CISNE-P: A Global Scheduling Oriented to NOW Environments*

M. Torchinsky, M. Hanzich, P. Hernández, E. Luque
Dept. Computer Architecture & Operating Systems
University Autonoma of Barcelona, Spain
{matias, mauricio}@aomail.uab.es, {porfidio.hernandez, emilio.luque}@uab.es

and
F. Giné, F. Solsona, J.L. Lérida
Dept. Computer Science
University of Lleida, Spain
{sisco,francesc,jlerida}@diei.udl.es

Abstract

In this work, we present an integral scheduling system for non-dedicated clusters, termed CISNE-P, which ensures the performance required by the local applications, while simultaneously allocating cluster resources to parallel jobs. Our approach solves the problem efficiently by using a social contract technique. This kind of technique is based on reserving computational resources, preserving a predetermined response time to local users.

CISNE-P is a middleware which includes both a previously developed space-sharing job scheduler and a dynamic coscheduling system, a time sharing scheduling component. The experimentation performed in a Linux cluster shows that these two scheduler components are complementary and a good coordination improves global performance significantly. We also compare two different CISNE-P implementations: one developed inside the kernel, and the other entirely implemented in the user space.

Keywords: space and time sharing scheduling, coscheduling, social contract.

1 Introduction

The use of non-dedicated systems for parallel computation is based on various studies that prove the effectiveness of making good use of the idle CPU cycles by executing distributed applications. [1] showed the low utilization of resources in environments such as an open laboratory in a university. The main motivation of using these resources is the low cost at which it is possible to do parallel computation.

In this article, we present a new system named CISNE-P. Our system combines space sharing and time sharing scheduling techniques in order to take advantage of the idle computer resources available across the cluster by executing parallel jobs without damaging the local users excessively. CISNE-P is made up basically of a dynamic coscheduling technique and a job scheduler.

The parallel job scheduler of CISNE-P is named LoRaS (Long Range Scheduler). It is responsible for distributing the parallel workload among the cluster nodes. When a parallel job is submitted to the LoRaS, the job waits in a queue until it is scheduled and executed. Thus, LoRaS must deal with the *Job Selection* process from a waiting queue, together with the problem of selecting the best set of nodes for executing a job (*Node Selection* policies). This is performed by taking into account the state of the cluster system together with the characteristics of the local and parallel workload.

The dynamic coscheduling system, termed CCS [5], is the time sharing scheduling component. Traditional dynamic coscheduling techniques [2] rely on the communication behavior of an application to schedule the communicating processes of a job simultaneously. Unlike those techniques, CCS takes its scheduling decisions from the occurrence of local events, such as: Communication, Memory, Input/Output and CPU, together with foreign events received from remote nodes. This allows CCS to provide a social contract [3] based on reserving a percentage of CPU and memory resources in order to assure the progress of parallel jobs without disturbing the local users, while coscheduling of communicating tasks is assured. Besides, the CCS algorithm uses status information from the cooperating nodes to re-balance the resources throughout the cluster when necessary.

CCS was firstly implemented in the Linux kernel [6]. In this article, we present the modifications that allow CCS to be incorporated into an integral cluster scheduling system, such as CISNE-P; a middleware entirely located in the user space. The new user level approach is more flexible than the previous one (implemented

*This work was supported by the MEyC under contract TIN 2004-03388.

in the Linux kernel). This allows more efficient and portable scheduling extensions on top of the existing operating systems. With this aim, we compared the performance of both implementations in a Linux cluster. Likewise, we evaluated the interaction between space and time sharing techniques, showing the need to combine coscheduling techniques simultaneously together with space-sharing scheduling policies.

The remainder of this paper is as follows: in Section 2, CISNE-P system is presented. The efficiency measurements of CISNE-P are performed in Section 3. Finally, the main conclusions and future work are explained in Section 4.

2 CISNE-P: A portable and integrated scheduler for non-dedicated environments

In order to provide a system oriented to executing parallel jobs over non-dedicated environments, we have developed a system called CISNE-P. This is an space and time sharing scheduling system, which is based on a social contract to preserve the assignment of resources to local users.

