Universidad Autónoma de Barcelona

Escuela Técnica Superior de Ingeniería

Departamento de Informática

Coscheduling Techniques for Non-Dedicated Cluster Computing

Memoria presentada por Francesc Solsona Tehàs para optar al grado de Doctor en Informática

Barcelona, Mayo de 2002

Coscheduling Techniques for Non-Dedicated Cluster Computing

Memoria presentada por Francesc Solsona Tehàs para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona. Trabajo realizado en el Departamento de Informática de la Escuela Técnica Superior de Ingeniería de la Universidad Autónoma de Barcelona, dentro del programa de doctorado "Arquitectura de Computadores y Procesamiento Paralelo", bajo la dirección del Dr. Porfidio Hernández Budé.

Barcelona, Mayo de 2002

Vo. Bo. Director de Tesis

Fdo. Porfidio Hernández Budé

Deseo expresar mi agradecimiento al Dr. Porfidio Hernández Budé por su magistral dirección y constante dedicación a la supervisión del trabajo.

A Francesc Giné, por su ayuda y colaboración en varias fases del trabajo. A Concepció Roig, que habiendo coincidido en la realización de la Tesis ha sido en todo momento un importante punto de apoyo. Y en general a toda el área de Arquitectura del Departamento de Informática de la Universidad de Lleida.

A Emilio Luque y a todo el grupo de investigación de la Unidad de Arquitectura y Sistemas Operativos del Departamento de Informática de la UAB, que gracias a su colaboración y apoyo ha sido posible la realización de esta Tesis.

Y como no, a toda mi familia. A mis padres, hermanos, a mi mujer, Montse y a los pequeños, Sara y Roger, que han sabido entender en todo momento la dedicación que me ha supuesto la realización del proyecto.

Abstract

When general-purpose workstations are interconnected by a Local Area Network (LAN) they form the so-called Network Of Workstations (NOW) or Cluster system. This kind of workstation is designed for developing and executing applications without excessive computing requirements. These applications, named "local" or "user" in this project, are usually very interactive or I/O bound. Their execution rarely exceeds the computing power provided by the hardware resources of the workstation.

Large CPU-bound applications require more complex computer systems. Supercomputers and MPPs (Massively Parallel Processors) have been the most widely used platforms to execute this kind of application. Basically, they are formed by a large number of Processing Elements (PE) with shared (all the PEs share a global memory) or distributed memory (a local memory is assigned to each PE). Depending on the memory access pattern, the applications can be classified as parallel or distributed. This also differentiates two different research fields, namely parallel and distributed processing. In the parallel case, communication/synchronization between processes (located at different PEs) making up parallel applications is performed through the Shared Memory. In contrast, in distributed processing, as local memory in one PE cannot be accessed by another one, the remote processes forming the distributed applications, communicate/synchronize between them by means of message passing.

The key questions are why not to use the Clusters in distributed processing, and what the advantages/drawbacks in doing so are. Many researchers work in this research field of Cluster Computing trying to answer these questions. Apparently, the workstations are not designed to execute distributed applications efficiently, but recent advances in their hardware components and the increasing speed of

LANs provide the NOWs with performance capabilities ever closer to the that of the MPPs. Another factor to be mentioned is the constant evolution of different Distributed Computing Environments (DCE), such as PVM (Parallel Virtual Machine) and MPI (Message Passing Interface), which are also applicable in Cluster computing and simplify the design and implementation of distributed applications for such systems. As a result, the use of NOWs or Clusters in distributed processing is increasing and with time is becoming more and more popular.

There are important drawbacks to be considered if both distributed and local applications are executed in parallel in a Cluster system. The performance of distributed applications depends on the behavior of all their forming processes, spread over the Cluster nodes, so the performance of distributed applications can decrease significantly if the nodes (even only one node) of the Cluster are (is) heavily loaded by local jobs. On the other hand, nodes with large distributed workload can disturb the local jobs excessively. The local tasks are normally I/O bounded, so the term “disturb” would mean here: “increase the response time”. If the local tasks were CPU-bound, “disturb” would mean in this case: “increase the return time”.

As a consequence of the above commented dissertation, a new research goal in Cluster computing appeared recently: how to coordinate local and distributed applications when they are executed in parallel in a Cluster system. We also focus our research on this area. More precisely, our efforts are centered on constructing a Virtual Machine over a Cluster system that provides the double functionality of executing traditional workstation jobs as well as distributed applications efficiently.

The problem is focused on Clusters in which the forming nodes have *multiprocessing* capabilities. If not, no manner of executing distributed and local applications in parallel would be possible.

Moreover, the execution of local applications (which usually require low average response times) in the Cluster implies that the operating system of each node must also be *time-sharing*. This property attempts to split the CPU time between all the executing tasks, so the average response time of local tasks can be reduced. Furthermore, distributed applications with a high (even low) communication degree will be affected if this property is not provided: an excessive delay in the

execution of distributed tasks will not favor the rapid interchanging of messages, and consequently their performance would be low.

To solve the problem, two major considerations must be addressed:

- How share and schedule the workstation resources (especially the CPU) between the local and distributed applications.
- How to manage and control the overall system for the efficient execution of both application kinds.

Coscheduling is the base principle used for the sharing and scheduling of the CPU. As will be seen later in greater depth, *coscheduling* is based on reducing the communication waiting time of distributed applications by scheduling their forming tasks, or a subset of them at the same time. Consequently, non-communicating distributed applications (CPU bound ones) will not be favored by the application of *coscheduling*. Only the performance of distributed applications with remote communication can be increased with *coscheduling*.

Coscheduling techniques follow two major trends: *explicit* and *implicit control*. This classification is based on the way the distributed tasks are managed and controlled. Basically, in *explicit-control*, such work is carried out by specialized processes and (or) processors. In contrast, in *implicit-control*, coscheduling is performed by making local scheduling decisions depending on the events occurring in each workstation.

Two coscheduling mechanisms which follow the two different control trends are presented in this project. They also provide additional features including usability, good performance in the execution of distributed applications, simultaneous execution of distributed applications, low overhead and also low impact on local workload performance. The design of the coscheduling techniques was mainly influenced by the optimization of these features.

An *implicit-control* coscheduling model is also presented. Some of the features it provides include collecting on-time performance statistics and the usefulness as a basic scheme for developing new coscheduling policies. The presented *implicit-control* mechanism is based on this model.

The good scheduling behavior of the coscheduling models presented is shown firstly by simulation, and their performance compared with other coscheduling

techniques in the literature. A great effort is also made to implement the principal studied coscheduling techniques in a real Cluster system. Thus, it is possible to collect performance measurements of the different coscheduling techniques and compare them in the same environment. The study of the results obtained will provide an important orientation for future research in coscheduling because, to our knowledge, no similar work (in the literature) has been done before.

Measurements were made by using various distributed benchmarks with different message patterns: regular and irregular communication patterns, token rings, all-to-all and so on. Also, communication primitives such as barriers and basic sending and receiving using one and two directional links were separately measured. By using this broad range of distributed applications, an accurate analysis of the usefulness and applicability of the presented coscheduling techniques in Cluster computing is performed.

Another point to mention is that general-purpose workstations do not differentiate between distributed and local jobs. All of these are simply jobs. Consequently, some sort of mechanism for differencing these two kinds of jobs is considered when new solutions are proposed.

Furthermore, the implementation of the coscheduling techniques may be performed in the user or the system space. So, it is necessary to analyze the main features the Cluster communication subsystem provides in the design of the coscheduling techniques. Moreover, the scheduling mechanisms (located in the operating system) of the forming nodes must also be studied. PVM and Linux are respectively the DCE and the operating system which were used in this project as base platform. The communication subsystem composed by PVM and Linux and also the Linux scheduler were accordingly studied in depth.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The studies in [4] indicate that the workstations in a NOW (Network of Workstations) are normally underloaded. So, a great number of researchers have been exploring ways to make better use of the wasted NOWs capacity for parallel computing. As a result, Networks of Workstations have become important and cost-effective parallel platforms for scientific computation.

Basically, there are three methods of using these CPU idle cycles. The first area is the analysis, design and development of remote execution facilities, *Condor* [5], *Process Server* [6], *Sprite* [3] and so on.

The second area is task migration [5, 7]. In a NOW, in accordance with the research by Arpaci [16], the unpredictable behavior of local users may lower the effectiveness of this method. An alternative is load balancing, which may be excessively time costly in Cluster computing [34]. In addition, load balancing may cause subsequent problems, such as redirection of messages (due to the migration of tasks), and extra overhead and communication traffic in managing and controlling the overall system.

The third and last area is job scheduling [8, 49, 18]. This deals with the design of schedulers, which must ensure the requested interactive behavior of a workstation. At the same time, this technique attempts to give as many CPU cycles (including the wasted ones) as possible to parallel tasks.

Job scheduling is the area chosen to tackle the problem, and more specifically parallel-task *coscheduling* rather than job scheduling.

In practice, a NOW system is heterogeneous and non-dedicated. These two unique factors make scheduling policies on multiprocessor/multicomputer systems unsuitable for NOWs, but the coscheduling principle is still an important basis for parallel process scheduling in these environments.

The heterogeneity of a NOW can be modeled by using the Power Weight, developed by Zhang in [50]. Each node making up the Cluster has a value (the power weight) between [0..1] assigned. This value represents the execution speed with respect to the fastest node in the system. Nevertheless, the non-dedicated feature means that two issues must be addressed, these being how to coordinate the simultaneous execution of the processes forming a parallel application, and how to manage the interaction between parallel and local user jobs. Combining parallel and sequential workloads on a non-dedicated Cluster system, with reasonable performance for both kinds of computation is an open research goal. Efforts in this project have been focused on the proposal of new solutions in this research field.

To define the problem to be solved, a more accurate study of the Cluster systems must be performed. Section 1.1 presents the location of NOWs inside a general classification of parallel architectures together with their principal features.

The coscheduling principle is defined in section 1.2. The different kinds of coscheduling are also presented in this section.

An in depth study of the state of the art is performed in section 1.3. Special attention is given to the coscheduling techniques which consider simultaneous executions of various distributed applications jointly with local or user workload.

Next, by means of a simple example, the reason for using coscheduling in Cluster computing is demonstrated (in section 1.4). After presenting the most relevant work in coscheduling and one specific motivation example, the main objectives of this project can be more easily understood. The presentation of the main aim as well as the detailed objectives of this thesis occupy the last part of this section.

Finally, the remaining contents of this project are presented in the Overview section (section 1.5).

1.1 Cluster Systems

A taxonomy of parallel systems, done by Tanenbaum in [36], is shown in Fig. 1.1. Here, the location where the Clusters are can be observed.

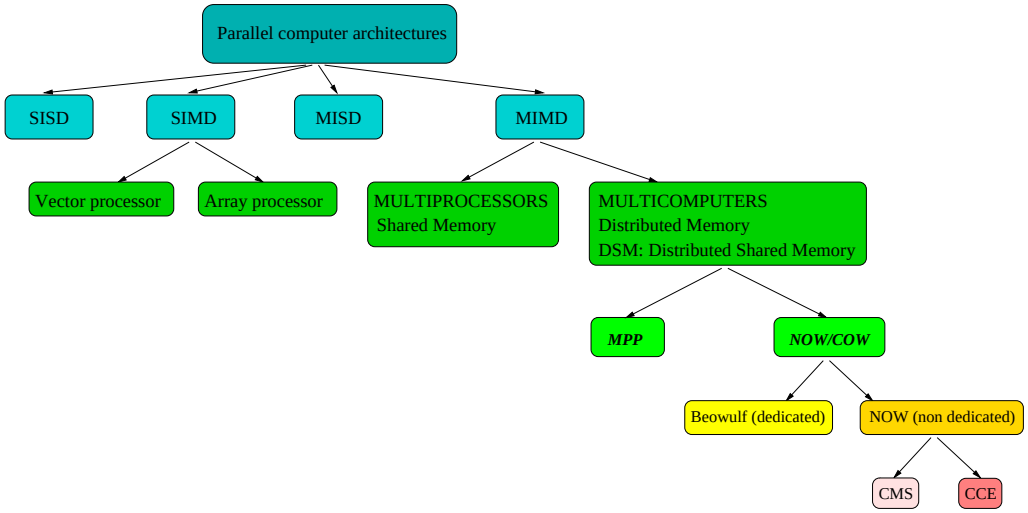


Figure 1.1: Taxonomy of Parallel Architectures.

The first level of the Tanenbaum taxonomy (see Fig. 1.1) is based on Flynn's classification of parallel computers.

Flynn used the concepts "instruction" and "data" stream as base terms for classifying the different kinds of architecture. Each Instruction stream has an associated Program Counter. A data stream consists of a set of operands. For example, SISD is the acronym of Single Instruction stream Single Data stream and corresponds to the classic Von Newman's Machine. We are interested in the MIMD (Multiple Instruction stream Multiple Data stream) category. In this case, multiple processes will execute in parallel by using various data sets. This is the branch where the NOWs are located, so the rest of Flynn's classification will not be considered henceforth.

The MIMD classification includes Multiprocessors (shared-memory machines) and Multicomputers (message-passing machines). Multiprocessor classification is based on the way the shared memory is implemented in them. Accordingly, no more comments nor explanations about Multiprocessors are necessary.

The Multicomputers are divided into MPPs (Massively Parallel Processors) and NOWs/COWs (Network/Cluster Of Workstations). The MPPs are expensive Supercomputers consisting of many CPUs “tightly coupled” by a high-speed interconnection network. The Cray T3E and IBM SP/2 are two well-known examples. NOWs, which are also named COWs, consist of regular PCs or workstations “thinly coupled” by commercial off-the-self interconnection technology.

Since the early 1980s, a particular trend in parallel computing is to move away from specialized platforms (such as MPPs) to general purpose systems, and especially to Cluster systems [74].

The most interesting comment to be made before introducing the NOWs is that they are based on distributed memory (see Fig. 1.1). However, some developed environments for distributed-memory systems implement the so-called DSM (Distributed Shared Memory). DSM allows the execution of shared-memory applications in distributed memory systems. These environments are discarded because we are interested in pure message-passing communication systems.

Distributed computing is a more accurate terminology for parallel computing in Cluster systems. Message passing is the paradigm used in communicating/synchronizing distributed tasks and no semaphores or shared memory utilities like those in parallel computing are used. This can be a drawback because to date, the most important work and research was been performed for shared-memory parallel applications. Furthermore, applications designated for serial systems are more easily migrated to parallel systems than distributed ones, because both serial and parallel systems use the shared-memory paradigm for communication/synchronization. Distributed applications use message passing instead. Researchers have made a great effort investigating automatic tools (i.e. interpreters and compilers) for migrating applications from shared to distributed memory systems, but a lot of work still remains to be done in this area.

Definition of Cluster: a "Cluster" is a local computing system comprising a set of independent computers and a network which interconnects them. The constituent computer nodes are general purpose workstations and the interconnection network usually employs an isolated local area network (LAN).

A "Beowulf-class system" ([61]) is a dedicated Cluster which is generally composed of personal computers (PC), integrated by local area networks (LAN), and normally hosting an open source Unix-like operating system in each node. Non-dedicated Clusters are also referenced as NOWs (Network Of Workstations) or simply Clusters.

In our case, the system used (for implementing and testing the coscheduling techniques presented in this thesis) is a NOW or Cluster. The system is non-dedicated because we are interested in executing in parallel user (local or interactive) as well as distributed applications.

The NOWs can be also divided between Cluster Management Software (CMS) and Cluster Computing Environments (CCE). The difference between CMS and CCE is that in the first, the addition of new functionality to the NOW is performed without modifying the operating system of the component nodes (additions/modifications are made in the user space). Instead, the CCE term usually serves to designate NOWs where the operating system incorporates parallel (distributed) facilities and services (additions/modifications are made in the system space). As will be seen, our work deals with both kinds of NOW.

The increasing use of Cluster systems in distributed computing is due to many factors. The most important ones are mentioned below:

- Continuous advances in high speed networks, microprocessor and component performance. Furthermore, the ratio between performance and price overtakes the contemporary Supercomputers.

- Commercial hardware and software technologies can also serve in high performance computing using Cluster systems. As a consequence, their acquisition does not require the heavy investment of the Supercomputers.

Exactly what attributes a Cluster system should possess is still an open question. The most desirable features a Cluster should provide are listed below:

- Stability: the most important characteristics are robustness against crashes of nodes (or processes) and usability under heavy load.

- Performance: bandwidth and latency of the interconnecting network, as well as memory management, process scheduling, I/O and communication protocols of each component node, should be as efficient as possible.

- Scalability: the scalability of a Cluster is mainly influenced by the provision of the contained nodes (i.e. maximum number of processes, user addressable space and so on). Also, the performance characteristics of the interconnection network have an important influence on scalability.

- Support: unlike large Supercomputers, the support cost of any Cluster is normally much lower because Cluster components are developed for the commercial and consumer computer markets.

- Heterogeneity: a Cluster system does not necessarily consist of homogeneous hardware and software. It can be made up of different kinds of nodes (PCs, workstations, even Supercomputers), operating systems (Unix, Windows, NT Windows and so on), etc ...

Another kind of distributed computing is Metacomputing. Metacomputing systems must be able to handle heterogeneous computer systems, wide area distribution, heterogeneous data, multiple user identities, and multiple security systems. Basically, differences between Cluster computing and Metacomputing are in the kind of network used in interconnecting the forming nodes. Usually, Cluster computing deals with NOWs, whose interconnection network is a LAN. Meanwhile, the term Metacomputing is associated to LANs which are also interconnected by means of WANs (Wide Area Networks). Thus, Metacomputing is a more generic term than Cluster Computing. For example, in addition to the problems of Cluster Computing, Metacomputing also includes user verification, and security control and maintenance. The work presented in this thesis is centered exclusively in Cluster Computing.

1.1.1 Cluster Components

The main components of a Cluster system are studied next separately. This study is necessary for understanding the terminology used throughout this document. Figure 1.2 shows the scheme of the general resources making up a Cluster system. Particular features of the Cluster used in the implementation of the different coscheduling techniques presented in this document appear in parenthesis. A particular characteristic (not shown in the figure) is that the nodes making up the Cluster are uniprocessor.

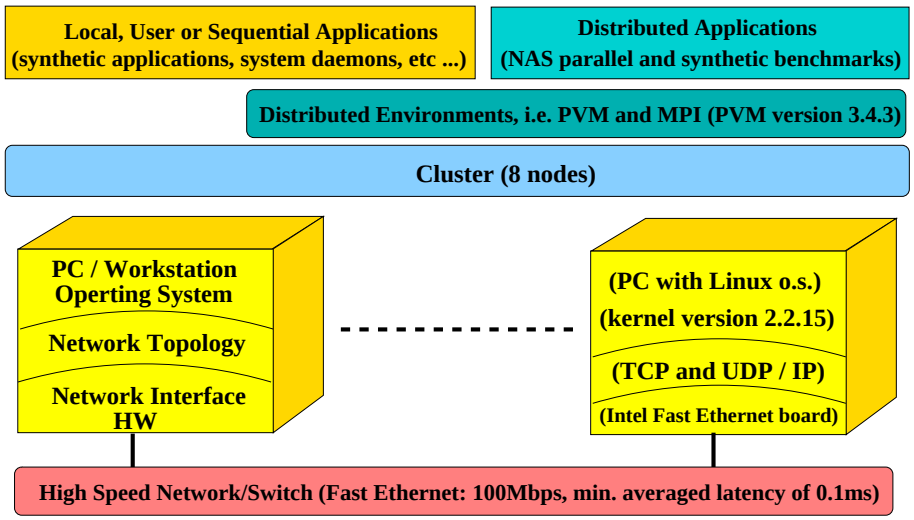


Figure 1.2: Cluster architecture

One of the most important Cluster components is the Operating System running in each Cluster node. Special attention is paid to the Linux o.s., this being the one chosen for this project.

Three Cluster Network Topologies are also commented on. The first one is the Internet Protocol, which is the kind of network used in this project. The other two are Active Messages and Fast Messages. As we will see later, Active and Fast Messages were used in implementing the two most important coscheduling techniques we based this project on.

The most commonly used Distributed Computing Environments (DCE) in Cluster computing are PVM (mainly used in this project) and MPI. Their prin-

cipal features and differences are also presented in this section.

Finally, the final part of this section contains the definition and some additional comments about the local applications.

1.1.1.1 Operating System

We are interested in investigating non-dedicated Clusters. This is, the study of the efficient use of Cluster systems for executing both distributed and user (interactive or local workload) applications simultaneously. So, based on that and taking into account the Cluster definition, the list of requirements the underlying o.s. must provide are as follows:

- Multiprocessing or multiuser capabilities: multiple execution of different processes (or users) must be allowed. If not, only serial execution would be possible. In a multiuser system, each process belongs to a user. Furthermore, each user has an specific environment (i.e. file system directory, rights, environment variables and so on), which serves for controlling system security and managing resource accounting. The environment is associated and reset for the permitted user in the login phase (when entering the system).

- Time-sharing: this feature guarantees rapid response time for interactive, I/O-bound, or even message passing intensive applications. If time-sharing were not provided by the underlying o.s., most of both distributed and local applications would perform poorly when various of them were executed in parallel. Normally, time-sharing is accomplished by using some sort of Round-Robin scheduler policy, which is based on repeatedly assigning a maximum execution time slice (quantum) to each process.

- Extensibility: the o.s. should allow the integration of coscheduling mechanisms and Cluster-specific extensions. For example, as we will see, some modifications to the original scheduler (in the kernel space) will be performed on the implementation of a new coscheduling technique. Also, some specific system information (for example, number and size of buffered communicating messages) which is not reported by the original system will

be accessed in doing so. Thus, new extensions to the o.s. must be easily incorporated.

