TESI DOCTORAL

DYNAMIC SCHEDULING OF PARALLEL APPLICATIONS ON SHARED-MEMORY MULTIPROCESSORS

Autor: XAVIER MARTORELL BOFILL

Chapter 6. Comparison with Previous/Related Work

Abstract

In this chapter we present the comparison of our work with existing research/commercial projects. The comparison is divided in two main sections: User and kernel levels. Nevertheless, some of the projects to compare with also take into account both levels, so the comparison is extended to focus also in the relationship between both levels in each of those cases.

"... Si l'home conegués la seva capacitat de procurar-se moments de placidesa en les grans crisis, no tindria tantes pors i seria més feliç."

"Ronda naval sota la boira", Pere Calders, Edicions 62, octubre 1994.

6.1. Comparison with existing user-level run-time packages

In this section, we present the comparison of the NANOS execution environment with existing user-level thread packages. Some of them are used mainly for parallel application programmers; others are used specifically for parallel compilers to generate code; some of them are designed to support both kinds of users.

6.1.1. Pthreads

The Pthreads package is the implementation of the POSIX Threads standard definition from the IEEE organization of standards [19][65][124]. It is a general purpose threads package. It provides the basic services for thread creation and joining, thread scheduling policies and priorities, synchronization primitives (mutex and condition variables), thread local data management and thread cancellation and signal management.

The Pthreads package is user oriented in the sense that it is thought to be used by programmers, not by a compiler to produce parallel code. Nevertheless, it is sometimes used as a basis to build compiler-oriented packages. This is the case of the MP libraries provided both by Compaq/DEC on Digital UNIX [41][42] and Portland Group on Linux [145][144].

The generality of the Pthreads package is the disadvantage to use it as an execution environment to be used directly by a compiler. The Pthreads standard interface would provide high portability to an execution environment built on it. But the existing implementations of Pthreads usually are heavily dependent on the underlying operating system, respecting to thread scheduling and thread local data support. Also, the environment (thread creation and joining) is too heavy-weight to efficiently support multiple levels of parallelism and fine grain parallelism. For this reason, when the package is used from a compiler, it is only used to create the virtual processors at the beginning of the application and destroy them at the end.

6.1.2. CThreads

The CThreads package is provided with the Mach [1][18] operating system and OSF/1 [77][76][78][79][36]. It was originally developed at the Carnegie Mellon University [29]. Although the CThreads interface has remained nearly the same, several different implementations have been also provided by the OSF organization (now, Open Group).

CThreads is a simple threads package (similar to Pthreads with respect the interface). It was designed to efficiently support the thread management needed inside the Mach operating system. The Mach micro-kernel and several of the UNIX subsystems running on top of it are based on CThreads to manage their internal parallelism/concurrence.

As occurs in the Pthreads case, the CThreads package is not designed to be used directly by a compiler, but by an experienced programmer, which decides which tasks are going to be parallel. The package is too heavy-weight to efficiently support both multiple levels of parallelism and fine-grain parallelism.

6.1.3. Cilk

The Cilk programming environment [17] comprises the Cilk language and the Cilk run-time library. It is being developed by the Supercomputing Technologies Group, at the MIT Laboratory for Computer Science. It its designed for shared-memory multiprocessors and it also offers support to distributed shared memory. The following comparison applies to the shared memory implementation.

Cilk is based on "fully strict" (well-structured) programs. Using the underlying model of fully strict computations [16], it can be proved that the Cilk scheduler achieves execution space, time and communication bounds all within a constant factor of optimal.

The Cilk language extends ANSI C in such a way that allows to express parallelism through the *thread* construct. A thread is defined to be a subroutine receiving some arguments. The Cilk preprocessor translates each thread to a C routine receiving a pointer to its arguments (called the *closure* of the thread). Threads are spawned using the *spawn* construct. This construct creates the closure for the thread, fills all available arguments and sets the *join value*, which is the number of arguments that remain unavailable when the thread is created. Unavailable arguments are considered like futures [149]. They are usually filled by other threads. When the join value reaches zero, the thread is ready to be executed.

