A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU MANAGER

Xavier Martorell[†], Julita Corbalán[†], Dimitrios S. Nikolopoulos[‡], Nacho Navarro[†],
Eleftherios D. Polychronopoulos[‡], Theodore S. Papatheodorou[‡], Jesús Labarta[†]

†European Center for Parallelism of Barcelona
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya, Spain

‡High Performance Information Systems Laboratory
Department of Computer Engineering And Informatics
University of Patras, Greece

e-mail:{xavim,juli,nacho,jesus}@ac.upc.es, {dsn,edp,tsp}@hpclab.ceid.upatras.gr
web: http://www.ac.upc.es/NANOS

Abstract

Scheduling parallel applications on shared–memory multiprocessors is a difficult task that requires a lot of tuning from application programmers, as well as operating system developers and system managers.

In this paper, we present the characteristics related to kernel–level scheduling of the NANOS environment and the results we are achieving. The NANOS environment is designed and tuned specifically to achieve high performance in current shared–memory multiprocessors.

Taking advantage of the wide and efficient dialog established between applications and the NANOS environment, we are designing powerful scheduling policies. The information exchanged ranges from simply communicating the number of requested processors to providing information of the current speedup achieved by the applications. We have devised several scheduling policies that use this interface, such as Equipartition, Variable Time Quantum DSS and Dynamic Performance Analysis.

The results we have obtained with these policies indicate that there is a lot of work to do in the search for a "good" scheduling policy, which can include characteristics like sustainable execution times, fairness and throughput. For instance, we show through several experiments that benefits in execution time range from 15% to 100%, depending on the policy used and the characteristics of the workload.

1 Introduction

Current multiprocessor machines are used by multiple users to execute several parallel applications at a time, which may share the available resources. Processor scheduling strategies on contemporary shared–memory multiprocessors range between the two extremes of time–sharing and space–sharing. Under time–sharing, applications share processors in time slices. The partitioning is highly dynamic and processors are allowed to freely flow among applications. On the other hand, under space–sharing, partitions of the machine are established at the starting of the execution and maintained for a long–term period.

Time–sharing usually causes lots of synchronization inefficiencies to applications due to the uncontrolled movements of processors. On the other side, space–sharing looses processor power when applications do not use the same number of processors along their execution (e.g. an application that has significant sequential portions). As a result, the scheduling of parallel applications should take the good characteristics of both environments to achieve good performance from applications and processors.

Current machines, from low–end workstations to big mainframes, provide execution environments that fit between the previous two extremes. Small shared–memory multiprocessor machines are available from a great number of computer builders: Silicon Graphics, SUN Microsystems, Compaq/DEC, Hewlett Packard, Intel, Sequent, Data General, etc. Some of them also build big mainframes. Examples of current shared–memory multiprocessor mainframes are the Origin2000 [17] from Silicon Graphics, the SUN Enterprise 10000 [6], the Digital AlphaServer [11], etc. Small–scale SMP systems rely usually on variants of time–sharing to support multidisciplinary workloads, while large–scale servers provide both time–sharing and space–sharing capabilities, usually realized through batch queueing systems.

Experience on real systems shows that with contem-

porary kernel schedulers, parallel applications suffer from performance degradation when executed in an open multiprogrammed environment. As a consequence, intervention from the system administrator is usually required, in order to guarantee a minimum quality of service with respect to the resources allocated to each parallel application (CPU time, memory etc.). Although the use of sophisticated queuing systems and system administration policies may improve the execution conditions for parallel applications, the use of hard limits for the execution of parallel jobs with queuing systems may jeopardize global system performance in terms of utilization and fairness.

Even with convenient queueing systems and system administrator's policies, application and system performance may still suffer because users are only able to provide very coarse descriptions of the resource requirements of their jobs (number of processors, CPU time, etc.). Fine–grain events that happen at execution time (spawning parallelism, sequential code, synchronizations, etc.), which are very important for performance, can only be handled at the level of the runtime system, through an efficient communication interface with the operating system.

In this paper, we present the work related to the previous issues that we are doing in the context of the NANOS Esprit Project. NANOS tackles medium to short–term scheduling and pursues global utilization of the system at any time. The NANOS execution environment consists of three main levels of operation, namely application, user–level execution environment and operating system. In the paper, we will focus at the operating system level to provide high performance to parallel applications.

Kernel–level scheduling solves the problem of having a limited number of physical resources where to execute the applications. Each application maps user–level threads (nano–threads in our model) to the virtual processors offered by the operating system. The operating system maps the virtual processors to physical processors, allowing that all applications execute in a shared environment.

Usually, each application assumes that the operating system assigns a physical processor to each one of its virtual processors. This is not always possible because the demand for virtual processors in the system can exceed the number of physical processors. The total current demand for virtual processors is known as the load of the system.

The role of the operating system dealing with processor scheduling becomes important when the load of the machine is high, when physical processors must be shared by a larger number of virtual processors. This work concentrates in providing new techniques and mechanisms for supporting well–known and new scheduling policies.

Evaluating scheduling policies at kernel–level on real systems usually requires kernel modifications and root privileges, which is very restrictive and limits the ability of the kernel developers to experiment extensively and tune their policies. We have developed a framework for developing and analyzing scheduling policies entirely at user–level, based on a user–level process, the CPU MANAGER, which interacts with the applications running under its control.

This framework not only serves as a tool for comparing scheduling policies, but also as a tool for improving throughput on an actual production system, since the performance of some of the scheduling policies developed in the CPU MANAGER is comparable or better to the performance of native kernel schedulers such as that of IRIX.

The paper is organized as follows: Section 2 presents an overview of the NANOS execution environment, and Section 3 is centered on the kernel–level scheduling issues and compares NANOS with related work. Section 4 presents the characteristics of some of the policies implemented inside the CPU MANAGER. Section 5 sketches the implementation of the CPU MANAGER and Section 6 presents the evaluation of the scheduling policies to demonstrate the usefulness of the tool. Finally, Section 7 presents the conclusions of this paper and the work we have planned for the future.

2 Execution Environment

In the NANOS environment, OpenMP [26] applications compete for the resources of the parallel machine. Applications are parallelized through the NANOSCOMPILER [4, 2, 3] and the parallel code is executed on top of the NTHLIB [21, 20] threads library.

2.1 The nano-threads library (NTHLIB)

The Nano-threads Library, NTHLIB is a user–level threads package specially designed for supporting parallel applications. The role of NTHLIB is two–fold. On one hand, NTHLIB provides the user–level execution environment in which applications execute. On the other hand, NTHLIB cooperates with the operating system level. NTHLIB and the operating system cooperate by interchanging significant fine–grain information on accurate machine state and resource utilization, throughout the execution of the parallel application.