- Usability: an absolute necessity is remote system administration. For example, some initial configuration should be performed by scripts which access nodes remotely. If remote access was not available, it must be performed manually, thus leading to hard, tedious work.

In this thesis, and principally in the implementation sections, comments and assumptions are mostly based on Linux [62, 63, 65, 66, 64]. Also, this was the o.s. chosen for implementing the coscheduling techniques in each Cluster node because Linux fulfils the above commented requirements and also for five additional reasons:

1. It is free.
2. It is an open source operating system: anyone is free to modify or customize the kernel for their own specific needs. This means that it can incorporate I/O optimizations, such as specialized drivers for new high-speed networks.
3. Research and development focus: Linus Torvalds started a personal project aimed at extending the MINIX o.s. [67]. He converted it into a UNIX reply. Nowadays, Linux is still an incomplete project. From its public launch in 1991, it has undergone a continuous and rapid evolution because many scientists base their research on this o.s.. As a consequence, extensive documentation is available nowadays as well as s/w and h/w resources adapted to it.
4. Acceptance: there is a growing acceptance in the last years over a great range of computer systems (specially in PCs or Workstations). Also, computer firms trust more and more in Linux as a base platform for developing their applications or components. Since its initial development for the Intel processors, it has been ported to other processor families as for example SPARC, Alpha and MIPS.
5. For historic reasons: Donald Becker selected Linux for the original Beowulf Cluster and thus, Beowulf-derived systems have also used Linux.

1.1.1.2 Cluster Networks

There are a broad range of interconnection technologies available for Cluster computing. So, only the ones referred to or used in the document are commented on.

Internet Protocol: The Internet Protocol (IP) [71] is the *de facto* standard for networking worldwide. IP offers messaging service between two computers that have an IP address. The Transmission Control Protocol (TCP) [70] and the User Datagram Protocol (UDP) [69] are both transport layer protocols built over the Internet Protocol. TCP (or TCP/IP) offers a reliable, connection-oriented service between two hosts on a network. UDP is an unreliable, connectionless transport layer service. The de facto standard BSD sockets is the most common Application Programmer's Interface (API) for TCP and UDP. Traditionally, TCP and UDP protocols are typically implemented using operating system services and one or more buffers in main memory.

The most important drawback in the use of TCP(UDP)/IP is that as network hardware became faster and faster, the overhead of the communication protocols remains significantly larger than the actual hardware transmission time for messages. That is, the operating system latencies in sending messages, have not evolved as rapidly as the network ones.

Active Messages: messages in Active Messages [24] are synchronous: each message contains the address of a user-level handler at its head. The handler is executed on message arrival. The role of the handler is to get the message out of the network and to yield it to the corresponding process. Like a traditional pipeline, the sender blocks until the message can be injected into the network and the handler executes immediately on arrival. Then, the handler will transfer the message from the network to the receive buffer, in the user memory.

The main advantage of Active Messages is that no buffering in system memory is performed.

Fast Messages: Fast Messages [72] was developed at the University of Illinois. It is a very similar protocol to Active Messages. Fast Messages extends

Active Messages by guaranteeing that all messages arrive reliably and in-order, even if the underlying network hardware does not. It does this in part by using flow control to ensure that a fast sender cannot overrun a slow receiver, thus causing messages to be lost.

Authors assure that Fast messages provides MPP-like communication performance on workstation Clusters.

1.1.1.3 Distributed Computing Environments (DCEs)

A DCE (Distributed Computing Environment) is a high-level message passing system, used for developing, executing, testing and controlling distributed applications.

Currently, the two most popular DCEs are the *de facto* standard PVM [30] (Parallel Virtual Machine), from the Oak Ridge National Laboratory and the standard MPI [32, 33] (Message Passing Interface), defined by the MPI Forum.

Geist et al. provide an excellent comparison between these two environments in [31], summarized in Table 1.1.

PVM	MPI
Both MPI and PVM run on MPPs and heterogeneous Clusters	
virtual machine concept	no such abstraction
simple message passing	rich messaging support
communication topology unspecified	supports logical communication topologies
portability over performance	performance over flexibility
robust fault tolerance	more susceptible to faults
PVM can only use IP	MPI can use a wide range of protocols
contains resource management, load balancing and process control primitives	primarily concerned with messaging
programs in C, C++, or Fortran may freely intercommunicate	interlanguage communication not supported

Table 1.1: PVM versus MPI

We are not interested in any particular environment because we do not want to modify or improve any of those aspects listed in Table 1.1. Also, the study and applicability of coscheduling do not depend on the choice of one of them.

However, PVM has more resource management facilities (such as the console) and task management capabilities. These facts simplify enormously the use and control of the overall system and as we will see, they also have an important role in the implementation of new coscheduling mechanisms. Consequently, PVM was finally the chosen distributed computing environment.

1.1.1.4 Local Applications

Different kinds of applications can be executed in a Cluster. We only distinguish between distributed and local ones. Distributed applications are formed by one or various distributed tasks. This is also true for local applications. The main difference between them is that all the local tasks making up a local application reside in one unique node. On the contrary, distributed tasks making up distributed applications can reside in different nodes. So, remote (only local) communication or synchronization between distributed (local) tasks may occur.

Throughout this document the local applications are denoted as local, user or interactive workload. Also, the term "*workload*" will be used to denote the local tasks residing in a particular Cluster node.

The term "*workload index*" will be used to quantify the number of local tasks residing in each Cluster node. A wide variety of workload indices have been used in the literature [53, 54, 55]. From among, the Ready to run Queue (RQ) length was chosen. This was decided because the work done by Ferrari [54] shows, by a wide range of experimentation, that the length of the RQ is a good index for measuring the load of a workstation.

Ferrari compared various workload indices. First, he differentiated the ones based on resource utilization and the ones based on resource queue length. Various resources were considered: the CPU, Main Memory (MM) and I/O (the different disk queues were treated as a single I/O queue). It was shown that resource utilization cannot reflect the load level exactly. As an example, a resource with an average queue lengths of 3 and 6, probably has a utilization close to 100%, while they are obviously very differently loaded. Also differences in workload indices based only on CPU queue length and those considering also I/O and MM queue lengths gave very close values. He also proved that the CPU is the most predom-

inant resource. The conclusion is that the average RQ length is a good workload index.

In [55], using a synthetic, executable workload, the experiments were conducted to determine the effect of different load indices. It was also shown that by using a simple workload descriptor, such as the number of tasks in the ready queue, one can achieve an equal or better performance compared with more complex load descriptors.

In this project we are interested in observing the interaction between the distributed and local workloads. A wide range of distributed applications (i.e. synthetic and well known benchmarks) will form the distributed workload. Instead, the local or user workload characterization in each node of the Cluster is carried out by means of running synthetic applications, which perform basically floating point operations indefinitely (or a variable number of times). Taking into account the work done by Ferrari, it is a good characterization of the workload because the key question is based on varying the mean RQ length (and this way, the workload). Furthermore, it is more helpful in obtaining and comparing performance of the different proposed environments because the workload is not a variable factor to be taken into account (it can be fixed at a predetermined value).

1.2 The Coscheduling Concept

1.2.1 Scheduling Schemes

Parallel/Distributed scheduling is usually decomposed into two independent steps. The first step, space slicing, determines the mapping of a parallel/distributed application into a set of processors. The second step, time slicing (or time sharing), dispatches those allocated processes over time.

A comprehensive classification of the scheduling techniques for multiprogrammed parallel systems was done by Feitelson in [13]. It was based on the way the computing resources are shared: space slicing, time slicing, or both. Table 1.2 shows a more reduced Feitelson's scheme with one or two examples of real systems for each particular case. On the time slicing axis, the main distinction was made between mechanisms that operate independently in each PE (Processor Ele-

ment) and *Gang scheduling* (defined later), that handles a group of PEs as a single unit. Mechanisms for independent PEs are further divided into those that use local queues (requiring processes to be mapped to PEs before their scheduling), and those which use a shared global queue.

space slicing	time slicing			
	yes			no
	independent PEs		gang scheduling	
	global queue	local queues		
yes	Mach	Paragon, Meiko,	Medusa, SGI,	IBM SP2,
no	IRIX on SGI,	StarOS, Payche,	MasPar MP2	Illiac IV

Table 1.2: Time-slicing and space-slicing classification.

As can be seen in Table 1.2, the first and second scheduling steps are independent. We focus on the second step of time slicing processes over time. Thus, no more space slicing techniques will be considered.

One of the most popular time slicing mechanisms is *Gang scheduling* ([2]). *Gang scheduling* is based on the preemptive and simultaneously scheduling of a certain set of processes (or threads) on distinct PEs, with a one-to-one mapping of threads to PEs. Thus, it requires coordinated context switching across the PEs, which is harder to implement than independent context switching. Basically, the benefits are reached when the overhead of frequent context switching is saved and the need for buffering during communication is reduced.

Gang scheduling is gaining in popularity, and an increasing number of commercial systems provide gang scheduling. Some examples of real implementations in different computer systems are: Medusa, SGI, Butterfly, Tera, Cray T3E, Meiko CS-2, Intel Paragon, CM-5, Cedar, IBM SP2 and so on.

1.2.2 Coscheduling

In gang scheduling (coscheduling's ancestor) all the threads in a job are scheduled and de-scheduled at the same time, so threads composing jobs should be known in advance. This information, in distributed systems like Clusters or NOWs, is very difficult to obtain (or maintain). The alternative is to identify them during

execution [12]. Furthermore, if the processes synchronize with each other at fine granularity, it would be beneficial to schedule just some of them simultaneously, leading to *coscheduling* rather than gang scheduling.

Coscheduling was originally defined by Ousterhout in [2] to describe systems where the operating system attempts to schedule a set of processes simultaneously on distinct PEs, as in gang scheduling, but if it cannot, then it resorts to scheduling only a subset of the processes simultaneously.

Another interesting and perhaps a bit more self-contained definition, by Feitelson in [12], is as follows: *Coscheduling* ensures that no process will wait for a non-scheduled process for synchronization/communication and will minimize the waiting time at the synchronization points.

In a Cluster or NOW system, coscheduling is applied to reduce message waiting time (by synchronizing tasks composing fine grained distributed applications -those with a high communicating degree-).

The distinction between gang scheduling and coscheduling is subtle but significant. Coscheduling is a variant of gang scheduling that does not guarantees that all the processes (or threads) making up a distributed (or parallel) application will always run simultaneously.

A gang feature (derived from the above mentioned distinction) which also differentiate those mechanisms is that gang scheduling allows guarantees about the performance to be given. This is so because applications execute in an environment that is essentially the same as a dedicated machine, except for some additional overheads. Coscheduling, on the other hand, has unknown performance implications. If the application processes are largely independent, they can make progress even if the whole gang is not scheduled. Coscheduling can be performed between groups of processes which form the gang. In this case, coscheduling can be highly beneficial.

Researchers in this area have shown that coscheduling can offer good performance, although the literature demonstrates that coscheduling is critical for parallel programs in order to achieve acceptable performance. The key question is how coscheduling techniques must combine parallel and sequential workloads with reasonable performance for both computation kinds.

1.2.3 Coscheduling Classification

The coscheduling techniques follow two control-based trends:

- **Explicit-control coscheduling**: in explicit-control coscheduling, cooperating components (i.e. specialized nodes or processes) explicitly contact other local or remote components for control or state information. They are responsible for making decisions and controlling the overall system.

  Another level of classification was indicated by Poovendran in [20]. Explicit-control coscheduling may be accomplished statically with the provision of a global scheduler responsible of making decisions in advance, or dynamically, where decisions are made at each context switch. In the last case, controlling and information interchange must be performed between the processes or nodes responsible for implementing explicit-control coscheduling in a distributed manner.

- **Implicit-control coscheduling**: in implicit-control coscheduling, components infer remote state by observing naturally-occurring local events and their corresponding implicit information, i.e., incoming (outgoing) messages to (from) a Cluster node, local workload, etc...

  It deals with implicit information: each node in the Cluster acts according to the interpretation of some sort of event, and determining from this the convenience for coscheduling processes with other cooperating distributed tasks.

Implicit-control simplifies the construction of distributed system services. Components neither query nor control remote components in their actions. Thus, an implicit-control system does not contain additional communication beyond that which is inherent in constructing the service.

Unlike implicit-control coscheduling, explicit-control techniques provide a means for controlling the overall system, and information concerning the global behavior must be maintained, thus more accurate decisions can be taken. The drawback is in the overheads introduced in the addition of (information and control) message exchanging between the processes that implement the mechanism.

The choice between explicit or implicit-control coscheduling mechanisms depends on the requirements of the final distributed system.

1.3 Related Work

Researchers and efforts in coscheduling have grown continuously since their origin. Nowadays, it still remains as an open question. As a consequence of such research work, many tendencies have arisen or are being developed. So, an accurate study of the state of the art must be performed.

Feitelson and Rudolph in [11], compared the performance of gang scheduling using busy-waiting synchronization to that of independent (uncoordinated) time sharing using blocking synchronization. They found that for applications with fine-grain synchronization, performance could degrade severely under uncoordinated time sharing when compared with gang scheduling. In an example where processes synchronized about every 160μs on a multiprocessor with 4-MIPS processing nodes, applications took roughly twice as long to execute under uncoordinated scheduling as they did under gang scheduling.

Another interesting study was performed by Crovella et al. in [8]. They showed that independent time sharing without regard for synchronization produced significantly greater slowdowns than coscheduling. Chandra et al. have reported similar results in [10]: in some cases independent time sharing is as much as 40% slower than coscheduling.

In general, the results cited above agree with the claims advanced by Ousterhout in [2]: under independent time sharing, multiprogrammed parallel job loads will suffer large numbers of context switches, with attendant overhead due to cache and TLB reloads. The extra context switches result from attempts to synchronize with de-scheduled processes.

As was mentioned in section 1.2, the coscheduling techniques follow two major trends: explicit and implicit control. Moreover, the coscheduling techniques studied (from the literature) and the ones presented in this document are located in one of these two categories. So, the study of the previous work, developed in this section, attempts to separate the main coscheduling contributions to date according to this classification.

Table 1.3 resumes the most relevant work done in coscheduling to date, and presented later in this section.

The table is split into two parts. In the upper part, a classification of the most important explicit and implicit-control techniques is given. In the bottom part, the main features of each model are listed. It summarizes schematically the main differences between the models. The abbreviations used are the following: Y - Yes; N - No; ϕ - no observations have been made. DCE - used Distributed Computing Environment (S: Simulated); OS - Operating System; P - Protocol (IP: TCP(UDP)/IP, AM: Active Messages, FM: Fast Messages); N - Network type (* - the coscheduling model was based on this kind of network); V - simultaneous executions of Various distributed applications considered; L - Local tasks starvation considered.

Classification		
explicit control	A	SONiC [15]
	B	self coordinated local scheduler [49]
	C	buffered coscheduling [23]
	D	ticket coscheduling [22]
implicit control	E	implicit coscheduling [16, 17, 18, 19, 20, 21]
	F	demand-based (dynamic) coscheduling [28, 29]

	Properties						
	DCE	OS	P	N	V	L	Based on
A	SONiC	Mach	IP	ϕ	Y	N	priority
B	S	*UNIX SVRC	IP	ϕ	Y	Y	priority
C	S	ϕ	ϕ	*Myrinet, *Switched	Y	N	buffering and scheduling commnication messages
D	S	ϕ	ϕ	*Switched	Y	Y	assigning execution tickets
E	MPI	Solaris	AM	Myrinet	N	N	spin-block on blocking recv
F	MPI	Solaris	FM	Myrinet	Y	Y	preemption in favour to receiving tasks

Table 1.3: Coscheduling Techniques.

1.3.1 Explicit-control Coscheduling

In [15, 49] two variants of explicit-control coscheduling were implemented.

In [15] a Mach-based scheduling server was developed. It allows for partitioning a workstation among parallel and interactive tasks. This server is an integral part of SONiC, the “Shared Objects Net-interconnected Computer”. SONiC implements an object-based distributed shared memory and remote execution services and allows for execution of parallel programs in workstation environments. The scheduling server relies on Mach’s fixed-priority scheduling policy and it is based on giving as many CPU cycles as possible to parallel tasks. By manipulating task priorities it overrides the operating system scheduler. The main contribution was that in contrast with other “remote execution systems”, SONIC does not have to be restricted to the use of only purely idle workstations.

X. Du [49], based on the coscheduling principle, implemented the so-called "self-coordinated local scheduler", which guarantees the performance of both local and parallel jobs in a NOW by a time-sharing and priority-based operating system. The scheme can be applied to schedule multiple parallel jobs. Each local scheduler adjusts the execution rate of a parallel process. The coordination of parallel processes is performed independently in each workstation. The priority of the processes was varied according to the power usage agreement between local and parallel jobs. The effectiveness of this method was demonstrated by means of simulation.

Another variation of explicit-control coscheduling is Buffered Coscheduling [23]. Instead of synchronizing distributed applications, the messages are buffered and delivered later, thus reducing the need to coschedule communicating tasks. The communication generated by each processor is buffered and performed at the end of regular intervals (or time-slices) in order to amortize the communication and scheduling overhead. By delaying communication, a global scheduling of the communication pattern can be performed. Next, a strobing mechanism performs a total exchange of control information at the end of each time-slice in order to move from isolated scheduling algorithms to more global system scheduling algorithms.

An important advantage of Buffered Coscheduling is that, instead of overlapping computation with communication and I/O within a single parallel program,

all the communication and I/O which arise from a set of parallel programs can be overlapped with the computations of the overall programs. However, additional overhead should be introduced because an explicit manipulation of messages must be done.

Although no real implementation has been performed yet, the obtained simulation results make this a technique to be considered. Also a more accurate study of how the messaging delay affects performance of the communicating tasks must be done.

1.3.2 Implicit-control Coscheduling

In [28], Sobalvarro introduced the concept of *demand-based* coscheduling, a new approach to scheduling parallel computations on time-shared multiprogrammed multiprocessors. Under *demand-based* coscheduling, processes are scheduled simultaneously only if they communicate; communication is treated as a demand for coordinated scheduling. Also, it indicated that the programmer would not even need to identify the processes that constitute a parallel job.

That method does not follow the coscheduling line explained above (in explicit-control). In *demand-based* coscheduling, potential coscheduling is detected by the implicit information (so it follows an implicit-control trend) between communicating processes; avoiding not only the need to implement a system controller, but also extra communication traffic, and fault tolerance problems. Sobalvarro divided *demand-based* coscheduling between *Dynamic* and *Predictive* coscheduling.

Dynamic coscheduling uses the arrival of a message as a means of signaling the scheduler that the process to which the message is addressed should be immediately scheduled. In the first *Dynamic* algorithm he presented (named the *Always Schedule* dynamic coscheduling algorithm), the reception of a message always caused a context switch in favor of the receiving task (thus preempting the task being executed in the CPU). In the most interesting algorithm, the second (named *equalizing* dynamic coscheduling algorithm), a mechanism was also provided to avoid local task starvation, where equal shares of the CPU for both distributed and local tasks were considered.

In [29], only the execution of the *equalizing* algorithm for one distributed ap-

plication was evaluated. Due to the limitations of Fast Messages [72] (the user messaging level where the *equalizing* algorithm was implemented), no more than one concurrent execution of parallel applications was possible.

The *Predictive* method is based on coscheduling at the same time those processes that have recently communicated with each other (the so-called correspondent processes). It deals with detecting these kind of processes implicitly and efficiently coscheduling them. No previous work in this area has been performed to date, thus Predictive coscheduling is a new and open research field.

A variation of Dynamic coscheduling, named *Implicit coscheduling* in [16, 17, 18], is based on the spin-blocking technique. In this, a process waiting for messages spins (i.e. performs an empty loop) for a determined time and if the response is received before the time expires, it continues execution. If not, the requesting process is blocked and another one is scheduled. In [17], the measurements in a Cluster of 16 workstations showed how multiple competing parallel jobs can be coscheduled implicitly with good performance. These results hold for programs that communicate either continuously or in bulk synchronous style, as well as applications with a mix of communication characteristics.

Dynamic coscheduling, in contrast to Implicit coscheduling, deals with all message arrivals (not just those directed to blocked processes), thus increasing the range of potential cases for coscheduling. This produces an important Implicit drawback because it supposes that multiple executions of distributed applications cannot be efficiently coscheduled.

In [19], Implicit coscheduling was implemented in the Berkeley Cluster, which contains 105 workstations with 16-port Myrinet switches. The communication subsystem in each node was formed by the MPI ([32, 33]) environment MPICH ([27]), and the Abstract Device Interface was implemented by Active Messages [24] operations. Results obtained in the execution of various NAS kernel benchmarks ([56]) shown as implicit coscheduling reduced the communication waiting time in the message passing applications. Their performance was improved by a factor of as much as 10, reducing the applications slowdown to 1.5 in the worst case. However, the implications that the gain in message passing applications produced in the local workload were not studied.

In [20], a mathematical framework for analyzing, interpreting, and extending

the extensive simulation results for Implicit coscheduling reported in the literature was presented. It provided arbitrary distributions for message arrival to distributed processes, in contrast to previous results, where only uniform ones where taken into account.

Anglano [21] evaluated different Implicit coscheduling algorithms. Basically, these algorithms were obtained as a result of combining different coscheduling strategies based on the spin-block technique explained above. Strategies were divided between: spin indefinitely, classical block, spin-block (as in [16, 17, 18]) and spin-yield (after spinning, the priority of the receiving process is decreased). Simulation results showed the importance of coscheduling a receiving process as soon as possible. But if I/O-bound (or interactive) jobs are present, more complex strategies are required in order to achieve satisfactory performance for parallel jobs. The results concerning the performance of local job performance showed that the worst strategies were those based on pure spin.