CISNE-P is a middleware entirely developed in the user space. CISNE-P includes 2 main components, LoRaS and CCS. LoRaS solves the space scheduling problem. It is responsible for distributing parallel applications throughout the cluster using information about the system state, the applications to be launched and the characteristics of CCS as a dynamic time sharing scheduler. CCS uses a time-slicing technique to exploit the unused computing capacity of a non-dedicated cluster without disturbing local jobs excessively. With this aim, CCS limits the CPU and Memory resources assigned to parallel tasks by applying a social contract. CCS tries to exploit the rest of the resources of the NOW for parallel execution by means of combining balancing of computational resources and coscheduling between parallel jobs. In doing so, each CCS node assigns its resources dynamically based on a combination of runtime information, provided by its own o.s. and its cooperating nodes, together with architecture information and system-wide information. Thus, local decisions are coordinated across the NOW.

Figure 1 shows the integration of both systems. This figure also illustrates the distribution of the system with a server node and several other nodes that execute parallel and local applications.

Next, we explain the main features of LoRaS and CCS, respectively. For efficiency reasons, CCS was firstly implemented in the kernel space. Successive versions of CCS have migrated progressively to the user space. Nowadays, CCS is entirely implemented in the user space. We analyze the main advantages and differences of both implementations, and how the CCS components interact with each other.

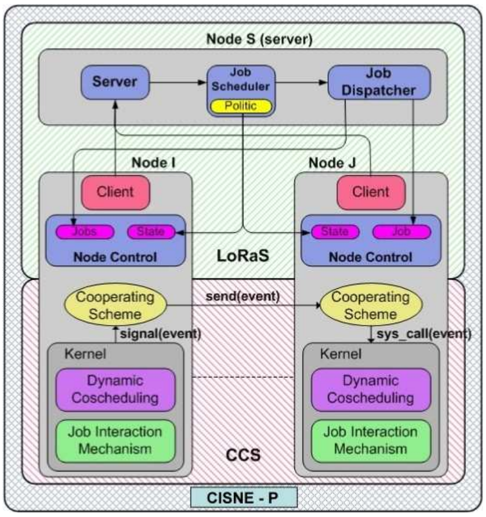

Figure 1: CISNE-P Architecture

2.1 LoRaS (Long Range Scheduler) System

LoRaS implements a Job Scheduler in the user space, which provides an Space-Sharing scheduling mechanism.

The LoRaS system has a master-slave architecture and consequently there are two kinds of node, master and slave. The jobs are delivered to the system from the client nodes. The clients request work from servers by sending them a Job Execution Request (JER). The admittance of new JERs for execution is performed by the server node.

Among the LoRaS responsibilities we consider (see Figure 1):

- *The admittance of new jobs to be executed*: this is done by the *Server* module, located in the server node. It is responsible for admitting new jobs into the system, sent by a parallel user using the *client module*, located in the slave nodes.

- *The management of queued jobs*: the LoRaS system has to schedule and then dispatch every queued job using some scheduling policy. The scheduling is performed by the *Job Scheduler*, located in the server node. The *Job Scheduler* allows the execution of a JER in a specific cluster and state in the amount of resources requested in the JER according to the job scheduling policy specified in this module. If there is no possibility of executing the job on arrival, then the JER is placed in the *Waiting Queue*, waiting to be scheduled.

  The *Job Dispatcher* formats the jobs accordingly by setting the parameters and the environment variables and then it dispatches the job by launching it in the cluster nodes specified by the *Job Scheduler*. Both PVM and MPI jobs are supported.

- *Job Execution Control* and System state gathering: the *Node Control* module (located in the slaves) monitors the execution control of every job and takes care of the state of every cluster node. It also informs the *Job Scheduler* so that it can take better scheduling decisions.

The next section explains the job scheduling policies applied by LoRaS.

2.1.1 Job Scheduling in LoRaS

Job scheduling is determined by the *Job Ordering*, *Job Selection* and *Node Selection* policies.