When the load of the system is high, each application should run as if executing in a smaller (dedicated) machine. In this case, resource sharing is unavoidable and usually prevents achieving a performance comparable to the individual applications execution, due to conflicts in processors, memory, etc.

Supplying accurate information to the operating system about resource needs is a key aspect for getting a good compromise between time– and space–sharing. When the application shrinks its parallelism, processors could become

quickly available for other applications running in the system, reducing idle time. On the other hand, when an application needs more processors, it is guaranteed that the request will be taken into account in a short enough amount of time. Applications generated by the NANOSCOMPILER are malleable [14] and adapt their structure of parallelism to the available resources.

2.2 Application adaptability to the available resources

The execution of a nano–threaded application is able to adapt to changes in the number of processors assigned to it. The adaptation is dynamic, at run–time, and includes three important aspects: first, the amount of parallelism that the application generates at any time is limited someway by both the number of processors assigned to the application and the current amount of work already pending to be executed. Second, the application is able to request and release processors at any time. And third, the application should be able to adapt to processor preemptions and allocations resulting from the operating-system allocation decisions.

With respect to the first aspect, the nano-thread starting the execution of a parallel region takes the decision about how many processors to use for spawning the parallelism. The operating system has to provide some interface to allow the application to check which is the number of processors available for spawning parallelism. By checking the number just before spawning parallelism, the application ensures that it is going to use the processors currently allocated to it.

The second aspect, enabling the request for processors, requires from the operating system interface to set the number of processors each application wants to run on. The operating system should guarantee that the number of requested processors from each application is considered as soon as it distributes processors among applications.

The third aspect, applications being able to adapt to processor preemptions, requires also some help from the operating system. The operating system moves processors from one application to another following some scheduling policy (e.g., time–sharing). The requirement from the application point of view is that preemptions must not occur. As this is usually not possible, the run-time execution environment may help to provide such a feeling, by recovering preemptions. A good solution from the operating system point of view is, on one hand, to provide some mechanism to reduce preemptions at a minimum. And on the other hand, to provide a complete interface for preemption recovery.

2.3 Operating system scheduling policies

Kernel–level scheduling consists of a set of policies to distribute processors to applications. Several kernel–level scheduling policies are already developed in order to achieve good performance results in the NANOS environment.

At any time, there is a current active scheduling policy, applied to all applications running in the system. The active policy can be dynamically changed without incurring any overhead to the running applications. Applications notice only the performance differences obtained from the processor allocation decisions taken by the policy newly established. Different application workloads can benefit from different policies [10, 18].

The active scheduling policy is in charge of looking at the requirements of all running applications and decide which resources to allocate to each one. Each parallel application is considered as a whole. This is the way space-sharing is established in the NANOS environment. As long as the policy decides to allocate a number of processors to each application, a portion of the machine is effectively given to that application and the application decides what to do with the processors. The mechanism in charge of determining the exact processors to be assigned to each application ensures that the processors assigned to the application are going to be the ones that more recently have been running on it, if any, thus enforcing data locality. Specific architectural characteristics, such as a NUMA memory subsystem can also be taken into account at that point.

The benefit of looking at applications as a whole is that in short term scheduling decisions, processors know where to look first for work (the application where they are assigned to). In case the application has no work to perform, its cooperation with the operating system makes it to release some processors, which will search for work in other applications. The scheduling policies implemented and evaluated in this work are presented in Section 4.

3 Kernel–Level Scheduling in NANOS

In this section, we present the main characteristics of the kernel–level scheduling in the NANOS environment, and compare them with the related work.

3.1 Sharing information with the upper levels

Each application executing on the NANOS parallel execution environment shares information with the operating system. The information dynamically flows from the application to the operating system and vice–versa.

The information includes, but is not limited to, the number of processors on which the application wants to run at

any moment and the number of processors currently allocated by the operating system to the application. From the number of requested processors, the operating system decides, in a first step, how many processors to allocate to each application. Processors are moved, in a second step, from one application to another. Each moved processor leaves a virtual processor preempted in the source application. It is possible that between the two steps, some time passes to allow the application to voluntarily release the processors to be moved. This functionality is designed to avoid as much as possible the preemption of running processes by the operating system.

Along with the number of requested and allocated processors, information about each one of the virtual processors can be checked by the user–level execution environment during synchronizations, and help the application progress when the operating system decides to reallocate processors to another application.

The amount and quality of the information shared between an application and the operating system is a significant difference with other related work. Process Control [38] already proposed to share a counter of running processes, but the amount of parallelism was not set by the application, but deduced by the operating system (knowing the number of processes created by the application).

Process Control, Scheduler Activations [1] and First–Class Threads [19] use signals or upcalls to inform the user–level about preemptions. The best mechanism for this is shared memory, which is asynchronous and does not disturb further the execution of the application. The application does not need to know about preemptions till it reaches a synchronization point, where it can check the shared memory.

Our approach is similar to the kernel–level NanoThreads [9], which provides a per-application shared memory area containing register save areas (RSA's) to save/restore the user–level threads state at blocking and preemption points.

The SGI–MP LIBRARY shipped with IRIX 6.4/6.5 incorporates a user–level mechanism [34] for detecting situations in which the load of the system is high and the application performs bad and tries to correct this situation by reducing the number of active processes. This solution has the problem that the view of the status of the system obtained by each application can be different and nothing ensures that the response given by the individual applications can help to solve the global problem of the system. A similar mechanism was also proposed in [32].

3.2   Synchronization and processor preemptions

Each time an application needs to do some operation which depends on the number of running processors, it uses the number of processors allocated provided by the NANOS kernel interface [29]. This ensures that, at least during a short amount of time, such processors are available (in average, during half a scheduling period or quantum). This means that most of the times, threads are not going to lose a synchronization point, so they are not going to delay the whole application execution.

In these situations, the behavior of the application depends, during a certain amount of time, on the number of processors allocated. Typically, this happens when the application spawns parallelism, checking the number of processors allocated to know how many processors are going to participate in the parallelism. From that point to the next synchronization point, in a barrier, or while joining the parallelism, the processors should remain assigned to the application, avoiding that a delay in the synchronization slows down the execution of the application. If the operating system decides to reallocate some processors during the execution of the parallelism, some of the virtual processors will be preempted. This can occur, and the NANOS user–level execution environment will be always informed, thus detecting the preemptions when reaching the next synchronization point. No time will be lost waiting for a synchronization with a preempted processor.

Also, when a preemption is detected, any processor of the application (usually the one that detects the preemption) can be directly transferred to execute the preempted work, recovering the application progress.

3.3   The application as the scheduling target

The NANOS operating system environment distributes processors among the running applications, having into account the applications as a whole and their exact requests. Looking at the requests of all the running applications, along with their priorities, the operating system can figure out which is the load of the machine, which applications have more priority to be executed and it can distribute processors accordingly.