Furthermore, Gupta et al. showed similar results in [9]. They had already concluded that the use of non-blocking (spinning) synchronization primitives would result in even worse performance under moderate multiprogrammed loads because, while the extra context switches are avoided, the spinning added an important overhead. Also, the above mentioned work by Feitelson and Rudolph in [11] corroborates these conclusions.

Anglano reconsidered the situation in [22], motivated by the fact that spinning does not allow performance of parallel applications to be achieved when various of them were executed simultaneously with local workload. In that work, he proposed a ticket distribution between users and processes. It is based on distributing execution tickets to the users depending on the global shared resource requirements. The strategy moved thus from implicit to explicit-control coscheduling. Experiments were performed by simulation. Note that in Cluster computing, resource requirements change rapidly, continuously and drastically, so the management of the global user requirements, can be very inefficient.

1.4 Motivation and Objectives

1.4.1 Motivation

The necessity and the main goal for coscheduling distributed tasks in Cluster systems is explained next by means of an example.

Let three intensive message-passing distributed applications be A_1, A_2 and A_3, and a Cluster C made up of three identical machines, M_1, M_2 and M_3. Each distributed application is composed of two tasks (i.e. $A_1 = \{A_{1,1}, A_{1,2}\}$, $A_2 = \{A_{2,1}, A_{2,2}\}$, $A_3 = \{A_{3,1}, A_{3,2}\}$) (see Fig. 1.3), which are also mapped into three different machines. The Mapping Assignment (MA) for each distributed application is: $MA(A_1) = \left\{A^1_{1,1}, A^2_{1,2}\right\}$, $MA(A_2) = \left\{A^2_{2,1}, A^3_{2,2}\right\}$, $MA(A_3) = \left\{A^2_{3,1}, A^3_{3,2}\right\}$, where $A^k_{i,j}$ means task j composing distributed application i is assigned to machine k.

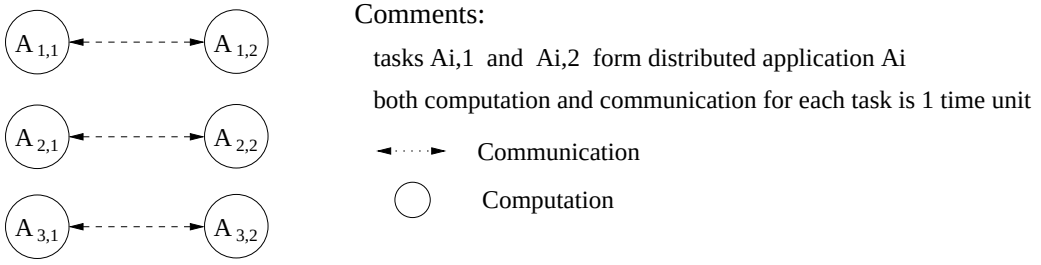


Figure 1.3: Applications A_1, A_2 and A_3

It is assumed that when a distributed task cannot communicate, it only performs its computations and will need another Time Slice for communication. However, if the distributed task can communicate with its correspondent, it can perform computation and communication at the same time, because the resources to do this are different. For simplicity reasons we suppose that each task of every application needs one unit of time for computation and another time unit for communication.

It is also supposed that there are local tasks in each machine: $\{LOCAL^1_1, LOCAL^1_2\}$ in machine M_1, $\{LOCAL^2_1\}$ in machine M_2 and $\{LOCAL^3_1, LOCAL^3_2\}$ in machine M_3. In this case, notation $LOCAL^k_i$ indicates local task i executing in

machine *k*. The local tasks in each machine execute jointly with the distributed ones, but depending on the coscheduling policy, their execution (return) time can vary significantly.

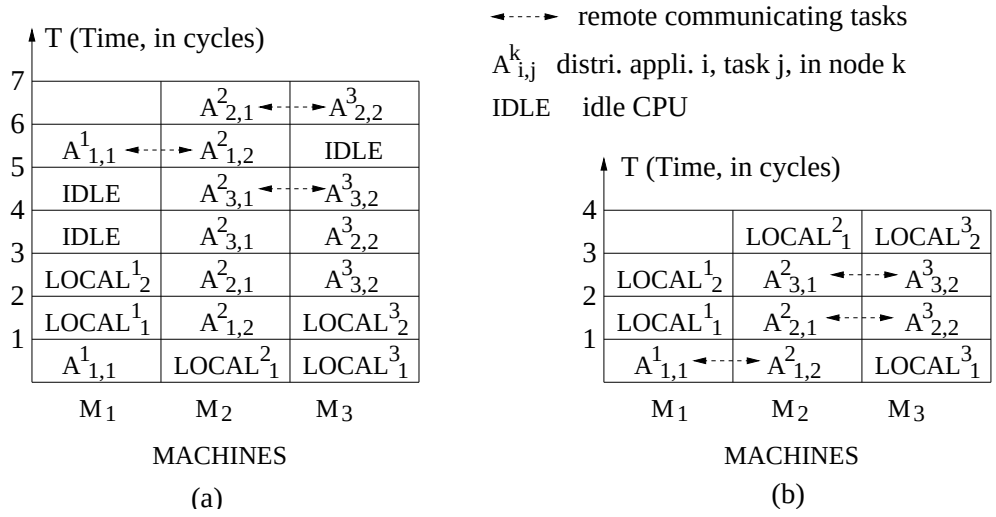


Figure 1.4: Coscheduling benefits.

If any kind of coscheduling between communicating tasks is applied (and as Fig. 1.4(a) shows), the executing time for each distributed application ($T(A_i)$), is $T(A_1) = 6$, $T(A_2) = 7$ and $T(A_3) = 5$, and its averaged execution time (TD_{mean}) is $TD_{mean} = 6$.

On the other hand, a coscheduling technique based, for example, on increasing the scheduling priority of distributed applications is used in Fig. 1.4(b). In this case, the results obtained are $T(A_1) = 1$, $T(A_2) = 2$, $T(A_3) = 3$ and $TD_{mean} = 2$. The correspondents of each application are coscheduled at the same time in Fig. 1.4(b), and for this reason an important gain in the distributed applications is produced.

Note that in the example, when the coscheduling technique is applied, local tasks, denoted as $LOCAL_1^2$ (in machine M_2) and $LOCAL_2^3$ (in machine M_3), are delayed by three and two cycles respectively. To simplify the problem, time is in cycle units. In real time-sharing systems, each cycle corresponds to some variable amount of time (often named *Time Slice*). In turn, a *Quantum* is the maximum *Time Slice* a process is allowed for executing uninterruptedly in the CPU. Follow-

ing the example, if one cycle is equivalent to a *Quantum*, it would mean that each process spends all the maximum allowed time executing inside the CPU each time it is dispatched. In Linux, the value of the Quantum is 210 ms, and if it is used as a reference in this example, $LOCAL_1^2$ would be delayed 630 ms. In this case, the response time (one of the most important system performance indices for local applications) of $LOCAL_1^2$ is not increased excessively, but real coscheduling techniques may even cause starvation of the local (even distributed) tasks.

This simple example shows us the motivation for implementing coscheduling techniques, but the observed gain is not always reached. There are many other factors to be considered, for example, how to obtain the communicating frequency, avoid local tasks starvation and so on.

1.4.2 Objectives

In Cluster or NOW systems, combining parallel and sequential workloads with reasonable performance for both computation kinds is an open research goal. Or in other words, how to build a NOW that runs parallel programs with performance equivalent to a MPP (Massively Parallel Processor) with the ability to execute sequential programs like a dedicated uniprocessor, is still an open question to be solved.

Furthermore, as mentioned at the beginning of this chapter, workstations in a NOW are normally underloaded. Thus, executing distributed applications in these systems will lead to making good use of the CPU idle cycles. Consequently, the use of NOWs in distributed processing is also justified.

Certainly, taking into account these previous comments, the Cluster system features and the definition of coscheduling, the goals of this thesis will be more easily understood.

Before going into detail on the objectives of this project, the main and secondary goals are described in general terms.

Main objective: *to execute distributed and local (or user) applications efficiently in a Cluster of NOW system by applying coscheduling techniques*. Efforts are directed towards understanding coscheduling mechanisms in a Cluster

or NOW system when distributed jobs are executed jointly with local workloads, balancing parallel performance against the local interactive response.

To carry out this objective, new coscheduling techniques are presented in this thesis.

Furthermore, various coscheduling mechanisms proposed in the literature were tested by simulation and some of them were also evaluated in a real Linux Cluster. Comparison of their performance in the same simulator system and in a real Cluster was the *second objective* of this thesis.

These two generic objectives are summarized in the design and implementation of two new coscheduling techniques (one with explicit and another with implicit control), which are detailed below.

Explicit-Control: the realization of a real implementation of explicit coscheduling in a NOW is the first objective. We will attempt to ensure that user of the parallel machine regularly has all the computing power of the NOW available during a short period of time. The expectation in doing so is to increase the performance of message-passing distributed applications without excessively damaging the local one.

Our contribution in this field is based on the work carried out in [37, 38, 39], where an explicit-control coscheduling algorithm was implemented in a PVM-Linux Cluster.

Implicit-Control: the realization of a Predictive coscheduling algorithm in a non-dedicated Cluster is the second specific aim of this thesis. It consists of coscheduling the sets of correspondent -recent communicated- processes at the same time. To do so, the correspondents must be known in advance, but in distributed (i.e. Cluster) systems, this information is very difficult to obtain.

Our proposal is based on the assumption that high receive-send message frequencies imply that potential coscheduling with remote processes is met. We propose to identify the correspondent processes in each node by taking into account the receive-send message frequency. Normally, the most recent communicating processes need to communicate again soon, so their

scheduling priority must be increased without damaging the local task performance excessively. Not only is message receiving considered -as in [16, 17, 18, 19, 28, 29]-, since tasks performing only message receiving do not justify the need for coscheduling. This is also true for only sending processes. This way, an approximation to Predictive coscheduling is performed.

Also, as in explicit-control, the contributions presented in implicit-control are based on some previously presented work in [39, 41, 42, 43].

1.5 Overview

The remainder of this document is organized as follows.

In chapter 2, two new coscheduling models are presented. They are defined and analyzed separately. First, in section 2.1, an explicit-control mechanism is presented, named also Explicit.

A new Predictive coscheduling mechanism which follows an implicit-control policy is presented in section 2.2. It includes an implicit-control coscheduling model (named CMC: Coscheduling Model for non-dedicated Clusters). Sections 2.2.1 and 2.2.2 respectively describe the notation used and a new set of CMC related performance metrics. Their applicability in a generic scheduler is introduced next in section 2.2.3. Also, a Predictive coscheduling algorithm based on this model is developed in section 2.2.4.

Finally, Dynamic coscheduling, a variant of the Predictive coscheduling is explained in the last section (sec. 2.3) of chapter 2.

In chapter 3, various implemented coscheduling prototypes are presented.

The first prototype presented (section 3.2) is the DisTributed coScheduler (DTS), an environment that implements the explicit-control model presented in chapter 2, section 2.1. A variation of DTS, which always assigns more scheduling priority to distributed tasks, named High Priority Distributed Tasks (HPDT), is explained in section 3.3.

Next, and also in chapter 3, three implicit-control prototypes are presented:

- Implicit (section 3.4): implements a variation of the spin-block technique

which was previously developed by other authors in [16, 17, 18, 19]. It was also studied in [20, 21].

- Predictive (section 3.5): implements the Predictive mechanism presented in section 2.2.

- Dynamic (section 3.6): implements a Predictive variation, named Dynamic, presented in section 2.3.

The performance of the coscheduling models are evaluated by simulation in chapter 4. Extensive performance analysis, based on the presented CMC model and metrics demonstrate their applicability in Cluster computing. Also, the proposed coscheduling algorithms are evaluated and compared with other coscheduling policies previously presented in the literature.

In chapter 5, the behavior of the implemented prototypes is checked in a Cluster made up of 8 nodes, by means of measuring the execution and communication times on two distributed application kinds: synthetic and parallel benchmarks (kernel benchmarks from the NAS [56], and the PARKBENCH low level benchmarks [57]).

Finally, the conclusions and future work are detailed in the last chapter of this document (chapter 6).

Chapter 2

Coscheduling Techniques

In this chapter two new coscheduling techniques with different control trends (explicit and implicit) are presented.

First of all, coscheduling with explicit-control mechanisms is studied. In doing so, an explicit coscheduling method is presented. The main idea is to split execution time in each node between distributed and local tasks. Theoretically, the model will provide distributed (and local) applications with a means for executing in the Cluster as if it were entirely dedicated to them. Due to the broad range of existing communicating patterns and synchronization needs of distributed applications, three different operation modes have been provided for this explicit scheme.

One important observation is that, as mentioned in chapter 1, gang scheduling allows guarantees about the performance to be given. In the explicit model we present, the performance is guaranteed between the overall group of distributed applications executed simultaneously in the Cluster. In some sense, this is accomplished by assigning them periodically their own execution interval. We consider this property an important advantage over environments where performance implications cannot be determined in advance, as for example plain Linux based Clusters or even ones that implement some kind of implicit-control mechanism.

Coscheduling techniques based on implicit-control have also been studied. In implicit-control, the major drawbacks (basically the added overhead) derived from the controlling and the managing needs of explicit-control coscheduling are saved.

Moreover, one additional explicit-control drawback to bear in mind is related to fault tolerance: crashing of controlling nodes or processes could cause a failure of the overall system. The study and proposal of different alternatives will also occupy our attention.

Predictive coscheduling is the second method presented in this thesis. Its behavior, as in other implicit-control methods, is autonomous. That is, it does not depend on local or remote controlling processes. It is based on assigning more scheduling priority to processes with higher communication frequency.

To our knowledge the presented Predictive model is the first one that attempts to coschedule the correspondent processes (those which have recently communicated between them). So an accurate study of this has been performed. In doing so, a model, named CMC (Coscheduling Model for non-dedicated Clusters), which provides a means for developing and testing the performance of implicit-control techniques in time-sharing based systems is presented. Multiple concurrent execution of distributed applications is supported by this model, but the clustering nodes are restricted to uniprocessors. A proposal for a new algorithm based on the CMC model will occupy the last part of the Predictive presentation.

Finally, a variation of the Predictive method, named Dynamic coscheduling is presented. Unlike Predictive, the Dynamic method is based only on message receiving, like the Dynamic model presented by Sobalvarro in [28] and [29], so it has been named the same.

All these mechanisms were implemented in a real Cluster system, as described in chapter 1, section 1.1.1. The realization of the different prototypes or environments which implement them are explained in the following chapter (chapter 3).

2.1 Explicit Coscheduling

The proposed explicit coscheduling technique is based on assigning an execution period to the distributed tasks and another to the interactive ones in the overall active nodes of the NOW. The set of active nodes (where the explicit mechanism is activated) are denoted by VM (Virtual Machine). In such a way that, the CPU time of each active node is split into two different periods, the *Parallel Slice* (*PS*) and the *Local Slice* (*LS*).

Algorithm 1 Explicit Scheduler

```
  assign high scheduling priority to distributed tasks
sync_point:
  while (distributed tasks ) do
    set(PS)
      schedule distributed tasks during a PS interval
    set(LS)
      schedule local tasks during a LS interval
  endwhile
```

The main work performed in the execution of explicit coscheduling is located in one module named *Explicit Scheduler* (see Algorithm 1). The function of each *Explicit Scheduler* (as can be seen in Algorithm 1) is the dynamic variation of the amount of CPU cycles exclusively assigned to execute distributed tasks (*PS: Parallel Slice*), and the amount of time assigned to local tasks (*LS: Local Slice*). *IP* (Iteration Period) is defined as $IP = PS + LS$.

Figure 2.1 shows a Gantt diagram of the behavior of the explicit mechanism: each time the *Explicit Scheduler* (in each VM node) is executed, it concedes the CPU alternatively to the distributed (for a *PS* unit of time) tasks and to the Local ones (for an *LS* unit of time).

Although the figure depicts the *Explicit Scheduler* execution intervals, it can seem that it adds an important overhead in the system. These intervals are really very small. They usually have similar length as the CS (elapsed time in doing a context switch). They have been drawn in this form to help in the understanding of the explicit mechanism. *SI* is the Synchronization Interval and will be defined

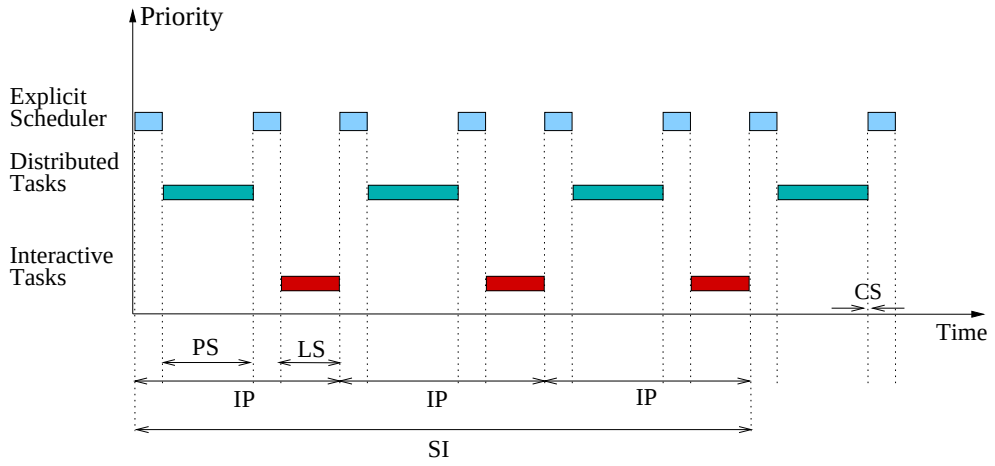


Figure 2.1: Explicit environment behavior

later when the different execution modes of the explicit mechanism are introduced.

The *Explicit Scheduler* is meant for executing in the user space, and behaves as an additional Scheduler which controls the execution times of distributed and local applications. First of all, each *Explicit Scheduler* assigns an scheduling priority to the distributed tasks which is higher than the local ones. Thus, the activation of the *PS* (*LS*) period can be performed for example, by simply waking up (suspending) the distributed tasks. Note that the form of doing so will depend on the DCE used and the underlying o.s. residing in each VM node.

The tasks composing a distributed application are composed basically of (a) CPU or (b) message passing intensive phases. In the first case, when distributed tasks are mainly performing computation, it is unnecessary to synchronize the tasks.

On the other hand, in the second situation (case (b)), as was already pointed out in [16], the synchronization between the communicating tasks can increase the global performance of the method. In our case, that will imply the synchronization of the *PS* (even *LS*) over the VM. For this reason, the explicit model has three different operating modes: STATIC, BALANCED and DISTRIBUTED.

In the presentation of the different modes, it is supposed that there are three types of modules, the *Explicit Scheduler* (already explained), the *Load* and the *Console*. As in the *Explicit Scheduler*, the *Load* is composed of distributed pro-

cesses running on each active workstation. The *Load* processes collect the interactive load on every workstation. The *Console* is executed in one VM node (named *master*) and is responsible for managing and controlling the overall system.

In the STATIC mode, the *PS* and *LS* periods are synchronized over all the distributed tasks. This synchronization is based on starting the *PS* interval at the same time in all the nodes forming the VM. This is accomplished by sending one synchronization message from the *Console* to the different *Explicit Schedulers*. The asynchronous reception of such message causes an unconditional jump of the corresponding *Explicit Coscheduler* to the beginning of the main loop, at the *sync_point* label (see Algorithm 1). In addition, the BALANCED mode sets the *PS* and *LS* intervals according to the mean local workload of the Cluster. The DISTRIBUTED mode is meant for situations where distributed tasks do not synchronize/communicate between each other. That is, the distributed tasks are mainly CPU bound. The *PS* and *LS* are updated according to the workload in each local node.

Next, the three operating modes are explained separately in more detail.

2.1.1 STATIC Mode

This mode consists of synchronizing the parallel and local periods of each workstation every *SI* (Synchronization Interval) unit of time. *PS* and *LS* are determined in the *Console* at the beginning of the startup phase of the Explicit environment. Next, they will be sent to the overall *Explicit Schedulers* of the nodes forming the VM.

This mode is based on synchronizing the PS (even the LS) interval throughout the overall VM. To do so, every *SI* unit of time, the *Console* sends a broadcast Datagram message (containing the PS and LS intervals) to every *Explicit Scheduler*, which in turn set their own *PS* and *LS* (equal in the overall VM). We need not delay the master due to message delivery latency because the slave side of the master will also receive a synchronization message.

At the end of each SI, the workstations remain synchronized for a long period of time. So, the *SI* interval would have to be much greater than *IP* ($SI \gg IP$). By assuming that *SI* is large enough, the added overhead in synchronizing *PS* and *LS*

can be depreciated. Consequently, no more considerations on optimal values for *SI* nor its associated overhead are made.

The search for optimal values for the intervals *IP*, *PS* and *LS* is investigated later in the experimentation chapter (chapter 5).

Although the explicit mechanisms may generate an additional overhead, the global performance of the system can be increased using this model. However, considering the workload in each workstation, as in the following mode (BALANCED), the expectation of performance gain also increases.

2.1.2 BALANCED Mode

In the BALANCED mode, in addition to the synchronization of the *PS* and *LS* intervals provided in the STATIC mode, we are also interested in varying these periods according to the local load average of the NOW. If the mean workload is low, it seems very reasonable to enlarge *PS* (decrease *LS*) in each node. On the other hand, if the mean workload is high, it would be necessary to decrease *PS* (increase *LS*). Also, as in the STATIC mode, the same *PS* and *LS* intervals have to be kept equal and synchronized in the overall VM.