The parallel jobs, when entering in the system are placed in the *Waiting Queue* according to one of the following job ordering policies: FCFS (First Come First Serve), SJF (Shortest Job First) and SNPF (Smallest Number of Processors First).

Next, the jobs are selected from the *Waiting Queue* according to one of the following *Job Selection* policies: Best Fit, First Fit and Just First.

Finally, the best set of nodes to map a given job and the current cluster state is obtained. This is done according to two different *Node Selection* policies:

- *Uniform*. This policy merges communication and computation bound applications in the same node and tasks making up a pair of jobs are mapped in the same set of nodes, balancing the workload across the cluster.
- *Normal*. Unlike the uniform policy, it merges the parallel jobs independently of their communication/computation characteristics and placement over the cluster.

Uniform and Normal policies limit the resources used by the parallel applications across the cluster. Both policies launch an application on any set of nodes where the fact of executing it does not mean exceeding a system usage limit for any resource. This acceptable limit is established by means of a *social contract* defined by the CCS system, and establishes the maximum *parallel MultiProgramming Level* (MPL) or the percentage of *memory* or *CPU*, that could be used by the parallel applications on each node. Thus, those jobs mapped in nodes whose load has reached the threshold fixed by the social contract are stopped by CCS.

2.2 CCS (Cooperating CoScheduling) System

The time sharing system of CISNE-P provides an execution environment where the parallel applications can be dynamically coscheduled. The resources are balanced and the interactive responsiveness of the local applications is fully preserved.

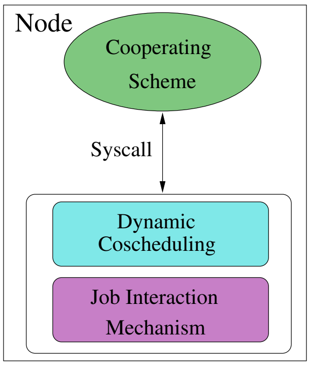

Figure 2: CCS Architecture.

Our dynamic coscheduling mechanism, located in a daemon of the forming clustering nodes, makes scheduling decisions based on the occurrence of local and remote events involved in the social contract. This coscheduling is slightly different from traditional dynamic coschedulers, mainly because it not only tries to schedule the communicating tasks making up the jobs, but it also tries to balance the assignment of the resources between local and parallel tasks, preserving in all the cases the portion of computational resources fixed by the social contract.

Firstly, we will explain CCS architecture (see Figure 2), the modules that integrate the system and how they work to achieve its goals. Later, we present different scenarios that we have been working on towards a portable solution.

The main components of the CCS system, residing in each node, are the following:

- Dynamic Coscheduling (DYN): is the module which guarantees that no processes must wait for a non-scheduled process for synchronization/communication. This is achieved by means of increasing the communicating task priority, even causing CPU preemption [2].
- Job Interaction Mechanism (JIM): it preserves the local user tasks responsiveness. In order to reach this goal, the JIM module manages the interaction between local and parallel jobs by means of a social contract. It means that both kinds of user, local and parallel, compromise for the cession of a minimum percentage (L) of CPU and memory for parallel tasks. The minimum term is related to the fact that if parallel tasks require a bigger percentage than L, then they will be able to use the portion allocated to local tasks, whenever local tasks are not using this.
- Cooperating Scheme: this module collaborates with the JIM module in order to balance the resources (CPU and Memory) assigned to tasks belonging to parallel jobs running throughout the

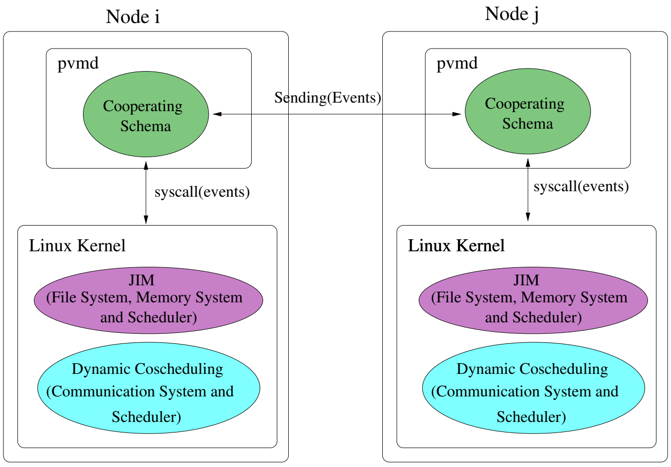
Node i
Node j
pvmd
Cooperating Schema
Sending(Events)
syscall(events)
Linux Kernel
JIM (File System, Memory System and Scheduler)
Dynamic Coscheduling (Communication System and Scheduler)

