determine an $n \times 1$ vector x whose elements can take on only the value 0 or 1. The vector must satisfy the constraint

$$Ax \geq b \tag{10-1}$$

and the function

$$f(x) = c^T x \tag{10-2}$$

must be minimized. The set S is the set of all values of the vector x that satisfy the equation (10-1). The cost function f (a scalar product) is evaluated for each vector x in S. The vector $x_{opt} \in S$ which minimizes equation (10-2) is the DOP solution. ■

Example 10-2. The eight-puzzle problem

The Eight-puzzle problem consists of a 3×3 grid containing eight tiles, numbered one through eight. One of the grid segments (called the *blank*) is empty. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine a shortest sequence of moves that transforms the initial configuration to the final configuration. Figure 10.1 illustrates a sequence of moves leading from the initial configuration to the final configuration.

The set S for this problem is the set of all sequences of moves that lead from the initial to the final configurations. The cost function f of an element in S is defined as the number of moves in the sequence. ■

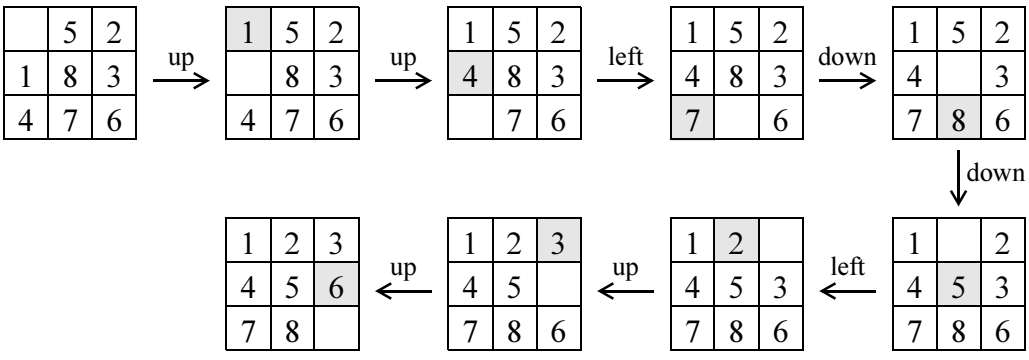


Fig. 10.1 An Eight-puzzle problem instance: sequence of moves leading from the initial to the final configuration

Why are DOPs so difficult to solve? DOP's feasible solution spaces S are enormous in size and they grow exponentially with the number of discrete choices to be resolved. For example, a problem requiring a modest 200 independent, binary decisions has $2^{200} = 10^{60}$ solutions to consider. A case with 201 binary decisions has twice as many. The enumeration of even a tiny fraction of the set of solutions is computationally untenable. Unfortunately, the state of our present knowledge regarding most DOPs is that we need to enumerate all, or at least a significant fraction, of the solution space to solve them. Except a few well solved DOPs, most DOPs belong to the *NP-Equivalent* [Parker88] class of problems, which is a strong reason to believe that quite fundamental limits of our mathematics and computing machines will leave most DOPs permanently in this *enumeration required* category.

One may argue that it is pointless to apply parallel processing to these problems, since we could probably never reduce their run time to a polynomial without using exponentially many processors. This is a misrepresentation of complexity theory. NP-Equivalent problems are equivalent only in the sense that no algorithm is likely to be discovered that resolves *every* instance in polynomially bounded time. This does not preclude doing well on smaller instances. Heuristic search algorithms for many problems have a polynomial average-time complexity. Furthermore, there are heuristic search algorithm that find suboptimal solutions for specific problems in polynomial time. In such cases, larger problem instances can be solved using parallel processing.

10.3. Heuristics

For some problems, it is possible to estimate the cost to reach the goal state from an intermediate state. This cost is called a *heuristic estimate*. Let $h(x)$ denote the heuristic estimate of reaching the goal state from state x and $g(x)$ denote the cost of reaching state x from the initial state s along the current path. The function h is called a *heuristic function*. If $h(x)$ is a lower bound on the cost of reaching the goal state from state x for all x, then h is called *admissible*. We define function $l(x)$ as the sum $h(x) + g(x)$. If h is admissible, the $l(x)$ is a lower bound on the cost of the path to a goal state that can be obtained by extending the current path between s and x. In subsequent sections we will see how an admissible heuristic can be used to determine the least-cost sequence of moves from the initial state to a goal state.

Example 10-3. An admissible heuristic function for the Eight-puzzle

Assume that each position in the Eight-puzzle grid represented as a pair (i, j). The distance between position (i, j) and (k, l) is defined as the *Manhattan distance* $|i - k| + |j - l|$. The sum of Manhattan distances between the initial and final positions of all tiles is an estimate of the number of moves required to transform the current configuration into the final configuration. This estimate is called the *Manhattan heuristic*. Note that if $h(x)$ is the Manhattan distance between configuration x and the final configuration, then $h(x)$ is also a lower bound on the number of moves from configuration x to the final configuration. Hence the Manhattan heuristic is admissible. ■

10.4. Sequential search algorithms

In this section we present some sequential search algorithms to solve DOPs that are formulated as *tree* or *graph* search problem[1]. In a tree, each new successor leads to an unexplored part of the search space. In a graph, however, a state can be reached along multiple paths. For such problems, whenever a state is generated, it is necessary to check if the state has already been generated. If this check is not performed, the graph is unfolded into a tree (repeating some states). Clearly, unfolding increases the search space. For many problems the search space increases only by a small factor, but for some problems it grows exponentially. Figure 10.2 illustrates the unfolding process.

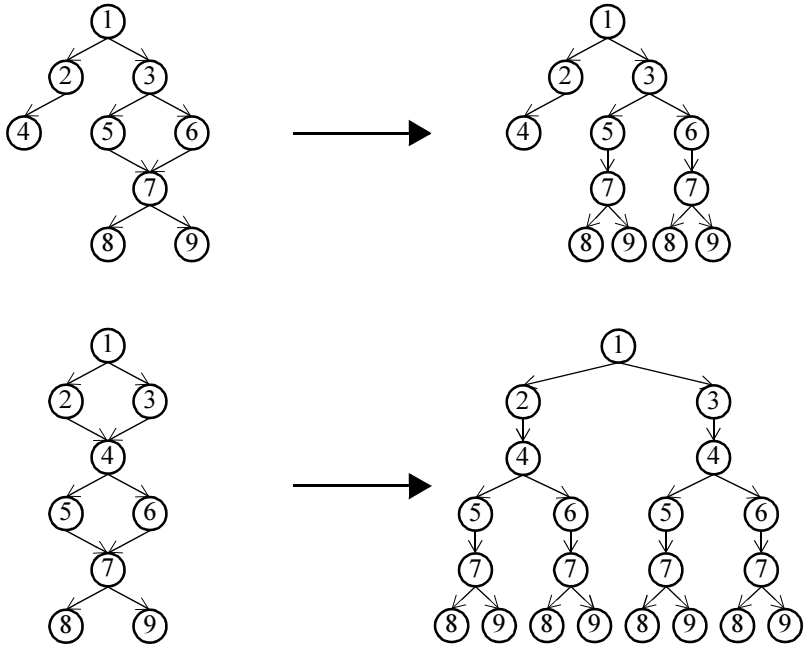


Fig. 10.2 Unfolding graphs into trees

Example 10-4. Tree-search problem: The 0/1 integer-linear-programming problem

Let us consider an instance of the 0/1 integer-linear-programming defined in example 10-1. Each elements of vector x can take the value 0 or 1, i.e. there are 2^m possible vectors. However, many of these values do not satisfy the problem's constraints corresponding to equation (10-1). The problem can be formulated as a tree-search problem. The initial node represents the state in which none of the elements of vector x have been assigned values. The initial node generates two nodes corresponding to $x_1 = 0$ and $x_1 = 1$. After a variable x_i has been assigned a value, it is called a *fixed variable*. All variables that are not fixed are called *free variables*. After instantiating a