An additional feature of the BALANCED mode is that it has been designed for use in heterogeneous systems.

According to the work done by Ferrari [54], the load is obtained based on the Ready to run Queue (RQ) length. The length of each RQ node is sampled each second by the *Load* processes. Every Load Interval ($LI = 10$s), the ten-second average RQ length (q_i), only in the Local Slice (*LS*), is obtained. Next, the *Load Index*, denoted as Q_i (the index used to obtain the average RQ length) is computed, and a message containing it is sent to the *Console* if $\left(\left|Q_i^j - Q_{i-1}^j\right| \geq Load\_Treshold\right)$, with a default value for $Load\_Threshold = 1$. Exponential smoothing is used to compute the *Load Index*, defined as follows:

$$Q_i = Q_{i-1}e^{-P} + q_i(1 - e^{-P}),\ i \geq 1,\ Q_0 = 0, \tag{2.1}$$

where Q_{i-1} is the last computed *Load Index*, q_i is the ten-second average RQ length and $P = \frac{1}{LI*N}$. Taking into account the studies done by Ferrari [54], an *LI* of 10s and *N* of 6 were chosen. This way, exponential smoothing over the last

$LI * N = 60$ seconds (= 1 minute) is used to compute Q_i.

When *Load* collects q_i, as it is supposed that this operation is performed in the *LS* interval, the distributed tasks are stopped, waiting out of the Ready Queue. For this reason, the distributed tasks are not computed. In another situation, for example, systems where the priority of distributed tasks is increased and decreased periodically, the need to distinguish between distributed and interactive tasks would add a great overhead to the system.

In the reception of the Load indices from the active nodes or after a timeout, the *Console* computes the *Relative Load Average* (RLA), which is used to fix the parallel and local slices on each workstation. The RLA is defined as follows:

$$RLA = \frac{\sum_{j=1}^{NW} \frac{Q_i^j}{W_j(A)}}{NW} \tag{2.2}$$

where Q_i^j is the *load index* of workstation j, NW is the number of workstations in the VM and $W_j(A)$ is the *power weight* of workstation j. Initially, in the startup phase of the BALANCED mode, all the active nodes send their respective Load indices to the *Console.*

The *power weight* [50] of one machine M_i with respect to one application of size A is defined as follows:

$$W_i(A) = \frac{min_{j=1}^{NW}\{T_j(A)\}}{T_i(A)},\ j = 1...NW \tag{2.3}$$

where $T_i(A)$ is the execution time to solve the serial application of size A in a dedicated workstation M_i.

Note that this method for obtaining the RLA is designed to be used in heterogeneous systems. It is worthwhile to point out that two machines with the same Q_i and with different power weight will not be equally loaded. However, its quotient (see formula 2.2) gives a number which sort the load between the different machines of the VM. It represents a more accurate approximation of the workload of each node with respect to the others. For example, suppose a serial application of size A and two nodes M_m and M_n, with $Q_i^m = 2$, $W_m(A) = 1$, and $Q_i^n = 1$ and $W_n(A) = 0.5$. Then, $Q_i^m/W_m(A) = 2$, and $Q_i^n/W_n(A) = 2$. This simple example shows how this pair of workstations are equal loaded. Note that $Q_i^m = 2 * Q_i^n$, but

also $W_m(A) = 2 * W_n(A)$, so they are evenly loaded because as M_m has double computing capacity it also requires the double load to be as loaded as M_n. In our case, the Cluster is homogeneous, thus for any serial application A, $W_j(A) = W_j = 1$, where j = 1...NW.

RLA	***LS***	***PS***
$0 \leq RLA \leq 0.5$	10	90
$0.5 < RLA \leq 1$	20	80
$1 < RLA \leq 1.5$	30	70
$1.5 < RLA \leq 2$	40	60
$2 < RLA \leq 2.5$	50	50
$2.5 < RLA \leq 3$	60	40
$3 < RLA \leq 3.5$	70	30
$3.5 < RLA \leq 4$	80	20
$4 < RLA$	90	10

Table 2.1: Relation between *RLA*, *PS* and *LS*

Table 2.1 (which shows the relation between the *RLA*, *PS* and *LS*) is used to compute *PS* and *LS*. The values of *PS* and *LS* shown in the table are percentages of the *IP* period. For example, if $IP = 1$s and $RLA = 0.5$, *LS* and *PS* will be set respectively to 100ms and 900ms.

Note that the range of values in table 2.1 depends on the chosen interval for which a variation in the RLA is significant enough to change the *PS* and *LS* intervals. A variation in the load lower than 0.5 can hardly be appreciated and will add excessive overhead. On the other hand, higher values would decrease the effectiveness of the model. An interval of 0.5 was therefore chosen.

Finally, the *Console* sends *PS* and *LS* to all the *Explicit Schedulers* modules (if $|RLA - RLA_{prev}| > 0.5$, where RLA_{prev} is the previous RLA) by sending a broadcast message (for example). Broadcast delivery is a good method of doing so because multicasting or sending as many messages as nodes making up the VM would add an excessive overhead.

2.1.3 DISTRIBUTED Mode

This explicit mode is designed for situations in which the message exchange between tasks forming distributed applications is very low, even when the communication is null. In these cases, no global adjustment of the *PS* and *LS* intervals need be performed. Also, there is no need for synchronizing these intervals between the VM nodes. The only factor to take into account is the efficient sharing of the CPU between the distributed and local tasks in each workstation, so that each workstation sets the *PS* and *LS* intervals according to its own local workload.

The *Load* module in each node computes and then sends the *PS* and *LS* values to the local *Explicit Scheduler* according to Table 2.1 too, but in this case, RLA is substituted by Q_i (the Load Index). For example, if $IP = 100$ms and $Q_i = 0.75$, *LS* and *PS* will be set respectively to 20ms and 80ms.

In this mode, apart of assigning more scheduling priority to distributed tasks, the *Explicit Scheduler* will be only responsible for setting the *PS* and *LS* periods on receiving them from the Load module (residing at the same node) and then starting (when the *LS* period has elapsed) and stopping (when the *PS* period has elapsed) the distributed tasks.

2.1.4 Explicit Synchronization

Algorithm 2 resumes the synchronization performed in the different explicit modes. The algorithm (written in pseudo-code), shows the interaction between the modules residing in each VM node ($Load_{\mathrm{j}}$ and $Explicit\ Scheduler_{\mathrm{j}}$) and *Console*, which is located in the *master* node.

Algorithm 2 Explicit Synchronization Algorithm.

Load$_j$: $\forall\, M_j \in VM$
$Q^j_{i-1} = 0$
Each LI interval **do**
if (MODE ! = STATIC)
compute (q^j_i);
compute (Q^j_i);
if (MODE == DISTRIBUTED)
compute (*PS*&IS);
send (Explicit Scheduler$_j$, *PS*&*LS*);
endif
else
if $\left(\left|Q^j_i - Q^j_{i-1}\right| \leq Load\_Threshold\right)$ **send** (Console, Q^j_i); **endif**
endelse
endif
enddo

Console: Node Master
if (MODE == STATIC)
Each SI interval **do broadcast** (*PS*&*LS*);
endif
if (MODE == BALANCED)
do
while (not timeout) **do**
for each $M_j \in$ VM **async_receive** (Q^j_i);
compute (RLA, *PS*&*LS*);
if ($|RLA - RLA_{prev}| > 0.5$)
broadcast (*PS*&*LS*); $RLA_{prev} = RLA$;
endif
while (true)
endif

Explicit Scheduler$_j$: $\forall\, M_j \in$ VM
async_receive (*PS*&*LS*);
set *PS*&*LS;*
if (MODE ! = DISTRIBUTED)
goto sync_point
endif

2.2 Predictive Coscheduling

The proposal of a Predictive coscheduling mechanism for non-dedicated Clusters and with an implicit-control trend will occupy our attention next.

The Predictive technique is formalized by the definition and design of a model for Cluster systems, named CMC (Coscheduling Model for non-dedicated Clusters). This model is made up of a set of performance metrics and the Local Coscheduler. A Predictive Coscheduling Algorithm (PCA), based on the CMC model is also presented.

First of all, various assumptions on the system in which to apply the model must be made:

1. This model assumes that not necessarily all the nodes in a non-dedicated Cluster or NOW must be under the control of this coscheduling scheme. This is an important difference with respect to the previously presented explicit model, where all the nodes of the Cluster making up the VM were explicitly managed and controlled by the explicit mechanism.

2. The distributed applications are composed of a suite of tasks which are already mapped in the nodes making up the Cluster.

3. Currently in Cluster computing, tasks making up the distributed applications are executed in each node as simple local tasks. For this reason no initial assumptions are made about knowledge of the distributed tasks of each node composing the distributed applications (i.e. composition, identification and mapping).

4. Each node in the Cluster is supposed to be uniprocessor, with a time-sharing operating system. It is also assumed that the local CPU Scheduler in each node deals with the ready to run tasks list, the *Ready Queue* (RQ) and has an appropriative Round-Robin (R-R) policy, with a variable time slice (TS: requesting CPU execution cycles) for each task. Note that this scheduling coincides (with some specific characteristics) with most real time-shared o.s.'s (i.e. Solaris, Linux and so on).

The CMC model provides the following features:

- Multiple concurrent execution of distributed applications is supported by this model.
- Dynamic identification of distributed processes: remote communicating processes (which are considered as distributed ones) are identified at run-time.
- Coordinated scheduling: communicating processes from the same parallel job are scheduled almost simultaneously across a set of workstations, thus achieving its performance. Coordination of processes should minimize the waiting time for messages.
- Local performance maintenance: participating in the Cluster should not degrade excessively the performance of the workload of a single node (local or user jobs). Also, the response time for interactive local jobs does not drop excessively.
- Autonomy: each node maintain control over its own actions, participating in the system in a non-imposed manner.
- Reconfigurability: workstations can join and leave the Cluster dynamically without restarting the system services.
- Reliability: the failure of any node of the system does not affect the behavior of the system.

2.2.1 Notation

In a machine with a time-sharing o.s., a task may be in different states (ready to run or simply ready, blocked, etc ...). A ready task l, meaning in the RQ, will be denoted as $T[l]$. However, if task l is not ready, it will be referenced simply as l, without the T prefix. The method used to uniquely identify a task without depending on its state, is by means of the task identifier function (tid). For example, $T[l]$ and h are the same task if $tid(T[l]) = tid(h)$.

Let a ready to run task be l ($T[l]$), some l-related basic notation is defined as follows (time is in cycles):

- $T[l]$: task l -can be local or distributed- of the RQ. Special RQ tasks are $l = 0$ (*top*) and $l = \infty$ (*bottom*). "*top*" task is the currently executing task in the CPU and "*bottom*" is the latest one to be executed.

- $T[l].c$ ($T[l].tc$) -*c*: cycles; *tc*: total cycles-: executing cycles for task l, since the last time such a task reached the RQ (*c*), or since the start of the execution (*tc*).

- $T[l].pco$ -*pco*: potential coscheduling-: boolean variable that informs of potential coscheduling of the task l. When a task enters the RQ and its current communication frequency is higher than 0, its associated field *pco* is activated. This fact tells that such a task is a candidate for coscheduling (and thus needs to be scheduled as soon as possible).

- $T[l].co$ ($T[l].tco$) -*co*: coscheduling, *tco*: total coscheduling-: number of coscheduled cycles (= executed cycles) of task l when potential coscheduling was met (the field *pco* was activated on insertion into the RQ), since the last time such a task reached the RQ (*co*), or since the start of the execution (*tco*). The coscheduling condition, as will be seen in section 2.2.4, is the most important cause for increasing the priority (overtaking other tasks of the RQ) of the distributed tasks.

- $T[l].th$ ($T[l].tth$) -*th*: thrashing, *tth*: total thrashing-: number of lost coscheduling cycles (not executed cycles) of task l when potential coscheduling was met, since the last time such a task reached the RQ (*th*), or since the start of the execution (*tth*).

- $T[l].de$ -*de*: delay-: number of times that task l has been overtaken in the RQ by another task due to a coscheduling cause, since the last time such a task reached the RQ.

- $T[l].d$ ($T[l].td$) -*d*: delay, *td*: total delay-: number of delayed cycles for task l due to a coscheduling cause, since the last time such a task reached the RQ (*d*), or since the start of the execution (*td*).

- $T[l].ovt[MCO]$: array of *MCO* (Maximum number of Coscheduling Overtakings) task identifiers (*tid*). When a task *h* overtakes task *l* (due to a coscheduling reason) in the RQ, its corresponding *h* identifier (*tid*(*h*)) is saved in one of the *l MCO* fields.

If a task is not ready, only fields *tc*, *td*, *tco* and *tth* are used, because they save information about the overall execution and not only since the last time the task reached the RQ. For example, the field *tc* of task *h* will be referenced as *h.tc*.

Let a Cluster be C, made up of *n* nodes ($C = \{N[k]\}$, $k = 1...n$). Also, for referring field tc of task l of a node k, we will use notation $N[k].T[l].tc$ for a ready task $T[l]$ and $N[k].l.tc$, otherwise.

In this model, it is supposed that incoming (out-going) messages to (from) a Cluster node $N[k]$, are buffered in a Receiving Message Queue, *RMQ* (Sending Message Queue, *SMQ*). This is a non-arbitrary assumption. The communication system of most time-shared environments has some sort of queue of this type. For example (as is explained in more detail in chapter 3, section 3.1.1), in a PVM-Linux environment, there is a queue named *pvmrxlist* in the PVM [30] level (in the user space), which buffers receiving fragments (the PVM transmission unit) and another one that buffers sending fragments, *txlist*. There are two queues in the kernel level (in the system space), named *receive_queue* (which buffers receiving packets: the socket transmission unit) and *write_queue* (which buffers sending packets). These queues have the same behavior and functionality as RMQ and SMQ. The choice of the level at which to apply the coscheduling techniques is an implementation decision that does not depend on the model.

The Predictive coscheduling algorithm -proposed in this project (see section 2.2.4)- is based on communication frequency. Taking into account the existence of the above mentioned message buffers, the following information can be obtained (and used later in the proposed algorithms) for a task l:

- $T[l].cur\_freq_r$: current message receiving frequency.
- $T[l].cur\_freq_s$: current message sending frequency.
- $T[l].freq_r$: past message receiving frequency.

- $T[l].freq_s$: past message sending frequency.
- $T[l].freq$: sending and receiving frequency.

2.2.2 Performance metrics

The performance metrics defined in this section use the CMC notation defined in section 2.2.1. As all of them are computed for tasks in the RQ, we use notation $T[l]$ for referencing tasks, and $N[k].T[l]$ for referencing task l of node k. These metrics can be used to measure performance of a coscheduling algorithm, and in particular a Predictive coscheduling algorithm like the one proposed in section 2.2.4, and are defined as follows:

- Task Delay ($TaskD(T[l])$). Informs of the delay introduced into tasks due to a coscheduling policy. For example, if the ready task $T[l]$ was overtaken by another ready task $T[h]$ due to a coscheduling policy, the execution time of $T[h]$ will be added to task $T[l]$ as a delay. This will be very useful for measuring the impact of a coscheduling policy on local (and also distributed) tasks. This information is maintained by the model in the *"td"* field of the corresponding task, that is:

$$TaskD(T[l]) = T[l].td \tag{2.4}$$

- Node Delay ($NodeD(N[k])$). Delay introduced into a node k.

$$NodeD(N[k]) = \sum_{l} T[l].td \tag{2.5}$$

- System Delay ($SystemD$). Delay introduced into the overall system.

$$SystemD = \sum_{k}\sum_{l}(N[k].T[l].td) \tag{2.6}$$

- Task Coscheduling Degree ($TaskCoDe(T[l])$). Provides information about the good performance of a coscheduling technique. For example, if a high-frequency task is always scheduled first, this task would have a good *Task-*

CoDe. It is defined as the relation between the total coscheduled cycles ($T[l].tco$) when potential coscheduling was met (field $T[l].pco$ was activated) and all the possible coscheduling ones ($T[l].tco + T[l].tth$) for task $T[l]$.

$$TaskCoDe(T[l]) = \frac{T[l].tco}{T[l].tco + T[l].tth} \tag{2.7}$$

- Node Coscheduling Degree (*NodeCoDe*($N[k]$)). It is the average *TaskCoDe* metric of the overall tasks in a node.

$$NodeCoDe(N[k]) = \frac{\sum_l T[l].tco}{\sum_l (T[l].tco + T[l].tth)} \tag{2.8}$$

- System Coscheduling Degree (*SystemCoDe*). Relation between coscheduled cycles and all the possible coscheduling ones in the overall system.

$$SystemCoDe = \frac{\sum_k \sum_l N[k].T[l].tco}{\sum_k \sum_l (N[k].T[l].tco + N[k].T[l].tth)} \tag{2.9}$$

- Task Thrashing, Node Thrashing and System Thrashing Degree (*TaskThDe*($T[l]$), *NodeThDe*($N[k]$) and *SystemThDe*). In contrast to the **CoDe* metrics, they provide information about poor use of the potential coscheduling by a coscheduling technique. In this case, $TaskThDe(T[l]) = 1 - TaskCoDe(T[l])$, $NodeThDe(N[k]) = 1 - NodeCoDe(N[k])$ and $SystemThDe = 1 - SystemCoDe$.

To sum up, a good coscheduling algorithm should maximize the **CoDe* ($* = \{Task, Node, System\}$) metrics and minimize the **D* and **ThDe* ones. This is the main goal of the Predictive algorithm presented in section 2.2.4.

Note that these metrics do not consider the time spent by a process waiting for an event to occur (normally in blocking states). The coscheduling performance mainly depends on a good choice of the task to be dispatched to the CPU (among all those which form the RQ) in each scheduling phase. Thus, blocking times when processes are, for example, waiting for messages, I/O or synchronization events are discarded, which in addition may depend on other factors, such as network and system latencies, and in general of the underlying hardware resources.

2.2.3 Local Coscheduler (LC)

Algorithm 3 shows the pseudo-code of the Round-Robin Local Coscheduler (LC) proposed for a time-shared o.s. of a Cluster node $N[k]$.

Algorithm 3 Local Coscheduler (LC) of node $N[k]$.

```
1 do forever
2   if (T[0] ≠ NULL)
3     dispatch(T[0]);
4   ACCOUNTING:
5     T[0].{c,tc} += exe_time;
6     if (T[0].pco) T[0].{co,tco} += exe_time; endif;
7     for (i=0; T[i+1] ≠ NULL; i++)
8       if (T[i+1].pco) T[i+1].{th,tth} += exe_time; endif;
9       if (T[i+1].ovt[j] == tid(0) for any j = 0..MPO-1)
10        T[i+1].{d,td} += exe_time;
11      endif;
12    endfor;
13    T[0].d = 0; T[0].c; T[0].co = 0; T[0].th = 0; T[0].de = 0;
14    for (j=0; j<MPO; j++) T[0].ovt[j] = NULL; endfor;
15    if (T[0].cur_freq==0) T[0].pco=false; endif;
16    delete_RQ (T[0]); insert_RQ (T[0]);
17  endif;
18 enddo;
```

This algorithm is very similar to a real Local Scheduler. It performs the same work as another typical scheduler of a time-sharing o.s., but with coscheduling capabilities. Other typical scheduler functions (like saving/restoring task contexts) that do not influence our model are not considered. Next, the LC operation is discussed.

We are interested in providing the Local Coscheduler (LC) jointly with the

Predictive Coscheduling Algorithm (PCA, presented in the following section), with the following three features:

- Priority based policy. The scheduler must dispatch processes with the highest potential degree to be coscheduled with remote tasks as soon as possible by assigning them more scheduling priority.
- To avoid local task starvation. Local tasks do not have to suffer excessive delay penalties. Response time and also return time have to be limited to a predetermined amount.
- No excessive overhead must be introduced. The added work the LC must carry out does not have to drop node (and system) performance excessively.

LC works only if the RQ is not empty ($T[0] \neq$ NULL). First of all, it dispatches (assigns the CPU to) the top task (line 3). When the executing task finishes execution, or its assigned time slice expires, or is preempted by another task, the execution in the CPU continues in line 4 (label *ACCOUNTING*), where some coscheduling accounting related to such task must be performed (lines 5 to 15).

In line 5, the execution time (*exe_time*) is added to the out-going CPU task. Note that this information is generally carried out by the scheduler of a time-shared o.s.. After doing coscheduling accounting in line 6, the next loop (lines 7-12) performs the modifications of the thrashed and delayed task fields when required.

In line 16, *delete_RQ* ($T[0]$) means that task $T[0]$ is first removed from the RQ, and then *insert_RQ* ($T[0]$) indicates that the top task $T[0]$, is inserted into the RQ again in accordance with the PCA algorithm. This way, a slight variation of the Round-Robin scheduling policy is implemented. This kind of movement in the RQ is performed when $T[0]$ has not completed its execution time requirement.

The task fields *d*, *c*, *co*, *th* and *de* must be initialized (line 13) before doing the movement, because their respective coscheduling information is "since the last time the task reached the RQ". Lines 14-15 perform the remaining initializations.

Note that the difference between fields *d*, *co* and *th* with their associated fields t* (*: *d*, *co* and *th*) is that they inform of the coscheduling policy behavior in each moment (or in each change in the RQ), and not in a determined period of time

(or the overall execution), as the t* ones do. Thus, they may be more valuable for obtaining on-time system performance. However, they do not serve for obtaining performance of a coscheduling mechanism based on the regular behavior of the distributed tasks. Consequently, the CMC metrics (sec. 2.2.2) were defined by using the fields t*.

Table 2.2 shows the CMC task field and the global variable initializations. These fields should be added to the Process Control Block, the information structure associated with each task in a real time-shared o.s.. All of them should be initialized when the task is created.