Figure 3: CCS in the kernel space.

cluster. It is responsible for exchanging several events between cooperating nodes[1], such as the login (LOCAL) or logout (NO_LOCAL) of a local user into a specific node, or the stopping (restarting) event generated by the JIM module, which stops (restarts) a parallel job. This happens whenever it has to preserve the local responsiveness.

2.2.1 CCS in kernel space

Firstly, *CCS* was implemented in a PVM (v.3.4) - Linux (v.2.4.18) cluster.

As we can see in Figure 3, the *Cooperating Schema* was implemented inside the daemon of the PVM system [4]. PVM provides useful information for implementing the algorithms for sending/receiving events between the cooperating nodes. Every node of a PVM system has a daemon, which maintains information about the PVM jobs under its management. It contains the identifier of each job (*ptid*) and a host table with the addresses of its cooperating nodes. This way, each node in the CISNE system knows where its cooperating nodes are.

Likewise, the *Dynamic Coscheduling* and *Job Interaction (JIM)* modules are implemented in the kernel space. This solution was adapted because thus CCS can adapt quickly to the continuous changes experimented by the environment, guaranteeing fast answer time for local users with interactivity needs as well as a high coscheduling likelihood for parallel jobs. A patch, with the following modifications must be introduced into the Linux Kernel:

File System: CCS sends the LOCAL (NO_LOCAL) events by means of the Cooperating module to the rest of the cooperating nodes when there is (no) local user interactivity for more than 1 minute. This value ensures that the machine is likely to remain available and does not lead the system to squander a large amount of idle resources [9]. At the beginning of every scheduling epoch, the access time to the keyboard and mouse files is checked, setting a new kernel variable *(LOCAL_USER*) to True or False.

Communication System: A new kernel function is implemented to collect the sending/receiving packets from the socket queues in the Linux kernel.

Memory System: The Linux swapping is modified to guarantee the memory portion fixed by the social contract for local and parallel tasks.

Scheduler: In order to select a task to run, the Linux scheduler considers the *dynamic priority* of each task, which is the sum of the base time quantum (*static priority*) and the number of remaining CPU ticks by the task in the last epoch. Whenever an epoch finishes, the dynamic priority is recomputed. The implementation of the social contract technique involves the modification of the base time quantum. Whenever a parallel task is stopped due to memory being overloaded, the scheduler assigns such task a quantum equal to zero. On the other hand, the scheduler decreases the time slice of the parallel task proportionally to the percentage L fixed by the social contract, whenever there is a local user in such a node. Likewise, the scheduler was modified to implement dynamic coscheduling. The coscheduling implementation increases the dynamic priority of each parallel task inserted in the Ready Queue according to the number of packets in the receive/send socket queue. Thus, the current scheduled task can be preempted by the task inserted into the RQ with most pending messages. Coscheduling is thus achieved.

However, this installation was complex and unportable, mainly because the Linux kernel had to be modified to make use of the CCS system. For these reasons, a more portable implementation was caught.

[1] nodes executing the same parallel job.