[1] Other DOP solving techniques exist, for example: *polyhedral* description algorithms [Parker88], *linear relaxation* strategies, *cutting*, *branch-and-cut* [Naddef99][Naddef00]

variable to 0 or 1, it is possible to check whether an instantiation of the remaining free variables can lead to a feasible solution. If not, the exploration of the remaining tree hanging to the current node is aborted. Otherwise, if the node could lead to a feasible solution, the next variable is selected and assigned a value. This process continues until all the variables have been assigned and the feasible set has been generated. The cost function f from equation (10-2) is evaluated for each generated node, the solution is the one which minimize f. Figure 10.3 illustrates this process. ■

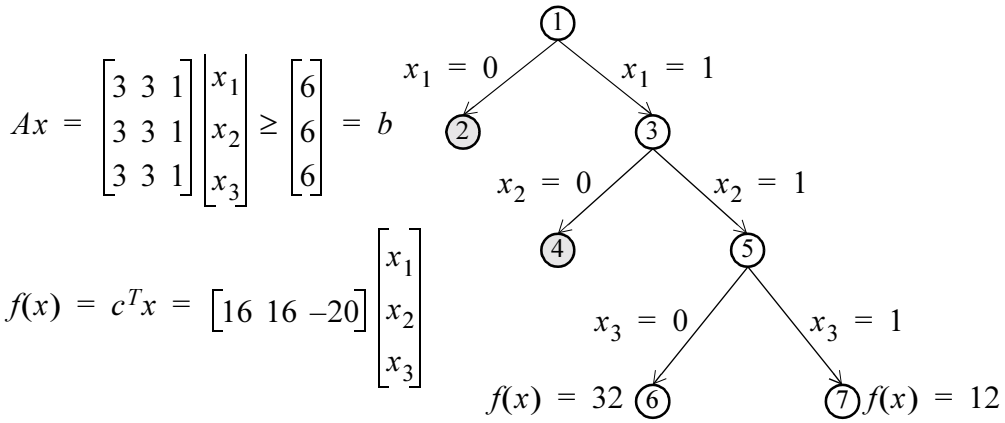


Fig. 10.3 Tree-search problem: The 0/1 integer-linear-programming
Nodes are numerated in exploration order, shaded nodes do not satisfy problem constraints, node 6 is a feasible solution but does not minimize the cost function, node 7 is feasible and minimizes f, i.e. it's the solution.

Example 10-5. Graph-search problem: The eight-puzzle

Considering the eight-puzzle problem as defined in example 10-2, this problem can be formulated as a graph-search problem. The initial node represents the initial configuration. The generated nodes correspond to the four (or less) possible moves. Each configuration appears only once in the graph. Each time the final configuration appears in the graph, a feasible solution has been found, i.e. a sequence of moves bringing from the initial to the final configuration. In such cases the cost function (the number of moves) is evaluated and the graph-search under the current node is terminated. The graph exploration continues until the whole feasible set has been generated. Figure10.4 illustrates the first steps of this process. ■

In both examples (10-4, 10-5) the cost function $f(x)$ is evaluated for each of the feasible solution in S; the solution with the minimum value is the desired solution. Note that it is unnecessary to generate the entire feasible set to determine the solution. As we will see, several search algorithms can determine an optimal solution by searching only a portion of the graph. The following presented search algorithms are the most commonly used. However, the list is non-exhaustive, also several algorithm combinations are possible.

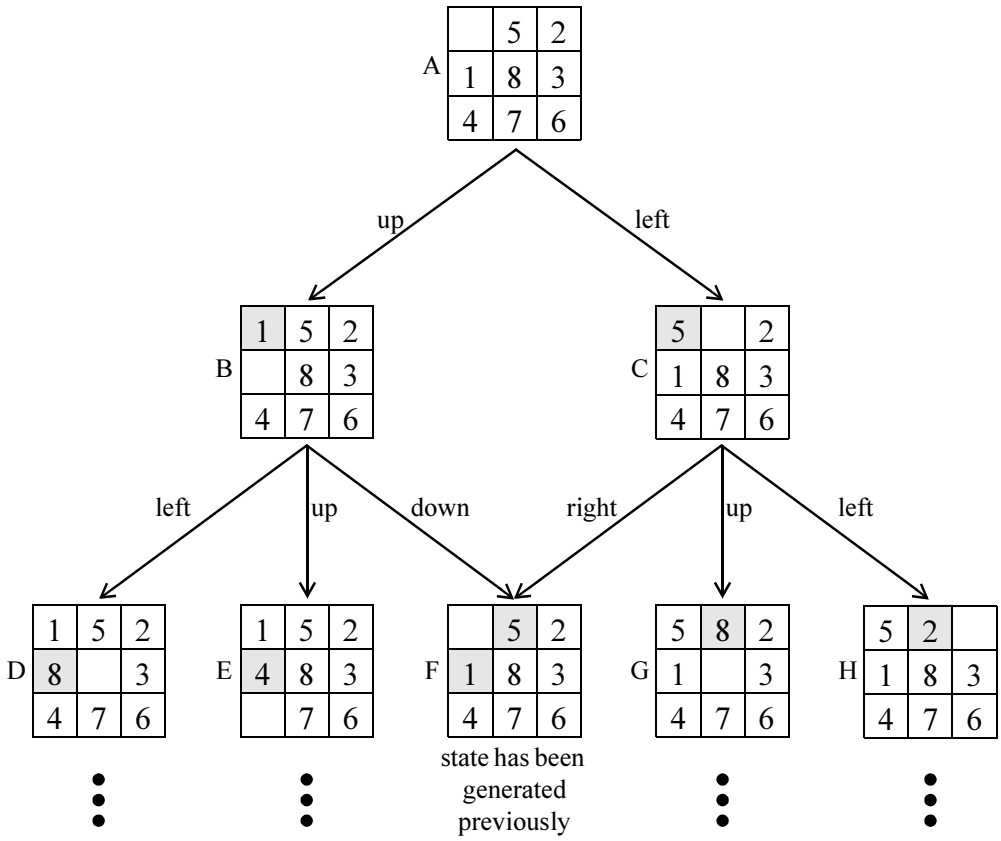


Fig. 10.4 Graph-search problem: The eight-puzzle

10.4.1. Depth-first search

Depth-first search (DFS) algorithms solve DOPs that can be formulated as tree-search problems. DFS begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), then DFS backtracks and expands a different node. We talk about *simple backtracking* if the search terminates upon finding the first solution. Thus, it is not guaranteed to find a minimum-cost solution. A variant, *ordered backtracking*, does use heuristics to order the successors of an expanded node.

The *iterative deepening depth-first search* (ID-DFS) consists of imposing a bound on the depth to which the DFS algorithms searches. If no solution is found, then the search process is repeated with a larger bound. ID-DFS does not guaranty to find a least-cost solution, but the one with the smallest number of edges. *Iterative deepening A** (IDA*) is a variant of ID-DFS. IDA* does not use a fixed depth bound. Instead, each node evaluates a cost-function, if this value is higher than a fixed cost, then the algorithm backtracks. IDA* performs cost-bounded DFS over the search space. Like the ID-DFS, if no solution is found, then the entire state space is searched again using a larger cost bound.

A major advantage of DFS is that its required memory size is linear to the depth of the state space being searched. Figure 10.5a illustrates the DFS search process.

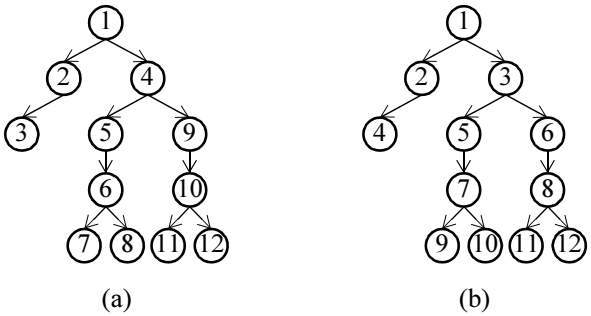


Fig. 10.5 Search order for (a) Depth-first search, (b) Breath-first search

10.4.2. Breath-first search

The *breath-first search* (BRFS) is a tree-search algorithm. BRFS begins by expanding the initial node, generating all nodes at a distance of one level from the root node. Next, all the nodes at a distance of two levels from the root will be examined, and so forth, until the entire tree is examined. The nodes are examined in order of increasing depth. This technique can be implemented via a queue. The BRFS algorithm explores the entire search space and locates the solution minimizing the cost function. The major disadvantage of BRFS algorithms is the large memory consumption. Figure 10.5b illustrates the BRFS algorithm.