To minimize movements of processors between applications, a processor allocated to an application searches for work in that application first. In case there is no ready virtual processor to run in its application, the processor is allowed to automatically assign itself to another application and get work from it. Usually, the scheduling policy applied at each quantum prepares a list of applications which have been given less processors than requested. Those applications are the candidates to receive the processors that become free due to some application termination.

The scheduling policies implemented on the NANOS environment range from the well–known equipartition, batch or round–robin policies to other kind of policies that can make more use of the information provided by the ap-

plications. They are described in section 4.

Sometimes, operating systems offer Gang Scheduling [27], combined with two–level synchronization methods at user–level [15, 39, 14]. This solution is not general enough to perform well in all situations. For instance, in current versions of IRIX, it is not recommended to run applications using Gang Scheduling. We have observed that even with Gang Scheduling is very hard that processes remain assigned as a whole to an application. A process doing I/O or taking a page fault may motivate a context switch in its physical processor, which can take off all the processors of the application, causing cascades of movements across the system and degrading performance.

Another drawback of gang scheduling is that it often compromises the memory performance of parallel applications. When multiple programs are gang-scheduled on the same processors, their working sets interfere in the caches, thus incurring the cost of cache reloads. Parallel applications running on multiprogrammed shared-memory multiprocessors and particularly on NUMA systems, are extremely sensitive to this form of interferences [15].

3.4 Processor affinity

Processor affinity is an important issue to consider in kernel–level scheduling because of the different access latencies to cached, local and remote memory locations. Cache memory is always of importance, both in SMP and CC-NUMA machines [37, 24, 36]. When a processor runs inside an application, the processor caches are filled with data which is usually accessed several times. Moving processors from one application to another causes a total or partial cache corruption. Processor affinity is useful to take advantage of the data remaining in the cache when the processor is allocated again to the same application.

In CC-NUMA machines, local and remote memory accesses are also important to consider due to the different access times, which can range from 0.3 to 2 microseconds. Usually, in NUMA machines, the operating system places data near the processor that has accessed it for the very first time. This means that other application threads accessing the same data can benefit of being scheduled on the same processor. The benefits in this case will be greater, if the data already is in the cache of the processor. Otherwise, at least the cost accessing local memory will be lower than accessing remote memory.

Scheduling at the operating system level in the NANOS execution environment uses two levels of affinity. In a first step, a processor is assigned to an application on which it has run before. In a second step, inside an application, a processor is assigned to a virtual processor on which it run before, if any.

4 Kernel–Level Scheduling Policies

In this paper, we are using the environment presented to test several policies and compare them with existing operating system scheduling policies. The policies proposed have been already explained in more detail in [29, 30, 7]. We summarize here their characteristics.

4.1 Equipartition (Equip)

Equipartition [23, 14, 22] is the simplest policy included in the NANOS environment. This policy divides the number of physical processors by the number of running applications, and assigns the resulting number of processors to each application. When an application requests less processors than the result of the division, the processors exceeding that number are not assigned to any application. In case there are more applications than processors, only the first P applications are executed, assuming there are P processors.

Some processors can remain unallocated when using this policy, depending on the amount of applications and the requests of each application. Equipartition is implemented to obtain a reference, given by such a simple policy, to which the behavior of the other policies could be compared.

4.2 Processor Clustering (Cluster)

The Processor Clustering policy allocates processors in clusters of four processors. The reasons for selecting this number are to achieve good locality in systems with physically distributed shared–memory. Allocating four processors at a time allows the CPU Manager to better select four neightbour processors, which will improve performance due to smaller memory latencies, assuming first–touch placement. In addition, our experience indicates that applications usually request a number of processors which is a multiple of four. Also, applications usually get better performance when running on an even number of processors.

In a first step, this policy allocates a cluster of four processors to all running applications. If some applications are not receiving processors because there is a large number of applications in the system, they are candidates to receive any processor released from the applications selected to run during the next quantum. In case a number of processors remain unallocated, this policy starts a second step, allocating again in clusters of four. And so on, till all the processors have been allocated or all the requests have been satisfied. When less than four processors remain to be allocated, some applications can receive two or even one processor to maintain working all processors available in the machine.

4.3 Dynamic Space Sharing (DSS)

Dynamic Space Sharing (DSS) [30, 29] is a two–level scheduling policy. The high level space–shares the machine resources among the applications running in the system. The low level improves the memory performance of each program by enforcing the affinity of kernel threads to specific physical processors.

DSS distributes processors as evenly as possible among applications taking into account the full workload of the system and the number of processors requested by each application. Each application receives a number of processors which is proportional to its request and inversely proportional to the total workload of the system, expressed as the sum of processor requests of all jobs in the system.

Time–sharing is then applied to DSS to obtain several derived policies, which are explained in the following subsections.

4.3.1 Sliding Window DSS (SW–DSS)

Sliding Window DSS partitions all applications on execution in groups of applications called *gangs*. A gang is defined as a group of applications requesting a number of processors which is not more than kP, where P is the number of processors and k is a tunable parameter of the policy.

Each gang is submitted for execution during a constant time quantum (usually of 100 ms.) After the expiration of the quantum, the next neighbouring gang of applications is scheduled and so on. The view of these gangs passing through execution is as a moving window sliding across the workload.

Each gang is usually evolving during the execution of the workload. Each time a new application starts, an application finishes, or an application changes its requests, the gang accommodates more or less applications to fit the condition of having less than kP processors requested.

4.3.2 Step Sliding Window DSS (SSW–DSS)

The Step Sliding Window DSS policy is designed to get the benefits of DSS, the time–sharing provided by SW–DSS and improve cache performance. In SSW–DSS the scheduling step is not a discrete new window, like in SW-DSS. Instead, it is the same window as in the previous quantum leaving out the first process and filling the gang with none, one or more processes to satisfy that the total requests does not exceed kP.

With the support of DSS, this policy ensures that in each scheduling step, all but the first applications will be executed again, and most of the processors will reuse the footprints in their cache memories.

4.3.3 Variable Time Quantum DSS (VTQ–DSS)

The Variable Time Quantum DSS policy searches for equalizing the CPU time received by all the applications in the workload. This means that applications executed on more processors are going to receive such processors during a smaller amount of time than applications requesting less processors. The time quantum is different for each application.

In this policy, applications in the workload are again organized in gangs, which are executed like in the previous policies with the restriction of requesting less than kP processors. In VTQ, the applications at the end of the ready queue that do not compose a gang are enqueued in a repository queue.

When an application starts, it is assigned an initial quantum of 100ms. After execution, the quantum is updated to try to equalize the amount of CPU time received by all the applications in the gang. As a result, when the time quantum of an application expires, the application is stopped and another application from the repository queue is taken to use the processors up to the end of the gang. The same mechanism is used when an application finishes, leaving a hole in its gang.

