Cooperating CoScheduling: A Coscheduling Proposal Aimed at Non-Dedicated Heterogeneous NOWs

Article *in* Journal of Computer Science and Technology · September 2007

DOI: 10.1007/s11390-007-9082-y · Source: DBLP

5 authors, including:

Francesc Giné
Universitat de Lleida
66 PUBLICATIONS **298** CITATIONS
SEE PROFILE

Francesc Solsona
Universitat de Lleida
163 PUBLICATIONS **717** CITATIONS
SEE PROFILE

Mauricio Hanzich
Barcelona Supercomputing Center
80 PUBLICATIONS **511** CITATIONS
SEE PROFILE

Porfidio Hernández
Autonomous University of Barcelona
82 PUBLICATIONS **382** CITATIONS
SEE PROFILE

Giné F, Solsona F, Hanzich M *et al.* Cooperating CoScheduling: A coscheduling proposal aimed at non-dedicated heterogeneous NOWs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 22(5): 695~710 Sept. 2007

Cooperating CoScheduling: A Coscheduling Proposal Aimed at Non-Dedicated Heterogeneous NOWs

Francesc Giné[1], Francesc Solsona[1], Mauricio Hanzich[2], Porfidio Hernández[2], and Emilio Luque[2]

[1] *Department of Computer Science, Universitat de Lleida, Spain*

[2] *Department of Computer Architecture & Operating Systems, Universitat Autònoma de Barcelona, Spain*

E-mail: {sisco,francesc}@diei.udl.cat; mauricio@aomail.uab.es; {porfidio.hernandez,emilio.luque}@uab.es

Received January 9, 2007; revised July 2, 2007.

Abstract Implicit coscheduling techniques applied to non-dedicated homogeneous Networks Of Workstations (NOWs) have shown they can perform well when many local users compete with a single parallel job. Implicit coscheduling deals with minimizing the communication waiting time of parallel processes by identifying the processes in need of coscheduling through gathering and analyzing implicit runtime information, basically communication events. Unfortunately, implicit coscheduling techniques do not guarantee the performance of local and parallel jobs, when the number of parallel jobs competing against each other is increased. Thus, a low efficiency use of the idle computational resources is achieved.

In order to solve these problems, a new technique, named Cooperating CoScheduling (CCS), is presented in this work. Unlike traditional implicit coscheduling techniques, under CCS, each node takes its scheduling decisions from the occurrence of local events, basically communication, memory, Input/Output and CPU, together with foreign events received from cooperating nodes. This allows CCS to provide a social contract based on reserving a percentage of CPU and memory resources to ensure the progress of parallel jobs without disturbing the local users, while coscheduling of communicating tasks is ensured. Besides, the CCS algorithm uses status information from the cooperating nodes to balance the resources across the cluster when necessary. Experimental results in a non-dedicated heterogeneous NOW reveal that CCS allows the idle resources to be exploited efficiently, thus obtaining a satisfactory speedup and provoking an overhead that is imperceptible to the local user.

Keywords job scheduling, non-dedicated and heterogeneous NOW computing, resource allocation

1 Introduction

Several studies[1,2] have revealed that a high part of computing resources (CPU and memory) in a Network Of Workstations (NOW/cluster) are idle. The possibility of using this computing power to execute distributed applications with a performance equivalent to a Massively Parallel Processors (MPP) and without perturbing the performance of the local user's applications on each workstation has led to new scheduling proposals. Several of these proposals are set out in the following frameworks.

• *Process Migration*: these techniques allow idle NOW resources to be used to execute rigid jobs made up of a fixed number of processes, which do not interact frequently. Migration involves the remapping of processes to processors during execution due to the need to relinquish a workstation and return it to its owner[3].

• *Remote Execution Services*: this approach is oriented to malleable jobs. It means that each job is executed by a variable number of worker processes, which are added and deleted as resources become available or are reclaimed by their owners[4].

• *Coscheduling*: if the processes of a parallel application communicate and synchronize frequently, it may be beneficial for them to execute simultaneously on different processors; this saves the overhead of frequent context switches and reduces the need for buffering during communication[5]. Such simultaneous execution is provided by gang scheduling[6]. In gang scheduling, all the processes in a job are scheduled and de-scheduled at the same time, so the processes making up jobs should be known in advance. In distributed systems like NOWs, this information is very difficult to obtain. The alternative is to identify the processes in need of coscheduling, also named *cooperating tasks*, during execution by gathering and analyzing implicit runtime information, basically communication events[7]. Thus, only a sub-set of the processes are scheduled together, leading to *coscheduling* rather than gang scheduling.

Both *remote execution services* and *process migration* permit parallel jobs to run only when a worksta-

Regular Paper

This work was supported by the MEyC under Grant No. TIN 2004-03388.

tion's owner is not using this machine. Thus, such systems focus on coarse-grained idle periods (on a scale of minutes or hours). However, there are other cycles available that such systems do not harvest. This is due to the fact that even when the user is actively working on the machine, the processor is idle for a substantial fraction of the time. Coscheduling techniques exploit these fine-grain idle cycles keeping both local and parallel jobs together. Thus, they achieve a more efficient use of computing resources.

Previous works[8,9] have revealed the good performance achieved by coscheduling techniques when a single parallel job is executed in a non-dedicated NOW. However, the rapid improvement in the computational capabilities of NOW systems increases the inactivity of resources such as Memory, CPU, communication links, and so on, even more. In order to increase the efficient use of these idle computational resources, various authors[10~12] have pointed to the need to increase the number of parallel jobs executed concurrently, a term named MultiProgramming Level of parallel jobs (MPL).

The rise in the MPL means that the complexity of the coscheduling problem has increased. As we will show in this article, an effective coscheduling technique must deal with the following aspects.

1) *Adaptive and Balanced Resource Allocation.* The MPL should be dynamically adapted according to the CPU and Memory requirements of each job and the heterogeneity of the NOW. Likewise, the computational resources assigned to tasks belonging to the same job must be balanced.

2) *Coscheduling of the Communication-Synchronization Processes.* No processes should wait for a non-scheduled process (cooperating) for synchronization/communication and the waiting time at the synchronization points must be minimized.

3) *Social Contract.* The performance of local applications must be guaranteed under a specific satisfaction threshold.

A new proposal which guarantees the above coscheduling features, named Cooperating CoScheduling (CCS), is presented in this work. Unlike traditional coscheduling techniques, under Cooperating CoScheduling each node takes its scheduling decisions from the occurrence of local events, basically communication, memory, Input/Output and CPU, together with foreign events received from remote nodes. This allows CCS to provide a social contract based on reserving a part 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 NOW when necessary.

CCS was implemented in a Linux-PVM environment. This implementation allowed its performance to be evaluated in comparison with traditional coscheduling techniques in two different heterogeneous environments, namely a controlled NOW and a production NOW. This experimentation shows that CCS obtains better performance than the rest of coscheduling techniques evaluated, both from the point of view of the local user as that of the parallel applications user.

The rest of the paper is organized as follows. The related work is described in Section 2. In Section 3, the system model is defined. Based on this model, the CCS coscheduling algorithm is developed in Section 4. The performance of CCS is evaluated and compared in an heterogeneous NOW in Section 5. Finally, the conclusions and future work are detailed.

2 Related Work

Coscheduling[5] deals with minimizing synchronization/communication waiting time between remote processes. According to the mechanism used by coscheduling the communicating tasks, the literature [12] classifies these techniques into three groups: gang scheduling, communication-driven coscheduling and coscheduling extensions.

Gang scheduling, also named *explicit coscheduling*[6], is an efficient coscheduling algorithm for fine-grained parallel processes, which has been used mainly in supercomputers (CM-5, SGI workstations and so on). The advantage of a gang scheduler is the faster completion time because the processes of a job are scheduled together, while its disadvantage is the global synchronization overhead needed to coordinate a set of processes, the high overhead introduced into local tasks and the lack of latency hiding. Regarding this last point, it is worthwhile pointing out that Buffered Coscheduling[13] is the only pure gang scheduling approach that supports hiding communication latency but it heavily relies on a BSP-like programming model. These reasons have driven researchers to search for more flexible solutions that can be adapted better to environments as dynamic as non-dedicated heterogeneous NOWs.

Unlike gang scheduling, with a *communication-driven coscheduling* like dynamic coscheduling[9], predictive coscheduling[14], implicit coscheduling[15] or periodic boost[16], each node in a NOW has an independent scheduler, which coordinates the communicating processes of a parallel job. All these coscheduling algorithms rely primarily on one of the two local events (arrival of a message and waiting for a message) to determine which process to schedule and when.