10.4.3. Best-first search

Best-first search (BFS) algorithms can search both graphs and trees. These algorithms use heuristics to direct the search to portions of the search space likely to yield solutions. Heuristic evaluation functions assign smaller values to more promising nodes. The *A* algorithm* is one of the common BFS technique. The A* algorithm uses the lower bound function $l(x) = h(x) + g(x)$ (see Section 10.3) as a heuristic. BFS algorithms maintain an *open* list and a *closed* list. Nodes in the open list are ordered according to increasing value of the heuristic evaluation function. At the beginning, the initial node is placed on the open list. At each step, the node with the smallest heuristic value, i.e. the most promising node, is removed from the open list and placed in the close list. Its successors are inserted into the open list, unless there are already in the closed list. The BFS algorithm terminates once the goal node is reached. If the heuristic function is admissible (see Section 10.3), then the BFS algorithm finds an optimal solution.

The main drawback of any BFS algorithm is that its memory requirement is linear to the size of the search space explored, which for many problems, grows exponentially with the depth of the explored tree.

10.4.4. Branch and bound

Branch and bound (BB) algorithms exhaustively search through the state space, represented as a tree or a graph, seeking for an optimal solution. The BB technique can be used in combination with all the previously discussed search algorithms. Whenever it finds a new solution, it updates the current best solution. BB abort exploration of nodes whose extensions are guaranteed to be worse than the current best solution. Consider the eight-puzzle problem described in example 10-2, we assume the initial and final configurations showed in Figure 10.1, and suppose that the illustrated 8-moves solution has already been found. We consider a DFS algorithm exploring the state space of the eight-puzzle problem. Supposing that our search algorithm reaches the state C illustrated in Figure 10.4. Since the current node is directly connected to the root, the cost function $g(C)$, as defined in Section 10.3, equals 1. The Manhantan distance from the current node to the final configuration is given by $h(C) = 9$. The evaluation of the admissible Manhantan heuristic $l(C)$ gives $l(C) = g(C) + h(C) = 10$. Therefore, the current node C will not lead to a better solution than the current best 8-moves solution. Therefore, the BB search algorithm skips the exploration of the nodes below the node C.

A variant of the BB algorithm is the *alpha-beta minimax search* (ABMS). The ABMS algorithm is used by several game-playing programs to search for the best move to make next. Instead of using one bound value to prune some part of the state search space, ABMS algorithms use two threshold values: alpha and beta, each corresponding to one player. Like in BB algorithms, the alpha-beta strategy use these two values to avoid the exploration of sub-graphs whose evaluation cannot influence the outcome of the search, see [Naddef00].

BB algorithms, like linear programming, and other techniques could be considered as a partial enumeration strategy. These strategies may reduce considerably the size of the search state space. They are especially well-adapted to problems for which a lower bound function exists and is easily computable. The design of suitable bounding values is at the heart of most BB research. Linear relaxations is one of the common techniques to provide an obvious source of bounds [Parker88].

10.4.5. Generic sequential search

Figure 10.6 lists a generic sequential search process [Roosta00]. It involves two lists: *open* and *closed*. The open list contains nodes to be examined, whereas the closed list has those nodes that are already examined. Line 4 selects a single node from the open list. Line 5 generates successors of the selected node. Line 6 removes from the successor list those nodes that are either on the open or closed list. Line 7 merges two lists to form a new open list. Line 8 forms a new closed list. This algorithm is generic in that it uses no particular search strategy. A particular search strategy can be embedded by modifying the merge step at line 7. For example, DFS can be readily implemented by inserting the successors in front of the open list. BRFS can be realized by inserting the successors at the end of the open list. The BFS search strategy can also be easily implemented by inserting the successors into the open list in ascending order of heuristic values.

```
1.  open = initial_state
2. closed = ∅
3. Repeat
4.    selected_node ← select(open)
5.    succ ← expand(selected_node)
6.     succ ← filter(succ, open, closed)
7.    open ← merge(succ, open)
8.    closed ← merge(selected_node, closed)
9. Until(goal_state ∈ succ) or (open = ∅)
```

Fig. 10.6 Generic sequential search algorithm

10.5. Parallel search

Parallel search algorithms incur overhead from several sources: communication overhead, idle time due to load imbalance, and contention for shared data structures. Thus, if both the sequential and parallel formulations of an algorithm do the same amount of work, the speedup of parallel search on p processors is less than p. However, the amount of work done by a parallel formulation is often different from that done by the corresponding sequential formulation because they may explore different parts of the search space. In this case, some *speedup anomalies* could occur [Lai83]. Some executions of parallel versions could find a solution after generating fewer nodes than the sequential version, making it possible to obtain a superlinear speedup (*acceleration anomalies*). Other executions let the parallel versions finding a solution after generating more nodes, resulting in sublinear speedup (*deceleration anomalies*). If the speedup is superlinear, i.e. if the speedup is greater than p using p processors, it indicates that the serial search algorithm is not the fastest algorithm for solving the problem.

The critical issue of parallel search algorithms is the distribution of the search space among the processors. By statically assigning a node in the tree to a processor, it is possible to expand the whole subtree rooted at that node without communicating with another processor. Potentially, it seems that such a static allocation yields a good parallel search algorithm. Unfortunately, static partitioning of unstructured trees yields poor performance because of substantial variation in the size of partitions of the search space rooted at different nodes. Furthermore, since the search space is usually generated dynamically, it is difficult to get a good estimate of the size the search space beforehand. Therefore, it is necessary to balance the search space among processors dynamically, i.e. doing *dynamic load balancing*.

At run time, when a processor (the *recipient* processor) runs out of work, it should get more work from another processor (the *donor* processor) that has work. Several dynamic load balancing schemes are described in the literature, the difference between them is the manner how the donor processor is chosen. It could be chosen by a master (*global round robin*) or in a distributed manner (*asynchronous round robin*). It could also be chosen in a probabilistic way (*random polling*). A detailed dynamic load balancing scheme for parallel Branch and Bound algorithm is presented in [Lüling92].

In order to distribute the load, the donor processor splits his work and transmits some nodes to the recipient processor which runs out of work. How many and which graph nodes should be transmitted? If too little work is sent, the recipient quickly becomes idle; if too much, the donor becomes idle. Ideally, the total work is split into equal pieces such that the size of the search space is the same on the two processors (*half-split* strategy). It is difficult to get a good estimate of the size of the tree rooted at an unexpanded alternative. However, deep nodes tend to have smaller trees rooted at them than nodes close to the root. Three splitting strategies can be chosen: (1) send nodes near from the root, (2) send nodes near to the leaf and (3) send intermediate nodes. The nature of the search space (uniform, irregular, with a good heuristic, etc.) determines which strategy is more suitable [Kumar94].

Once a general dynamic load balancing system is installed, simple search algorithms, like DFS or BRFS, can be readily implemented. The more sophisticated search algorithm that need to maintain a global status, like BFS or BB, need to exchange several data structures at run time. The realization details of these data exchanges are dependent on both the target parallel system (shared/distributed memory, low/high latency network) and on the parallel framework (low level message passing system like MPI or higher level parallel scheduling language like CAP) used to develop the application. The most common referenced data exchange techniques are: the centralized strategy (master-slave), the random communication strategy, the ring communication strategy and the blackboard communication strategy, see [Parker88].

10.6. Travelling Salesman Problem: A didactical solution

The present section has been developed for teaching purposes. It has its place both as an advanced tutorial for the CAP framework or as the starting point for developing a more efficient TSP solution. Because of the speedup anomalies (see previous section), the performances are difficult to measure and are out of the scope of this tutorial.

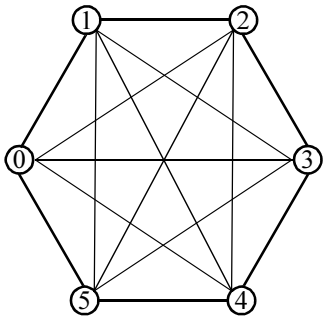


Fig. 10.7 TSP example with 6 cities: the travel costs are defined as the Euclidian distance between two cities. The bold weighted cycle is the solution.