Table 2.2: CMC task field and global variable initializations.

$N[k].T[l]$												glob. var.
c	tc	pco	co	tco	th	tth	de	d	td	$ovt[0]$	$ovt[1]$	MCO
0	0	0	0	0	0	0	0	0	0	0	0	2

2.2.4 Predictive Coscheduling Algorithm

In this section, a Predictive coscheduling algorithm (named PCA) is proposed and discussed (Algorithm 4 shows its pseudo-code).

Algorithm PCA is implemented inside a generic routine (named *insert_ RQ*). This is the routine chosen to implement Predictive coscheduling because all the ready to run tasks must pass it before their scheduling. The *INITIALIZATION* section is the place where the different initializations (these may be global variables) are done. Note that an original *insert_RQ* routine should only contain one line of the form *insert (h,∞)*, which would insert task *h* at the bottom of the RQ.

From lines 5 to 14, we can see that task *h* is inserted in the lowest (the closest to the top, or even overtaking it) possible position of the RQ, depending on two conditions, the *STARVATION* and *COSCHEDULING CONDITIONS* (*S_C* and *C_C* respectively). In line 8, *tid*(*h*) is saved in one *ovt* field of each overtaken task. In line 9, the delay field (*de*) of the overtaken tasks is increased.

The routine *insert*, appends a process to the RQ. For example, *insert* (*h*,0) (line 4) means that task *h* will be inserted in the top when the RQ is empty, or else task *h*

Algorithm 4 Predictive Coscheduling Algorithm (PCA). $S\_C \equiv T[i].de < MCO$. $C\_C \equiv h.freq > T[i].freq$.

```
1  insert_RQ (task h)
2  INITIALIZATION
3  if (h.cur_freq ≠ NULL) h.pco =true; endif;
4  if (T[0] == NULL) insert (h,0); endif;
5  else
6      i = bottom;
7      while ((S_C) and (C_C) and (i ≠ −1))
8          tmp = T[i].de; T[i].ovt[tmp] = tid(h);
9          T[i].de ++;
10         i − −;
11     endwhile;
12     if (i ≠ −1) insert (h,i); endif;
13     else context_switch(h,0); endelse;
14 endelse;
```

will be added after task $T[i]$ in the RQ (line 12), where *i* depends on the Predictive conditions (*S_C* and *C_C*). When task *h* overtakes the executing task (case i $= -$ 1), then a context switch (*context_switch*(*h*,0), in line 13) in favor to task *h* must be performed. The top task $T[0]$ will be preempted from the CPU and moved to position 1 of the RQ (it will become $T[1]$). Next, the new executing task *h* (i.e. *tid(h)*) will become $T[0]$.

The *STARVATION CONDITION* (*S_C*) is equal to $T[i].de < MCO$. *MCO* is defined above as the Maximum number of Predictive Overtakes or, in other words, the maximum number of task overtakes due to a coscheduling policy. The aim of this condition is to avoid starvation of local tasks. For this reason, the inserting task *h* overtakes the ones whose field *de* ($T[i].de$) is lower than the global coscheduling variable *MCO*. Thus, starvation of local tasks is avoided.

The default value for *MCO* is 2, as higher values may decrease the response time (or interactive performance) of the local tasks excessively. The field *de* is

selected to do this comparison because its purpose is to count the number of overtakes since such a task last reached the RQ. This is much more significant for interactivity information than for example, tde (not used in the model), which would deal with overtakes since the task started execution. The tde field is more appropriate than de for controlling starvation of intensive CPU workload (less representative of real local workload).

The *COSCHEDULING CONDITION* (*C_C*) is $h.freq > T[i].freq$, where $T[i].freq$ indicates the frequency of messages addressed to (delivered by) the task $T[i]$. Its goal is to increase the scheduling priority of the tasks according to their respective communication frequency. That is, higher scheduling priority is assigned to tasks with higher receive-send messaging frequency.

Before defining the sending and receiving frequency (*freq*) for a task *h* (following the notation of section 2.2.1, it will be denoted as $h.freq$), the *h* receiving ($h.freq_r$) and sending ($h.freq_s$) frequencies must be defined:

$$h.freq_r = P * h.freq_r + (1-P) * h.cur\_freq_r, \tag{2.10}$$

$$h.freq_s = P * h.freq_s + (1-P) * h.cur\_freq_s, \tag{2.11}$$

where P is the percentage assigned to the past receiving and sending frequency ($h.freq_r$ and $h.freq_s$ respectively). $(1-P)$ is the current receiving and sending frequency percentage ($h.cur\_freq_r$ and $h.cur\_freq_s$ respectively). Note that in these formulas, $h.freq_r$ and $h.freq_s$ act as past frequencies on the right side. This means that each time these frequencies are obtained, they will be used later as past history for obtaining new frequency values.

Finally, the communication frequency for a task *h* ($h.freq$) is defined as the sum of the task sending and receiving frequency:

$$h.freq = h.freq_r + h.freq_s, \tag{2.12}$$

All frequencies must be computed at regular intervals (*FI*: Frequency Interval). Moreover, to obtain the current frequencies it would be necessary to gather the number of messages of the *RMQ* and *SMQ* each Sampling Interval (*SI*), which

in turn must be smaller than *FI* (*SI* < *FI*).

Note that this method for obtaining $h.freq$ is analogous to the one presented in section 2.1.2 for obtaining the Load Index, Q_i (formula 2.1), in the explicit BALANCED mode.

The choice of the *FI* and *SI* intervals will depend on the kind of distributed applications to be executed. Also it must be considered in doing so that the shorter these intervals are, the more overhead will be introduced. For example, large *FI* should be chosen for large enough distributed applications. *SI* depends on the communication behavior: regular communications favor the choice of large SI intervals; instead, irregular communication will force the election of short *SI* intervals. However, in the following chapter (where a Predictive prototype implemented in a PVM-Linux environment is explained), we suggest a more efficient form of obtaining those frequencies without having to consider the *FI* and *SI* intervals.

In contrast with any other coscheduling technique, coscheduling in the proposed Predictive scheme is applied not only to blocked receiving tasks (as [16, 17, 18, 19, 28, 29]), but coscheduling can be applied to each movement (i.e. the quantum of the executing task has expired) and in each kind of task insertion in the RQ (i.e. when a task is reawakened not only from a blocked receiving state, but also when it was blocked for I/O, or waiting for a synchronization event).

2.3 Dynamic Coscheduling

The Dynamic technique was first defined by Sobalvarro in [28]. The most interesting algorithm he proposed (the *equalizing* dynamic coscheduling algorithm), attempts to provide equal sharing of the CPU for runnable tasks within some constant difference. That algorithm is as follows: on receipt of a message in a node $N[k]$, the number of executing CPU cycles of the target task ($T[l]$) since it reached the ready queue plus a constant (*share*) is compared with the same parameter ($N[k].T[0].c$) of the running task ($T[0]$). Thus, if $N[k].T[l].c + share < N[k].T[0].c$ then a context switch is performed in favor of $T[l]$.

Take for example, the following situation: there is one local (and other distributed) task executing in the CPU, which request 30ms (20ms) of execution

every 50ms (50ms too). For low *share* values and when the local task is executing, Sobalvarro's algorithm would always preempt the CPU in favor of the distributed task, thus adding too much overhead in context switching. On the contrary, high *share* values do not allow the realization of context switching. That in turn can damage the peer-coscheduling between the remote source(s) and the local target tasks excessively. Note that in the example, the distributed execution time never equals the local one. On the contrary, it tends to be lower. Therefore, it will be very difficult to obtain dynamically optimal *share* values for each scheduling situation.

Furthermore, local (even distributed) applications can be excessively delayed or preempted from the CPU. In an extreme case, if the number of mixed workload is high enough, the system nodes can enter in thrashing. Instead of progress in the execution of the Cluster workload, the system would perform mainly context switching.

Our proposed Dynamic coscheduling is a variation on the previously presented Predictive model. Unlike the Predictive model, the Dynamic version is based only on message reception frequency. This is both to assign more scheduling priority to high receiving frequency for increase coscheduling success between distributed tasks, and prevent local starvation by controlling the number of overtakings (with *MCO*) thus avoiding the problems of Sobalvarro's algorithm.

Taking into account the above mentioned considerations and also section 2.2.4, where Predictive Coscheduling was developed, the resulting Dynamic Coscheduling Algorithm (DCA) is a slight variation on PCA.

DCA only varies from PCA in the *COSCHEDULING CONDITION* (*C_C*). In PCA, *C_C* is equal to $h.freq > T[i].freq$, meaning that when task *h* is inserted in the RQ, it advances tasks with lower sending and receiving frequency. In DCA, only receiving frequencies must be taken into account. However, note that modifications for obtaining it are minimal. It is only necessary to change *C_C*. In this case *C_C* should be equal to $h.freq_r > T[i].freq_r$.

Only the messages waiting in the *RMQ* should be taken into account for obtaining the receiving message frequency. Thus, only the CMC fields $h.freq_r$ and $h.cur\_freq_r$ must be considered, and consequently $h.freq = h.freq_r$.

All the remaining structures, considerations and hypotheses are the same as in

the PCA case.

Note that having previously defined the CMC model, implicit-control coscheduling techniques can be more easily defined.

Chapter 3

Coscheduling Prototypes

In this chapter various coscheduling implementations in an heterogeneous and non-dedicated PVM-Linux Cluster are explained.

First of all, DTS (DisTributed coScheduler), a system which implements the explicit mechanism of chapter 2, section 2.1, is presented. Basically, DTS is an environment that explicitly deals (it follows an explicit-control philosophy) with both kinds of task: distributed and local.

Next, a variation of DTS, named HPDT (High Priority Distributed Tasks) is presented. HPDT also follows an explicit-control trend. It always assigns maximum scheduling priority to distributed tasks. The maximum distributed performance in Cluster computing should be reached when distributed processes making up the distributed applications always have more scheduling priority than the local ones. For this reason, this particular kind of coscheduling technique is studied next. Assuming that the maximum distributed performance will be reached using this method, this will serve as a performance referencing point for comparing the coscheduling techniques.

The first studied coscheduling technique based on implicit-control, is also named *Implicit* coscheduling. *Implicit* coscheduling has already been developed by other authors in [16, 17, 18, 19] and [20, 21]. Here, the main objective is to study the applicability of this technique in Clusters formed by general-purpose workstations. Basically, efforts have been directed towards the search for an optimal *"spin"* value for the spin-block procedure.

Finally, the implementation of the Predictive model is analyzed in depth. The implementation of the Dynamic version scarcely varies with respect to the Predictive one. So, only some observations on the changes performed in the Predictive model are made.

3.1 Preliminary Concepts

As we will see later in this chapter, *DTS*, *HPDT* and the *Implicit* models have been implemented in the user space. Moreover, the main modifications are located in the PVM environment.

On the other hand, the *Predictive* and *Dynamic* models have been implemented in the system space, mainly in the Linux Scheduler. So, it is necessary to study the Linux Scheduler behavior and its principal mechanisms and structures which have been used (or modified).

Furthermore, to better understanding the main decisions, taken in the design of these models, some prior concepts must be introduced about the communication system formed by PVM and Linux, in the user and system space respectively.

3.1.1 Analysis of the Communication System

In Linux Clusters, tasks forming the PVM distributed applications, communicate/synchronize between themselves by message-passing. This means that message delivering starts in the user space (inside PVM) and crosses the Linux communication subsystem to the network. The reception of messages passes through the same communication levels but in the opposite direction.

In this section, the main queues involved in the communication process are analyzed, from the PVM throughout the communication-system Linux layers, to the physical network device layer. The study is centered on the main PVM and Linux messaging structures and buffering mechanisms.

3.1.1.1 PVM Layer

PVM [30] allows the execution of distributed applications in two different communication modes: *RouteDirect* and *DontRoute* (see Fig. 3.1). In DontRoute,

communication between remote tasks is done through the *pvmd* daemon. In this way, the daemon-daemon communication uses the UDP protocol and the task-daemon communication is by means of TCP or UNIX Domain sockets. On the other hand, in the RouteDirect mode, communication between remote tasks uses the TCP protocol.

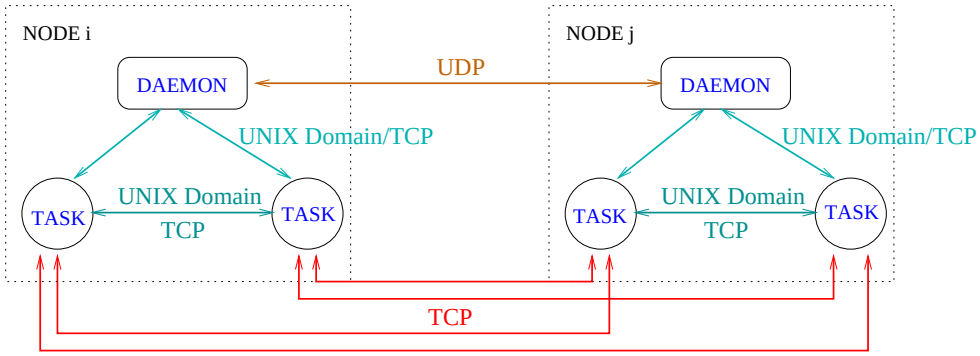


Figure 3.1: PVM protocols.

The PVM communication system provides a *message* interchanging facility. Every message (with variable length) is divided into *fragments*, which have fixed lengths (= 4096 bytes). The *fragment* is also the PVM transmission unit. Initially, a head fragment named *master* is created, then every time that a new fragment is filled up, another one is initialized and linked to the fragment list. Fig. 3.2 shows the structure of a PVM message made up of a master fragment and two data fragments (the first is full).

Every PVM task has an associated dynamic list named *pvmrxlist*, which stores the received messages, waiting for such task. On the other hand, all the messages sent by a PVM task are stored in a static queue named *txlist*, which has a maximum capacity of 100 messages.

3.1.1.2 Linux Layers

The responsibility for communicating processes by means of sockets belongs to the Linux communication system (see Fig. 3.3).

The first communication level, the BSD socket layer, is responsible for creating a new structure *socket* and providing a set of facilities to the underlying levels

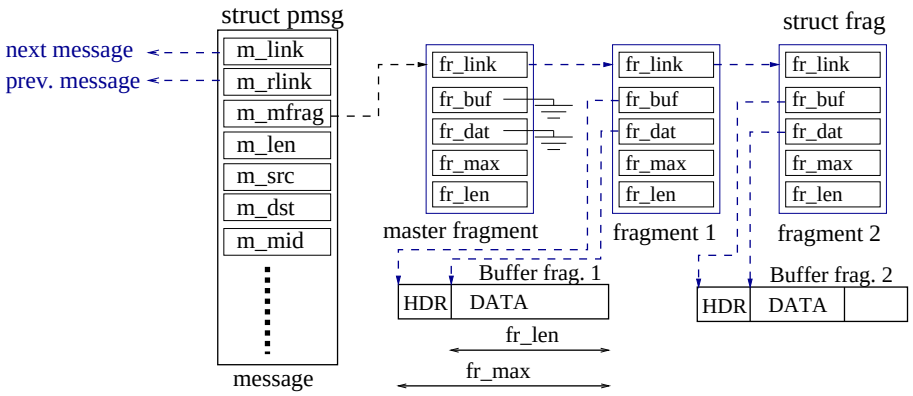


Figure 3.2: PVM message structure (*pmsg*)

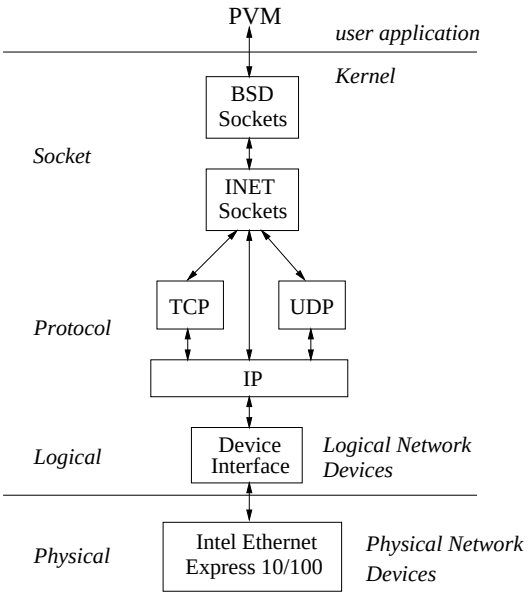


Figure 3.3: Linux communication levels.

for its access. By means of an associated *sock* structure, the *socket* structure is associated to the INET communication domain (or address family), in the INET socket layer. The *socket-sock* structures are usually denoted simply as *socket*. If, on the contrary, the communication must be performed by UNIX Domain sockets, the UNIX socket layer will be used instead.

The fragment sent by the PVM layer is decomposed into MTU (Maximum

Transmission Unit) size packets in the socket layer. Also, communication in the opposite direction will convert packets into fragments. A structure, named *sk_buff*, will be associated with every packet.

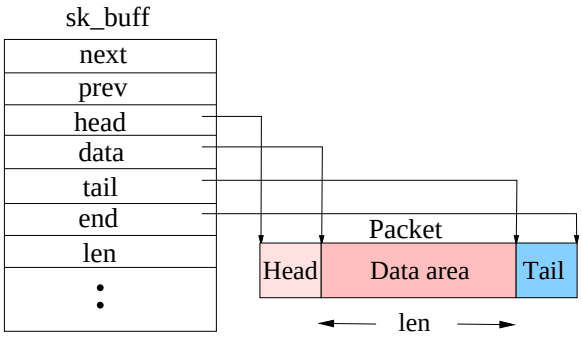


Figure 3.4: *sk_buff* and *packet* structures. The packet Data area contains the information to be transmitted.

In the emission of packets, the socket layer creates a new *sk_buff* and stores it in the *write_queue*. Similarly, in packet reception, the socket layer stores *sk_buff* structures in the *receive_queue.* Both queues have a maximum capacity of 65535 bytes.

The *sk_buff* structures are used by Linux to pass data through the TCP(UDP)/IP protocol layers [70, 71]. In emission/reception of packets, every protocol will add/extract control information to/from its reserved Head and Tail space (see Fig. 3.4).

At the logical layer, in transmission, the *sk_buff* structures (coming from the protocol layer) are stored in one of three buffering queues (with a max. capacity of 100 elements per queue). The choice of the queue will depend on the priority of the packet, *interactive* (highest priority), *normal* (PVM messages) and *background* (lowest priority). The head of every queue is stored in an array named *buffs*. On the other hand, the packets coming from the physical device are stored in a list named *backlog*, which has a maximum length of 300 buffers.

Our communication board is an Intel EtherExpress 10/100 Mbps, which has an i82558 microprocessor. The i82558 communicates with the kernel by means of a shared memory mechanism. This memory is divided into two different frame queues (also of *sk_buff* structures): *CBL* and *RFA*. The packets are named frames

in this layer. The *CBL* stores sending frames to the network. The *RFA*, instead, stores incoming frames from the network. The maximum number of elements in both queues is 14.

3.1.2 The Linux Scheduler

The Linux scheduler behavior is explained below. As can be seen in the Linux Scheduler scheme (see Figure 3.5), the scheduling phase starts when there is at least one task in the RQ (Ready to run task Queue). In Linux, the RQ is implemented by a double linked list of *task_struct* structures, the Linux PCBs (Process Control Block).

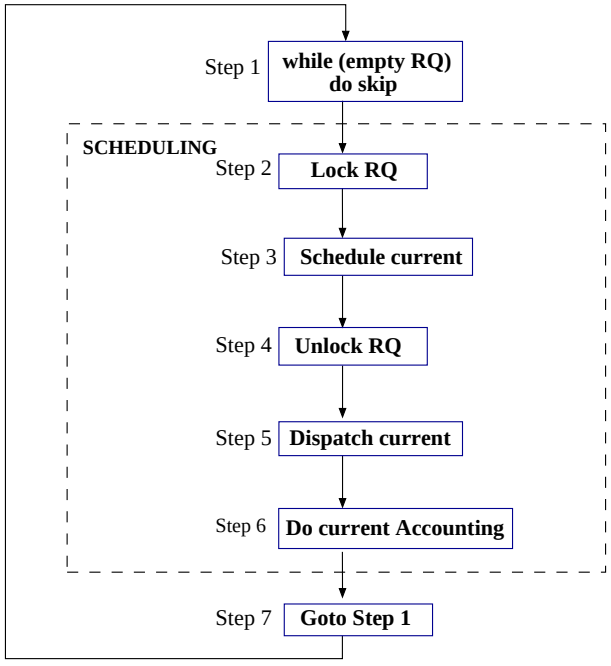


Figure 3.5: Linux Scheduler.

Then the RQ is locked because it must be accessed in an exclusive manner. Next, the scheduler picks up one task, "*current*" (task with the maximum returned value from an internal function named *goodness*). Afterwards, the scheduler unlocks the RQ and dispatches *current* (assigns it to the CPU). When the execution time slice is used up, the *current* accounting is saved in its PCB and if finished,

it is removed from the RQ. If not, it will wait to be scheduled again in the RQ. Finally, the scheduling process begins again.

The main Linux PCB fields which are involved in the scheduling process are:

- *policy*: scheduling policy. There are four scheduling policies in Linux: the "normal" and three "real-time" (with more scheduling priority than the normal ones).
- *rt_priority*: scheduling priority between real-time tasks.
- *priority* -"*static*" priority-: scheduling priority between normal tasks. This ranges from 1 (low priority) to 40 (high priority). Default Priority (DefPrio) is equal to 20.
- *counter* -"*dynamic*" priority-: when the task is executing, each tick[1], *counter* is decremented towards 0 in one unit; then the CPU is yielded. The initial value is equal to *priority*. Thus, the default Quantum value for normal tasks is 210 ms.
- *files*: open files structure. They save information about the open files.
- ***cur_freq***$_r$: added field, and used for saving the current receiving frequency. The initial value is 0.
- ***cur_freq***$_s$: added field, and used for saving the current sending frequency. The initial value is 0.
- ***freq***$_r$: added field, and used for saving the passed receiving frequency. The initial value is 0.
- ***freq***$_s$: added field, and used for saving the passed sending frequency. The initial value is 0.