In *implicit coscheduling*[15], a process waiting for

messages spins for a determined time before blocking. *Demand-based coscheduling*, divided between dynamic and predictive coscheduling, was first introduced in [7]. In contrast to implicit coscheduling, *dynamic coscheduling*[9] deals with all message arrivals (not just those directed to blocked tasks). It is based on increasing the receiving task priority, even causing CPU preemption of the task being executed inside. Predictive coscheduling is based on scheduling the correspondent — the most recent communicated — processes in the overall system at the same time. To our knowledge, the implementation in Solsona *et al.*[14] is the only implementation of this technique. In the Periodic Boost scheme[16], a periodic mechanism checks the endpoints of the parallel processes in a round-robin fashion and boost the priority of one of the processes with unconsumed messages based on some selection criteria. All these coscheduling schemes have been evaluated on non-dedicated homogeneous NOWs[8,17] and have shown variable performance depending on the kind of parallel and local workloads and the architecture of the NOW. In general, the conclusion is that in the presence of a good coscheduling algorithm, it is reasonably possible to allocate multiple parallel jobs without performance penalty. Such schemes not only reduce the response time per-job, but also increase the overall system throughput, because multiprogramming frees up other nodes. However, from the point of view of the local users[8,11], these schemes do not guarantee the performance of local users when the maximum number of parallel applications (MPL) is higher than one.

So, the single analysis of communication events is not enough to guarantee the performance of local and parallel jobs when the MPL is increased. Basically, this is due to: (a) communication-driven scheduling imposes no restriction on the resources assigned to parallel users[8,14]; (b) the load imbalance produced by the presence of local users[18,19] drastically reduces the performance of parallel jobs; (c) the CPU[20] and Memory[21] must be assigned according to the degree of heterogeneity of each node in the NOW and (d) an excessive memory paging due to the $MPL > 1$ may slow the entire system down[21~23]. Unlike these related works, which deal with each of these aspects in isolation way, CCS is able to confront all the above aspects globally in a coordinated way. Thus, CCS determines not only *when* and *which* process to schedule, like traditional coscheduling schemes, but also *how much* of the resources (CPU and Memory) may be assigned to each task.

In preliminary work[24], we presented initial benchmark results for CCS, running on a controlled NOW. CCS was further tuned and simplified and provided better performance results. Thus, CCS has been tested in a set of controlled NOWs with different levels of heterogeneity. In addition, this paper extends that work with new experiments on a production heterogeneous NOW, with real local users and using more realistic workloads.

3 System Model

In this section, our model for NOW systems is explained. The description of the model, according to the classification detailed in the previous section, is divided into two basic sub-models: the traditional coscheduling model, which is made up of CPU scheduling and communication subsystems, and the extension coscheduling model, which provides for the I/O subsystem, local user interaction and the state information to be exchanged between cooperating nodes of the NOW. Fig.1 shows all the subsystems taken into account by our non-dedicated heterogeneous NOW model.

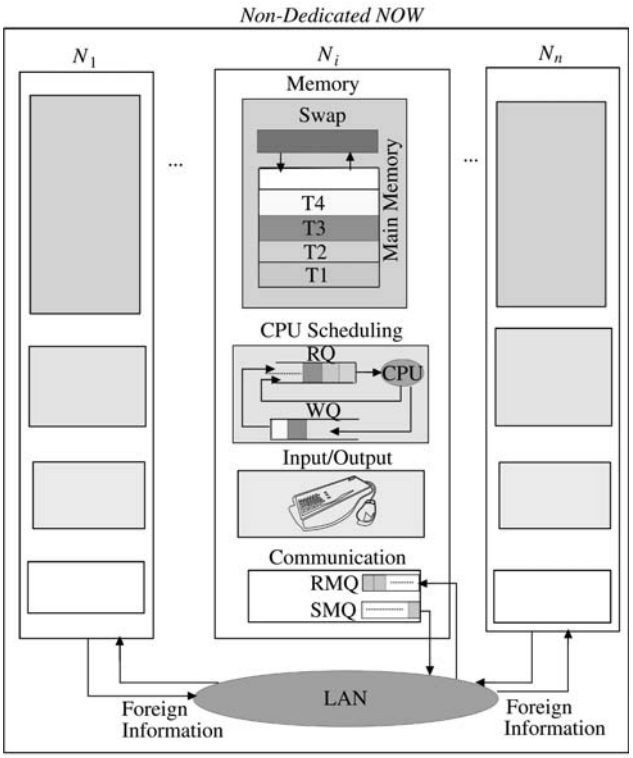

Fig.1. Non-dedicated heterogeneous NOW model.

3.1 Traditional Coscheduling Model

The traditional coscheduling model provides a framework for the execution of various distributed applications at the same time. This model assumes that all the nodes in a cluster (or NOW) are under the control of our coscheduling scheme. Based on that, some preliminary assumptions must be performed.

• The operating system of each node making up the NOW provides a time sharing scheduler with process preemption based on ranking processes according to their priority. It works by dividing the CPU time into *epochs*. In a single epoch, each process is assigned a specified quantum, which it is allowed to run. The epoch ends when all the processes in the Ready Queue have exhausted their quantum. Linux and Solaris are well known examples.

• Incoming (out-going) messages to (from) a NOW node are buffered in a Receiving Message Queue, *RMQ* (Sending Message Queue, *SMQ*). This is a non-arbitrary assumption. For example, Unix-like systems, with the standard *de facto* TCP (UDP)/IP protocol, maintain some sort of these buffers with specific differences.

• A networked workstation (node) is no longer the private resource of its local user (i.e., a computational laboratory of any institution). Rather, it is shared by other users and jobs.

Let a NOW or cluster $C = \{N_1, \ldots, N_n\}$, where n is the number of nodes in the cluster, and $Wrk = \{Job_1, \ldots, Job_m\}$, a set of m jobs executed into the cluster C. Each Job_k is composed of a set of h tasks distributed across the cluster, where $h = 1$ for local jobs and $h > 1$ for distributed jobs.

If the NOW C consists of a set of identical nodes, C is homogeneous. Otherwise, C is heterogeneous. Here we only consider the differences in computing capability among nodes. We use the power weight ($W(N_k)$) to refer to a node's computing capability relative to the fastest workstation in a cluster. According to the meaning given in [25], the power weight of node N_k is defined as follows:

$$W(N_k) = \frac{BOGO(N_k)}{BOGO(N_{\max})}, \tag{1}$$

where $BOGO(N_k/M_{\max})$ is a measurement of the computational capabilities of the k-th and fastest node, respectively. Note as this measurement is provided by some o.s. (p.e., bogomips in Linux) in the boot period. From the power weight of each node, we defined the heterogeneity of NOW C as follows:

$$Heterogeneity(C) = \frac{\sum_{k=1}^{n}(1 - W(N_k))}{n}, \tag{2}$$

where n is the number of nodes in the NOW C.

The basic notation used in the remainder of this article is:

• $RQ(N_k)$: ready queue of N_k. For this queue, we define two different operations: $Insert(RQ(N_k), T, l)$, which means that task T is inserted into the l-th position of the RQ, and $get(RQ(N_k), j)$, which returns the j-th task T of RQ. It is assumed that tasks are arranged according to their priority. Special cases are position 0 (currently executing task in the CPU) and position ∞ (the last one to be executed).

• $COM(T)$: number of current receiving and sending messages for task T in the RMQ and SMQ queues.

• $Quantum(T)$: dynamic quantum associated to task T. By default, at the beginning of each epoch, the quantum of every task T of the node N_k is set to $Def\_Quantum(N_k)$, defined as:

$$Def\_Quantum(N_k) = \frac{Def\_Q_{\text{o.s.}}}{W(N_k)}, \tag{3}$$

where $Def\_Q_{\text{o.s.}}$ is the base time slice of the operating system (o.s.) and $W(N_k)$ is the power weight of N_k. Note that it provides equitable cycles to all the tasks belonging to the same job, independently of the node where they are mapped. Thus, if they are started simultaneously in each epoch, coscheduling can be achieved as if they were executed in a virtual "homogeneous" cluster.

• $DE(T)$: number of times that task T has been overtaken in the RQ by another task due to a coscheduling cause, since the last time such a task reached the RQ.

• $UID(T)$: user identification of task T. For simplicity, we assume that all the parallel jobs belong to the same user, who is identified as $PARAL$.

3.2 Extension Coscheduling Model

Our model assumes that the o.s. of each node provides demand-paging virtual memory. In order to balance resources throughout the cluster, the model also allows the exchange of memory status and local user activity information between the cooperating nodes.

Each node maintains the following information for each active task (T).

• $RMS(T)$: resident memory size for task T. It is worth pointing out that the task resident size may vary randomly according to the LRU page replacement algorithm.

• $FLT(T)$: number of page faults for task T during the last second.

• $JID(T)$: job identifier of task T. We assume that all the tasks belonging to the same parallel job, also named *cooperating tasks*, share the same identifier.

• $State(T)$: this is the state of the Job_k, which T belongs to. It means that all tasks belonging to the same Job_k will share the same state. The possible values of this field are: (a) $LOCAL$: Job_k has computing resources limited due to the presence of a local user in some node; (b) $STOP$: Job_k is stopped due to memory overflow in some node; (c) $NULL$: value by default (local jobs have this value for all their lives).

In addition, each node (N_k) maintains the following information.

• $M(N_k)$: main memory size of N_k.