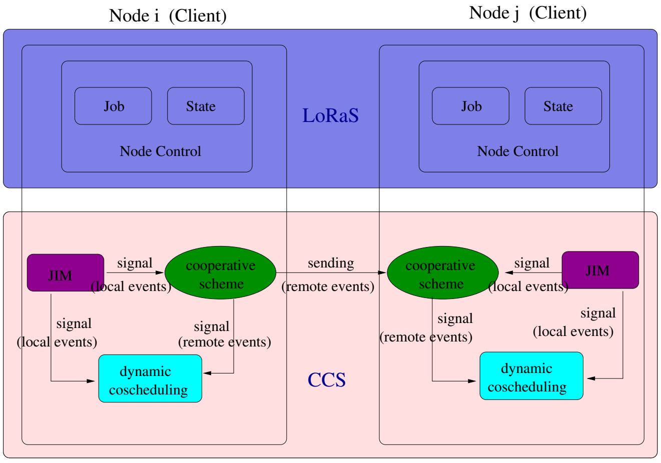

Figure 4: CISNE-P architecture in the user space.

2.2.2 CCS entirely in user space

For a more portable solution, we studied and applied the idea of moving the JIM component to the user space, as shown in Figure 4. To achieve this goal, we modified the JIM module to gather information about the resources consumed by parallel applications from the "/proc" file system. This way, CCS monitors the amount of resources (Memory and CPU) used by each parallel process and, as a consequence, it checks wether the social contract is being carried out. Whenever the parallel processes violate the social contract, they are penalized by means of lowering their priority or even stopping the parallel job until enough resources are available to restart it.

Working for a totally portable solution, the only module which was left running in kernel space was the dynamic coscheduling. So, in a second step, we moved the dynamic coscheduling to the user space, obtaining an entirely portable system and a kernel independent code. The dynamic coscheduling is achieved by lowering/raising priorities according to the number of packets in the socket queues. Thus, the dynamic coscheduling module is able to manipulate the priority of parallel jobs or even through stopping/restarting applications (from user space) by means of the *nice* / *renice* Unix commands.

Finally, the Cooperating schema was separated from the PVM daemon. Thus, the LoRaS daemon, residing on each node of the cluster, provides the cooperating module with the information required to exchange events between the cooperating nodes.

3 Experimentation

This experimentation was divided into two sections. The first section evaluates the need to use a coscheduling system over a non-dedicated Linux cluster. The second set of results compares the performance of CCS implemented in the kernel space (CISNE [7]) against CCS in the user space (CISNE-P). Likewise, these results are evaluated in relation to different sets of space-sharing scheduling policies.

In order to simulate a non-dedicated cluster, we need two different kinds of workloads: local and parallel.

The local user activity is represented by a benchmark that could be parametrized in such a way that it uses a percentage of CPU, Memory and Network. To parametrize this benchmark realistically, we measure our open laboratories for a couple of weeks and used the collected values to run the benchmark (15% CPU, 35% Mem., 0,5KB/sec LAN). Besides, and according to the values observed in the monitoring, we load 25% of the nodes with local workload in our experiments.

The parallel workload is a set of NAS parallel applications (CG, IS, MG, BT, LU and FT) with a size of 2, 4 or 8 tasks. These benchmarks were mixed in different ways according to our experimental purposes. The composition of each parallel workload is explained in the following sections.

Both workloads were executed in a Linux cluster made up of 16 P-IV (1,8GHz) nodes with 512MB of memory and a Fast Ethernet interconnection network.

3.1 Dynamic coscheduling

First, we show the impact of the DYN module (dynamic coscheduling in the kernel space) on the performance of the parallel workload in a non-dedicated/dedicated cluster. With this aim, we execute a parallel workload in different scenarios: (a) CCS with the DYN module activated and (b) CCS without the DYN module.

This evaluation was carried out by running three different parallel workloads (B, C and D), each one composed of a set of NAS parallel applications merged in such a way that it was possible to characterize the system bounding it by computation (B workload) and communication parameters (D Workload). Specifically, the B, C and D workloads were made up of the {MG and LU}, {SP and CG} and {IS and FT} set of benchmarks,