4.4 Dynamic Speedup Analysis

From the production point of view, it is interesting to ensure that applications getting a higher speedup should receive benefits compared to applications with poor performance. We propose the *Performance–Driven Processor Allocation* (PDPA) policy, which takes into account the speedup achieved at run–time in parallel regions. This feature avoids the assignment of physical processors to applications that are not able to take advantage of them. These processors can then be reassigned to other applications.

Traditionally, the speedup obtained by a parallel application has been computed doing several executions with 1 to P processors and the results have been provided to the scheduler.

We propose to dynamically compute the speedup and provide it to the scheduler in the same way the application informs about the number of requested processors. The speedup of an application is computed through the SelfAnalyzer library [7] which informs the CPU Manager at run–time. A similar approach is also used in [25] to compute the efficiency achieved by the applications.

The associated scheduling policy (*PDPA*) implements the state diagram presented in Figure 1. The state diagram is a search procedure, applied to each application and parameterized through three arguments: the number of processors assigned when the application starts (BASE), the increment / decrement on the number of processors applied when the

application is performing well / bad (STEP), and the desired minimum efficiency level based on the performance got by the application with the current number of processors (MIN_EFFICIENCY). The definition of efficiency is taken from [12].
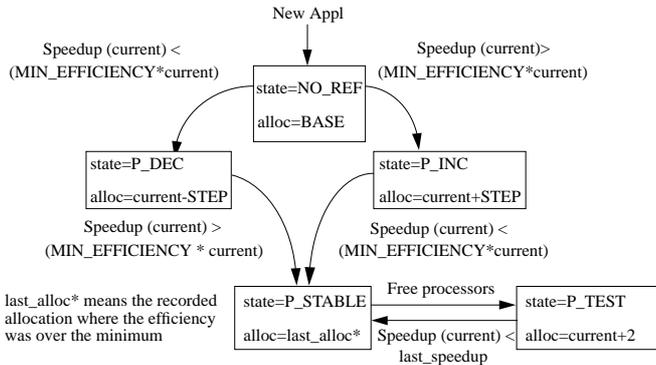
Figure 1. State diagram used in the *PDPA* policy

An application can be found in five different states: speedup unknown (NO_REF), increasing the number of processors (P_INC), decreasing the number of processors (P_DEC), stationary (P_STABLE) and trying to improve the speedup due to free processors (P_TEST). When the application informs about the speedup achieved with BASE processors, the *PDPA* policy decides to increment or decrement the number of processors allocated to the application.

The four transitions showed in the figure are followed when the associated condition becomes true. An application performing badly in the NO_REF state, goes to the P_DEC state. Otherwise, if it performs well, it goes to the P_INC state. In both states, when the application reaches a sustained speedup, it goes to the P_STABLE state, where it remains for some time. In this state, the number of assigned processors remains constant.

At some point, for instance when there are free processors, the scheduler can decide to provide more processors to an application, testing several times whether the application is able to improve its efficiency. In the current implementation, this test is done three times.

By tuning the BASE, STEP, and MIN_EFFICIENCY parameters, it is possible to establish the aggressiveness of the policy. Sometimes it is also useful to consider two different levels of efficiency, one for increasing the number of processors and another one for decreasing it.

It is interesting to note that this policy can influence the long–term scheduler by changing the degree of multiprogramming at a given point. The policy can detect whether there are several idle processors because the actual applications can not take advantage of them. In this situation, the degree of multiprogramming can be increased to accommodate more applications. When the efficiency of applications increases, the policy can indicate to the long–term scheduler to reduce the degree of multiprogramming.

5 Implementation

This sections describes the current implementation of the CPU MANAGER [8].

5.1 The user–level CPU MANAGER

The user–level CPU MANAGER implements the interface between the applications and the kernel, establishing the cooperation between the user–level execution environment provided by NTHLIB and the kernel level.

The CPU MANAGER is implemented by a server process and a portion of NTHLIB. The server establishes a shared memory area between itself and the applications. This area is used to implement the interface between the kernel and the applications, efficiently and with minimal overhead. The shared–memory area contains one slot for each application under the control of the CPU MANAGER. At initialization time, the CPU MANAGER starts the scheduler thread. Its mission is to apply any of the user–selectable scheduling policies to the application workload, forcing the application threads selected to run on a specific physical processor. It also communicates all its decisions to the applications.

5.2 Implementation of the kernel–level scheduling framework

As it has been stated in Section 3, the NANOS kernel–level scheduling is application–oriented and takes applications as the scheduling target. As it is shown in figure 2, the knowledge of the applications is the central point in the design of the CPU MANAGER. All data structures and algorithms are oriented to manage applications as a whole through the interface implemented in shared memory.

The kernel–level scheduling framework consists of two main data structures: the application slots shared with the applications and the work–list.

The left portion of the figure represents several applications attached to the shared–memory area. This area is divided in application slots, one slot for each application. The shared memory is readable and writable both from the applications and the CPU MANAGER. The information available in this area includes, inside each application slot, the number of processors requested, the number of processors assigned by the CPU MANAGER, whether the CPU MANAGER has preempted any thread of the application and
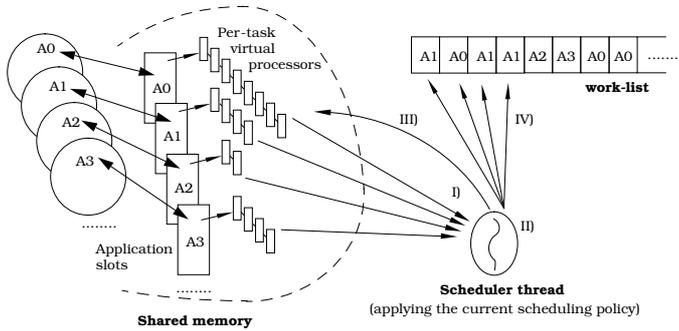
Figure 2. CPU MANAGER **environment**

other information useful for the scheduling policy, like the speedup obtained by the parallel application.

The right part of the figure represents the scheduler thread and the work–list structure. The scheduler thread is in charge of distributing processors every time quantum (100 ms.) The work–list is a list containing a reference to the applications which are requesting more processors than those allocated to them. It is used to maintain working during the current quantum all processors released by other applications running in the system due to some reasons (e.g., lack of parallel work inside the application). Processors, when released, go immediately to visit the work–list to find a new application where they can go to execute a ready virtual processor.

In this framework, physical processors are first assigned to an application, and then they choose a virtual processor belonging to that application for execution.

Figure 2 also presents the algorithm to apply the current scheduling policy. The scheduler thread starts executing the algorithm by collecting all the information about processor requests supplied by the applications (step I, in the figure). Then, the current scheduling policy decides how many processors each application is going to receive for the next time quantum (step II).