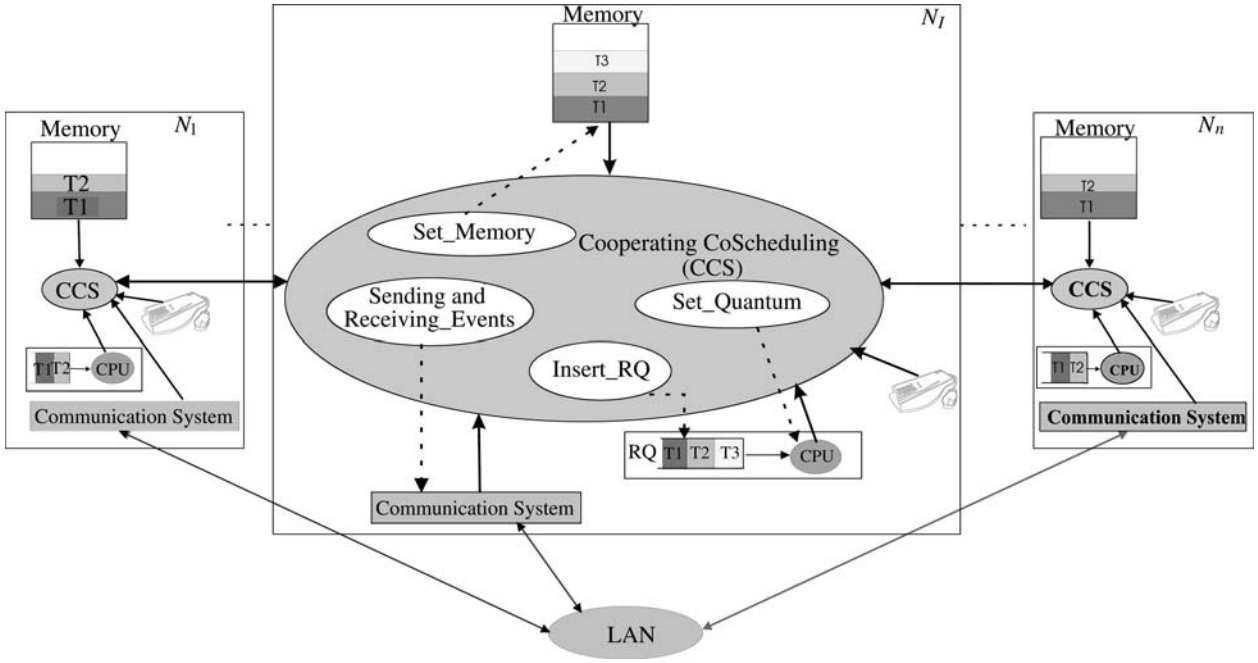

Fig.2. Cooperating CoScheduling architecture.

• *Mem/Mem_Paral*(N_k): sum of the memory requirements of all the/only parallel tasks in N_k.

• *Local_User*(N_k): this Boolean variable is set to TRUE when there is a local user in N_k.

• *MPL*(N_k): number of parallel tasks executing concurrently in N_k. In order to map the parallel tasks according to the power weight of each node, the maximum *MPL* of N_k is assigned as follows:

$$MPL(N_k) = MPL(N_{\max}) \cdot W(N_k), \qquad (4)$$

where $N_{\max}$ is a node of C, such that $W(N_{\max}) = 1$. According to this assignment and the quantum definition, given in (3), we can consider that the length of the epoch associated with parallel jobs on each node across the cluster will be the same.

• L: our model assumes that the computing power and the memory are explicitly divided into two parts: one part reserved for running parallel jobs (L) and the other reserved for the local ones ($1 - L$). L is fixed by the system administrator and it is assumed to be equal across the cluster.

• *M_Paral/M_Local*(N_k): portion of main memory assigned to parallel/local tasks.

4 CCS: Cooperating CoScheduling

Following the distributed nature of traditional communication-driven coscheduling techniques, Cooperating CoScheduling (CCS) resides in each node of the cluster.

Fig.2 shows the CCS architecture. CCS takes its scheduling decisions from the occurrence of local events, basically communication, memory, I/O and CPU, together with foreign events received from cooperating nodes. These events, shown by the continuous arrows, are dealt by five different modules: *Insert_RQ*, *Set_Quantum*, *Set_Memory*, *Sending_Events* and *Receiving_Events*. Each one is in charge of managing the subsystem indicated by the discontinuous arrow. The Insert_RQ module manages the priority of the processes in the RQ taking memory and communication events into account. Thus, CCS is able to *coschedule parallel applications under memory constraints*. Likewise, the Set_Quantum and Set_Memory modules assign the quantum length at the beginning of each scheduling epoch and the memory portion to each task respectively. Both of these modules implement the *social contract* between local and distributed users. Finally, the *adaptive and balanced resource allocation* is carried out by the Sending_Events and Receiving_Events modules.

The next subsections explains how the main modules of CCS work to reach the proposed expectations.

4.1 Coscheduling Under Memory Constraints

In previous work[21], we showed how the scheduling of the next task to be executed must take care of: (a) coscheduling of the communication-synchronization processes, (b) minimizing the number of page faults throughout the cluster and (c) preventing the starvation of the local jobs.

Accordingly, Algorithm 1 is proposed. It is implemented inside a routine (called Insert_RQ). The algorithm assigns a priority to tasks by means of arranging

them into the RQ according to the starvation condition, communication and memory requirements.

Algorithm 1. Insert_RQ(T). Inserts task T into the RQ of N_k.

$j := \infty$
$T_{\text{aux}} := get(RQ(N_k), j)$
if $(Mem(N_k) \leqslant M(N_k))$
  while $(COM(T_{\text{aux}}) < COM(T)$ and $DE(T_{\text{aux}}) < MNO$ and $j \neq 0)$
    $DE(T_{\text{aux}}) := DE(T_{\text{aux}}) + 1$
    $j := j - 1$
    $T_{\text{aux}} := get(RQ(N_k), j)$
else
  while $(FLT(T_{\text{aux}}) > FLT(T)$ and $DE(T_{\text{aux}}) < MNO$ and $j \neq 0)$
    if $(UID(T) \neq PARAL)$
    $DE(T_{\text{aux}}) := DE\ (T_{\text{aux}}) + 1$
    $j := j - 1$
    $T_{\text{aux}} := get(RQ(N_k), j)$
$Insert(RQ(N_k), T, j)$

In the nodes where the main memory is underloaded ($Mem(N_k) \leqslant M(N_k)$), a coscheduling technique based on assigning more scheduling priority to tasks with higher communication rates is applied to ensure that fine-grained distributed applications are coscheduled. In contrast to traditional coscheduling techniques, both sending and receiving frequencies are taken into account. Solsona[14] shows that processes performing both sending and receiving have higher potential coscheduling than those performing only sending or receiving.

Otherwise, if the main memory is overloaded ($Mem(N_k) > M(N_k)$), the scheduling condition will depend on the page fault rate of each task. More priority is assigned to tasks with a lower page fault rate. Taking into account the LRU page replacement algorithm assumed in the extension coscheduling model, the pages associated with tasks with less chance of being scheduled will get old early. So, every time a page fault happens, the older pages will be replaced by the missing pages. It means that with time, both more CPU time and more memory resources will be allocated to tasks with low page fault rates.

In order to avoid both the starvation of local tasks and the distributed ones with low communication rates, the inserting task overtakes only the tasks whose *delay* field ($DE(T)$) is not higher than a constant named *MNO* (Maximum Number of Overtakes). According to the results shown in [14], the default value for *MNO* is 2, as higher values may decrease the response time (or interactive performance) of local tasks excessively.

4.2 Social Contract

We propose to manage 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 a cession of a minimum percentage (L) of computing power and memory for parallel tasks on each node N_k. 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. L is fixed by the system administrator and it is assumed to be equal across the NOW.

Algorithm 2. Set_Quantum(N_k). CPU social contract algorithm for node N_k.

foreach $(T \in N_k)$
  if $((State(T) = STOP)$ or $(T = stoptask))$
    $Quantum(T) := 0$
  else
    if $((Local\_User(N_k)$ and $UID(T) = PARAL)$ or $(State(T) = LOCAL))$
      $Quantum(T) := Def\_Quantum(N_k) \cdot L$
    else
      $Quantum(T) := Def\_Quantum(N_k)$

Regarding the CPU, at the end of each epoch, the scheduler reassigns the quantum of every task according to Algorithm 2. Whenever a parallel task is stopped due to memory considerations, the scheduler assigns such task a quantum equal to zero. On the other hand, the scheduler decreases the quantum of the parallel task proportionally to the percentage L, whenever there is a local user in such a node N_k or the task is in a *LOCAL* state.

Memory ($M(N_k)$) is divided into two pools: one for the local tasks ($M\_Local(N_k)$) and the other for the parallel ones ($M\_Paral(N_k)$). The memory portion of parallel tasks is assigned according to the following equation:

$$M\_Paral(N_k) = (\min_{i=1}^{n} M(N_i)) \cdot L, \tag{5}$$

where the first factor is associated with the smallest memory size across the cluster. Thus, the same minimum memory size is assigned to parallel tasks in each node, independently of the memory size of each node. Likewise, the performance of local jobs is guaranteed because ($M\_Local(N_k) = M(N_k) - M\_Paral(N_k)$) is equal to or greater than what the local user expects ($M(N_k) \cdot (1 - L)$).