Fields ***cur_freq***$_r$, ***cur_freq***$_s$, ***freq***$_r$ and ***freq***$_s$ were added to the original Linux PCB, and they will be used later for implementing Predictive and Dynamic coscheduling (see sec. 2.2.4 and 2.3 respectively).

[1] 1 tick $\simeq$ 10 ms

There are other fields, such as *pid* (process identifier), *state* and so on, but they will have no influence on our coscheduling schemes and so no further comments about them are included (see [62, 63, 65, 66, 64] or the Linux source for more information).

Initially, tasks in Linux acquire a "normal" scheduling policy. Normal tasks have a Round Robin scheduling policy, with a variable time slice. Real-time tasks must acquire this condition explicitly. Also, the scheduling policy of each real-time task can be defined as Round Robin or FIFO.

In task creation, the field *counter* is set equal to the field *priority* and then the task (PCB) is appended to the RQ. The Linux scheduler picks up the next process to be executed (*current*) by means of the internal function *goodness* (Figure 3.5, Step 3). Basically, it returns *rt_priority* to real-time tasks or the sum of the *counter* and *priority* to the normal ones. The task with the highest value returned by the *goodness* function is scheduled. If all the returned RQ tasks values are 0 (the field counter is 0), the field *counter* of every normal task is reset to be equal to *priority* and the scheduling process begins again. In absence of real-time tasks, this mechanism avoids starvation of the normal ones.

As we will see, the *rt_priority* field, will have a great influence in the implementation of the DTS and HPDT explicit-control mechanisms. They assign a real-time priority to tasks making up distributed applications.

Instead, in the implementation of the implicit-control models Predictive and Dynamic, only the normal scheduling policy will be considered. Fields *priority* and *counter* will be used jointly with the new frequency ones ($***freq**_r$ and $***freq**_s$).

3.2 DTS

In this section, the system named DTS, DisTributed coScheduler, which implements the explicit mechanism (of chapter 2, sec. 2.1) is introduced. It has been fully implemented in the user space of a PVM-Linux Cluster.

DTS is composed of the three types of modules introduced in chapter 2, the *Explicit Scheduler* (named here *DTS Scheduler*), the *Load* and the *Console*. Figure 3.6 shows the architecture and the relationship between the different components of the DTS environment.

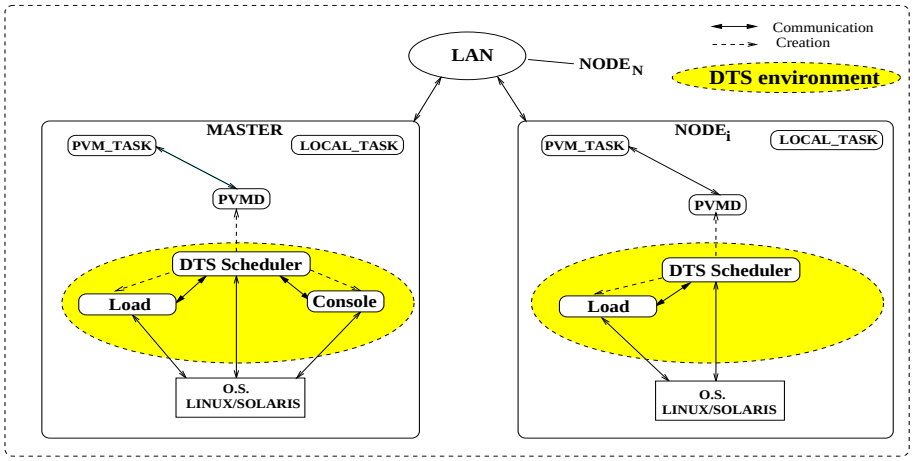


Figure 3.6: DTS environment.

The DTS startup process begins when the PVM is activated. In each workstation making up our VM (Virtual Machine), the shell script which starts *pvmd* (the PVM daemon) has been modified as follows: the sentence "*exec $PVM_ROOT/lib/pvmd3 $@*" has been changed to "*exec $PVM_ ROOT/lib/DTS_Scheduler $@*". This way, when the workstation is added/ activated to the virtual machine (even if it is the first) from the PVM console, the *DTS Scheduler* is executed.

DTS work is mainly centralized in a master node. If the algorithm were distributed, it would reduce the performance of the local applications and would increase the network activity due, for example, to the high communication generated by each *Load* module, which would have to send the load index to the overall VM.

3.2.1 DTS Scheduler

The pseudo-code that implements the *DTS Scheduler* is shown in Algorithm 5. This algorithm is very similar to the *Explicit Scheduler* (Algorithm 1), but more implementation considerations are introduced.

Our solution consists of promoting the distributed tasks (initially with a “normal” priority) to real-time. Furthermore, in each workstation, all the PVM tasks are put in the same group of processes, and led by *pvmd* (the PVM daemon). Con-

trol of their execution can thus be performed by sending them stop and resume signals from the *DTS Scheduler*.

Algorithm 5 DTS Scheduler.

```
  set PRI(DTS_Scheduler) = ((max(rt_priority) and SCHED_FIFO)
  fork&exec (Load)
  fork&exec (pvmd)
  set PRI(pvmd) = ((max(rt_priority) - 1) and SCHED_RR)
  set PRI(Load) = ((max(rt_priority) - 2) and SCHED_FIFO)
  set pvmd leader of pvm_tasks
sync_point:
  while (pvm_tasks) do
    sleep (PS)
    send_signal (STOP, pvm_tasks)
    sleep (LS)
    send_signal (CONTINUE, pvm_tasks)
  endwhile
```

At the start of the execution, the *DTS Scheduler*, which has root permissions, promotes itself to real-time class (initially time shared). After that, it forks and executes *Load* and *pvmd*, and also promotes *pvmd* and *Load* respectively to 1 and 2 real-time priority lower than the *DTS Scheduler*. Next, *DTS Scheduler*, promotes *pvmd* to become the leader of a new group of processes (denoted by *pvm_tasks*; this group will be made up of all the PVM tasks that *pvmd* will create).

The scheduling policy of every process (SCHED_FIFO or SCHED_RR) is also shown in the algorithm. They denote a FIFO or Round Robin scheduling policy respectively. The *DTS Scheduler* and *Load* have a FIFO policy because they need to finish their work completely before releasing the CPU. On the other hand, *pvmd* can block waiting for the receipt of a message (or event); meanwhile the CPU will be granted to another process. For this reason, the scheduling policy of *pvmd* has to be Round Robin.

This loop stops when there are no more PVM tasks (including the pvmd). This occurs when the workstation is deleted from the PVM console.

Thus, after the Parallel Slice (*PS*), all the Schedulers in the VM stop all the

PVM tasks by sending a STOP signal to the group of PVM processes led by *pvmd*. After that, these are resumed at the end of the local slice by sending them a CONTINUE signal. The distributed tasks run in the real-time class, so they have a higher global priority than the interactive ones. Thus, when they are resumed, they take control of the CPU. In this way, an interval of execution is assigned to distributed tasks (*PS*) and another interval (*LS*), which can be different, to the interactive ones.

Various control and information messages can be sent from the *Console* to the *DTS Schedulers*. One of these control messages (*init*) informs the *DTS Scheduler* processes to start delivering STOP and CONTINUE signals to their local high-priority distributed processes at regular intervals (depending on the *PS* and *LS* values). As was pointed out in section 2.1, the *PS* and *LS* intervals can be sent with the control (i.e. *init*) or synchronization messages.

The *Console* also provides various commands for controlling and managing the system. All the *Console* commands are listed in Appendix A.

3.2.2 DTS Overhead

The cost of implementing DTS in a PVM-Linux Cluster is analyzed next. The cost in this case is the overhead introduced into each Cluster node in the implementation of explicit coscheduling. More precisely, we are interested in finding a formula for measuring the overhead when changing from the *PS* to the *LS* period and vice-versa.

A change in each Cluster node to the *LS* period is performed when the *DTS Scheduler* sends a STOP signal to the distributed task group in the local node. On the other hand, a change to the *PS* period is performed by delivering them a CONTINUE signal.

When changing from period *LS* to *PS*, and taking into account that the Cluster nodes are uniprocessors, at most there will only be one local task executing into the CPU. Consequently, the time spent by “dts” stopping one local task and waking up a distributed one (T_{start}) is:

$$T_{start} = W_s(local) + W_w(dts) + S_{sig}(CONT) + W_s(dts) + W_w(dis), \qquad (3.1)$$

where W_w/W_s is the elapsed time in waking up/suspending the *DTS Scheduler* (*dts*), a local task (*local*) or a distributed task (*dis*). $S_{sig}(CONT)$ is the maximum elapsed time in sending a CONTINUE signal to all the distributed tasks in the node.

Also, at most only one distributed task will be executing when a change from the *PS* to the *LS* period is performed. When changing from period *PS* to *LS*, the time spent now in stopping a distributed task and waking up another, in this case local, is (T_{stop}):

$$T_{stop} = W_s(dis) + W_w(dts) + S_{sig}(STOP) + W_s(dts) + W_w(local), \qquad (3.2)$$

where $S_{sig}(STOP)$ is the maximum elapsed time in sending a STOP signal to all the distributed tasks in the node.

Because the time in delivering a signal to a group of processes does not depend on the signal to be delivered, we consider that $S_{sig}(STOP) = S_{sig}(CONT) = S_{sig}$. Similarly, the values W_w, W_s and W are considered to be equal ($W_w = W_s = W$). In consequence 3.1 and 3.2 can be reformulated as:

$$T_{ex} = T_{start} = T_{stop} = 4W + S_{sig}. \qquad (3.3)$$

3.3 High-Priority Distributed Tasks (HPDT)

HPDT (High Priority Distributed Tasks) is a very similar coscheduling environment to DTS. Its purpose is always to give the maximum priority to the distributed tasks in a PVM-LINUX environment.

To achieve maximum performance to distributed tasks it is only necessary to assign them a high priority throughout their lifetime. As in DTS, HPDT is implemented in the user space.

Taking into account the above consideration, the algorithm that implements

Algorithm 6 Priority Scheduler. Assigns real-time priority to PVM tasks.

set PRI(*Priority_Scheduler*) = ((max(*rt_priority*) **and** SCHED_FIFO)
fork&exec (*pvmd*)
set PRI(*pvmd*) = ((max(*rt_priority*) - 1) **and** SCHED_RR)

HPDT (Algorithm 6) is presented. This algorithm is located in a process named *Priority Scheduler*. There is one copy of *Priority Scheduler* in each Cluster node. The maximum real-time priority is also assigned to each copy.

The startup mechanism is identical to DTS. When a new node is activated from the PVM console, the *Priority Scheduler* is started instead of the DTS Scheduler (as in DTS). Then, in order to assign a high priority to distributed tasks, it is only necessary to promote *pvmd* (in its creation) one unit level less than *Priority Scheduler* (max(*rt_priority*) - 1). After that, all the *pvmd* descendant tasks will inherit this priority.

This technique does not introduce additional overhead, except in the initial startup phase, but for large enough distributed applications the starting process time can be depreciated. However, the average response and also the return time of the local tasks can be drastically increased.

3.4 Implicit

Implicit coscheduling was previously presented in [16], [17] and [18]. As its name suggest, it follows an implicit-control philosophy. This model only requires processes to block awaiting message arrivals for coscheduling to happen. The *two-phase* spin-blocking is the most efficient one between two variants of this technique. The waiting process spins for a determined time and if the response is received before the time expires then it continues its execution. Otherwise, the requesting process blocks and another one is scheduled. The other variant consists of only spinning, which produced poorer performance, as was shown by the authors.

One objective we have in mind is to measure the influence of the spinning phase on the performance of distributed applications, and also to find the optimal

spin value. The previous work based on that technique does not consider the interaction between distributed and local workload. So another purpose of the study of this model is to investigate the relationship between the *spin* value and the local slowdown in time sharing environments.

Algorithm *ImCoscheduling* (Algorithm 7) shows the pseudo-code of the Implicit coscheduling model. The DCEs provide facilities for dealing with messages, so Implicit coscheduling can be implemented in the user space (inside the DCE), but it can also be implemented in the system space, for example, in the Socket level. As was shown previously, this communication level uses packets (and its associated sk_buff structures), so the spinning should be implemented in this case as an active waiting for packets.

Algorithm 7 ImCoscheduling. Implements the Implicit coscheduling.

```
Initialize input_time, execution_time, spin
while (no_new_message) and (execution_time < spin)
 and (execution_time < timeout) do
    execution_time = current_time - input_time
if (no_new_message)
  if (timeout) block (timeout - execution_time)
    else block (indefinitely)
```

There are no appreciable differences in implementing Implicit coscheduling in the user or system space, except in the kind of message to be dealt with. Accordingly, it was decided to implement it in the user space, inside the PVM (the chosen DCE). This way, no modification of the Linux kernel in each VM workstation is required.

Messages in PVM are split into smaller transmission units, the *fragments* (see section 3.1.1). Thus, for implementing Implicit coscheduling in the PVM level, it is only necessary to change the word message by fragment in the Algorithm 7.

Next, it is also necessary to find the PVM libraries to incorporate the Implicit mechanism where reception of fragments may occur, and modify them according to the proposed algorithm. Only one PVM function, named *pvm_receive*, is responsible for doing the waiting for fragments, so the algorithm was placed entirely in this function.

In addition, the function *pvm_receive* is called from the best known PVM function *pvm_recv*. When the calling process returns from *pvm_receive*, it means that a new fragment is available to it. This process is repeated until the overall message is received or an error in the reception of the message is produced.

3.4.1 Implicit Coscheduling Overhead

Normally, in multiprogrammed systems, when a task is waiting for an event to occur (i.e. message arrival), it suspends execution until the event arrives. Then, it is reawaken and becomes ready to be scheduled again. In Implicit coscheduling, distributed tasks waiting for message arrival spin for a period of time before blocking, wasting this way CPU cycles that should be assigned to other ready to run tasks. Thus, the Implicit coscheduling overhead depends on the CPU intervals the distributed tasks spend in spinning.

In [17], the interval in which a process must active waiting the response of a requesting message for spinning to be beneficial was investigated. Based on the LogP model ([25, 26]), it was determined that the best benefits were obtained for a spin value equal to $5C$, where C is the context switch cost. It was assumed that C is a constant. Really, C depends on the size and number of processes in the system. Analytically, according to the LogP model, the time in sending messages is $L + 2o$, where L is the latency of the network and o is the processing overhead in each node. The optimal spin value in the requesting node was obtained by solving the following equation:

$$2L + 4o + spin < 2L + 4o + 5C, \tag{3.4}$$

where $5C$ is the context switching cost: $2C$ in both, the requesting and service nodes, and one additional C (fixed as the lower spin interval), realized only in the service node.

Distributed tasks can follow many types of communication pattern and the messages can arrive to distributed tasks asynchronously, at any time. We are only interested in spinning a distributed task at most during the time the system spends when the task is suspended immediately, which is the cost of blocking the distributed task and reawaking it later (when the message arrives). This is the cost

in doing two context switches. Accordingly, the maximum spin value ($spin_{max}$) is defined as follows:

$$spin_{max} = 2 * C \tag{3.5}$$

This fact give us a first reference to choose the spin value for Implicit coscheduling to be beneficial. It is also the overhead added -in each node- to the remaining tasks which are ready, waiting to be scheduled.

3.5 Predictive

In this section, a real implementation of the Predictive coscheduling technique in a Cluster system, as defined in chapter 2, section 2.2 is presented.

Modifications were mainly performed on the Linux scheduler (located in the *sched.c* module). To modify the scheduler policy of any computer system in the user space seems to be a bad idea. Scheduling is an inherent function of the operating system. If coscheduling were implemented in the user space, in the PVM sending or receiving libraries, no means for promoting the correspondents would be possible inside PVM (in the user space) when potential coscheduling was met. New system calls and libraries should be added to access operating system structures and services, leading into a very complicated user-system interface. Also, the extra overhead could reduce coscheduling performance.

Tasks making up distributed applications are normally executed in a Cluster, as are the local ones, so there is no advance information differentiating both kinds of task. As we are not interested in providing mechanisms which explicitly promote the distributed tasks to real-time (as DTS or HPDT do), only the normal task scheduling policy will be taken into account. However, a means for assigning more scheduling priority to distributed tasks must be provided to implement Predictive coscheduling in the normal task queue.

Implementing coscheduling in the kernel space gives transparency to the overall system and allows the application of coscheduling independently of the DCE used (PVM, MPI, etc ...).

The drawback is in portability. The kernel in each node must be modified to

incorporate coscheduling capabilities. Although it is an inconvenient, the modifications we propose are minimum.

3.5.1 Predictive Scheduler

The Linux scheduler, in the scheduling phase (Fig. 3.5, Step 3), picks up the task with the highest *goodness* value. Algorithm 8 shows how the *goodness* value (*weight*) for a task is obtained. In line 1, the *weight* for a real-time task is obtained with the help of the *policy* and *rt_priority* task fields. Both distributed and local tasks that we are dealing with always have "normal" priority, so lines 3 to 9 will be executed.

Algorithm 8 Function *goodness*(*task*).

```
1  if (task→policy == "real_time") weight = task→rt_priority + 1000;
2  else { //   "normal" priority
3    weight = task→counter;
4    if (! weight) {
5      weight + = task→priority;
6      task→cur_freq_r = n_packets(current,RMQ); // RMQ packets
7      freq_r = (1-P)*task→cur_freq_r + P*task→freq_r;
8      Distributed=freq_r || task→freq_s;
9      weight + = freq_r + task→freq_s + DefPrio * Distributed;
10   }
11 }
12 return weight;
```

The proposed Predictive model is based on increasing the priority of the distributed tasks (h) proportionally to their sending and receiving frequency ($h.freq$), defined in section 2.2.4, formula (2.12) as: $h.freq = h.freq_r + h.freq_s$. Thus, for implementing Predictive coscheduling, it is only necessary to increase the returned *goodness* value of the distributed tasks. This has been accomplished with the addition of lines 6 to 9 in the original *goodness* function (Algorithm 8) and by modifying the original Linux Scheduler (Algorithm 9).

The current receiving frequency is obtained in the scheduling phase (line 6 of the function *goodness*), where messages are waiting for service. However, in the sending case, it is obtained just after execution, in step 5.1 of the Predictive

version of the Linux Scheduler, Algorithm 9. Then, in Algorithm 9, steps 5.2 and 5.3, new passed frequencies are also computed and saved accordingly for later use in Algorithm 8, in lines 7 to 9.

Note that in the section 2.2.4, it was stated that each time the receiving and sending frequencies are obtained ($h.freq_r$ and $h.freq_s$ respectively), they will be used later as passed history in obtaining new frequency values.

Instead of obtaining the current sending and receiving frequencies as was explained in the previous chapter (chapter 2), the number of sending and receiving packets in the socket level was used. The reason for doing this is explained below, in the Additional Comments section (sec. 3.5.3). That work is performed in the function *n_packets* (explained in the following section, sec. 3.5.2).

Algorithm 9 Predictive version of the Linux Scheduler. Only the modifications (addition of Steps 5.1 to 5.3) with respect to the original Linux Scheduler (Fig. 3.5) are shown.

Step5: Dispatch *current* s
Step5.1: current→cur_freq$_s$ = n_packets(current,*SMQ*) // *SMQ* packets
Step5.2: current→freq$_s$ = (1-*P*)∗current→cur_freq$_s$ + *P*∗current→freq$_s$
Step5.3: current→freq$_r$ = (1-*P*)∗current→cur_freq$_r$ + *P*∗current→freq$_r$
Step6: Do *current* accounting

Unlike explicit-control methods (as DTS or HPDT), we are interested in detecting the correspondents in an implicit manner. In the DTS environment presented earlier, as every distributed task is started from the *DTS Scheduler*, all the distributed (PVM) tasks belong to the same group of processes and led by *pvmd*. As *pvmd* in each node is started from the *DTS Scheduler*, the *pvmd* descendant tasks (all the remaining distributed or PVM ones) can be easily identified. The implicit way of detecting the correspondent processes in the Predictive model is by obtaining the communication frequency of "normal" processes.

Initially, in plain Linux, as well as in the Predictive model, all the processes (even the distributed ones) begin execution with a "static" priority equal to DefPrio (=20), and remains constant during all their lifetime. So, the resulting priority of the distributed applications, with non-zero communication frequency (var. Distributed = 1 in line 8 of the Algorithm 8), will be higher than the local (user)

ones by adding they DefPrio (in line 9). Also the distributed application with the highest communication frequency will obtain the highest scheduling priority (see also the line 9).

Starvation of local tasks is avoided because when the *counter* field of any normal task reaches 0, it can not be executed while there are other tasks with non zero values in their respective *counter* fields (see Algorithm 8, lines 3 and 4).

3.5.2 Obtaining the current sending/receiving frequency

The implemented function *n_packets*(*task*,*queue*), depending on the value of the *queue* argument, obtains the number of packets received (queue = RMQ) or transmitted (queue = SMQ) by a task. Next, the obtained value is saved in the PCB as the current receiving (field cur_freq$_r$) or sending (field cur_freq$_s$) frequency. The pseudo-code of this function is shown in Algorithm 10.