Next, the allocation results are communicated to the applications (step III), including the number of processors allocated and which virtual processors have been preempted, if any. Finally, the work list structure is used to indicate applications that want to receive more processors when any becomes available (step IV).

Applying the previous algorithm, some applications are going to loose processors. Physical processors that have to move, get from the work–list a new application and assign to it. Figure 3 details the algorithm by which a processor is assigned to an application *allocate_cpu*. The *application* parameter contains a descriptor for each virtual processor from the application point of view. The algorithm searches first for an unallocated virtual processor in which to assign the physical processor. If one is found, the virtual processor

(thread) in this descriptor is bound to the physical processor, and unblocked (woken up), while updating the information shared with the application. In case there is no virtual processor available, this means that another free processor has filled the last one available, so the current processor remains unallocated and continues searching for work in the work–list.

In order to avoid that the IRIX operating system can disrupt the processor assignments, the CPU MANAGER uses system calls to block / unblock virtual processors and bind / unbind them to run on physical processors. Finally, the CPU MANAGER internal structure representing physical processors is updated accordingly.

This algorithm can be executed either by the CPU MANAGER to initially assign free processors to a running application or by NTHLIB, as part of the scheduler to transfer an application processor directly to another application.

```
int allocate_cpu (int cpu,
                  struct appl_info * application)
{
  int vp;

  /* Search for a virtual processor (vp) giving
     priority to (in this order) the vp used last
     time, a stolen vp, and any unallocated vp */
  vp = search_for_unallocated_vp (application);

  if (vp<application->n_cpus_requested) {
    /* Vp is valid, decrement number of preempted
       processors, when needed */
    if (application->cpu_info[vp].stat == CPU_STOLEN)
      --application->n_cpus_preempted;

    /* Assign thread/sproc to cpu through the OS */
    sys_bind_thread (
            application->cpu_info[vp].kthread, cpu);

    /* Assign cpu to the virtual processor vp and
       mark the vp running */
    application->cpu_info[vp].sys_sim_id = cpu;
    application->cpu_info[vp].stat = CPU_RUNNING;

    /* Unblock the associated thread/sproc */
    sys_unblock_thread (
                application->cpu_info[vp].kthread);

    /* Update current number of processors */
    ++application->n_cpus_current;

    /* Update the processor structure */
    phys_cpus[cpu].status = CPU_ALLOCATED;
    phys_cpus[cpu].curr_appl = application->appl_no;

    /* Return succesfully */
    return 0;
  }
  /* Return indicating processor unallocated */
  return -1;
}
```

Figure 3. Algorithm for allocating processors to applications

Applications detect at user–level, through the shared–memory area, that some processors have been stolen and

recover the work themselves using other processors already allocated. To implement this feature, the user–level idle loop of NTHLIB is slightly more complex than that of the other thread packages. Figure 4 shows how the idle code is responsible of freeing processors that are not going to be used by the application (through the primitive *cpus_release_self*).

After that, the idle code also checks whether the CPU MANAGER has preempted a virtual processor, and recovers the work stopped by transferring to it the current physical processor (through *cpus_processor_handoff*). This is implemented through the bind operating system primitive.

Finally, the idle code searches for work in the ready queues and executes it, if found.

```
struct nth_desc * nth_getwork (int vp)
{
    struct nth_desc * next;

    /* Dequeues work from the local ready queue */
    next = nth_lrq_dequeue (vp);

    if (next==NULL)
        /* The local queue is empty, tries to dequeue
           work from the global queue */
        next = nth_rq_dequeue ();

    return next;
}

void nth_idle (int vpid)
{
    struct nth_desc * next;
    work_t work;

    while (1) {
        if (cpus_asked_for ()) {
            /* The processor detects that the CPU
               Manager reclaims some processors
               and stops itself */
            cpus_release_self ();
            /* The processor returns here in case it is
               reassigned later */
        }
        else if (cpus_preempted_work () > 0) {
            /* Recovering preempted work (go & back) */
            work = cpus_get_preempted_work ();
            if (work!=NO_WORK) {
                cpus_processor_handoff (work);
                /* The processor returns here in case it is
                   reassigned later */
            }
        }
        /* Gets work from the ready queues */
        next = nth_getwork (vpid);
        /* Executes the work in the processor */
        if (next!=NULL) schedule (next);
    }
}
```

Figure 4. Idle loop in the nano–threads environment

The current implementation of the NANOS CPU MANAGER is equivalent in functionality with a pure kernel implementation. In this implementation, both the CPU MANAGER and NTHLIB participate in implementing the scheduling mechanisms. For this reason, the information shared among the applications and the CPU MANAGER is read/write for all the participating processors. This allows NTHLIB to transfer a physical processor to another application, when the CPU MANAGER requests to do so.

We have also implemented a different version of the CPU MANAGER, the MP CPU MANAGER, in which the design of the shared–memory areas enforces protection by using a different area for each application. This solution does not allow the transfer of a processor from an application directly to another one, and the CPU MANAGER itself has to intervene in each movement. The MP CPU MANAGER deals with IRIX application binaries running on top of the SGI–MP LIBRARY without neither recompiling nor relinking the source code.

6 Evaluation

This section presents the evaluation of the scheduling framework developed for the NANOS environment. The environment is also evaluated in [29, 30].

6.1 Experimentation platform

The design and implementation of the NANOS CPU MANAGER has been carried on a Silicon Graphics Origin2000 machine [17, 33], running the IRIX 6.5.5 operating system. The machine is located at the European Center for Parallelism of Barcelona (CEPBA [13]). It has sixty-four R10000 MIPS processors [16] running at 250 Mhz (chip revision 3.4). Each processor has separated 32 Kb. primary instruction and data caches and a common 4 Mb. secondary cache.

All benchmarks and applications used for the evaluation presented in this section have been compiled to run on both the native SGI–MP and NANOS environments. Input source programs were previously annotated with standard OpenMP directives. NANOS binaries have been obtained by preprocessing the applications through the NANOSCOMPILER [4].

The benchmarks are taken either from the NAS Benchmarks [5] or the SPEC95FP Benchmarks [35]. We have modified two of them to obtain two evolving benchmarks: EPD and ltomcatv. The EPD benchmark derives from EP (Embarrassingly Parallel) and consists of an external loop containing EP. At every iteration, the EPD benchmark requests a different number of processors, following the series: 2, 4, ... MAX-2, MAX, MAX, MAX-2, ...2, in order to give a variable (diamond shape) parallelism. The ltomcatv is a variation of the tomcatv from the SPEC95FP Benchmarks. In ltomcatv the main loop of the application is performed several times, generating a sequence of sequential and parallel regions. It requests one processor for the first

I/O portion of the external loop and then it requests MAX processors for the parallel portion.