Algorithm 3. Set_Memory(N_k). Memory social contract algorithm for node N_k.

if $(Mem\_Paral(N_k) < M\_Paral(N_k))$
  `swaptask` := *task with the largest RMS* (*resident memory size*)
else
  `swaptask` := *parallel task with the largest RMS*
  if $(Local\_User(N_k))$ `stoptask` := `swaptask`

The implementation of the memory social contract is shown in Algorithm 3. We can see how in the nodes with local users, where swapping is activated and the memory requirements of the paral-

lel tasks ($Mem\_Paral(N_k)$) exceed their assigned part ($M\_Paral(N_k)$), the parallel task with most pages mapped in memory (*swaptask*) is stopped. Thus, the resident memory size (RMS) of the remaining active tasks is increased and the swapping activity is also finished or, at least, reduced[21]. In this way, the multiprogramming level of parallel jobs is adapted dynamically to the memory requirements of the local and parallel users.

4.3 Adaptive and Balanced Resource Allocation

The social contract used to preserve local job performance limits the resources assigned to parallel tasks. This can cause non-desirable situations, as in Fig.3(a), which shows an unbalanced assignment of the resources throughout the cluster. A parallel job (Job_1) is executed on nodes N_1, N_2, N_3 and N_4. Simultaneously, another parallel job (Job_2) is executed on nodes N_3, N_4 and N_5. Nodes N_1 and N_2 assign 20% of computational resources (CPU and Memory) to Job_1, while the local user has the remaining 80% available. Nodes N_3 and N_4 assign half of the resources to each parallel job following a fairness policy, while N_5 assigns full resources to Job_2. This uneven taxing of resources creates a situation where some processes in the parallel program run slower than others. Thus, the performance of coscheduling techniques can be seriously degraded. However, if N_3 and N_4 had knowledge of the status of Job_1 and Job_2 in their cooperating nodes, they could take better decisions by giving Job_2 more resources than Job_1. Thus, the situation shown in Fig.3(b) would be achieved. In this case, Job_1 has the same amount of resources in each node, while Job_2 has more resources available than in the previous case. Likewise, local users retain the same resources as in Fig.3(a). In this way, the cluster resources are better balanced. This example illustrates the main aim of the adaptive and balanced resource allocation mechanism described in this subsection.

In order to balance resources throughout the cluster, we propose a distributed scheme where the cooperating nodes exchange status information. The key question is which information should be exchanged. An excess of information could slow down the performance of the system, increasing its complexity and limiting its scalability. For this reason, only the following four events are notified by each node N_k.

1) *LOCAL_U.* This event is sent whenever the local user activity begins. It means that the *Local_User* variable makes the transition from 0 to 1. In addition, this event is also sent when a new parallel task is mapped in the node N_k and there is a local user in such a node.

2) *NO_LOCAL.* This event is sent if the local activity finishes or the node has a local user and a parallel task has completed in such a node. This means that *MPL* variable is decreased.

3) *STOP.* This event is sent whenever a parallel task T has been stopped in the node N_k as a consequence of the social contract mechanism. This means that the *stoptask* variable makes the transition from NULL to *task* T.

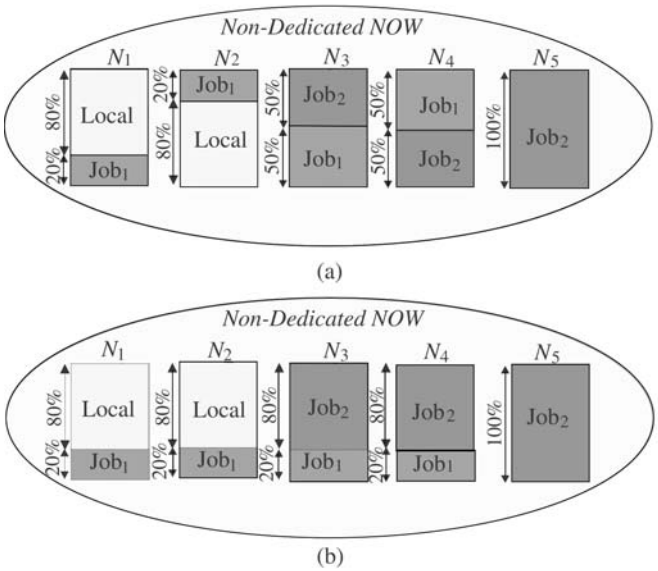

Fig.3. Examples of resource balancing.

4) *RESUME.* This event is sent whenever the *stopped task* T restarts its execution. This means that the *stoptask* variable makes the transition from T to NULL.

At the beginning of each epoch, the values of the *Local_User, MPL* and *stoptask* variables are checked by the *Sending_Events module* to determine if one of these events should be delivered. If so, each event is sent to all the nodes where there were tasks with the same job identification ($JID(T)$) of the parallel task T of N_k affected by these events.

When a cooperating node N_j receives one of the above events, the *Receiving_Events module* will reassign the resources according to the following steps. First, the task with the same received job identification must be found. Next, depending on the received event, this module will assign one of the following states to the target task.

1) *LOCAL_U.* CCS will assign a state equal to LOCAL. It means that Algorithm 2 (Set_Quantum) will assign a quantum equal to $(Def\_Quantum(N_j) \cdot L)$ to the parallel task notified by the event, whenever it was not stopped previously.

2) *NO_LOCAL.* CCS will assign the state NULL whenever the number of *LOCAL_U* received events coincides with the *NO_LOCAL* received events. In this

way, the algorithm ensures that the notified parallel job is not slowed down by any local user across the NOW. Note that the Set_Quantum algorithm will assign the complete quantum base ($Def\_Quantum(N_j)$) to tasks with the NULL state.

3) *RESUME*. As in the previous case, CCS will assign the state NULL and the Set_Quantum algorithm will act consistently.

4) *STOP*. In this case, the notified task will be stopped. However, if a node with overloaded memory receives the *STOP* event for a task different than the *stoptask*, the algorithm will stop the notified task and the last *stoptask* will be resumed. Finally, it is worth pointing out one deadlock situation taken into account on reception of a *STOP* event. If a node N_j receives a *STOP* event from N_k for a task belonging to one parallel job, just when this node N_j has sent another *STOP* event to the same transmitter node N_k for a task belonging to another job, a deadlock situation could be reached. Whenever the algorithm detects this situation, the parallel job with the lowest identifier *JID* will be stopped in both nodes and consequently, across the cluster.

4.4 Implementation of CCS in a *PVM-Linux* NOW

The implementation of *CCS* is shown in Fig.4. *CCS* was implemented in a PVM (v.3.4)-Linux (v.2.4.22) cluster. PVM provides useful information for implementing the algorithms for Sending and Receiving events described in Subsection 4.3. Every node of a PVM system has a daemon, which maintains information of the PVM jobs under its management. It contains the identifier of each job (*JID*) and a host table with the addresses of its cooperating nodes. These PVM 's characteristics made the implementation easier.

Likewise, the Set_Quantum, Set_Memory and Insert_RQ modules are implemented in the kernel space. This means that 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_U* (*NO_LOCAL*) events when there is (no) 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[26]. At the beginning of every epoch, the access time to the keyboard and mouse files is checked, setting a new kernel variable (*Local_U*ser) 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 implement the Set_Memory module and guarantee the memory portions for local (*M_Local*) and parallel (*M_Paral*) 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 Set_Quantum module involves the modification of the base time quantum according to the algorithm explained in Subsection 4.2. The coscheduling implementation (Insert_RQ module) increases the

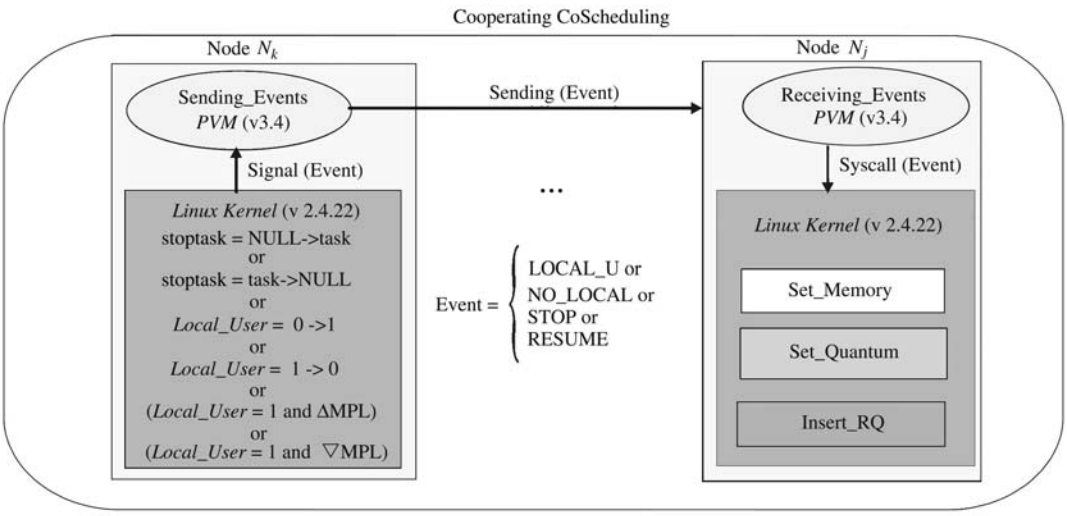

Fig.4. Implementation of CCS.