With these characteristics, programmers have to learn to code applications using the explicit continuation passing style, which forces to split program subroutines in two or more pieces in order to express parallelism. So, from the point of view of FORTRAN and C programmers, Cilk is introducing an extra effort to understand and learn to use new language constructs, which although can be well-suited for expressing parallelism, are not necessarily clarifying what the application is doing. For this reason we have selected the method of introducing directives in programs. In this way, we can take profit from existing applications and we limit the needs to re-code them.

Cilk is mainly oriented to express parallelism in recursive programs and does not provide any tool for expressing data-parallel constructs. For example, a Cilk programmer should code a data-parallel loop by means of a divide-and-conquer control structure [17].

The Cilk run-time library supports the Cilk language by implementing the mechanisms explained in the previous paragraph. In addition, it includes the Cilk scheduler. Each processor decides the next thread to run using the Cilk scheduler. It first searches for work in the processor's local queue. In case the local queue is empty, the Cilk scheduler applies work stealing to search for work in other processor's local queues. It selects randomly which processor is the victim of the work stealing mechanism. In computing intensive applications usually it is of much importance to achieve good data locality than to balance the processor's load. For this reason we think that it is not necessary to introduce that overhead and complexity to the scheduling mechanisms. It is important that applications could express their preferences for locality and leave opened an opportunity to tune the run-time load balancing mechanisms.

Multiple levels of parallelism and processor grouping are supported by means of recursion and language constructs to express processor affinity. The problem here is that the recursive version of an application is generally not as efficient as the iterative version.

With respect the implementation, Cilk uses a counter for each closure to wait till all its arguments are available. In this thesis, we present results showing that implementing join values is a source of major overhead, and this is specially noticeable when searching for fine-grain parallelism.

Finally, Cilk does not include any support for dynamic processor management and communication with the operating system. Execution of several Cilk applications requesting more processors at a time than those in the machine will suffer from the operating system scheduling decisions. This could lead to undesirable effects at Cilk threads synchronization points.

6.1.4. Filaments

The Filaments execution environment [44][82] was designed for shared-memory and has evolved to run in distributed shared memory multiprocessors. The package was developed in the Department of Computer Science at the University of Arizona. Filaments can be used from C applications like the Pthreads and CThreads packages and from the Sisal functional language [51].

A *filament* is a lightweight thread. Two classes of filaments are supported by the Filaments package: *Iterative* filaments give support to data-parallel execution. They execute repeatedly with an implicit barrier synchronization at the end of the loop code structure. *Fork/join* threads are used in recursive programs. Each thread creates other threads and then waits for their results.

The Filaments package supports fine-grain parallelism. Thread descriptors are created directly by the programmer using the tools provided by the Filaments package. Thread descriptors are small, so thousands of descriptors can be created before starting the parallelism. Each thread descriptor contains the Filaments arguments and the function to be executed by the thread. Parallelism is explicitly started calling another service of the Filaments package. Iterative filament descriptors are automatically reused during execution. Filaments includes tools for control of thread placement for data locality, efficient barrier and fork/join synchronization mechanisms and automatic load balancing for fork/join threads.

The programmer of a Filaments application must learn a new interface and code his/her applications using it, trying to express all the parallelism. There is an extra work to map each parallel construct to the different classes of threads. Existing applications have to be re-written to use the Filaments package.

Filaments support multiple levels of parallelism for the fork/join threads, because of the recursive nature of such kind of threads, like in Cilk. Nevertheless, as Filament threads do not provide an address space (stack), they do not allow creation of extra levels of parallelism, in the general case, when some data should ne declared locally to be used by the inner levels of parallelism.

As in the Cilk case, the Filaments package does not provide any possibility of interaction between the application and the operating system. For this reason, any scheduling decision taken at operating system level may cause synchronization delays due to undetected processor preemptions.

6.1.5. COOL

The COOL execution environment [23][22] comprises the COOL language and compiler and the COOL run-time system. The COOL language (Concurrent Object Oriented Language) extends C++ with tools for expressing concurrence and communication. It has been designed to exploit coarse grain parallelism at the task level in NUMA shared memory multiprocessors. COOL was developed in the Computer Systems Laboratory at the Stanford University.