The NAS Benchmarks scale well up to 32 processors on a dedicated machine. The tomcatv, ltomcatv and swim SPEC95FP benchmarks scale also well, even achieving super–linear speedup with some number of processors. The turb3d scales well up to 16–20 processors. Its speedup decreases with more processors. And finally, the SPEC95FP apsi does not scale at all. The proportion of parallel code is very small and by the Amdhal's Law the speedup that it can achieve is less than two. Even more, when increasing the number of processors, apsi does not suffer an slowdown. It simply does not take advantage of the processors.

Compilation of all benchmarks in both environments has been done using the same command line options to generate machine code with the native MIPSpro F77 compiler: –64 –MIPS4 –R10000 –O3 –LNO:PREFETCH_AHEAD=1.

The following subsections present the results obtained through the execution of several workloads and policies.

6.2   Workload on 32 processors (cpuset32)

This section presents the evaluation of a workload consisting of several NAS applications running on 32 processors. For this workload, the W class of each application is used. The EPD application is class S because class W is too large compared with the other applications. This workload was run inside a *cpuset* partition of 32 processors in our machine of 64 processors, while the other half of the processors in the machine were in production mode executing applications belonging to other users. Table 1 shows the applications participating in the workload, the number of processors requested by each one, and the total and system execution times of each application. Execution times are the arithmetic mean of the execution times obtained by the different instances of each application.

This workload shows the different behaviour between the SGI–MP environment and the NANOS Cluster policy. The largest difference is in CG. This is because CG requests 16 processors. Running in the SGI–MP environment, it tries to use as many processors as possible, but the SGI–MP LIBRARY does not detect that the application performs bad. As a result the time spent in system mode for the CG application raises up to 2.48 seconds per thread. In the Cluster policy, the CG application receives only 4 processors (one cluster), and performs highly better without sharing processors with other applications.

EPD performs better in SGI–MP than using the Cluster policy because it is a dynamic application. In the SGI–MP LIBRARY environment, dynamic applications create and destroy processes at every request. It seems that this mechanism benefits execution time because new processes are created with their dynamic priority high in IRIX. We

Figure 5. Workload execution on 32 processors

have to perform more experiments in this line to confirm this result.

SP and BT benchmarks are clearly better when using the NANOS Cluster Policy. And FT behaves the same, receiving only 4 processors with the Cluster Policy (given by the output trace generated by the CPU MANAGER) and up to 8 in the SGI–MP environment.

Figure 5 shows graphically the execution of 80 seconds of the workload using the PARAVER tool [28], recorded during the real execution. These plots present time in the X axis and applications in the Y axis. The names of the applications are displayed on the left-hand side of the figure, along with the number of processors that they are requesting (enclosed in parenthesis). For each application, an horizontal line is displayed. For each instance of an application, a different color is used to fill the horizontal line. Different colors represent, thus, the execution of the different instances of the corresponding application. Also, a flag is displayed when an application starts. The reader can observe how the NANOS environment (bottom plot) manages better the execution of parallel applications and more instances of each one are executed in the same amount of time.

6.3   Workload on 64 processors using Dynamic Space Sharing

In this subsection, we present a workload running on 64 processors, with the machine in dedicated mode. Table 2 presents the workload, consisting of six class A NAS benchmarks, plus the class W EPD benchmark, and

Table 1. Workload on cpuset 32

Benchmark	CPUS	Mean execution time (in s.)		Mean system time per thread (in s.)	
		SGI–MP LIBRARY	NANOS Cluster	SGI–MP LIBRARY	NANOS Cluster
sp.W	8	34.7	26.7	1.26	0.04
bt.W	12	18.8	7.51	2.01	0.01
epd.S	2-16	1.37	1.92	0.17	0.02
ft.W	8	1.03	1.04	0.10	0.03
cg.W	16	21.1	1.67	2.48	0.01
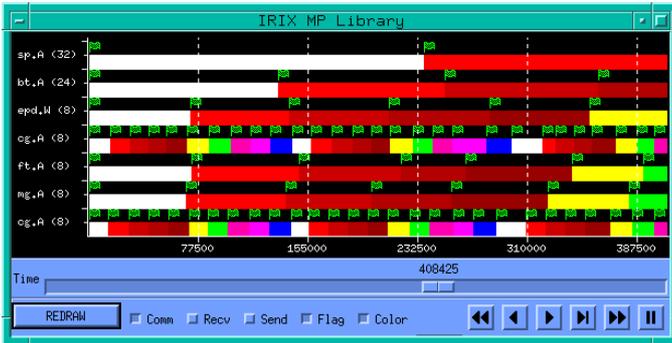
Figure 6. Workload execution on 64 processors (SGI–MP LIBRARY**)**
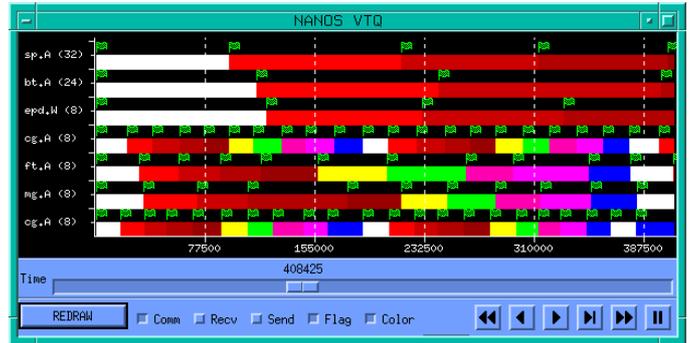
Figure 7. Workload execution on 64 processors (NANOS VTQ POLICY**)**

its results for three different policies (SGI–MP LIBRARY, NANOS SSWDSS and NANOS VTQ).

We want to highlight that the benchmark requesting more processors (SP, 32 processors) suffers a lot of performance problems in the SGI–MP LIBRARY environment due to resource sharing and preemptions. On the other hand, in the NANOS environment, all policies provide better response time for the SP benchmark. The differences in performance shown in the NANOS environment depend on the policy applied during execution.

FT and MG applications obtain also better performance in the NANOS environment, in general. CG and BT perform nearly the same and EPD is better in the SGI–MP environment.

The reasons for the better performance exposed by the NANOS environment are the better coordination between the user and kernel levels, solving synchronization problems earlier and allowing applications to work with a more stable number of processors.

Figures 6 and 7 show graphically the behaviour of the workload on the SGI–MP environment and under the VTQDSS policy. Up to four complete instances of the SP application are executed using the VTQDSS policy, compared with only two in the SGI–MP environment. This is a very good result that can be usually applied to applications with good scalability running in the NANOS environment

and requesting a large number of processors.

6.4 Workloads on 64 processors using Dynamic Speedup Analysis

In order to evaluate the performance of the *PDPA* policy we have prepared three different parallel workloads, trying to highlight different situations to see how the policy adapts the environment to them.