Table 1. Experimental Environments

Heterogeneity (C)	Node Characteristics	Network
$C(0)$	{16 PIVs 3GHz (512MB)}	Fast Ethernet
$C(0.15)$	{8 PIVs 3GHz (512MB), 8 PIVs 2.6GHz (256MB)}	Fast Ethernet
$C(0.25)$	{8 PIVs 2.6GHz (256MB), 8 PIVs 1.4GHz (256MB)}	Fast Ethernet

Table 2. Local User Requirements

Local	Distribution (%)	CPU_Load (%)	Memory	Network (Bytes/s) Rec. - Send
Shell	5	0.4 (0.2)	20 (15)	108-3 (30-1)
Xwin	62	0.15 (0.1)	35 (55)	608-30 (302-5)
Internet	33	0.2 (0.1)	60 (75)	3154-496 (1890-245)

Note: The standard deviation is shown between brackets.

dynamic priority of each parallel task inserted in the Ready Queue (RQ) according to the number of packets in the receive/send socket queue or the page fault rate calculated by the memory subsystem. Thus, the current scheduled task can be preempted by the task inserted into the RQ with most pending messages or lowest page fault rate. In this way, coscheduling is achieved.

5 CCS Performance in a Non-Dedicated NOW

This section discusses the performance characteristics of CCS in relation to the three communication-driven coscheduling techniques cited in Section 2: Implicit (spin is twice the context switching cost), Dynamic and Predictive coscheduling. The implementation of these techniques is based on the description given in [14]. In addition, these techniques were compared with the proposals of Zhang *et al.*[20] and Polychronopoulos *et al.*[22] described in Section 2.

This experimentation was done in the three different NOWs shown in Table 1. Every NOW C is identified by its heterogeneity, measured according to (2). Each NOW was used in two environments: the *production* and the *controlled NOW*. The *production NOW* is characterized by the fact that the NOW is open to the students. In this way, we could test the performance of CCS in a real non-dedicated environment.

In the *controlled NOW* case, the same laboratory is closed to students. The local workload was carried out by running one synthetic benchmark, called *local*. This benchmark alternates CPU activity with interactivity by running several system calls and different data transfers to memory. This is configured according to three different input parameters: the CPU load, memory requirements and network traffic. Thus, we can simulate any kind of local user profile. In order to assign these values in a realistic way, we monitored the average resources used by real users. According to this monitoring, we defined three local user profiles, characterized in Table 2: *Shell*, a user with high CPU requirements; *Xwin*, high interactivity requirements; and *Internet*, high communication requirements. The *distribution* column of Table 2 shows the percentage of monitored users for each profile. In order to simulate the high variability of local user behavior, the local execution time was modeled by a two-stage hyper-exponential distribution[26] with means, by default, 10 and 60 minutes and a weight of 0.6 and 0.4 for each stage. Each new local benchmark was executed in a node without any local users. At the end of its execution, the *local* benchmark returns the system call latency and the wall-clock execution time.

The performance of the parallel jobs was evaluated by running PVM jobs from the NAS suite with classes A and B. These benchmarks were gathered into three workloads according to their communication requirements: *COM_L* (*Low*), *COM_M* (*Medium*) and *COM_H* (*High*). Each workload was composed of 48 jobs chosen uniformly from the set of jobs depicted in Table 3. These 48 jobs were submitted following a Poisson distribution[6] in such a way that a maximum of *MPL* jobs were run into the NOW simultaneously. The size of each job (4, 8 or 16 tasks) was chosen according to an uniform distribution.

Table 3. Parallel Workloads (Wrk_{paral})

Wrk_{paral}	Benchmarks
COM_L	EP.A, LU.A, SP.A, EP.B, BT.B
COM_M	LU.A, FT.A, MG.B, SP.A, BT.B
COM_H	IS.A, IS.B, FT.A, MG.B, CG.B

The maximum *MPL* of each node N_k was calculated in function of its power weight ($W(N_k)$), according to (4). For instance, if the *MPL* of each fast node ($W(N_{\max}) = 1$) belonging to the NOW $C(0.15)$ is 3, then the *MPL* of each slow node of $C(0.15)$ will be $[3 \cdot 0.7] = 2$, where 0.7 is the power weight of each slow node. Henceforth, the maximum *MPL* of the NOW will be associated with the *MPL* of the fastest nodes.

We measured the *speedup* of each Wrk_{paral} accord-

ing to the following equation:

$$speedup(Wrk_{\text{paral}}) = \frac{Texec_N(Wrk_{\text{paral}})}{Texec_C(Wrk_{\text{paral}})}, \quad (6)$$

where $Texec_C(Wrk_{\text{paral}})$ and $Texec_N(Wrk_{\text{paral}})$ are the execution time of Wrk_{paral} (also called makespan) in the NOW and on a single node, respectively. In addition, we calculated the average speedup of every job J making up Wrk_{paral} ($Job_j \in Wrk_{\text{paral}}$) according to the following formula:

$$speedup(Job_j) = \frac{1}{m}\sum_{j=1}^{m}\frac{Texec_N(Job_j)}{Texec_C(Job_j)}, \quad (7)$$

where m is the number of jobs in the parallel workload. Note that (6) and (7) give a measure of the overall system throughput and the overall response time per-job, respectively.

5.1 Controlled NOW

Firstly, we evaluated the contribution of each module of CCS into the global performance separately. Starting from the dynamic technique, denoted as DYN, we measured the slowdown of local tasks and the speedup of distributed jobs, each time that a new module of the CCS was added to the system. These trials were done with the COM_M workload with the $MPL = 3$, and 8 local users. In order to evaluate the performance of every module under different memory constraints, we used two environments. In the first one, we used only *Shell local users* and all the tasks fitted in the main memory. The second one was made up exclusively of *Internet local users*, and the memory of such nodes with local users was overloaded.

Following the same order as in Section 4, the Insert_RQ module was the first one evaluated. Fig.5(left) shows how the Insert_RQ module, denoted as RQ, doubles the speedup of parallel applications in environments with high memory requirements. This is due to the memory control policy applied by this module, consisting of giving more execution priority to the distributed task with the lowest page fault rate. Thus, the task with the lowest memory requirements finishes its execution sooner and so the memory available for the rest of tasks is considerably increased, leading to a speedy execution of such tasks. Likewise, the slowdown of local tasks (see Fig.5(right)) is decreased considerably, around 20%, due to the limitation in the number of overtakes suffered by a task inserted in the RQ and the reduction of the page fault rate achieved by this module. However, this slowdown continues being over 3 for both kind of environments, which may be considered unacceptable by the local user. This reveals the need for adding the social contract mechanism, which is implemented by the Set_Quantum and Set_Memory modules.

Fig.5 shows the good behavior of the Set_Memory module. When the memory is overloaded (COM_M+Internet), the slowdown of local tasks improves drastically by over 50%; while the speedup of parallel tasks stays constant. Regarding the Set_Quantum module, denoted as Quantum, we can see a significant reduction of the slowdown when memory is underloaded. This improvement, of over 45%, is due to the fact that the quantum of the parallel tasks is decreased proportionally to the percentage L, whenever there is local activity in one node. This reduction of the quantum provokes the reduction of the waiting time of local tasks in the RQ. Unfortunately, this limitation in the quantum length makes the speedup of parallel jobs worse due to the fact that the resources assigned to parallel tasks belonging to the same job are unbalanced across the NOW. For this reason, the inclusion of Sending/Receiving_Events modules, denoted as Events, allows the resources to be assigned in a coordinated way and as a consequence, the speedup of parallel task is improved by over 40%, while the slowdown maintained lower than 2 for both environments.

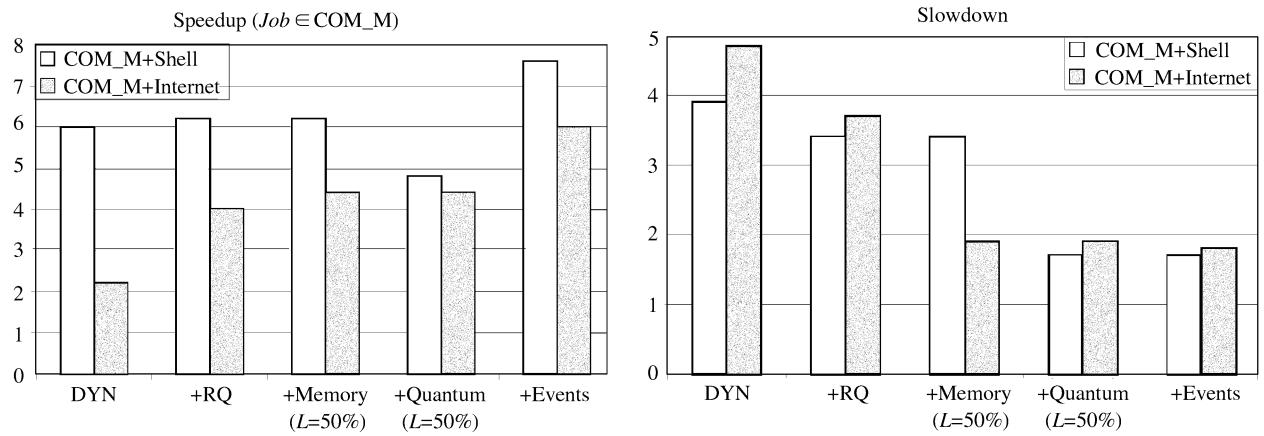

Fig.5. Contribution of each module to CCS performance.