The Traveling Salesman Problem (TSP) is easy to formulate: given a finite number of *cities* together with the *cost* of travel between each pair of them, find the cheapest way of visiting all the cities and returning to the starting point. We consider here only the case of the symmetric

TSP, where the travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X. We assume also that the traveling costs are integers. The solution of the TSP gives the order in which the cities must be visited to minimize the traveling costs. The simplicity of this problem, coupled with its apparent intractability, makes it an ideal platform for exploring new algorithmic ideas. Figure 10.7 illustrates a TSP instance.

The origin of the TSP are obscure. The mathematician and economist Karl Menger published the TSP problem in 1920. Several mathematician and statisticians popularized it until 1950. A breakthrough came with linear programming and Dantzig's *simplex* algorithm [Applegate98]. The TSP gained notoriety as the prototype of hard problem in combinatorial optimization. Indeed, the TSP belongs to the NP-Hard problems. It remains NP-Hard even if the travel costs are limited to the integers 0 and 1 - the TSP is NP-Hard in the *strong sense* [Parker88]. Nowadays, a lot of work has been done and published from several domains to solve the TSP: heuristic approaches, genetic algorithms, neural networks, tabu searches, approximation algorithms, Branch-and-Bound algorithms, Branch-and-Cut algorithms, and many more. Various algorithms and softwares are available in the public domain [Chvátal00] [Concorde00] [TSPBIB00] [TSPLIB00]. Some work has also been done for parallelizing the TSP, in particular, Stefan Tschöke and its colleagues published a remarkable paper describing the parallelization of the TSP on a 1024 processor network using a BB algorithm [Tschöke95].

```
1.  parameter int NbSlaveProcess = 4;
2.  process MasterProcessT {
3.  operations :
4.  };
5.  process SlaveProcessT {
6.  operations :
7.     SlaveRun in tspJobT* inputP out tspJobT* outputP;
8.  };
9.  process GlobalProcessT {
10. subprocesses :
11.    MasterProcessT MasterProcess;
12.    SlaveProcessT SlaveProcess[NbSlaveProcess];
13. operations :
14.    GlobalRun in void* inputP out void* outputP;
15. };
```

Fig. 10.8 CAP process hierarchy for the TSP

In this section, we present a parallel BFS-BB algorithm solving the TSP. In order to facilitate the development of the parallel application, we implemented it with the CAP language. We choose a master-slave strategy to achieve dynamic load balancing and data structure exchange. The slaves perform each a sequential BB algorithm on a part of the search space. The master distributes jobs to the slaves according to a BFS strategy. The master is responsible for maintaining a job list. The jobs are split into smaller subjobs and inserted in the list according to the BFS strategy. The master sends first the most promising jobs to the slaves. The slaves work on a job during a given amount of time. The chosen amount of time determines the parallelization grain. When the time is elapsed the job is returned to the master. The returned jobs are either

terminated or still active. In the last case they are inserted again in the job list. The program terminates when the job list is empty and all the jobs returned.

10.6.1. Process hierarchy

In order to implement the master-slave distribution strategy, we need one master process and several slave processes. Figure 10.8 shows such a CAP process hierarchy. The *GlobalProcessT* contains one master subprocess (line 11) and *NbSlaveProcess* slave subprocesses (line 12). Since *NbSlaveProcess* is a CAP parameter, it can be changed via the command line arguments (no compilation needed). *MasterProcessT* and *SlaveProcessT* are both system processes, *GlobalProcessT* is a higher-level CAP process abstraction; an abstraction that has no associated system process. The *GlobalRun* (line 14) operation in *GlobalProcessT* is the high-level CAP operation controlling the data flow. The *SlaveRun* (line 7) leaf operation in *SlaveProcessT* is the sequential C++ function implementing the BB algorithm. The *MasterProcessT* is the master execution thread, it does not contain any operation, it is responsible for executing the *Split* and *Merge* functions as we will see it later.

```
1.  configuration {
2.  processes :
3.     M  ("user");
4.     S0 ("128.178.75.10", "tsp.exe");
5.     S1 ("128.178.75.11", "tsp.exe");
6.     S2 ("128.178.75.12", "tsp.exe");
7.     S3 ("128.178.75.13", "tsp.exe");
8.  threads :
9.     "MasterProcess"   (M) ;
10.    "SlaveProcess[0]" (S0);
11.    "SlaveProcess[1]" (S1);
12.    "SlaveProcess[2]” (S2);
13.    "SlaveProcess[3]” (S3);
14. };
```

Fig. 10.9 CAP configuration file for the TSP: 1 master and 4 single-processor slave computers

```
1.  configuration {
2.  processes :
3.     M  ("user");
4.     S0 ("128.178.75.10", "tsp.exe");
5.     S1 ("128.178.75.11", "tsp.exe");
6.  threads :
7.     "MasterProcess"   (M) ;
8.     "SlaveProcess[0]" (S0);
9.     "SlaveProcess[1]" (S0);
10.    "SlaveProcess[2]" (S1);
11.    "SlaveProcess[3]" (S1);
12. };
```

Fig. 10.10 CAP configuration file for the TSP: 1 master and 2 bi-processor slave computers

This process hierarchy can be used with several CAP configuration files. Figure 10.9 shows an example with 5 single processor computers: 1 master and 4 slaves. If the slaves are bi-processor computers, then the example described in the configuration file of Figure 10.10 is appropriate.

10.6.2. Sequential part of the algorithm

The application is based on the master-slave paradigm: the master sends a token data structure to the slave, the slave performs some work on it, and sends it back to the master. Before describing the parallel aspect of the application, we first need to explain the individual work done by each slave. The token *tspJobT* exchanged between the master and the slaves is presented by Figure 10.11. The *tab* member is a fixed size integer array containing the subpath of visited cities. The *tripSize* member is the size of this subpath. Initially, *tab* contains only the city index 0.

```
1.  token tspJobT
2.  {
3.     int tab[MAX_SIZE];         // contains the actual subpath
4.     int swapTab[MAX_SIZE];     // internal usage, used for backtracking
5.     int backtrack;             // boolean value, set to TRUE if backtrack needed
6.     int tripSize;              // number of cities in the actual subpath
7.     int lowerBound;            // lower bound for the actual subpath
8.     int upperBound;            // best found upper bound
9.     int jobLowerBound;         // lower bound of the job
10.    int minTripSize;           // backtracking limit
11. };
```

Fig. 10.11 Description of the token data structure exchanged between the master and slaves

Then, the algorithm performs a DFS, i.e. when the DFS goes down in the search tree, a new city is appended to *tab* and when the algorithm backtracks, the last city is removed. The search is performed by two functions: *tspGetNextSubPath* and *tspbacktrack*. The first goes down in the search tree adding a new city to *tab*, the second backtracks by removing the last city from *tab*. In order to explore a new subpath when backtracking occurs, we use the *swapTab* integer array. This array contains indexes of cities chosen after backtracking. Figure 10.12 explains how the exploration of the search tree with four cities is performed. For example to reach state F from the initial state A, we need to call *tspGetNextSubPath* to go from A to B, then *tspGetNextSubPath* to go from B to C, then again *tspGetNextSubPath* to go from C to D, then *tspbacktrack* to backtrack from D to B (*tspbacktrack* jumps over state C since no unexplored subpaths are rooted at this point), then *tspGetNextSubPath* to go from B to E, and finally *tspGetNextSubPath* to go from E to the final state F.

The *lowerBound* (Fig. 10.11, line 7) member is the length (travel cost) of the current subpath. The *upperBound* is the shortest path going through all the cities (*Hamiltonian cycle*) found at this time. The *upperBound* is compared to the *lowerBound* within the scope of the BB algorithm. The *backtrack* boolean value determines if we need to backtrack or not. It is set to *TRUE* for backtracking, and to *FALSE* otherwise.