The COOL run-time system is mainly oriented to execute parallelism expressed through the *parallel* COOL construct. This construct applies to functions and object methods. Calling a parallel function/method implies the creation of a *task* to execute it. This task executes asynchronously with the caller. The parallelism exposed by the application can be constrained at invocation points just by indicating that the call must be done serially.

Mutual exclusion among parallel invocations of functions/object methods is achieved by means of a function definition attribute. All object methods declared with the *mutex* attribute execute in mutual exclusion with other methods in the same object. Mutual exclusion is taken at function entry and released at function exit.

Parallel functions are inserted in ready queues when created as task descriptors. Processors pick up work from the ready queues during execution. There are two kinds of ready queues in the COOL run-time system in order to achieve data locality in three different ways: Object, task and processor affinity. The programmer is able to express such affinity classes in each function invocation. In addition, idle processors steal tasks from busy processors' ready queues to improve load balancing.

The programmer is allowed to help the run-time system about data and object placement. This means to distribute object data across processor memories in NUMA machines. Also to migrate data from one processor to another when it is required by the data access pattern exhibited by the application.

As in the Cilk and Filaments cases, the COOL programmer must be aware of restructuring his/her application in order to fit with the specifics of the new language. Our approach simplifies the way the parallelism is expressed and it works fine in the general case, where application data is allocated near the processor that first touches it.

The COOL execution environment is related with the Process Control approach at the operating system level. The comparison of Process Control with our approach is presented in the next subsection.

6.1.6. Illinois Concert System

The Illinois Concert System [27] is a concurrent object-oriented language and a run-time system developed at the University of Illinois at Urbana-Champaign.

The Concert run-time system provides primitives for thread management, communication, and mechanisms for achieving data-locality and load balancing. Such primitives and mechanisms are efficiently implemented in such a way that they usually achieve performance an order of magnitude higher than do corresponding primitives in vendor-supplied communication and thread libraries [67]. They are designed to maintain a high level of performance over the unpredictable dynamic behavior characteristics of irregular applications.

The run-time system is implemented on both shared and distributed memory machines. Two main modules, language-independent and language-dependent provide the following functionalities: communication, thread management, memory management and object caching are language-independent. Users can add their own user-level scheduler. Memory management provides a distributed garbage-collection algorithm to reclaim unreferenced memory. Object support, contexts, futures and continuations are provided by the language-dependent module.

ICC++ [26][67] is an extension of the sequential programming language C++ [143]. ICC++ allows the programmer to express concurrency among C++ statements and inside loops. The compiler analyses the code and determines whether there are dependencies among the statements, through aggressive static global program analysis and transformations. The compiler uses a unified flow analysis framework to obtain information about the compiled program. From the previous analysis, four inter-procedural static optimizations that reduce the overhead associated with the concurrent object-oriented programming model are applied: Method inlining, procedure cloning, object inlining and access region expansion to search for larger basic blocks.

The Concert run-time system provides more functionality than NthLib. This is due to support for distributed memory machines and communication and also because of extended memory management and garbage-collection, futures and continuations. Memory management in NthLib is simplified and unified in the management of the nano-threads stacks. Futures are not needed because of the consideration that it is the compiler which should be able to determine whether a function call can be executed in parallel or not. Continuations, although are internally used by NthLib to implement thread blocking, are provided to the user by means of successor nano-threads.

6.1.7. Active Threads & pSather

Active Threads [150] is a threads package developed at the International Computer Science Institute of the University of California at Berkeley. Active Threads was designed to facilitate high-performance fine-grained platform-independent parallel programming, to help in taking advantage of the memory hierarchy on modern machines and to make possible performance profiling of threaded software [151].

In order to provide fine-grain parallelism support, the implementation searches for efficiency in the thread creation, context switch, and synchronization operations. Also, the interface allows to extend the scheduler by taking into account data locality or implementing memory-conscious scheduling. New scheduling modules can capture the dynamic structure of the evolving parallel computation and allow to schedule threads in a way that reflects this structure. The run-time system allows to schedule nested parallel constructs in a way that the most internal level can span all or a subset of the available processors. Each thread has an state, by which it is controlled by the library.