All along the workloads, the applications request 32 processors in a 64–processor machine. We have selected this number for two reasons: first, usually the policy of supercomputing centers is to limit the number of processors that can be used by an application. The second reason is that even when there are no limits established, the users guess that applications will not scale well up to the maximum number of processors and limit themselves to execute in a smaller number of processors.

We have executed the three workloads in dedicated mode and under two scheduling policies: the standard SGI–MP environment and the *PDPA* policy. This policy was setup with the following parameters: the BASE number of processors is set to 4, the STEP is set to 2 and the MIN_EFFICIENCY is set to 0.7. These arguments were determined through experimentation.

In these experiments, the NANOS environment controls

Table 2. Execution times (in s.) in 64 processors

Benchmark	CPUS	SGI–MP LIBRARY	NANOS SSWDSS	NANOS VTQ
sp.A	32	236.6	98.5	101.1
bt.A	24	120.0	113.5	133.1
epd.W	2–8	70.8	128.7	110.2
cg.A	8	14.7	19.1	18.9
ft.A	8	65.2	35.9	37.1
mg.A	8	63.6	35.5	38.2
cg.A	8	14.7	18.6	18.1

also the amount of applications that should be executed in parallel (the multiprogramming level). The CPU MANAGER is in charge of increasing the multiprogramming level when it detects that the current applications are not obtaining a good efficiency from the system. In this case, the CPU MANAGER can reduce the number of processors allocated to the applications and allow more applications to enter the system. Alternatively, when an application finishes, the CPU MANAGER can either test the performance of the remaining applications with more processors, thus decreasing the multiprogramming level, or allow another application to enter the system and maintain the multiprogramming level.

The first workload consists of five ltomcatv benchmarks. The degree of multiprogramming is set to five. Each instance of the ltomcatv is executed three times. The goal of this workload is to analyze what happens when the applications are highly dynamic. Table 3 shows the results.

This workload has the characteristic that the number of requested processors has a high variability since the applications change from sequential to parallel phases several times. The *PDPA* outperforms the SGI–MP policy. Several reasons can be given to this behaviour: first, since the applications enter the P_STABLE phase, their allocation does not change, even when other applications execute the I/O portion and release their processors. Second, in the SGI–MP environment, the MP LIBRARY is not able to track the different phases of the ltomcatv benchmark and this motivates synchronization problems. Although the speedup obtained by this benchmark can be quite good, the stable allocation with less processors avoids resource sharing and this improves overall performance.

The second workload is composed by three instances of the tomcatv and three of the turb3d, both from the SPECFP 95. In that case the multiprogramming level has been set to four, and each application is executed three times. The goal here is to show what happens when all the applications in the workload have good scalability up to 16–20 processors. Results are presented in Table 4. They show that the search procedure to determine the number of processors to achieve good efficiency is not introducing a noticeable overhead, al-

though it is executed in several of the very first iterations of each application. The execution times obtained are competitive compared with the results obtained from the SGI–MP environment.

And finally, the last workload consists of three swim and three apsi, showing what happens when several applications do not scale at all and are sharing the machine with others that scale well. This scenario reproduces one of the situations where the *PDPA* policy can extract more benefits since the speedups achieved by the applications are very different. In this case the multiprogramming level has been set to four, and each application is executed three times. Table 5 shows the results.

As we expected, the *PDPA* policy outperforms the SGI–MP policy. *PDPA* is able to detect that all apsi benchmarks obtain a very poor speedup and re–assigns their processors to swim benchmarks. The overall speedup achieved by the *PDPA* with respect to the SGI–MP environment reaches 4.5.

7 Conclusions and Future Work

In this paper, we have presented an execution environment to efficiently execute parallel applications in shared–memory multiprocessors, the NANOS environment. This paper has focused in the design and implementation of kernel–level scheduling inside the NANOS environment. A user–level scheduler has been presented, the CPU MANAGER, which efficiently cooperates with the applications. This tool has proven to be very powerful and useful to communicate the information the scheduler needs to distribute processors among the applications.

The CPU MANAGER also provides a good environment to implement and evaluate scheduling policies. In this paper, we have described a set of scheduling policies implemented in the CPU MANAGER, trying to cover a wide range of the possibilities that it offers. One of the major benefits of the CPU MANAGER is that it provides a real execution environment, and parallel applications can be executed on an actual system. This characteristic and the total control of the processor allocation provided by the CPU MANAGER,

Table 3. Results of the ltomcatv workload using Speedup Analysis

Policy	Mean execution time (in s.)		Total Speedup	Applications
	Ltomcatv	Total		
SGI–MP LIBRARY	165.9	606	1.0	15
Speedup Analysis	47.0	132	4.59	15

Table 4. Results of the tomcatv/turb3d workload using Speedup Analysis

Policy	Mean execution time (in s.)			Total Speedup	Applications
	Tomcatv	Turb3d	Total		
SGI–MP LIBRARY	33.36	48.69	251	1.0	18
Speedup Analysis	15.63	42.71	148	1.7	18

allows us a more accurate and realistic evaluation of the scheduling policies.

The scheduling policies have been evaluated comparing its performance with the SGI–MP LIBRARY. Results show that the NANOS environment is solid and flexible. Moreover, all the scheduling policies evaluated have outperformed the standard SGI–MP environment. When we compare the results achieved in individual applications, we see that some scheduling policies have outperformed the SGI–MP by a factor of more than two. And, when we compare the total execution times of some workloads, the NANOS environment achieves an speedup of 4 for some experimental workloads.

We are now working on porting this technology to run with native IRIX OpenMP applications. This means that we are developing an MP CPU MANAGER to deal with IRIX application binaries running on top of the SGI–MP LIBRARY without recompiling or relinking the source code. The high flexibility and tuning options offered by the SGI–MP environment allows to do that. The preliminary results indicate that the performance of these applications can also be improved through increasing the cooperation with the operating system scheduler represented by the MP CPU MANAGER.

We are also considering the design and implementation of some alternative kernel–level scheduling policies with memory locality considerations.

Acknowledgements

References

[1] T. Anderson, B. Bershad, E. Lazowska and H. Levy, "Scheduler Activations: Effective Kernel Support for the User–Level Management of Parallelism", Proceedings of the 13th. ACM Symposium on Operating System Principles (SOSP), October 1991.

[2] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzàlez and N. Navarro, "Exploiting Parallelism Through Directives on the Nano-Threads Programming Model", Proceedings of the 10th. Workshop on Language and Compilers for Parallel Computing (LCPC), Minneapolis, USA, August 1997.

[3] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzàlez and N. Navarro, "Exploiting Multiple Levels of Parallelism in Shared–memory Multiprocessors: a Case Study", Dept. d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, Technical Report: UPC-DAC-1998-48, November 1998.