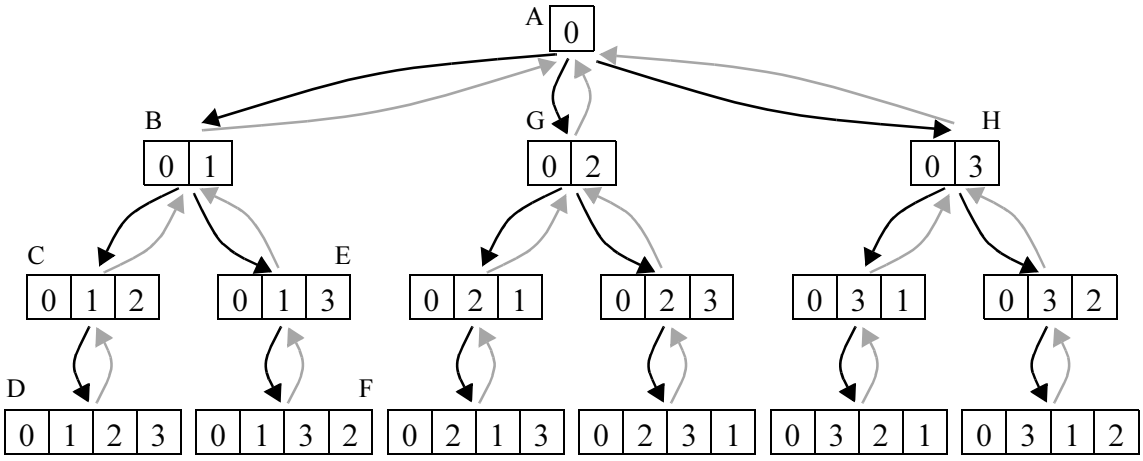


Fig. 10.12 DFS exploration of the TSP search tree with four cities
The black lines correspond to a call to *tspGetNextSubPath*, the light gray correspond to a *tspbacktrack* call

Looking again at Figure 10.12, we known that the whole search tree has been explored when the *tspbacktrack* function comes back to state A and that all the subpaths rooted there have been explored. During the whole search, the first city stays at the first position in *tab*. If we want to perform such a search we set the *minTripSize* member to 1 . Since we are interested in parallel processing, we need to be able to split a *tspJobT*. We could, for example, split the job starting at initial state A in three *tspJobT* starting respectively at states B, G and H. The two first cities of these jobs will never change during the search. For these jobs, *minTripSize* is set to 2 . The *minTripSize* member is set when a *tspJobT* is split; it represents the backtracking limit. The *jobLowerBound* member is the lower Bound of the job, i.e. the travel cost from the first city to the city at position *minTripSize* in *tab*.

Figure 10.13 lists the sequential CAP leaf operation performed on each slave. The *SlaveRun* leaf operation receives a *tspJobT* as input token. A Depth-First Search Branch and Bound (DFS-BB) algorithm is applied to this token, then the token is returned to the master. Lines 7-9 manage the case of an invalid *tspJobT*. Such an invalid job is send by the master when no more valid jobs are available. At line 10 begins the main loop performing the DFS-BB algorithm. The condition of this loop limits the computation time of a job. When the *count* variable exceeds the *ComputationSteps* CAP parameter, the loops ends and the *tspJobT* token returns to the master. The slave processes then another job previously received from the master. This computation limitation avoids the slave to spend too much time on exploring unpromising subtrees. Lines 11-19 implements a DFS algorithm calling the *tspGetNextSubPath* and *tspbacktrack* functions (Fig. 10.12). Lines 21-24 lists the BB part of the algorithm. When the *lowerBound* exceeds the best known *upperBound* backtracking occurs, since the current node will not lead to a better solution. Lines 25-28 treat the case where a solution better than the currently best known solution is found. In such a situation, the loop ends, and the new solution is immediately send to the master. The last lines check if the current path is *intersectionless* [Applegate99][DeCegama89].

```
1.  parameter int ComputationSteps = 200000;
2.  parameter int NumberOfCities = 20;
3.  leaf operation SlaveProcessT::SlaveRun in tspJobT* inputP out tspJobT* outputP
4.  {
5.     int count = 0;
6.     outputP = new tspJobT(*inputP);
7.     if(outputP->tripSize < 0) {         // invalid job
8.        return;
9.     }
10.    while(++count < ComputationSteps) {
11.       if(outputP->backtrack || outputP->tripSize > NumberOfCities) {
12.          tspbacktrack(outputP);        // DFS search (up)
13.          if(outputP->tripSize < outputP->minTripSize) {
14.             break;                     // job is terminated
15.          }
16.          outputP->backtrack = FALSE;
17.          continue;
18.       }
19.       tspGetNextSubPath(outputP);      // DFS search (down)
20.       // analysis of new subpath
21.       if(outputP->lowerBound > outputP->upperBound) {
22.          outputP->backtrack = TRUE;    // no better solution found here
23.          continue;
24.       }
25.       if(outputP->tripSize > NumberOfCities) {// if round trip completed:
26.          outputP->upperBound = outputP->lowerBound; // better sol: upperBound updated
27.          break;
28.       }
29.       if(!tspIsAcceptable(outputP)) {
30.          outputP->backtrack = TRUE;
31.          continue;                     // subpath is not intersectionless, backtrack
32.       }
33.    }
34. }
```

Fig. 10.13 Sequential CAP leaf operation performed on each slave (DFS-BB algorithm)

If the path contains an *intersecting pair*, then the subsequent exploration will not lead to the optimal solution. In such a case the algorithm backtracks.

10.6.3. Parallel part of the algorithm

The parallel algorithm implements a Best-First Search (BFS) technique (Fig. 10.14). An open list (see Section 10.4.3) is used to order the *tspJobPs* according to their *jobLowerBound*. The jobs are sent to slaves using a dynamic load balancing scheme. The best found job (*BestTspJobP*) is continuously updated and stored to allow the slave to perform a BB algorithm.

GlobalRun is the global CAP parallel operation handling the data flow. The *indexed parallel* statement generates new tokens (by calling the *Split* function), and sends them to the slave processes. The token generation stops when they are no jobs in the job list, and no currently running jobs. Once, the *indexed parallel* exits, the application terminates. Line 32 guaranties that the *Split* function is executed by the *MasterProcess*. The *flow_control* statement limits the number of currently sent jobs to *PipelineDepth*NbSlaveProcess*. When the pipeline reaches the steady

```
1.  parameter int CutOffDepth = 5;
2.  parameter int PipelineDepth = 2;
3.  int RunningJobs = 0;
4.  void Split(capTokenT* inputP, tspJobT*& outputP, int)
5.  {
6.     if(!(outputP = tspOrderedJobListRemove())) {// take most promising job
7.        outputP = new tspJobT();         // job list is empty: send an invalid job
8.        outputP->tripSize = -1;
9.        return;
10.    }
11.    if(outputP->tripSize > outputP->minTripSize &&       // if the job is not new and
12.       outputP->minTripSize < NumberOfCities - CutOffDepth) { // not too small then it is
13.       tspOrderedJobListExpand(outputP);                 // splitted into smaller jobs
14.    }                                                    // and inserted in the list
15.    outputP->upperBound = BestTspJobP->upperBound;
16.    RunningJobs++;
17. }
18. void Merge(capTokenT* outputP, tspJobT* inputP, int)
19. {  // returned jobs may be terminated, provide a better solution or remain active
20.    if(inputP->tripSize < 0) {
21.       return;                          // receiving an invalid job
22.    }
23.    RunningJobs--;
24.    if(inputP->tripSize < inputP->minTripSize) {
25.       return;                          // this job is terminated
26.    }
27.    tspUpdateBestJob(inputP);                   // job still active: update BestJob
28.    tspOrderedJobListInsert(new tspJobT(*inputP)); // and insert it in the job list
29. }
30. operation GlobalProcessT::GlobalRun in void* inputP out void* outputP
31. {
32.    MasterProcess.{} >->    // ensure that MasterProcess executes the split fct
33.    flow_control(PipelineDepth*NbSlaveProcess) // PipelineDepth jobs per slave
34.    indexed
35.    ( ; tspOrderedJobListSize() + RunningJobs > 0; )
36.    parallel (Split, Merge, MasterProcess, void result)
37.    (
38.       SlaveProcess [cap_fcindex0%NbSlaveProcess].SlaveRun// dynamic load balancing
39.    );
40. }
```

Fig. 10.14 Parallel CAP operation solving the TSP (BFS algorithm)

state, each slave job queue should contain *PipelineDepth* jobs. The *PipelineDepth* CAP parameter must be set at least to two. This ensures that slaves are never idle waiting for receiving a new job. If *PipelineDepth* is set to a value larger than 2, then the termination latency could increase due to a possible higher imbalance between slaves. The CAP *cap_fcindex0* keyword ensures dynamic load balancing, *cap_fcindex0%NbSlaveProcess* represents the index of the slave, which returned the most recently terminated job.