Threads are defined as units of (potentially parallel) execution that share an address space and other system resources. Groups of logically related threads are organized into *thread bundles*. Threads in the same bundle share a common thread scheduler. This is the mechanism that can be used to schedule parallel loops using groups of processors. At any given moment, one bundle has the *execution focus*. The bundle which has the focus receives the idle scheduling events and can assign work to the idle processors or it can propagate the idle event through the bundle activation tree.

The Active Threads implementation hides the number of available processors by using *virtual processors.* Threads are scheduled on top of virtual processors. There is no limit in the number of virtual processors, but it is recommended that threads accessing the same data be scheduled on the same virtual processor. The package ensures that those threads will be executed in the same physical processor.

The Active Threads library can be used directly or as a compilation target for parallel languages. Along with the basic thread management primitives, the Active Threads interface supports a great variety of synchronization mechanisms (spin-locks, mutexes, semaphores and condition variables).

The library works on several architectures and operating systems (SUN SPARC/ Solaris, Alpha Axp/Digital UNIX, Intel 386 and HPPA) but it is uncompleted on several operating systems in the sense that it does not allow to use more than one processor, by now.

The pSather language [93][142] is a parallel object oriented language which extends the Sather language. Sather was initially based on Eiffel but now incorporates ideas and approaches from several other languages. Thread extensions to Sather include the possibility of declare parallel sections of code and parallel loops. The pSather compiler generates C code from pSather programs in such a way that they are linked with the Active Threads package to run on parallel machines.

Active Threads shares some characteristics with our nano-threads package. It provides the bundle mechanism for supporting multiple levels of parallelism. And it allows the users to change the scheduling policy for each bundle, thus providing a method for processor grouping. It has also some extra functionality needed to support thread synchronization and code generation from object oriented languages.

6.1.8. Illinois-Intel Multithreading Library

The Illinois-Intel Multithreading Library (IML, [56][126]) is a user-level threads package targeted to shared memory multiprocessors. It supports multiple levels of general, unstructured parallelism. General (or functional) parallelism is provided by allowing the expression of task execution conditions through a directed acyclic graph. Multiple levels of parallelism is provided through ready queues.

Applications supply work to the library through ready queues of application tasks and the library schedules them on the available processors. There is one local ready queue for each processor. Task queues allow several task descriptions to be active at the same time. IML focuses on the design alternatives for implementing such task queues (centralized and/or distributed). The library is in charge of mapping the tasks to the available processors. The outer-most level of parallelism distributes work on the available processors. Inner levels of parallelism generate work on the local processor spawning the parallelism. IML is also in charge of load balancing. Library level scheduling allows processors to steal work from the neighbor processors. The library interface does not allow to express any data locality hint. Parallel loops are implicitly distributed by the IML among the participating processors.

IML supports code generation from Parafrase-2 and the Intel Fortran compilers. It runs on Intel platforms and Windows NT.

6.1.9. SGI IRIX MP Library

The SGI IRIX MP library [137][133] supports efficient parallel execution in the Silicon Graphics multiprocessor machines. Applications are parallelized either automatically or manually to run on top of the MP library. Automatic parallelization is achieved through the PFA (Parallel Fortran Analyzer) tool [134]. PFA gets a sequential program, analyses it and generates an optimized sequential program annotated with directives. Manual parallelization consists on manually annotating a sequential application using OpenMP directives. In either case, the MIPS Pro F77 compiler understands the OpenMP directives and generates the parallel code.

The compiler generates two versions of each one of the parallel regions of the program. The first one is the parallel version. In this version, the compiler encapsulates the parallel region in a function and it replaces the code by a call to the MP library to spawn parallelism on that function. The second one is a sequential version of the code to be used when there is only one processor available, avoiding the overhead of the spawning and joining mechanism. This version can also be used when the application detects, dynamically at run-time, that there is not enough work to spawn in a parallel region, or when the parallelism has already been spawned before.

The MP library provides a complete execution environment for each application, supporting thread creation, management, synchronization and NUMA features, such as memory placement. Also, the library is aware of the machine load, trying to adjust the parallelism which is exploiting to the available resources.