Next, the effect of the percentage of resources (L) assigned to parallel jobs was analyzed. We ran the three parallel workloads separately with an $MPL = 3$ and three L values: 0.25, 0.5 and 1. The performance of CCS with different L values was compared in relation to the Predictive, Implicit and Dynamic techniques. The number of local users was also varied from 0 to 16. Local users were chosen in the same proportion as the *Distribution* column in Table 2 (it means 5% of the Shell kind, etc.). These trials were done in the $C(0.15)$ NOW.

Fig.6 shows the speedup of the parallel jobs for each workload. We can see that there is a high dependency on the kind of workload. Regarding the COM_L workload, the performance of the three communication-driven coscheduling techniques (Implicit, Predictive and Dynamic) is similar. Likewise, we can see that the difference of CCS with $L = 100\%$ in relation to the rest of techniques increases with the number of local users. This is due to the capacity of CCS for balancing the idle resources between parallel tasks and, as a consequence, a better utilization of the NOW is achieved. When the communication requirements of the parallel workload are increased (see Fig.6(b)), the Predictive coscheduling technique obtains better results ($\simeq 20\%$) than Dynamic. This is due to the fact that Predictive coscheduling takes both the past and present messages into account, whereas Dynamic only considers the present ones, so the opportunity for promoting through the Ready Queue is greater in the Predictive case. Likewise, the bad behavior of the Implicit technique is surprising. This can be explained by the fact that the irregular message arrival provokes the inefficiency of the spinning period applied in this technique. As in the previous case, CCS with $L = 100\%$ also obtains the best performance.

The poor performance achieved by Dynamic, Predictive and Implicit with the COM_H workload (see Fig.6(c)) is due to the fact that with this workload and *MPL*, the memory is overloaded in those nodes with 256MB of memory and local users. As a consequence, the coscheduling benefits are totally hidden by the paging effects. In this particular case, we can see that CCS is able to maintain the speedup practically constant due to its capacity for analyzing the memory requirements and for varying the *MPL* dynamically.

Likewise, we can see, from the CCS performance, that the difference between $L = 100\%$ and 50% ($\simeq 10\%$) is much lower than that between $L = 50\%$ and 25% ($\simeq 25\%$). This asymmetric behavior is due to: (a) the time slice not being big enough to exploit the locality of the parallel application data into the cache[21] with an $L = 25\%$ and (b) this kind of benchmarks being characterized by executing short CPU periods, which do not expire all the default Linux quantum (210ms). So, if CCS reduces the quantum by 50%, its performance is not notably affected. For this reason, we can see that the influence of L decreases in relation to the communication requirements. On the other hand, an $L = 100\%$ value does not produce a significant benefit in the parallel workload, but disturbs the local one excessively.

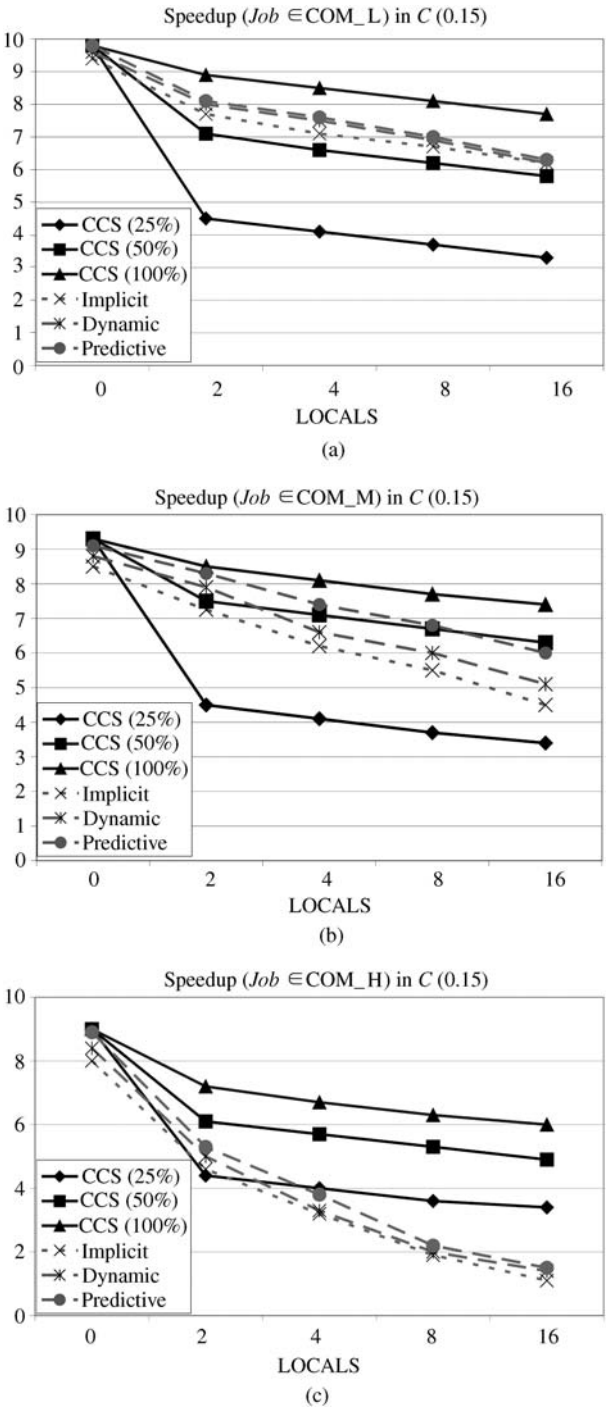

Fig.6. Average speedup of the parallel jobs according to the percentage of the assigned resources (L) and the number of local users (LOCALS).

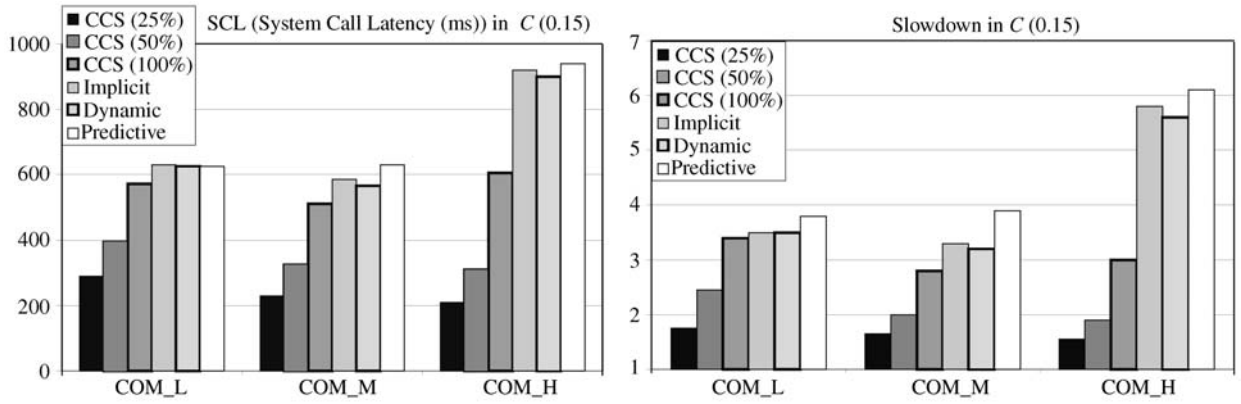

Fig.7. System call latency (left) and slowdown of local tasks (right).

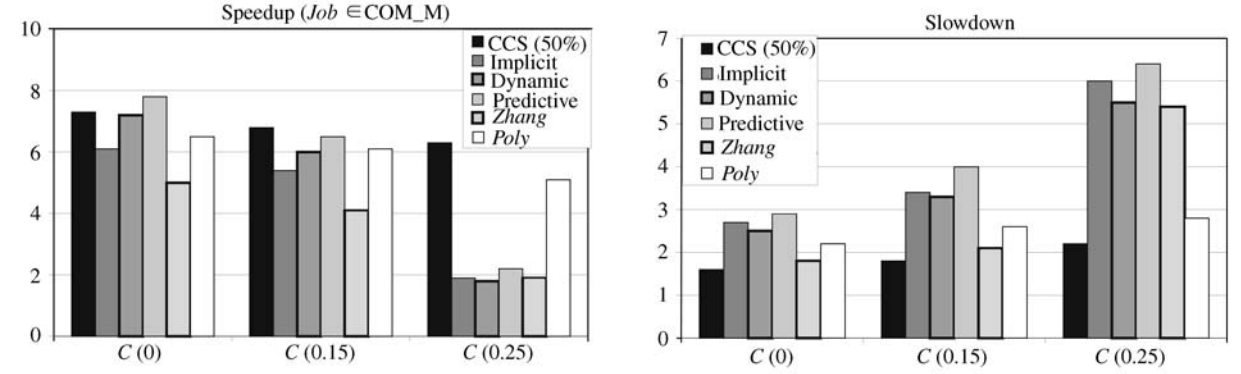

Fig.8. Average speedup of the parallel jobs belonging to COM_M workload (left) and slowdown of local jobs (right) in relation to the heterogeneity of the NOW.