The *Split* function gets the jobs from the job list (line 6). If the job list is empty, then an invalid job is sent (lines 7-9). The successors of the considered job are generated and inserted in the job list (line 13). The successors are not generated if they are too close to the leaf (*CutOff* CAP parameter) of the search tree. In order to avoid to insert in the list unpromising nodes, successors of jobs that have never been processed by a slave are also not generated (line 11). If the current

job is valid, it's *upperBound* is updated to the currently best found upper bound and the *RunningJobs* variable is increased.

The *Merge* function inserts the received job in the job list and updates, if necessary, the best found job. Invalid or terminated jobs (lines 20-22 and lines 24-26) are ignored. The *RunningJobs* variable is decreased if the received job is valid.

10.7. Summary

The parallel processing of discrete optimization problems is a large domain of research. Several theoretical results and applications are available. In this chapter, we give a didactical presentation of its central aspects. We develop the most common used search algorithm and several techniques to considerably reduce the huge search space. We implement a didactical CAP program solving the famous traveling salesman problem using a hybrid search algorithm. This program features dynamic load balancing, which is one of the critical issues when parallelizing discrete optimization problems. CAP enables to express data structures, sequential functions, and parallel scheduling in a clearly separated manner. This modularity lets the programmer concentrates himself on the main (sequential and parallel) algorithm strategies without being distract by low-level implementation details.

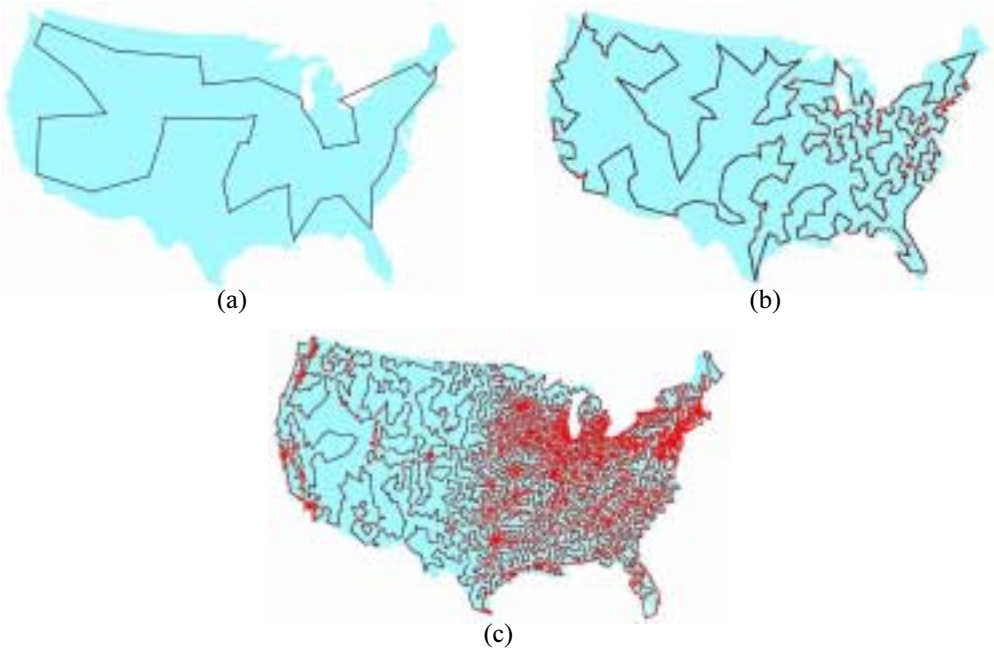


Fig. 10.15 TSP solutions for cities in the U.S.A., the cost of travel between two cities is defined by the road distance (a) 49 cities: 1 city from each of the 48 states and Washington D.C. (Selmer Johnson, 1954) (b) 532 AT&T switch locations in the USA (Padberg and Rinaldi 1987) (c) 13,509 cities in the USA with populations > 500 (Applegate, Bixby, Chvatal, and Cook 1998)

11 Conclusion

Various approaches have been explored in order to facilitate the development of parallel applications. In the context of this thesis, we explore an approach based on the explicit specification of parallel schedules. For this purpose, we use the CAP framework which allows to specify at a high level of abstraction the flow of data and parameters between operations running on the same or on different processors. Within the CAP language, the programmer specifies explicitly the parallel behavior of the application as a parallel schedule. The CAP C++ language extension execution model is based on a coarse grain decomposition. The CAP approach has proved to work on clusters between 10 to 50 PCs interconnected with a Fast Ethernet network.

In the first chapters we have introduced the CAP approach. We began by presenting the fundamentals and the underlying philosophy of the CAP language. We also presented technical programming aspects and advanced topics such as load balancing and serialization. In the next chapters, we presented several parallel applications developed within the CAP framework: parallel linear algebra computations, parallel image filtering, parallel Branch and Bound algorithms, dynamic load balancing of cellular automata and parallel computation combined with asynchronous data access. In all these applications, we demonstrate the ability of CAP to produce efficient parallel solutions.

The CAP framework is the result of a large collaborative effort between several members of the Peripheral Systems Laboratory (LSP) of the EPFL. In particular, the CAP language was created by Benoît Gennart, the parallel file striping and the message passing system have been developed by Vincent Messerli. The contribution of this thesis was (1) to contribute to the evolution of the CAP system by improving it with load balancing and serialization features; and (2) to validate the CAP approach through the development of several parallel applications.

The applications developed during this thesis demonstrate the ability of CAP to express custom parallel schedules. The presented performance results indicate that the developed applications have been successfully parallelized yielding very efficient programs making use of most of the underlying hardware capabilities. With the *Radiocontrol* application developed for industrial purposes, we demonstrated that existing programs can be relatively easily parallelized and that solutions can be found for graceful degradation in case of failure. We conclude that the CAP approach is valid and well adapted for developing efficient parallel cluster computing applications.

Future research efforts should aim at overcoming the current limitations of the CAP framework. In particular, CAP does not allow to create parallel schedules dynamically. All schedules are defined at compilation time. The next generation CAP framework will be based on dynamic parallel components. Each component will offer a set of parallel services. For example, one

component could be a parallel file system component, another a parallel image processing component, etc. We should offer the possibility of creating dynamically new schedules using the previously defined parallel components. For example, a new schedule could perform an out-of-core parallel image processing algorithm, using the parallel file system and parallel image processing components. This new schedule can in turn become itself a component. The next generation framework will offer a flexible solution for building efficient parallel component based services.

Within this thesis, we validate the schedule based approach for developing parallel applications. We demonstrate that the formulation of parallel schedules independently from the serial part of the program facilitates the programmer's work and offers valuable compositional aspects. Finally, the presented performances demonstrate the efficiency of the schedule based approach.