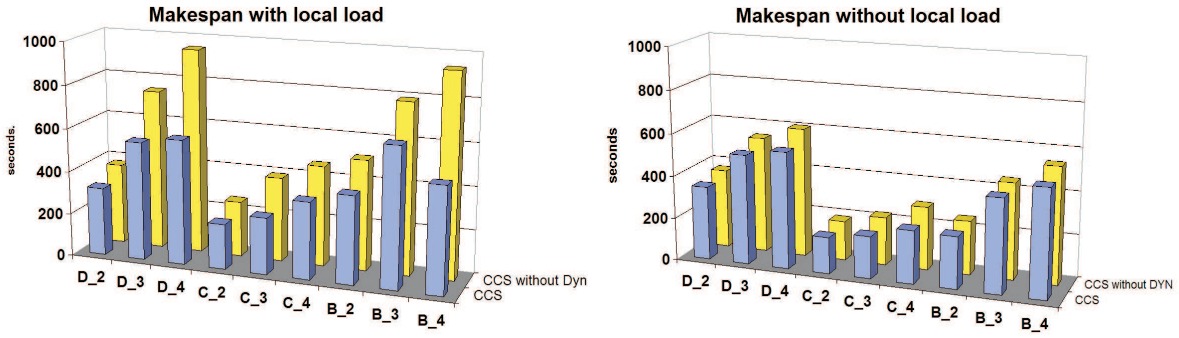

Figure 5: Makespan: CCS with Dyn vs CCS without Dyn.

respectively. Each workload was exercised several times for different Parallel Multiprogramming Level (MPL), from 2 to 4 instances of parallel applications chosen from the set defined by the workload in a round-robin manner (e.g.: class A - MPL 4: SP.A, BT.A, SP.A, BT.A). It is worthwhile pointing out that the threshold of MPL=4 was chosen taking the results shown in [6] into account. In that paper, we concluded that the response time of the local user for a social contract of *L*=0,5 and MPL=4 never exceeds significantly the 400ms stated as the acceptable limit for disturbing the local user responsiveness [8].

Figure 5 shows the makespan metric obtained when these parallel workloads were executed in a non-dedicated cluster (left) and a dedicated cluster (right). In general, these results show the effectiveness of the dynamic coscheduling. As was expected, the best values were obtained by the workload with the highest communication rate (D Workload). Likewise, we can see that this improvement increased according to the value of MPL. However, the gain of the dynamic coscheduling is reduced for the case of a dedicated cluster. This means that DYN performance behaves worse when the competing parallel tasks tend to be equal. This problem arises when some competing parallel processes have the same communication rate. In these cases, a situation where a set of different parallel processes have the same number of receiving/sending packets in their reception queues can happen frequently. In such cases, and taking into account the implementation of dynamic coscheduling in the kernel space, the scheduler assigns the same priority to all these processes so the next parallel process to run is selected randomly by the scheduler. In this way, there is a high likelihood that coscheduling was not achieved.

3.2 CISNE vs CISNE-P

In this section, we compare the performance of CCS implemented in the kernel space (CISNE) against CCS in the user space (CISNE-P).

In this case, the parallel workload was a list of 90 NAS parallel applications (CG, IS, MG, BT) with a size of 2, 4 or 8 tasks that reach to the system following a Poisson distribution. The parallel applications were merged so that the entire workload had a balanced requirement of computation and communication. It is important to remark that the MPL reached for the workload depends on the system state at each moment, but in any case it will surpass an MPL = 4. In order to validate our assumptions, the average *Turnaround* (see Figure 6) metric of the parallel jobs and the *makespan* (see Figure 7) of the workload were used.

This parallel workload was executed with two different combinations of *Job Ordering* and *Job Selection* policies (see section 2.1): FCFS-FFIT and SJF-JFIRST. Thus, we were able to evaluate the sensitivity of the CCS performance in relation to different space-sharing scheduling policies. In all the cases, a Uniform node selection policy was chosen.

The results in Figures 6 and 7 show that the penalization for moving from kernel to user space is lower than 10% for the turnaround and 30% for the makespan metric in the worst case (see Figure 7.right). The makespan metric is more sensitive to the CCS implementation than the turnaround due to the fact that a penalization in a specific job has a lower influence on an average metric than on the turnaround. Likewise, we can see that this behavior is similar for both kinds of environment, dedicated and non-dedicated cluster.