Fig.7 shows the system call latency (SCL) and slowdown of the local workload while running jointly with each parallel workload in the same environment described above. The SCL measures the overhead introduced into the local users with high interactivity, while the slowdown reflects the overhead introduced into CPU bound ones. From Fig.7(left), it is possible to see that the SCL for $L = 50\%$ and 25% never exceeds 400ms (stated in [27] as the acceptable limit in disturbing the local user responsiveness). Likewise, the difference between $L = 100\%$ and 50% is higher in the COM_H workload because the likelihood of stopping one parallel task in the nodes where paging is activated increases when L decreases and, as a consequence, the local user performance is improved. This figure also shows the unacceptable intrusion for CCS with $L = 100\%$, and the rest of evaluated techniques, and more specifically when the latency > 800ms. The bounded slowdown (see Fig.7(right)) follows a similar behavior. Note that it increases with the value of L but always below the *MPL*.

Next, the performance of the coscheduling techniques in relation to the level of heterogeneity of the NOW was analyzed. With this aim, the COM_M workload was run with an $MPL = 3$ in the three NOWs: $C(0)$, $C(0.15)$ and $C(0.25)$. The number of local users across the NOW was set at 8 in each NOW. In this environment, the performance of CCS with $L = 50\%$ was compared with the Implicit, Predictive and Dynamic coscheduling techniques. In addition, these techniques were compared with the proposals of Zhang *et al.*[20] and Polychronopoulos *et al.*[22] described in Section 2. Zhang's proposal, denoted as *Zhang,* corresponds with the isolated Set_Quantum module of CCS, whereas the proposal by Polychronopoulos *et al.*, denoted as *Poly,* is similar to the Insert_RQ plus Set_Memory modules of CCS described in Section 4.

The speedup of parallel jobs and the slowdown of local tasks are shown in Fig.8(left) and 8(right), respectively. Fig.8(left) shows the good performance of the Predictive technique for a homogeneous NOW ($C(0)$). However, its performance is degraded when the heterogeneity is increased due to the absence of any mechanism to coordinate the assignation of CPU and Memory of each node globally. Likewise, we can see that the slowdown introduced by the Predictive technique increases fast (> 3) with the level of heterogeneity. A similar behavior is found in the rest of

communication-driven coscheduling techniques.

On the other hand, the gain of CCS and the *Poly* proposals in relation to the rest of communication-driven coscheduling techniques increases with the level of heterogeneity. This is due to the fact that all these techniques assign the resources taking heterogeneity into account. It means that all parallel processes of the same job get the same number of CPU cycles and memory portion on average. Thus and according to the coscheduling principle, they can progress in a coordinated way. It is worth pointing out that the *Poly* proposal obtains a speedup near to CCS, although its slowdown is much worse due to the absence of any limitation of the use of the CPU by the parallel tasks. Regarding the *Zhang* proposal, we can see in Fig.8(right) that the slowdown in the $C(0)$ and $C(0.15)$ NOWs is lower than 2.25. This improvement is due to the fact that the quantum of the parallel tasks is decreased proportionally to the percentage L, whenever there is local activity in one node. Unfortunately, this limitation in the quantum length makes the speedup of parallel jobs worse (< 5) due to the fact that the resources assigned to parallel tasks belonging to the same job are unbalanced across the NOW. The increase of the slowdown associated to the *Zhang* proposal into the $C(0.25)$ NOW is due to the fact that the memory of some nodes is overloaded and, as a consequence, the slowdown of local tasks is degraded.

5.2 Production NOW

In order to test CCS in a NOW with real local users, the COM_L and COM_H parallel workloads were executed during sessions where the laboratory $C(0.15)$ was being used by students. Table 4 shows the three experimental scenarios. In addition, this table shows the average percentage of machines (% $Non\_Idle(C)$), CPU load ($CPU(N_k)$) and part of memory of each machine ($\%Mem\_Local(N_k)$) used by the students (local users).

Table 4. Experimental Scenarios

Environments	$\%Non\_Idle(C)$	$\%Mem\_Local(N_k)$	$CPU(N_k)$
Lab_Open	45 (20)	40 (50)	0.15 (0.25)
Lab_Statistic	100 (0)	60 (10)	0.35 (0.12)
Lab_EDI	100 (0)	35 (7)	0.10 (0.07)

Note: The standard deviation is shown between brackets

The Lab_Open scenario corresponds to the session where the laboratory is open to the students to work freely. Therefore, this scenario is characterized by a wide range of users with very different requirements. The Lab_Statistic scenario is associated with statistics practices, characterized by the use of a statistical package with high computation requirements. Finally, the Lab_EDI scenario is basic programming practices, where the students use a simple editor together with the standard *gcc* compiler to develop short C programs.

This experimentation was done according to the following methodology. First, we looked for the maximum system load, evaluated by means of the CPU load ($CPU_{\max}(N_k)$) and percentage of requested memory for each node ($\%Mem_{\max}(N_k)$)[28], supported by real local users without perceiving any delay in their work. With this aim, we measured the correspondence between the metrics used in the controlled NOW (slowdown and SCL) in relation to the CPU load and percentage of memory. We found that the maximum slowdown and SCL values, fixed in the controlled NOW at 2 and 400ms respectively, were achieved for a $CPU_{\max}(N_k)$ and $\%Mem_{\max}(N_k)$ of 1.7 and 80% for the Dynamic and Predictive coscheduling and 2.2 and 120% for CCS (50%). It is worthwhile pointing out that CCS is able to reduce dynamically the percentage of memory requested by parallel applications by around 25% and for this reason it is able to work with a percentage of memory over 100%.

As soon as $CPU_{\max}(N_k)$ and $\%Mem_{\max}(N_k)$ were fixed, the parallel workload was executed with an *MPL* adapted dynamically to the variation of these parameters. This means that both parameters were sampled periodically (Sampling period of 30s) in each node in order to determine when one of these parameters reached the maximum value. If it was reached in a single node, the last launched parallel job was killed and, as a consequence, the *MPL* was decreased.

Fig.9 shows the speedup of COM_L and COM_H workloads in each of the scenarios in Table 4. Likewise, the speedup obtained in the same NOW, when it was dedicated to a parallel workload, is shown. The maximum *MPL* achieved by each policy and scenario is shown at the top of each bar.

Fig.9 shows that the best results for both workloads were obtained in the Lab_EDI scenario, due to the low use of resources by the students, whereas the worst result was obtained in the Lab_Statistic. Except for the dedicated case, we can see that Dynamic and Predictive are only able to reach an $MPL = 2$ in the best case, whereas CCS is able to work with an $MPL = 4$ the Lab_EDI scenario. This high MPL is translated into a better speedup (> 7.5). Note that the maximum gain of CCS in relation to the rest of policies ($\simeq 45\%$) is achieved in the Lab_Open scenario.

Table 5. Average CPU Local and Percentage of Memory Used Across the NOW in the Lab_Open Scenario

	COM_L		COM_H	
	$CPU(N_k)$	$\%Mem(N_k)$	$CPU(N_k)$	$\%Mem(N_k)$
CCS	1.8 (0.4)	82 (15)	1.95 (0.25)	84 (18)
Dynamic	0.85 (0.2)	46 (21)	0.4 (0.3)	49 (31)
Predictive	0.85 (0.2)	44 (23)	0.5 (0.3)	50 (36)

Note: The standard deviation is shown between brackets.

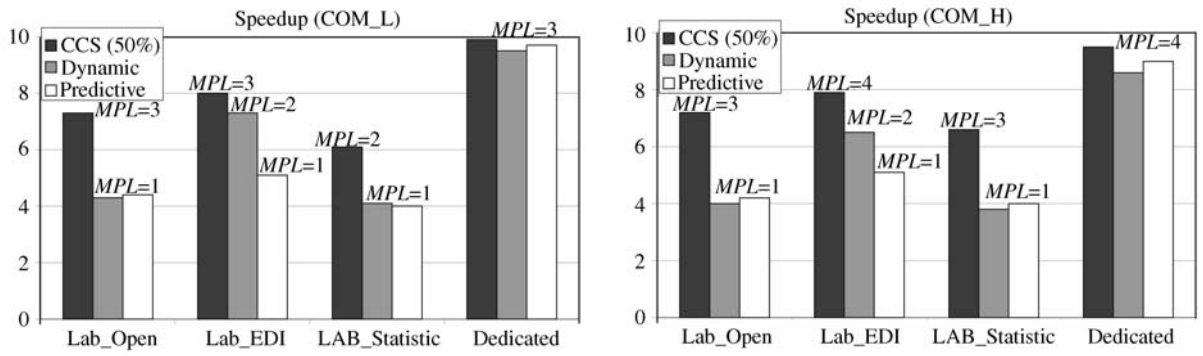

Fig.9. Speedup of COM_L (left) and COM_H (right) parallel workloads.

This scenario, characterized by high scattering into the idle resources, allows CCS to exploit its capacity to adapt the idle resources dynamically according to both architecture information and system-wide state. This behavior of CCS is reflected in Table 5, where the average CPU load ($CPU(N_k)$) and percentage of memory used ($\%Mem(N_k)$) across the NOW are shown. We can see that CCS is able to optimize the resource usage, achieving a $CPU(N_k)$ and $\%Mem(N_k)$ near to the maximum threshold supported by the real users.