Bibliography

[Agerwala82] T. Agerwala, Arvind, “Data Flow Systems: Guest Editors’s Introduction,” *IEEE Computer*, February 1982, Vol. 15, No. 2, 10-13

[Amdahl67] G. S. Amdahl, “Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities”, *Proc. AFIPS*, Conf. 30, AFIPS Press, Reston Va., 1967, 483-485

[Amdahl88] G. S. Amdahl, “Limits of Expectations”, *The International Journal of Supercomputer Applications*, Vol. 2, No. 1, 1988, 88-94

[Anderson95] E. Anderson, *LAPack User’s Guide*, 2nd edition, Society for Industrial and Applied Mathematics, Philadelphia, 1995

[Applegate98] D. Applegate, R. Bixby, V. Chvátal, W. Cook, “On the solution of Travelling Salesman Problems”, *Documenta Mathematica*, Vol. 3, Extra volume ICM, 1998, 645-656

[Applegate99] D. Applegate, R. Bixby, V. Chvátal, W. Cook, “Finding tours in the TSP”, Tech. Rep. TR99-05, Departement of Computational and Applied Mathematics, Rice University, 1999

[Beguelin90] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. S. Sunderam, “A User’s Guide to PVM Parallel Virtual Machine,” *Oak Ridge National Laboratory Report ORNL/TM-11826*, July 1990

[Beguelin92] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. S. Sunderam, “HeNCE: Graphical Development Tools for Network-Based Concurrent Computing”, *Proc. of SHPCC-92*, IEEE Computer Society Press, Los Alamitos, California, 1992, 126-136

[Bichsel98] M. Bichsel, “Method for the compression of recordings of ambient noise, method for the detection of program elements therein, and device therefor”, *European Patent Application EP 0 598 682 A1*, applicant LIECHTI AG, issued 30.12.1998

[BSD93] "Computing Science and Systems: The UNIX System", *AT&T Bell Laboratories Technical Journal 63*, October 1984, No. 6 Part 2, 1577-1593, see also http://www.bsd.org

[Buyya99] I. Buyya, Rajkumar, *High Performance Cluster Computing,* Prenttice Hall, 1999

[Carriero89a] N. Carriero, D. Gelernter, “Linda in Context,” *Communications of the ACM*, April 1989, Vol. 32, No. 4, 444-458

[Carriero89b] N. Carriero, D. Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed,” *Journal of the ACM*, September 1989, Vol. 21, No. 3, 323-357

[Chvátal00] V. Chvátal, http://www.cs.rutgers.edu/~chvatal/tsp.html

[Cohen98] A. Cohen, M. Woodring, *Win32 Multithreaded Programming*, O’Reilly & Associates, January 1998

[Concorde00] D. Applegate, http://www.keck.caam.rice.edu/concorde.html

[Cornell97] G. Cornell, C. S. Horstmann, *Core Java*, SunSoft Press, 1997

[Cosnard95] M. Cosnard, D. Trystram, *Parallel Algorithms and Architectures*, Int. Thomson Computer Press, 1995

[Crichlow97] J. M. Crichlow, *An Introduction to Distributed and Parallel Computing*, Prentice Hall, 1997

[Croes58] G. Croes, “A method for solving Traveling-Salesman Problems”, *Operations Research 5*, 1958, 791-812

[DeCegama89] A. L. DeCegama, *The Technology of Parallel Processing*, Prentice Hall, 1989

[Denning86] P. J. Denning, “Parallel Computing and its Evolution”, *Communications of the ACM*, Dec. 1986, Vol. 29, No. 12, 1163-1167

[Dennis75] J. Dennis, “First Version of a Data Flow Procedure Language,” *MIT Technical Report TR-673*, MIT, Cambridge, Massachusetts, USA, May 1975

[Dongarra96] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker, “A message passing standard for MPP and workstations”, *Communications of the ACM*, 1996, Vol. 39, No. 7, 84-90

[Equitz95] W. Equitz, “Image searching in a shipping product”, *Proc. Conf. IS&T/SPIE-Storage and Retrieval for Image And Video Databases III*, California, February 1995, SPIE Vol. 2420, 186-196

[Foster94] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1994

[Galletly96] J. Galletly, *Occam-2*, UCL Press, 1996

[Gennart98a] B. A. Gennart, “The CAP Computer-Aided Paralllelization Tool: Language Reference Manual”, EPFL internal report, July 1998

[Ghezzi82] C. Ghezzi, M. Jazayeri, *Programming Language Concepts*, John Wiley, 1982

[Ghezzi85] C. Ghezzi, “Concurrency in Programming Languages: A Survey”, *Parallel Computing*, November 1985, Vol. 2, 229-241

[Golub96] G. H. Golub, C. F. Van Loan, *Matrix computations*, Johns Hopkins University Press, Third edition, 1996

[Gray92] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1992

[Grimshaw93a] A. S. Grimshaw, W. T. Strayer; P. Narayan, "Dynamic Object-Oriented Parallel Processing", IEEE Parallel & Distributed Technology: Systems & Applications, May 1993, Vol. 1, No. 2 , 33 -47

[Grimshaw93b] A. S. Grimshaw, "Easy-to-Use Object-Oriented Parallel Processing with Mentat", *IEEE Computer*, May 1993, Vol. 26, No. 5, 39-51

[Gustafson88] J. L. Gustafson, "Reevaluating Amdahl's Law", *CACM*, May 1988, 532-533

[Halstead85] Robert H. Halstead Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, Oct. 1985, Vol. 7, No. 4, 501-538

[Hansen75] P. B. Hansen, "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering*, June 1975, Vol. SE-1, No. 2, 199-207

[Hatcher91] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, R. R. Jones, "Data-parallel programming on MIMD computers", *IEEE Transaction on Parallel and Distributed Systems*, 1991, Vol. 2, No. 3, 377-383

[Hersch93] R. D. Hersch, "Parallel Storage and Retrieval of Pixmap Images", *12th IEEE Symposium on Mass Storage Systems*, Montherey CA, Digest of papers, IEEE Computer Society Press, April 1993, 221-226

[Hersch00] R. D. Hersch, B. Gennart, O. Figueiredo, M. Mazzariol, J. Tarraga, S. Vetsch, V. Messerli, R. Welz, L. Bidaut, "The Visible Human Slice Web Server: A first Assessment", *Proc. IS&T/SPIE Conference on Internet Imaging*, San Jose CA, Jan. 2000, SPIE Vol. 3964, 253-258

[IBW99] http://www.ibwag.com/

[IMT99] http://www-imt.unine.ch/Radiocontrol/

[Inmos85] Inmos Limited, *Occam Programming Manual*, 1985

[Jain89] A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1989

[Koelbel94] C. H. Koelbel, D. B. Loveman, R.S. Schreiber, Jr. G. L. Steele, M. E. Zosel, *The High Performance Fortran Handbook*, Scientific and Engineering Computations, MIT Press, Cambridge, MA, 1994

[Kumar94] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*, Benjamin/ Cummings, 1994

[Lai83] T.-H. Lai, S. Sahni, “Anomalies in parallel branch-and-bound algorithms”, *IEEE Parallel Processing Proc.*, August 1983, 183-190

[Lee80] R. B. Lee, “Empirical results on the speed, efficiency, redundancy and quality of parallel compuations”, *Proc. International Conference on Parallel Processing*, 1980, 91-100

[Leiss95] E. L. Leiss, *Parallel and Vector Computing*, MCGraw-Hill, 1995

[Lewis92] T. G. Lewis, H. El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, 1992

[Liechti99] http://www.innofair.ch/technologiestandort/sp_d/99/pages/cebit_5.html

[Loveman93] D. B. Loveman, “High Performance Fortan,” *IEEE Parallel & Distributed Technology*, February 1993, Vol. 1, No. 1, 25-42

[Luckham93] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, F. Belz, “Partial orderings of events sets and their application to prototyping concurrent, timed systems”, *Journal of Systems and Software*, 21(3), June 1993, 253-265

[Lüling92] R. Lüling, B. Monien, “Load Balancing for Distributed Branch & Bound Algorithms”, *Proc. of the 6th International Parallel Processing Symposium*, IEEE Computer Society Press, 1992, 543-549

[Manzanera99] A. Manzanera, T. M. Bernard, F. Prêteux, B. Longuet, “Ultra-Fast Skeleton Based on an Isotropic Fully Parallel Algorithm”, *DGCI’99,* Springer-Verlag, LNCS 1568, 1999, 313-324

[Mazzariol97] Marc Mazzariol, Benoît A. Gennart, Vincent Messerli, Roger D. Hersch, “Performance of CAP-specified linear algebra algorithms”, *EuroPVM-MPI'97*, Springer-Verlag, LNCS 1332, 351-358

[Mazzariol98] B. A. Gennart, M. Mazzariol, V. Messerli, R.D. Hersch, “Synthesizing Parallel Imaging Applications using the CAP Computer-Aided Parallelization tool”, *IS&T/SPIE's Symposium on Electronic Imaging'98*, Conf. Storage & Retrieval for Image and Video Databases VI, SPIE Vol. 3312, 446-458

[Mazzariol00a] M. Mazzariol, B. Gennart, R.D. Hersch, “Dynamic load balancing of parallel cellular automata”, *Proc. SPIE Conference on Parallel and Distributed Methods for Image Processing IV*, San Diego, July 2000, SPIE Vol. 4118, 21-29