4 Conclusions and future work

This work presents a totally portable and integral system termed CISNE-P, which provides a space and time sharing scheduling applied to a non-dedicated cluster. It includes both a previously developed dynamic coscheduling system and a space-sharing job scheduler to make better scheduling decisions than they can do separately. CISNE-P allows multiple parallel application to be executed concurrently in a non dedicated Linux cluster with good performance, as much from the point of view of the local user as that of the parallel application user.

Using this framework, we evaluated two different scenarios of CISNE-P implementations, one located inside

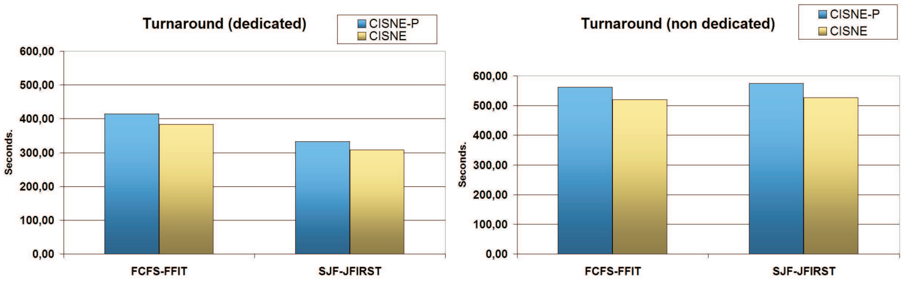

Figure 6: Turnaround of parallel applications: CISNE vs CISNE-P.

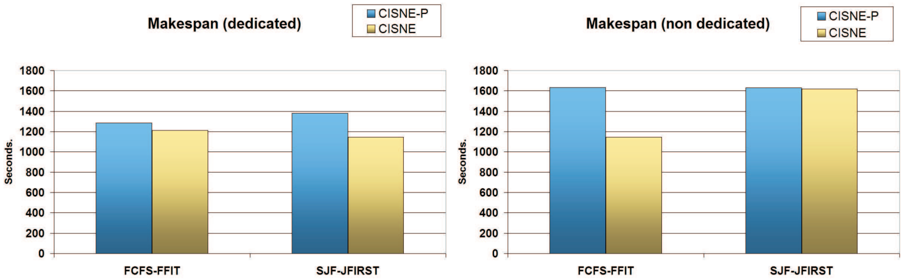

Figure 7: Makespan of parallel applications: CISNE vs CISNE-P.

the kernel, and the other, entirely implemented in the user space. The experimentation showed that the dynamic coscheduling and the Job Interaction Mechanism can be moved to the user space. The penalization in moving from the kernel to the user space is almost insignificant. Beside, the system improved in portability. We evaluated the influence of the dynamic coscheduling inside the CISNE-P environment. The results obtained show that the performance of parallel jobs is increased when coscheduling is applied.

Future work is oriented towards extending the functionalities of the CISNE-P system to provide facilities to execute Soft Real-Time jobs (local/parallel). Likewise, we are interested in extending the CISNE-P architecture to multicluster systems. Thus, we will be able to make better use of the computational resources of any kind of organization.

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A case for now (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[2] C. Anglano. A comparative evaluation of implicit coscheduling strategies for networks of workstations. *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 221–228, August 2000.

[3] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE 1995*, pages 267–277, 1995.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM:Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press Pub., 1994.

[5] F. Giné, F. Solsona, P. Hernández, and E. Luque. Cooperating coscheduling in a non-dedicated cluster. *EuroPar 2004, Lecture Notes in Computer Science*, 2790:212–218, 2004.

[6] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Coscheduling and multiprogramming level in a non-dedicated cluster. *EuroPVM/MPI 2004, Lecture Notes in Computer Science*, 3241:327–336, 2004.

[7] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Cisne: A new integral approach for scheduling parallel applications on non-dedicated clusters. *EuroPar 2005, Lecture Notes in Computer Science*, 3648:220–230, 2005.

[8] R. Miller. Response time in man-computer conversational transactions. *AFIPS Fall Joint Computer Conference Proceedings*, 33:267–277, 1968.

[9] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *J. Performance Evaluation*, 12(4):269–284, 1991.