Algorithm 10 Function n_packets(task,queue).

```
1  if (task→files)
2    for (fd = 0; fd < task→files→max_fds; fd++) {
3      file = task→files→fd[fd];
4      if (file) {
5        inode = file→f_dentry→d_inode;
6        if (inode && inode→i_sock && socket = socki_lookup(inode))
7          if (queue == RMQ)
             cur_freq_r += ⌈socket→sk→rmem_alloc.counter/4096⌉;
8          else cur_freq_s += ⌈socket→sk→wmem_alloc.counter/4096⌉;
9      }
10   }
11   if (queue == RMQ) return cur_freq_r; else return cur_freq_s;
```

In Unix (Linux), the sockets are treated as files. Thus, first of all, the open files that correspond to sockets must be identified. To do so, it is necessary to access the structure that represents each file in Linux, the *inode*. As Fig. 3.7 shows, this will be carried out by descending through the following structures: *task_struct*, *files_struct*, *file*, *dentry*, and finally *inode*. Then, if the inode corresponds to a socket (condition “inode->i_sock” of the Algorithm 10) it will be necessary to access the socket related structures (*socket* and *sock*) by means of the obtained

pointer to the socket, socket = *socki_lookup(inode)*, where *socki_lookup* is an internal Linux function.

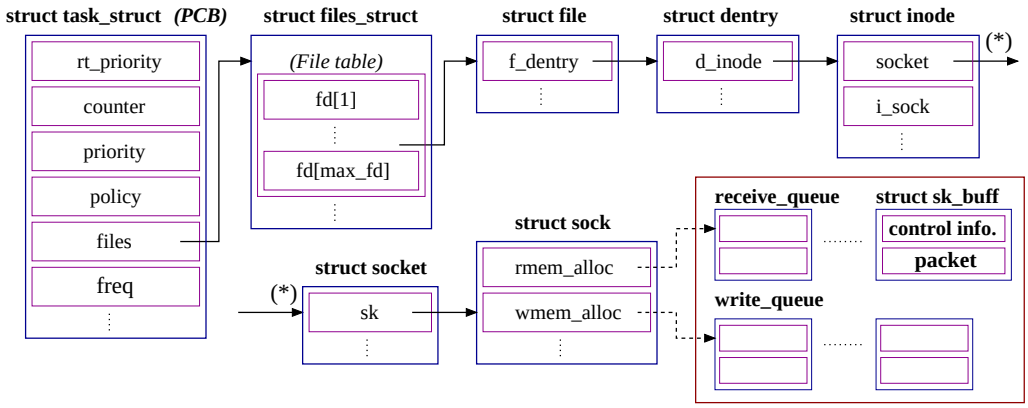


Figure 3.7: Most frequently used Linux structures (and their fields).

The *sock* structure points to two lists of *sk_buf* structures, which buffer packets (the socket transmission unit[2]). One of these lists, named *receive_ queue* (the *RMQ*, defined in section 2.2.1), buffers receiving packets, and the other, named *write_queue* (the *SMQ*), buffers packets to be transmitted. The *rmem_alloc* sock field saves the number of bytes in the *receive_queue*, and the *wmem_alloc* field saves the number of bytes in the *write_queue*.

3.5.3 Additional Comments

The packet-buffering queues (*receive_queue* and *write_queue*), in the socket Linux layer, has been chosen to collect message communication information. Accordingly, packets would be a more accurate terminology for messages. As was shown in the previous section, these queues can be accessed from the PCB of each process and consequently, all the packets sent towards (delivered by) such process can also be obtained. It would be more difficult to obtain the same information on other communication levels (i.e., inside the PVM or in the protocol layer), or it would even be more inefficient and add more additional overhead.

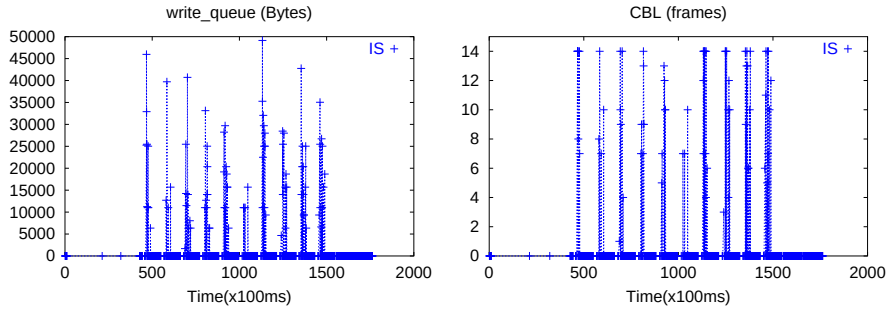
[2] 4096 bytes (4Kbytes)

MCO, defined as the Maximum number of Coscheduling Overtakings in chapter 2, was not considered in the implementation of the Predictive model. Its use was unnecessary because, as mentioned above, the Linux scheduler already incorporates a valid mechanism for avoiding starvation of "normal" tasks. Perhaps, in heavily loaded workstations, the Linux method may produce excessive response time in local tasks, but really it is very difficult for commodity Clusters to deal with this kind of load, so no more studies have been performed in this field.

Another Predictive feature to be mentioned concerns the way the current communication frequencies are obtained. As was explained previously, the current sending frequency is obtained by gathering the number of packets in the *SMQ* queue of the process which has just released the CPU. If an accurate frequency metric was obtained, the added overhead in gathering information each SI (Sampling Interval) and the computation of frequencies of the overall system processes in regular intervals (FI: Frequency interval), as was commented in chapter 2 section 2.2.4, will add an extra overhead. Similarly, the obtaining of the current receiving frequency is substituted by collecting the *RMQ* length in the scheduling phase. This information even gives more information of real coscheduling necessities than the proper current receiving frequency and also adds less overhead to the system.

Also, the current frequencies could for example, be obtained in the send and receive functions provided by the socket layer. But this method has been also discarded because each time these functions were called, it must access the system list of all the PCBs to update statistics. Such a method would add another excessive time overhead in fine grained distributed applications.

In our method, the system communication overhead plays a favorable role. Transmission packets are buffered for a short period of time, which is, however, sufficient to collect information. With the help of Monito [40, 82] (a communication monitoring tool for Linux environments), each 100μs, occupation information was gathered in the execution of IS (a NAS parallel benchmark described in chapter 5) of the *write_queue* (which buffers transmission packets, in the Socket layer) and the CBL queue (which buffers transmission frames, in the physical network device). The results are shown in Fig. 3.8. Note that due to the delay introduced by the network latency, the frames are buffered for a short period of time in the



(a) (b)

Figure 3.8: (a) write_queue and (b) CBL transmission queues.

CBL queue (see Fig. 3.8(b)). In addition, the system latency also increases the time that the packets must be keep in the *write_queue* (Fig. 3.8(a)), waiting for their transmission.

The design decisions depend on the final system of the Predictive (or other coscheduling) model to be applied. In this case, due to the previous study of the communication subsystem and the Linux scheduler behavior, the way the current frequency is obtained and the control of the coscheduling overtakes (by using the MCO parameter) for avoiding local task starvation has scarcely been modified.

3.6 The Dynamic Version

In this section, the Dynamic coscheduling technique defined in chapter 2, section 2.3 is explained. To implement the Dynamic version of the Predictive model it is only necessary to discard the sending frequencies. Modifications to be made are:

1. Function *goodness(task)* (Algorithm 8), line 8, must be changed to a more simple sentence (**Distributed**=$\boldsymbol{freq_r}$), because only the receiving frequency must be taken into account. Line 9 must also be changed to (***weight*** $+=$ $\boldsymbol{freq_r}$ $+$ **DefPrio** $*$ **Distributed**).

2. The Predictive version of the Linux Scheduler (Algorithm 9) must only save the passed receiving frequency. So steps 5.1 and 5.2 must be omitted.

3. Function *n_packets(task, queue)* (Algorithm 10) must not be modified. Unlike the Predictive case, this function will always be named with the queue argument equal to *RMQ*.

No more modifications to the Predictive method need to be performed in implementing the Dynamic mechanism. This fact demonstrates the importance of defining the CMC model. Having in mind the CMC model, different coscheduling techniques can easily be first designed and then implemented in real environments.

Note that the implementation of the Dynamic version require fewer modifications to be performed in the original Linux source. It also produces less extra overhead in the execution of distributed applications because the access to the *SMQ* queue is avoided.

One final comment is that both models can coexist in the same Cluster. If we assume the existence of a system parameter which specifies the coscheduling policy (Dynamic, Predictive or even plain Linux), one of the three models will be applied. This parameter can also be easily changed when required by a command (and/or system call). This is the way this has been performed for obtaining performance results in chapter 5. Appendix A presents an implementation of the Linux command and system call which has been implemented for changing the coscheduling model.

Chapter 4

Experimental Results (Simulation)

In this chapter, a broad experimentation based on simulation is performed. The simulator, named SCluster (Simulator Cluster), was implemented in C++ (and compiled with the public gnu g++ compiler) with the statistical CNCL (Communication Network Class Library) library [77]. CNCL was created at Communication Networks, Aachen University of Technology, in Germany. It is a class library featuring generic C++ classes which provides very useful random number generation and statistics libraries as well as an event driven simulation.

The main advantage of SCluster is that it simulates a Cluster with a variable number of nodes. The program defines a generic station class. Inside it, all the different parameters and functionality a Cluster node must provide are very easily defined and structured. Further extensions or modifications are located in a unique structure, the *station* class. Also, communication links between different Cluster nodes are incorporated in this class, so network topologies, or communication patterns of distributed applications can easily be incorporated.

The global simulator parameters (passed as input arguments at program startup), are the following (see fig. 4.1):

- *NSTATIONS*: number of nodes making up the Cluster.
- *MRQL* (Mean Ready Queue Length): the average number of tasks in each Cluster node. The overall experimentation was done for two *MRQL* values (2 and 5). Values of 2 and 5 represent a medium and high workload respectively.

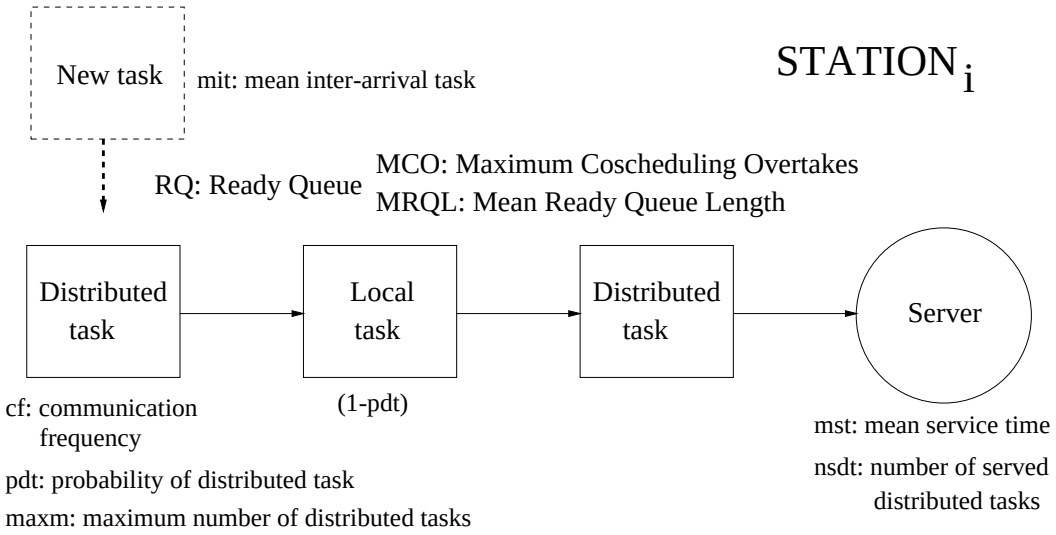


Figure 4.1: SCluster input arguments.

- mean inter-arrival time (*mit*): mean time for tasks to arrive at the RQ (Ready Queue). The density function is exponential with mean = *mit*. The *mit* is constant (with a value of 10) in the overall experimentation.
- mean service time (*mst*): mean time in serving tasks (by the Server, which simulates a uniprocessor Cluster node). The chosen density function is an exponential with mean = *mst*. Taking into account the equilibrium condition, *mst* is computed as follows:

  for a predetermined *MRQL* value, as *mit* is fixed to a constant value equal to 10, the *mst* parameter is calculated by means of the following equation:

$$MRQL = \frac{\rho}{1-\rho}, \tag{4.1}$$

  where $\rho = \frac{\lambda}{\mu} = \frac{1/mit}{1/mst}$. $\lambda = 1/mit$ and $\mu = 1/mst$.

- *nsdt* (number of served distributed tasks): finishing simulation parameter. All the experimentation was performed with a value of *nsdt* = 10.000.
- *pdt* (probability of distributed task). Each generated task is distributed with probability *pdt*, and local with probability 1 - *pdt*. The density function

is a Bernoulli with a *pdt* probability. The experiments were done with the following *pdt* values: 0.2, 0.5 and 0.8.

- *maxm* (maximum number of distributed tasks): maximum number of distributed applications which can be simultaneously executed in the Cluster. It is also the maximum number of distributed tasks residing in a Cluster node. Experimentation was performed for *maxm* values equal to 1, 3 and 5. These values are very realistic because the hardware resources capabilities of the nodes making up the Cluster (as for example memory capacity) are generally not designed for higher *maxm* values.

- *cf* (communication frequency): each distributed task (application) has an associated *cf* value in the range [1...*maxm*]. Depending on the correspondent *cf* value, the message sending frequency of each distributed task is generated. The density function of each generated distributed task is uniformly discrete in the interval [0...*cf*].

- *MCO* Maximum Coscheduling Overtakes: number of times a task can be overtaken in the RQ by another one with higher priority. In the experimentation, the *MCO* values are varied in the range [$MRQL-2$, $MRQL+2$].

The models (and their principal parameters) that were evaluated are represented by the following notation:

- Linux (LIN): it represents the Linux scheduler. As mentioned in chapter 3, section 3.5, the scheduling policy of "normal" tasks (the kind of tasks we are interested in) has a Round Robin behavior.

- Implicit (IMP): represents the spin-block technique of the Implicit coscheduling technique presented in chapter 3, section 3.4. The spin interval is equal to $mst/10$.

- Explicit (EXP): represents the STATIC operation mode of the Explicit (DTS) coscheduling environment, explained in chapter 2, sec. 2.1 (chapter 3, sec. 3.2). To obtain the best possible results a modification to the original DTS model was introduced. In the *PS* (*LS*) interval, if there are no distributed

(local) tasks in the RQ, an execution chance arises for the local (distributed) ones. Moreover, *PS*=*LS*=100. In the simulation there is no overhead in their setting or synchronization over the Cluster.

- High Priority Distributed Tasks (HPDT): this acronym is used to denote the coscheduling model with the same name, HPDT, described in chapter 3, section 3.3. A high priority is always assigned to distributed tasks. To simulate this model (and also in the EXP one), a previous identification of the distributed tasks is needed. How to implement this in a Cluster system by using a DCE with distributed-tasks management capabilities (i.e. PVM) was explained in Chapter 3 .

- Predictive (PRE): denotes the Predictive model, defined in chapter 2, section 2.2.

- Dynamic (DYN): denotes the Dynamic model, defined in chapter 2, section 2.3.

Times in the simulator are in Time Units. For example, a value for *mit*=10, represents 10 units of time. This generic unit of time has no influence in the results obtained with SCluster.

4.1 Coscheduling Models Evaluation

The most significant CMC metrics are those referring to system performance, this is *SystemCoDe* (System Coscheduling Degree) and *SystemThDe* (System Thrashing Degree), described in chapter 2 section 2.2.2. They reflect the behavior of distributed tasks in the overall system. Also, as the System Thrashing Degree is the complement of 1 of *SystemCoDe* (*SystemThDe* $=$ 1 - *SystemCoDe*), there is no need to obtain *SystemThDe.*

Only *SystemCoDe* statistics are shown in the simulation because more information about the overall system is reported using this metric. Deduced from its definition, *SystemCoDe* will be low (high) if the average *NodeCoDe* is also low (high), so there is no need to compare *NodeCoDe* results. The same can be said for the *TaskCoDe* metric.

We are interested in finding coscheduling techniques with good values (close to 1) for *SystemCoDe*, which also in turn implicate good ones (close to 0) for *SystemThDe*.

In doing the comparison between the models, two kinds of figures are used:

- SCODE: evaluates the *SystemCoDe* (System Coscheduling Degree). This metric is applied only to Distributed tasks. It serves to compare coscheduling performance between the different models and was formally defined in formula 2.9 as:

$$SystemCoDe = \frac{\sum_k \sum_l N[k].T[l].tco}{\sum_k \sum_l (N[k].T[l].tco + N[k].T[l].tth)}$$

  As can be seen in the formula, *SystemCoDe* is the relation between the total coscheduled cycles ($T[l].tco$) when potential coscheduling was met (field $T[l].pco$ was activated) and all the possible coscheduling ones ($T[l].tco + T[l].tth$). Potential coscheduling for a task $T[l]$ is indicated with the activation of the $T[l].pco$ field. The field $T[l].pco$ is activated when the current frequency is higher than 0. Thus, local and distributed tasks with the current frequency equal to 0 are not computed in this formula because no need for coscheduling is met.

  In all the measurements of the PRE and DYN models, *MCO=MRQL* and P $\simeq$ 0.

- TIMES: in this kind of figure, the average distributed and local times are obtained. As in the SCODE figures, in the PRE and DYN models *MCO=MRQL*, and P $\simeq$ 0. The chosen times in this case are:

  - *Dret*: Distributed return time. Average return time (elapsed time from the start to the end of execution) for all the distributed tasks in the overall Cluster.
  - *Dwait*: Distributed waiting time. Average waiting time in the RQ for all the distributed tasks in the overall Cluster.
  - *Lret*: Local return time. Average return time for all the local tasks in the overall Cluster.

- *Lwait*: Local waiting time. Average waiting time in the RQ for all the local tasks in the overall Cluster.

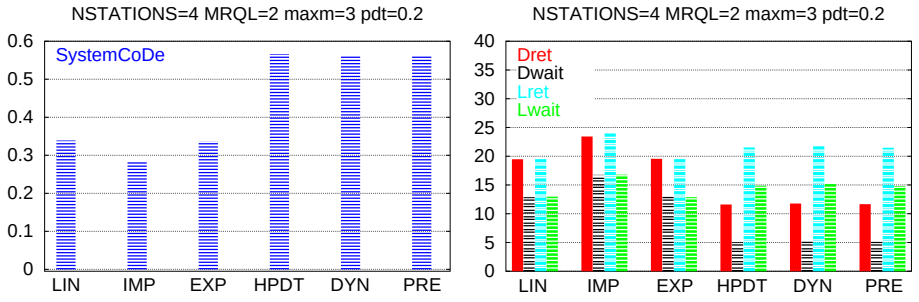
Figure 4.2 shows the results obtained for the above mentioned kinds of metric (on the left (right) the SCODE (TIMES) figures show the *SystemCoDe* (*Dret, Dwait, Lret* and *Lwait*) results) for *NSTATIONS*=4, *MRQL*=2 and *maxm*=3. The different figures show the results obtained for *pdt*=[0.2, 0.5, 0.8]. The *MRQL*=2 represents a moderate workload in each Cluster node. As the maximum number of distributed applications is 3 (*maxm*=3), a value of 2 for the *MRQL* is very appropriate if we also consider that eventually there will be owner tasks in the RQ.

Specific characteristics of distributed applications (i.e. different message patterns, and effects of varying the communication frequency and the message lengths) are not studied in this chapter. This experimentation aspect is covered in the following chapter (chapter 5), where different kinds of distributed applications are executed in a real Cluster system.

By observing the Figure 4.2, it can be seen how *SystemCoDe* and TIMES results are very closely related with each other. High (low) *SystemCoDe* results generally give low (high) *Dret* and *Dwait* values in the distributed tasks (see the TIMES figures). Note as times in the TIMES figures are in time units. All the obtained TIMES values are also very coherent because they are in the same order of magnitude than for example *mit* (=10). This is, they ranges in multiples of 10 (and not for example in multiples of 100 or 1000).

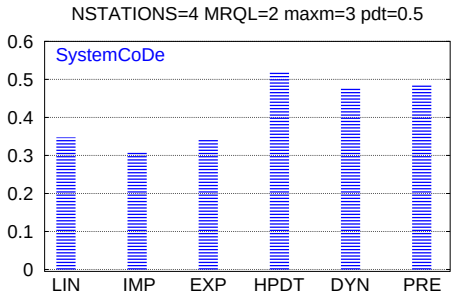
Also, high (low) values for *SystemCoDe* imply high (low) return and waiting times in the local tasks (metrics *Lret* and *Lwait* respectively). The added overhead in local tasks is an important factor to be taken into account because as was commented in the Objectives section of chapter 1, we are interested in increasing the performance of distributed tasks without disturbing the local ones excessively.

The results obtained validate the usefulness of the *SystemCoDe* metric in obtaining on-time distributed performance of the overall system. By extension, *NodeCoDe* (*TaskCoDe*) will serve for obtaining on-time performance of any particular node (distributed tasks forming a distributed application in a particular node).