[Mazzariol00b] M. Mazzariol, B. Gennart, R.D. Hersch, M. Gomez, P. Balsiger, F. Pellandini, M. Leder, D. Wüthrich, J. Feitknecht, “Parallel Computation of Radio Listening Rates”, *Proc. SPIE Conference*, Parallel and Distributed Methods for Image Processing IV, San Diego, July 2000, SPIE Vol. 4118, 146-153

[Messerli99a] V. Messerli, *Tools for parallel I/O and compute intensive applications*, Ph.D. Thesis, EPFL, No. 1915, 1999

[Messerli99b] V. Messerli, O. Figueiredo, B. Gennart, R.D. Hersch, "Parallelizing I/O intensive Image Access and Processing Applications", *IEEE Concurrency*, April-June 1999, Vol. 7, No. 2, 28-37

[Microsoft96] Microsoft Corporation, *Windows Sockets 2 Application Program*, 1996

[Miller88] P. C. Miller, C. E. St. John, S. W. Hawkinson, "FPS T Series Parallel Processor," in Robert G. Babb II, editor, *Programming Parallel Processors*, Addison-Wesley, 1988

[MPI94] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard", *The Int. Journal of Supercomputer Applications and High Performance Computing*, Vol. 8, 1994, 159-416

[MPI97] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," *Technical Report*, July 1997, http://www.mpi-forum.org

[MSDN00] MSDN library, http://www.msdn.microsoft.com, July 2000

[Naddef99] D. Naddef, S. Thienel, "Efficient Separation Routines for the Symmetric Traveling Salesman Problem", working paper, http://www-apache.imag.fr/~naddef/

[Naddef00] D. Naddef, G. Rinaldi, "Branch-and-cut Algorithms", chapter of a forthcoming book on the Vehicle Routing Problem, http://www-apache.imag.fr/~naddef/

[Pachero97] P. S. Pachero, *Parallel Programming with MPI*, Morgan Kaufmann, 1997

[Parasoft90] ParaSoft Corporation, *Express C User's Guide (Version 3.0)*, 1990

[Parker88] R. G. Parker, R. L. Rardin, *Discrete Optimization*, Academic Press, 1988

[Patterson97] D. A. Patterson, J. L. Hennessy, *Computer organization and design: the hardware / software interface*, Morgan Kaufmann Publishers, 1997

[Quinn87] J. M. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987

[Rajkumar99a] B. Rajkumar, *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999

[Rajkumar99b] B. Rajkumar, *High Performance Cluster Computing*, Vol. 2, Prentice Hall, 1999

[Roosta00] S. H. Roosta, *Parallel Processing and Parallel Algorithms: Theory and Computation*, Springer-Verlag, 2000

[Schalkoff89] R. J. Schalkoff, *Digital Image Processing and Computer Vision*, Wiley, 1989

[Shu91] W. Shu, L. V. Kale, "Chare Kernel - A runtime support system for parallel computations", *Journal of Parallel Distributed Computing*, 1991, Vol. 11, 198-211

[Sipper97] M. Sipper, *Evolution of Parallel Cellular Machines*, Springer-Verlag, LNCS 1194, 1997

[Snir98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI-The Complete Reference*, MIT Press, 1998

[Sohn95] A. Sohn, J-L. Gaudiot, "Programmability and Performance Issues of Multiprocessors on Hard Nonnumeric Problems", *Advanced topics in dataflow computing and multithreading*, G. R. Gao, L. Bic, J-L. Gaudiot, IEEE Computer Society Press, 1995, 143-166

[Srini86] V. P. Srini, "An Architectural Comparison of Dataflow Systems," *IEEE Computer*, March 1986, Vol. 19, No. 3, 68-88

[SSR99] http://tora.sri.ch/gd/fr/home/fr_textarchiv_1299.html#hörer

[Stevens90] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1990

[Stevens94] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols,* Addison-Wesley, 1994

[Sun99] S. Sun, C. Calame, M. Gomez, P. Balsiger, F. Pellandini, D. Wuthrich, P. Mishcler, J. Feitknecht, "Very Low-Power Terminal for Media Control Applications", *International Workshop on Intelligent Communication Technologies*, Cost Workshop 254, Neuchatel, Switzerland, 1999, http://www-imt.unine.ch/cost/

[Sunderam90] V. S. Sunderam, PVM: "A Framework for Parallel Distributed Computing", *Concurrency: Practice & Experience*, December 1990, Vol. 2, No. 4, 315-339

[Toffoli87] Toffoli, Margolus, *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, 1987

[Tschöke95] S. Tschöke, M. Räcke, R. Lüling, B. Monien, "Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm, *Proc. of the 9th International Parallel Processing Symposium (IPPS'95)*, IEEE Computer Society Press, 1995, 182-189

[TSPBIB00] P. Moscato, http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html

[TSPLIB00] G. Reinelt, http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/

[Veen86] A. H. Veen, "Dataflow Machine Architecture," *ACM Computing Survey*, December 1986, Vol. 18, No. 4, 365-396

[VisibleHuman98]http://visiblehuman.epfl.ch/

[Watt90] D. A. Watt, *Programming Language Concepts and Paradigms*, Prentice-Hall, 1990

[Weeks96] A. R. Weeks, *Fundamentals of Electronic Image Processing*, SPIE/IEEE Series on Imaging Science & Engineering, 1996, 127-144

[Wilson96] G. V. Wilson, P. Lu (Eds), *Parallel Programming Using C++*, The MIT Press, 1996

[Wolfe82] M. J. Wolfe, *Optimizing supercompilers for supercomputers*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-82-11005, 1982

[Wright95] G. R. Wright, W. R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley Professional Computing Series, Reading, Massachusetts, January 1995

[Wüthrich94] D. Wüthrich, "Verfaren zur Ermittlung von Radiohörerverhalten und Vorrichtung dazu", *European Patent Application EP 0 598 682 A1*, applicant LIECHTI AG, published 25.05.1994

[XDR95] RFC 1832, http://www.cis.ohio-state.edu/Services/rfc/index.html

[Xu01] M. Z. Xu, "Effective Metacomputing using LSF MultiCluster", *Proc. of IEEE/ACM Symposium on Cluster Computing and the Grid*, May 2001, 100-105, see also http://www.platform.com

Biography

Marc Mazzariol was born in the city of Geneva, Switzerland, on July 9th 1973. He received his high school degree from *Collège Calvin* in Geneva. At the same time he started as trainer of a competition team (national and international level) in trampolin. He pursued his studies at the *Ecole Polytechnique Fédérale de Lausanne* (EPFL), Switzerland and graduated as a Computer Science Engineer in 1997. He has been a research assistant at the Peripheral Systems Laboratory (LSP) of the EPFL for the past four years. His research interests include parallel and distributed computing, computer-aided parallelization tools, network programming, and design patterns in C++.

Publications

- M. Mazzariol, B. Gennart, R. D. Hersch, M. Gomez, P. Balsiger, F. Pellandini, M. Leder, D. Wüthrich, J. Feitknecht, "Parallel Computation of Radio Listening Rates", Parallel and Distributed Methods for Image Processing IV, July 2000, San Diego, USA, SPIE vol. 4118, 146-153

- M. Mazzariol, B. Gennart, R.D. Hersch, "Dynamic load balancing of parallel cellular automata", Parallel and Distributed Methods for Image Processing IV, July 2000, San Diego, USA, SPIE vol. 4118, 21-29

- B. A. Gennart, M. Mazzariol, V. Messerli, R. D. Hersch, "Synthesizing Parallel Imaging Applications using the CAP Computer-Aided Parallelization tool", IS&T/SPIE's Symposium, Electronic Imaging'98 Conf., Storage & Retrieval for Image and Video Databases VI, San Jose, Calif., 1998, SPIE vol. 3312, 446-458

- Marc Mazzariol, B. A. Gennart, V. Messerli, R. D. Hersch, "Performance of CAP-specified linear algebra algorithms", EuroPVM-MPI'97, Krakow, Poland, 1997, Springer-Verlag, LNCS 1332, 351-358

Other Reference

- The Visible Human Slice Server, **http://visiblehuman.epfl.ch/**