In relation to the comparison with the values obtained by CCS in a dedicated environment, Fig.9 shows that even in the worst case (COM_L and Lab_Statistic scenario) the difference is less than 50%. In this sense, we must remark that the speedup obtained by executing over a non-dedicated NOW is zero-cost for the institution or company that owns the NOW. Finally, it is worthwhile noting that the workload speedup profit is almost always greater than the reduction in the speedup of each parallel application instance. Therefore, each parallel application loses less than the whole workload gains.

6 Conclusions and Future Work

In this paper, a new coscheduling technique, named Cooperating CoScheduling (CCS), is applied to schedule parallel applications effectively in a heterogeneous non-dedicated NOW. CCS uses a time-slicing technique to exploit the unused computing capacity of a non-dedicated NOW without disturbing local jobs excessively. With this aim, CCS limits the CPU and memory resources assigned to parallel tasks by means of 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 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 operating system and its cooperating nodes, together with architecture information and system-wide information. Thus, local decisions are coordinated across the NOW.

The performance of CCS was tested in a heterogeneous non-dedicated NOW and compared with other coscheduling policies. The results obtained reveal that CCS allows the idle resources to be exploited efficiently, thus obtaining a satisfactory speedup (in many cases higher than half the number of tasks of the application) and provoking an overhead that is imperceptible to the local user.

Our next work is directed towards applying CCS together with a space-slicing technique. Our aim is to divide the NOW considering both architecture information and system-wide state and execute multiple parallel programs in every NOW partition by means of applying CCS. Likewise, we are interested in extending our system to support soft-real-time applications from both the local and parallel user points of view.

References

[1] Acharya A, Edjlali G, Saltz J. The utility of exploiting idle workstations for parallel computations. In *Proc. the ACM SIGMETRICS/PERFORMANCE*, USA, 1997, pp.225~236.

[2] Acharya A, Setia S. Availability and utility of idle memory in workstation clusters. In *Proc. the ACM SIGMETRICS/PERFORMANCE*, USA, 1999, pp.35~46.

[3] Carriero N, Freedman E, Gelernter D, Kaminsky D. Adaptive parallelism and piranha. *Computer*, 1995, 28(1): 40~49.

[4] Litzkow M, Livny M, Mutka M. Condor — A hunter of idle workstations. In *Proc. the 8th Int. Conf. Distributed Computing Systems*, USA, 1988, pp.104~111.

[5] Ousterhout J. Scheduling strategies for concurrent systems. In *Proc. the 3rd Int. Conf. Distributed Computing Systems*, USA, 1982, pp.22~30.

[6] Feitelson D. Packing schemes for gang scheduling. *Lecture Notes in Computer Science*, 1996, 1162: 89~110.

[7] Sobalvarro P, Weihl W. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. *Lecture Notes in Computer Science*, 1995, 949: 106~126.

[8] Anglano C. A comparative evaluation of implicit coscheduling strategies for networks of workstations. In *Proc. the 9th Int. Symp. High Performance Distributed Computing*, Japan, 2000, pp.221~228.

[9] Sobalvarro P, Pakin S, Weihl W, Chien A. Dynamic coscheduling on workstation clusters. *Lecture Notes in Computer Science*, 1998, 1459: 231~256.
[10] Frachtenberg E, Feitelson D, Petrini F, Fernandez J. Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *Proc. the Int. Parallel and Distributed Processing Symposium* (*IPDPS*), France, 2003.
[11] Hanzich M, Giné F, Hernández P *et al.* Coscheduling and multiprogramming level in a non-dedicated cluster. *Lecture Notes in Computer Science*, 2004, 3241: 327~336.
[12] Sodan A. Loosely coordinated coscheduling in the context of other approaches for dynamic job scheduling: A survey. *Concurrency and Computation: Practice and Experience*, 2005, 17(16): 1725~1781.
[13] Petrini F, Feng W. Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems. In *Proc. the Int. Parallel and Distributed Processing Symposium* (*IPDPS*), Mexico, 2000.
[14] Solsona F. Coscheduling techniques for non-dedicated cluster computing [Dissertation]. Dept. Computer Science, Universitat Autonoma de Barcelona, Spain, 2002.
[15] Dusseau A. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 2001, 19(3): 283~331.
[16] Nagar S, Banerjee A, Sivasubramaniam A, Das C. Alternatives to coscheduling a network of workstations. *J. Parallel and Distributed Computing*, 1999, 59: 302~327.
[17] Choi G, Agarwal S, Kim J, Yoo A, Das C. Impact of job allocation strategies on communication-driven coscheduling in clusters. *Lecture Notes in Computer Science*, 2003, 2790: 160~169.
[18] Frachtenberg E, Petrini F, Feitelson D, Fernandez J. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Trans. Parallel and Distributed Systems*, 2005, 16(11): 1066~1077.
[19] Yu J, Azougagh D, Kim J, Maeng S. Impact of exploiting load imbalance on coscheduling in workstation clusters. In *Proc. the Int. Conference on Parallel Processing* (*ICPP*), Norway, 2005, pp.595~602.
[20] Du X, Zhang X. Coordinating parallel processes on networks of workstations. *J. Parallel and Distributed Computing*, 1997, 46(2): 125~135.
[21] Giné F, Solsona F, Hernández P, Luque E. Dealing with memory constraints in a non-dedicated linux cluster. *International Journal of High Performance Computing Applications*, 2003, 17(1): 39~48.
[22] Nikolopoulos D, Polychronopoulos C. Adaptive scheduling under memory constraints on a non-dedicated computational farms. *Future Generation Computer Systems*, 2003, 19(4): 505~519.
[23] Ryu K, Pachapurkar N, Fong L. Adaptive memory paging for efficient gang scheduling of parallel applications. In *Proc. the Int. Parallel and Distributed Processing Symposium* (*IPDPS*), Mexico, 2004, pp.30~40.
[24] Giné F, Solsona F, Hernández P, Luque E. Cooperating CoScheduling in a non-dedicated cluster. *Lecture Notes in Computer Science*, 2004, 2790: 212~218.
[25] Zhang X, Yan Y. Modeling and characterizing parallel computing performance on heterogeneous NOW. In *Proc. the Seventh IEEE Symposium on Parallel and Distributed Processing*, USA, 1995, pp.25~34.
[26] Mutka M, Livny M. The available capacity of a privately owned workstation environment. *J. Performance Evaluation*, 1991, 12(4): 269~284.
[27] Nielsen J. Designing Web Usability: The Practice of Simplicity. New Riders Publishing, 2000.
[28] Batat A, Feitelson D. Gang scheduling with memory considerations. In *Proc. the 14th Int. Parallel Distributed Processing Symposium (IPDPS)*, Mexico, 2000, pp.109~114.

Francesc Giné received the B.S. degree in telecommunication engineering from the Universitat Politecnica de Catalunya (UPC), Spain, in 1993 and the M.S. and Ph.D degrees in computer science from the Universitat Autonoma de Barcelona (UAB), Spain, in 1999 and 2004, respectively. He is currently an associate professor of computer arquitecture at University of Lleida (UdL), Spain. His research interests include cluster, multicluster and peer-to-peer computing and scheduling-mapping for parallel processing.

Francesc Solsona received the B.S., M.S. and Ph.D. degrees in computer science from the Universitat Autonoma de Barcelona, Spain, in 1991, 1994 and 2002 respectively. Currently, he is an associate professor and chairman of the Department of Computer Science at the University of Lleida (UdL), Spain. His research interests include distributed processing, cluster, multicluster and peer-to-peer computing, and administration and monitoring tools for distributed systems.

Mauricio Hanzich received his B.S. degree from the Universidad Nacional of Comahue, Argentina, and his M.S. and Ph.D. degrees in computer science from the Universitat Autonoma of Barcelona (UAB), Spain, in 2002, 2004 and 2006 respectively. He is currently an assistant professor at the UAB. His research interests include cluster scheduling and parallel computing.

Porfidio Hernández received the B.S., M.S. and Ph.D. degrees in computer science from the Universitat Autonoma de Barcelona (UAB), Spain, in 1984, 1986 and 1991, respectively. He is currently an associate professor of operating systems at UAB. His research interests include operating systems, cluster computing and multimedia systems.

Emilio Luque is currently a professor of computer science and chairman of the Computer Architecture and Operating System (CAOS) Department at University Autonoma of Barcelona (UAB), Spain. He received the Licenciate (Master) and Ph.D. degrees, both in physics, from the University Complutense of Madrid (UCM) in 1968 and 1973 respectively. He has been involved, from 1994 and heading his group, in several research projects (SEPP, HPCTI, FOREMMS, APART, INFLAME, SPREAD, CrossGrid, $\cdots$) founded by the European Union (EU). He has been invited lecturer/researcher in Universities of USA, Argentina, Brazil, Poland, Ireland, Cuba, Italy, Germany and P.R. China, and keynote speakers in several conferences. He has published more than 35 papers in technical journals and more than 100 papers at international conferences and his current/major research areas are: computer architecture, interconnection networks, task scheduling in parallel systems, parallel and distributed simulation, environment and programming tools for automatic performance tuning in parallel systems, cluster and Grid computing, computational science oriented to environmental applications (forest fire simulation, forest monitoring) and distributed video on demand (VoD) systems.