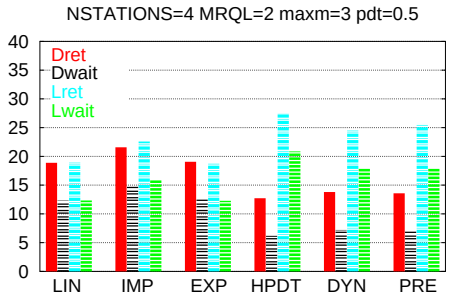


(a)

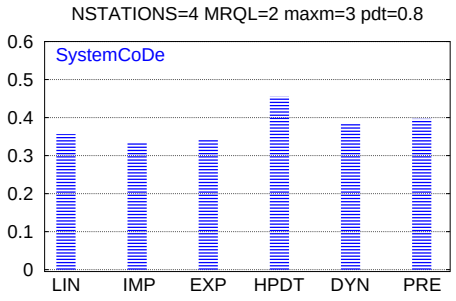
(b)



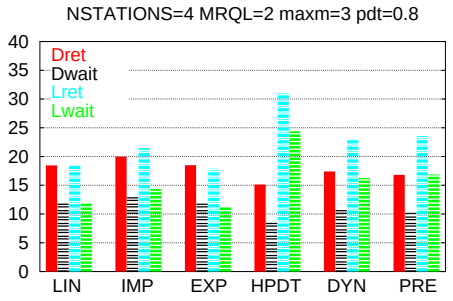
(c)



(d)



(e)



(f)

Figure 4.2: *NSTATIONS*=4, *MRQL*=2. (left) SCODE (right) TIMES.

The performance obtained in each model and a comparison between them is discussed below.

In general, the performance statistics of the different models are very coherent. For example, note that the LIN model always gives the same performance for both distributed and local tasks (see the TIMES figures). The expectation and the obtained results coincide, because LIN implements a Round Robin scheduling policy, with no differentiation between distributed and local tasks.

The distributed performance of the IMP model is the poorest in all cases. This fact is provoked by the high spin interval we chose (which is excessively high with respect to *mst*, the mean service time). In the following chapter (chapter 5), the *spin* interval is selected according to the explanation in section 3.4.1, which is based on the context switch cost (which has a null value in the simulator and so we cannot base this on it), and so more coscheduling benefits are reached.

It can be seen how the LIN and EXP performance always remains in intermediate locations.

Note that LIN and EXP results are very close to each other because, in the simulation, no overhead is added in the EXP model. Also, overhead in switching between the *PS* and *LS* periods is not considered. Thus, in EXP, distributed and local tasks perform the same Round Robin as in LIN, but at separate intervals. This fact favors the performance of distributed applications due to the benefits obtained from coscheduling them in the same period (*PS*).

In general, HPDT, DYN and PRE gave better coscheduling performance (*SystemCoDe*) than LIN, IMP and EXP. The simulation results confirm the expectations about HPDT, which assigns more scheduling priority to distributed tasks, so this model obtained the best SystemCoDe results in all the situations. However, DYN and PRE models obtained results very close to HPDT, especially for low distributed workload (i.e. $pdt = 0.2$). This fact demonstrates their effectiveness in making good use of potential coscheduling. However, no significant differences between the PRE and DYN models were produced. Only one slight advantage for PRE can be appreciated in these figures. We must await the results obtained in real implementations (chapter 5) to observe more significant differences between these models.

Unlike the other models, the LIN and IMP *SystemCoDe* scarcely increase with

pdt. By increasing *pdt*, the number of distributed tasks in each node also increases, but as LIN does not take into account the need for coscheduling (it performs a simple Round Robin), no important gains are produced. Note as for low *pdt* values (and especially when *MRQL*=5) the IMP performance of both distributed and local tasks is very poor. This means that there is no need for spinning when the probability of receiving a message is low. To spin in this situation only adds an unnecessary overhead in the system.

In IMP, by increasing *pdt* the active waiting for messages favored distributed applications and in turn also harmed the local ones. This result is also very coherent because it means that tasks only performing computation (local) are more affected than ones which also communicate (distributed).

On the contrary, EXP *SystemCoDe* decreases with *pdt* (major differences can be appreciated in Fig. 4.3, where the workload is increased, this is *MRQL*=5). By increasing the number of distributed tasks, the *PS* interval must also be increased accordingly, otherwise it becomes insufficient for executing all the distributed tasks. The synchronization phases have an additional overhead here, by stopping distributed (and also local) tasks being served before the time slice for execution has expired.

In the EXP model, local task performance scarcely increases with *pdt* (major differences can also be appreciated in Fig. 4.3), because they have an interval in which its execution is assured. By increasing *pdt*, the number of local tasks decreases, thus there are fewer tasks to split benefits by executing them in the same *LS* interval. This fact is not so significant in Fig. 4.2 because the local tasks are less affected when *MRQL* is low.

In the HPDT, DYN and PRE models, *SystemCoDe* also decreases with *pdt*. In this case, higher values for *pdt* produced the splitting of benefits (reported by a more efficient coscheduling) between an increasing number of competing distributed tasks. This is, when a distributed task is favored, proportional penalties are introduced into the others. On the contrary, when the *pdt* is low, the benefits reached by a distributed task do not affect other distributed tasks negatively as many times as before. These results show us the limit of distributed tasks to be executed in a node composing a Cluster system for coscheduling to be beneficial.

For low *pdt* values the distributed gains obtained in the DYN and PRE (and

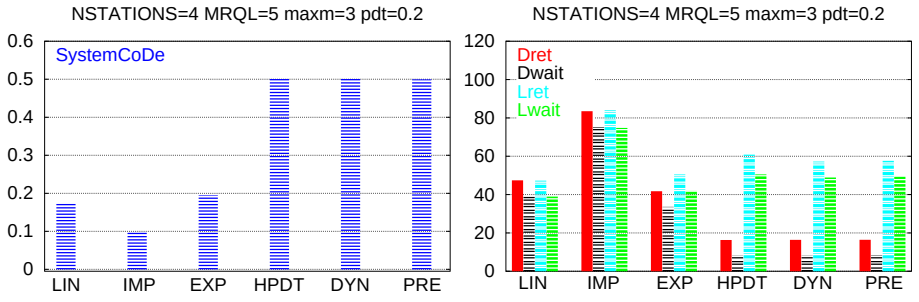
especially in the HPDT) models exceed the added overhead into the local tasks (see Fig. 4.2(b) and 4.3(b)). When *pdt* is high, the local overhead can increase excessively (see Fig. 4.2(f) and 4.3(f)). This also informs of the upper bound of distributed applications to be executed in a Cluster system.

As was expected, in the overall models, performance of both distributed and local tasks decreased with *MRQL*. With *MRQL* the probability increases that gains produced in one task affect other ones negatively (even in the coscheduling models). In sum, the negative effects produced in the affected tasks may exceed the gain obtained in the task being favored.

From the comparison between the graphs in the Fig. 4.2 and Fig. 4.3, it can be seen as *SystemCoDe* fall in increasing *MRQL* is more accentuated in LIN, IMP and EXP than in HPDT, DYN and PRE. If the number of competing tasks increases obviously, the Round Robin algorithm (LIN), the spinning mechanism (LIN) and the dropping of chances for executing distributed tasks in the *PS* interval (EXP) affect their respective *SystemCoDe* negatively. In the HPDT, DYN and PRE models, the distributed tasks have more chances to be executed than in the other ones, and consequently major gains are produced in the *SystemCoDe* metric. However, special attention should be paid to the *MCO* parameter in the DYN and PRE models when *MRQL* increases (studied in the section 4.2).

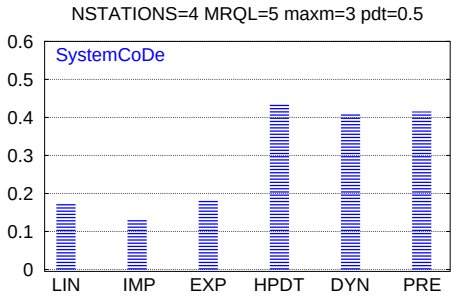
No significant variations in the behavior of the different models were obtained by varying the *maxm* value. As the *SystemCoDe* is computed in the overall distributed applications, the differences between executing one or various distributed applications are minimum. Only a scarce improvement can be observed (when *maxm*=3) for the HPDT, DYN and PRE models. This also demonstrates the most efficient coscheduling of these models. Furthermore, it can be appreciated that the PRE model behaves better than DYN when *maxm* is increased. These facts can be appreciated for example by comparing the SCODE Fig. 4.3(a) with Fig. 4.4(a) (and Fig. 4.3(e) with Fig. 4.4(b)).

Figures B.2 and B.3 (in Appendix B) show the good behavior of the simulator for *NSTATIONS*=8 and *NSTATIONS*=16 respectively. All the comments made for *NSTATIONS*=4 are also valid for these cases. This confirms the space scalability of the simulator. Consequently, it also proves the applicability and usefulness of the coscheduling models presented, and especially the PRE and DYN ones.

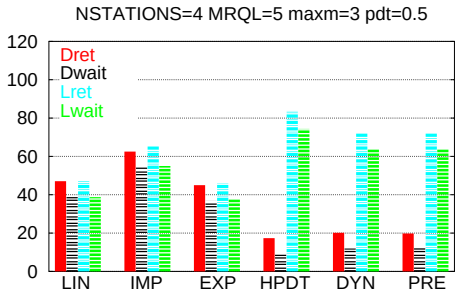


(a)

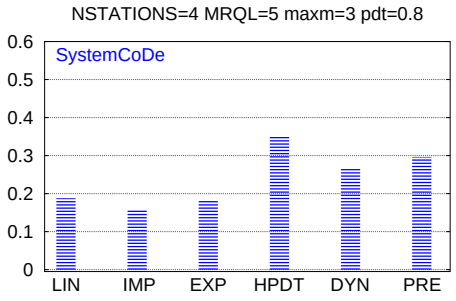
(b)



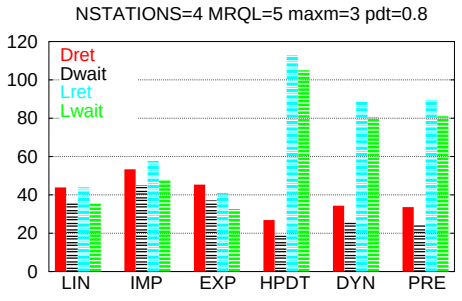
(c)



(d)

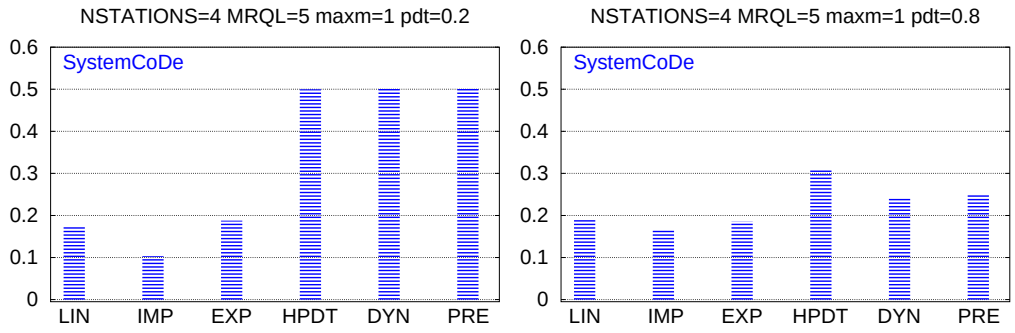


(e)



(f)

Figure 4.3: *NSTATIONS*=4, *MRQL*=5. (left) SCODE (right) TIMES.



(a) (b)

Figure 4.4: SCODE: *NSTATIONS*=4, *MRQL*=5, *maxm*=1.

The good behavior of the coscheduling models is supposed in ideal conditions when the system is scaled. This is when the network latency and system latencies in each Cluster node do not increase with the number of nodes. Normally, this is not at all true. Due to the bandwidth capacity of the network, when communication traffic increases, communication performance decreases. Collisions and network latencies grow and this can reduce coscheduling effectiveness. Also, overhead in communication buffering and information gathering in each node is a factor which negatively affects performance. These effects will be covered in the following chapter.

4.2 Predictive and Dynamic

In this section, Dynamic and Predictive performance are compared. In doing the comparison between these models, one kind of figure is used:

- MCO: performance of both DYN and PRE are compared separately by varying the *MCO* between *MRQL*-2 and *MRQL*$+2$. Also, in both models P $\simeq$ 0.

Once the usefulness of the *SystemCoDe* metric had been proved in the previous section, it was used in the current one for measuring performance when the *MCO*

was varied. Only the DYN and PRE models were compared. They are the only models whose performance can be influenced by *MCO*.

Fig. 4.5 shows the results obtained for both DYN and PRE when *NSTATIONS*=4. Also, measurements were made for *MRQL*=2 and *MRQL*=5 (on the left and right side respectively).

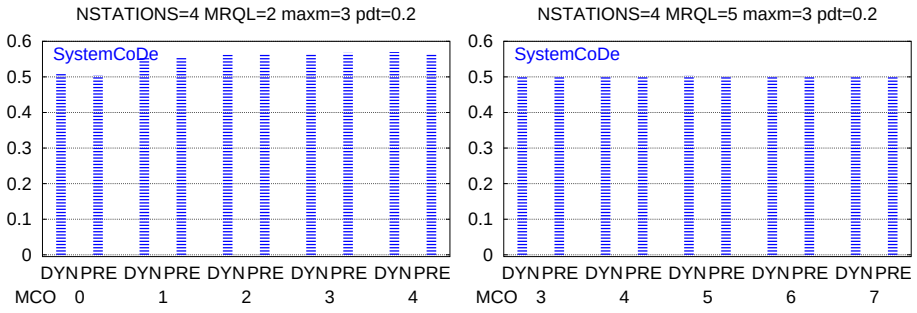
In general, it can be observed how the *SystemCoDe* metric increases with the *MCO*. The *SystemCoDe* metric increases while the *MCO* is less than or equal to *MRQL*. It tends to be stable above this threshold. It means that an $MCO > MRQL$ is not as important in distributed-tasks performance as an $MCO \leq MRQL$. According to these results *MCO* must vary between 0 and *MRQL*.

From the comparison between the graphs in Fig. 4.5, we can observe how the influence of *pdt* (the distributed workload) and *MCO* on performance depends on the total workload (*MRQL*). That is, when the overall workload increases, no gains are produced by varying the *MCO* until *pdt* reaches an optimal value (which also depends on the *MRQL*). Above this optimal *pdt* value, the *MCO* tends to decrease its influence on the performance of distributed tasks (see, for example, Figures 4.5 (a), (c) and (e)).

The PRE model behaves slightly better than DYN when *pdt* is increased. This tell us that this model works finer than DYN when the number of distributed tasks is also increased. The Predictive technique takes both the sending and receiving frequency into account, whereas DYN only works with the receiving one, and this implies that more coscheduling chances arise in the PRE model.

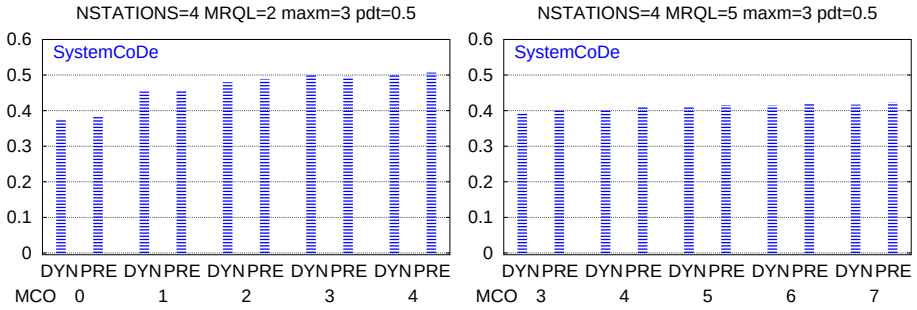
Similar results were obtained for different values of P, so they have been omitted. The study of the influence of the P parameter on coscheduling performance is covered in the following chapter.

In Appendix B, the figures B.5 and B.6 show the results obtained for *NSTATIONS*=8 and *NSTATIONS*=16 respectively. As in the previous section, the *NSTATIONS* parameter does not affect performance of the different models because when the system was spatially scaled, no variations on network bandwidth, latency and bottlenecks were produced. By doing simple modifications to the simulator program, it should be possible to measure the performance implications of these system parameters, but here, we are only interested in measuring coscheduling performance. Thus, these were not taken into account in this experimentation.



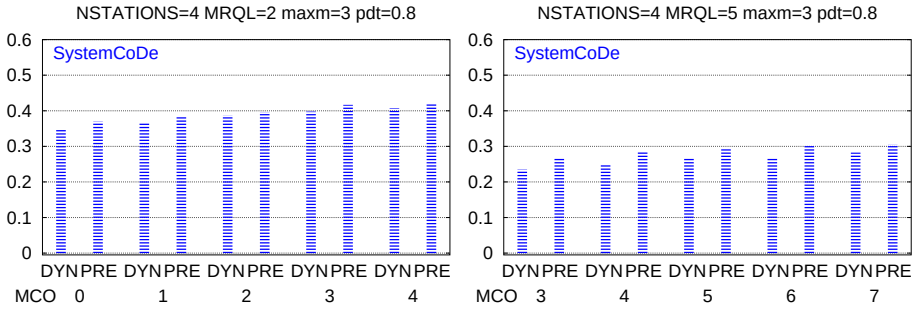
(a)

(b)



(c)

(d)



(e)

(f)

Figure 4.5: MCO. *NSTATIONS*=4. (left) *MRQL*=2 (right) *MRQL*=5.

4.3 Summary

The results obtained are summarized in table 4.1. This table shows the *SystemCoDe* behavior of the different models (LIN, IMP, EXP, HPDT, DYN and PRE) when various simulation parameters (*NSTATIONS*, *MRQL*, *maxm*, *pdt* and *MCO*) are varied. The LIN column represents the behavior of *SystemCoDe* (C: remains Constant, I: Increases; SI: Scarcely Increases, SD: Scarcely Decreases, D: Decreases) when the simulation parameters are increased. The columns of the remaining models represent the same *SystemCoDe* behavior, but with respect to LIN. We are not only interested in summarizing the performance of the different models but we also want to compare them with respect to LIN.

SystemCoDe	**MODELS**					
		SystemCoDe respect to LIN				
PARAMETER	LIN	IMP	EXP	HPDT	DYN	PRE
NSTATIONS	C	C	C	C	C	C
MRQL	D	C	SI	I	I	I
maxm	C	C	C	SI	SI	SI
pdt	SI	SI	SD	D	D	D
MCO	-	-	-	-	I	I

Table 4.1: Simulation Summary.

As can be seen in table 4.1, the LIN performance is degraded by increasing *MRQL*. That is, *SystemCoDe* decreases by increasing *MRQL*. Unlike *pdt* (which scarcely increases *SystemCoDe*), the remaining parameters do not affect LIN performance.

The C in the IMP cell of the *MRQL* row means that this model behaves like LIN when the workload increases. That is, the *SystemCoDe* of this model does not vary with respect to LIN (performance differences between the two models remain constant).

On the contrary, the EXP model behaves better (the *SystemCoDe* increases) with *MRQL*, and even more so with HPDT, DYN and PRE. This means that coscheduling of these models is more efficient than the remaining ones in heavily loaded systems.

When comparing the IMP model with respect to LIN, it was observed that only *pdt* increased *SystemCoDe*. Moreover, with low values of *pdt*, IMP obtained significantly lower results than LIN, but by increasing *pdt*, the *SystemCoDe* of IMP also increases and tends to reach the LIN one.

Unlike IMP, *pdt* harms EXP. With low *pdt* values, EXP behaves better than LIN and tends to decrease in performance when *pdt* is increased. For high enough *pdt* values, the performance is even worse than LIN. This fact can be better appreciated (it is also more accentuated) in high *MRQL* (see Fig. 4.3). EXP works fine when the system is heavily loaded, but as was commented, by increasing *pdt*, distributed gain is also split between an increasing number of distributed tasks.

The same explanation for *pdt* and *MRQL* in EXP is also true for HPDT, DYN and PRE, but gains in distributed performance are significantly better in these models. In addition, PRE tends to give slightly better results than DYN and both models also increase performance while $MCO \leq MRQL$ (above this threshold *SystemCoDe* tends to be stable).

The performance of local tasks was excessively decreased in the HPDT, PRE and DYN cases on increasing *pdt*. HPDT provides no mechanism for reducing this overhead. In PRE and DYN in contrast, the *MCO* parameter can be used for tuning the slowdown of the local workload.

DYN and PRE models (when *pdt* increases) tend to lower in performance with respect to HPDT. Unlike HPDT, the *MCO* parameter limits the distributed gain and consequently, *SystemCoDe* decreases by increasing the number of distributed tasks.

The most interesting appointment to be done about *maxm* is that by increasing this argument the PRE model behaves slightly better than DYN. This also confirms the advantage of PRE in front of the DYN model.

4.4 Concluding Remarks

It is worthwhile to point out that in real environments, coscheduling behavior can be overridden by non-scheduling factors. Some of these factors can also vary depending on the system workload, such as memory contention, network latency and bottlenecks, DCE servicing rate, context switching cost, operating system

latency and so on. So, the simulation has served to measure the efficiency of the coscheduling policies by only considering their scheduling behavior.

Simulation provided a mean for obtaining the behavior of the coscheduling techniques in a wide range of situations. Moreover, the trend of the distributed gains and overheads can be more easily studied than it can in the implementation of the coscheduling techniques (in the following chapter).

Note that the simulation parameters (and specially *MRQL*, *maxm* and *pdt*) were chosen to simulate the behavior of a non-dedicated Cluster as accurate as possible. They are the most important parameters to take into account in the analysis of the applicability of coscheduling techniques when both distributed and local applications coexist in the same system (i.e. Cluster).

Also, preliminary simulation results were very useful in the design of the coscheduling models described in chapter 3. In the following chapter (chapter 5), the different coscheduling polices are implemented in a real Cluster system. So, their performance were measured considering also those additional effects produced by the execution of distributed applications in a real environment.

Finally, we want to emphasize the usefulness of the CMC model in implementing new coscheduling policies in heterogeneous and non-dedicated Cluster systems. The implementation of new coscheduling techniques can be more easily performed by taking the model into account.