[4] E. Ayguadé, M. Gonzàlez, J. Labarta, X. Martorell, N. Navarro and J. Oliver, "NanosCompiler: A Research Platform for OpenMP Extensions", Dept. d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, Technical Report: UPC-DAC-1999-39, 1999.

[5] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA, December 1995.

[6] A. Charlesworth, "STARFIRE: Extending the SMP Envelope", IEEE Micro, Jan/Feb 1998.

Table 5. Results of the swim/apsi workload using Speedup Analysis

Policy	Mean execution time (in s.)			Total Speedup	Applications
	Swim	Apsi	Total		
SGI–MP LIBRARY	74.46	354	1082	1.0	18
Speedup Analysis	7.28	112	237	4.56	18

[7] J. Corbalán, J. Labarta, "Dynamic Speedup Calculation through Self-Analysis", Dept. d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, Technical Report: UPC-DAC-1999-43, 1999.

[8] J. Corbalán, X. Martorell and J. Labarta, "A Processor Scheduler: The CpuManager", Dept. d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, Technical Report: UPC-DAC-1999-69, 1999.

[9] D. Craig, "An Integrated Kernel– and User–Level Paradigm for Efficient Multiprogramming Support", M.S. thesis, Department of Computer Science, University of Illinois at Urbana–Champaign, 1999.

[10] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, E. Markatos, "Multiprogramming on Multiprocessors", Technical Report 385, University of Rochester, February 1991 (revised May 1991).

[11] Digital Equipment Corporation / Compaq Computer Corporation, "AlphaServer 8x00 Technical Summary", http://www.digital.com/alphaserver/alphasrv8400/8x00_summ.html, 1999.

[12] D.L. Eager, J. Zahorjan and E.D. Lazowska, "Speedup Versus Efficiency in Parallel Systems", IEEE Transactions on Computers, pp. 408-423, Vol. 38, No. 3, March 1989.

[13] European Center for Parallelism of Barcelona (CEPBA), http://www.cepba.upc.es.

[14] D. Feitelson, "Job Scheduling in Multiprogrammed Parallel Systems", IBM Research Report 19790, Aug. 1997.

[15] A. Gupta, A. Tucker and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications", ACM SIGMETRICS Performance Evaluation Review, Vol. 19(1), pp. 120-132, May 1991.

[16] J. Heinrich, "MIPS R10000 Microprocessor User's Manual", version 2.0, MIPS Technologies, Inc., January 1997.

[17] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server", Proceedings of the 24th. Annual International Symposium on Computer Architecture, pp. 241-251, Denver, Colorado, June 1997.

[18] S. T. Leutenegger, M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", Proceedings of the ACM SIGMETRICS Conference, pp. 226-236, May 22-25, 1990.

[19] B. Marsh, M. Scott, T. LeBlanc and E. Markatos, "First–Class User–Level Threads", Proceedings of the 13th. ACM Symposium on Operating System Principles (SOSP), October 1991.

[20] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, "Nano-Threads Library Design, Implementation and Evaluation", Technical Report, Universitat Politècnica de Catalunya, UPC-DAC-1995-33, Sep. 1995.

[21] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, "A Library Implementation of the Nano–Threads Programming Model", Proceedings of the 2nd. Euro–Par Conference, Lyon, France, August. 1996.

[22] C. McCann, R. Vaswani, J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared–Memory Multiprocessors", ACM Transactions on Computer Systems, 11(2), pp. 146-178, May 1993.

[23] V. K. Naik, S. K. Setia, M. S. Squillante, "Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments", Proceedings of the Supercomputing '93, 1993.

[24] V. K. Naik, M. S. Squillante, "Analysis of Cache Effects and Resource Scheduling in Distributed Parallel Processing Systems", Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM Press, 1995.

[25] T.D. Nguyen, R. Vaswani, J. Zahorjan, "Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling", Proc. of the workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, Springer–Verlag, 1996.

[26] OpenMP Architecture Review Board, "Fortran Language Specification, v 1.0", www.openmp.org/openmp/mp–documents/fspec.ps, October 1997.

[27] J.K. Ousterhout, "Scheduling Techniques for Concurrent Systems", Proceedings of the 3rd. International Conference on Distributed Computing and Systems, pp.22-30, 1982.

[28] V. Pillet, J. Labarta, T. Cortés, S. Girona, "PARAVER: A Tool to Visualize and Analyse Parallel Code", WoTUG-18, pp. 17-31, Manchester, April 1995. Also as Technical Report UPC-CEPBA-95-03.

[29] E.D. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. S. Papatheodorou and N. Navarro, "Kernel–level Scheduling for the Nano-Threads Programming Model", Proceedings of the 12th ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, July 1998.

[30] E.D. Polychronopoulos, D.S. Nikolopoulos, T.S. Papatheodorou, X. Martorell, J. Labarta, N. Navarro, "An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors", Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems (PDCS'99), Fort Lauderdale (Florida - USA), August 18-20, 1999.

[31] E. D. Polychronopoulos, D. Nikolopoulos, T. Papatheodorou, X. Martorell, N. Navarro and J. Labarta, "TSDSS – An Efficient and Effective Kernel–Level Scheduling Methodology for Shared–Memory Multiprocessors", Technical Report HPISL-010399, High Performance Information Systems Laboratory, University of Patras, 1999.

[32] C. Severance, R. Enbody, "Automatic Self–Allocating Threads (ASAT) on an SGI Challenge", Proc. of the International Conference on Parallel Processing (ICPP96), Vol.3, pp. 132-139, August 1996.

[33] Silicon Graphics Computer Systems SGI, "Origin 200 and Origin 2000 Technical Report", 1996.

[34] Silicon Graphics Computer Systems (SGI), "IRIX 6.4/6.5 manual pages: mp(3F) & mp(3C)", IRIX online manuals, also in http://techpubs.sgi.com, 1997-1999.

[35] SPEC Organization, "The Standard Performance Evaluation Corporation", www.spec.org.

[36] M. S. Squillante, R. D. Nelson, "Analysis of Task Migration in Shared–Memory Multiprocessor Scheduling", ACM SIGMETRICS Performance Evaluation Review, vol. 19, no. 1, 1991.

[37] J. Torrellas, A. Tucker, A. Gupta, "Evaluating the Performance of Cache–Affinity Scheduling in Shared–Memory Multiprocessors", Journal on Parallel and Distributed Computing, 24(2), pp. 139-151, February 1995.

[38] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared–Memory Multiprocessors", Proceedings of the 12th. ACM Symposium on Operating System Principles (SOSP), December 1989.

[39] A. Tucker, "Efficient Scheduling on Multiprogrammed Shared–Memory Multiprocessors", Ph.D. Thesis, Stanford University, December 1993.