When a parallel program starts, a pool of lightweight processes (sprocs or kernel threads) are created, the number indicated by the user. When the execution reaches a parallel region, the master processor enters the MP library and uses a unique work descriptor, located at a fixed memory location, to spawn the parallelism. Other processes (the slaves) pick up work from that global descriptor. Each process executes the same block of code (encapsulated in the function containing the body of the parallel section [21]. Variables can be either shared or privatized, at the parallel function entry, as necessary.

This implementation restricts the parallelism that can be exploited at application level to a single level [133] because the descriptor cannot be reused until the previous parallelism has been joined. If the program reaches a parallel section while already executing in previous one, the sequential path is taken.

The SGI MP library provides three different implementations for thread joining: a distributed joining structure in shared memory (used by default), a shared counter updated through atomic machine level instructions (which provides bad performance when using more than a few processors because of the large amount of traffic), and a shared counter based on uncached atomic memory operations on the specialized hardware in the memory modules of the Origin2000 system.

When a slave process completes its portion of the work of a parallel region, it returns to the MP library code, where it picks up another portion of work if any work remains, or waits polling for a while, and then suspends itself until the next time it is needed. The MP library uses a two-phase synchronization mechanism. When a spinning process spends too much time to acquire a spin lock, the library decides to yield the processor to allow the progress of other processes in the system.

While this is a good solution, it is not general enough to perform well in all situations. A false processor utilization appears when an application reaches a sequential section and the slave threads are spinning for some time before yielding. And as the operating system knows nothing about the execution at user level, in a multiprogrammed machine, an application can loose a processor which, in the worst case, may be given to another spinning process.

The SGI MP environment supports a per application control of the degree of parallelism. An auxiliary process wakes up each approximately three seconds and suggests an optimal number of ready process based on system load information. This number is considered the next time a new parallel region is spawned. The application blocks/unblocks some processes, if necessary. Using this mechanism, the application parallelism is decreased progressively when the system load is high or when the application CPU usage is low due to input/output or page faults [3]. This feature is related with an environment variable (OMP_DYNAMIC, which can be set to TRUE or FALSE). It is activated by default.

The comparison between the SGI MP library and our approach is completed in the evaluation of Chapter 9.

6.1.10.Integrating functional and loop parallelism

Several programming systems are trying to provide multiple levels of parallelism by integrating functional (task) and loop (data) parallelism. They usually focus on allowing the coordination of several program modules in architectures mixing shared and distributed memory.

The Fx project [57] focuses on proposing a small set of Fortran directives to integrate task and data parallelism on a HPF framework [69]. Applications under consideration are mainly related to processing continuous streams of data sets. The PARADIGM project [120] proposes the use of controlling the degree of data parallelism in individual tasks through task parallelism, an approach similar to our processor grouping. They extend the model proposed in the Fx project to target a more general class of applications and to perform allocation and scheduling of processors to HPF data parallel routines automatically without any user intervention.

The HPF/MPI project [50] merges High Performance Fortran with the Message Passing Interface standard. Calling MPI from HPF avoids to introduce new parallel constructs in the language, providing a library based solution. This system allows the creation of a number of parallel tasks written in HPF, that run in a data-parallel way on a specified collection of machines, controlled by MPI. The tasks communicate with each other through the MPI interface. Other related works propose extensions to the HPF directive set to indicate that several calls to data parallel HPF routines can proceed in parallel [49].

6.2. Kernel approaches

In this section, we compare our approach for the kernel interface and scheduling techniques, the widely used gang scheduling [108][15], the SGI IRIX 6.4 operating system [30] and the user/kernel interface and scheduling techniques implemented recently in the SGI IRIX 6.5 operating system [32], as well as with well-known research projects, notably Process Control [147][148], First-Class Threads [84] and Scheduler Activations [5][53][70]. As all the projects are very related and there are more similarities than differences among them, we directly compare here the more interesting aspects to consider instead of comparing with each individual project.

We summarize the comparison between the different approaches in fifth relevant aspects: who is in charge of setting the degree of parallelism of each application; which is the target for processor scheduling, the application or the individual processes; the effect of the blocking events in the behavior of the scheduler; the synchronization issues related to scheduling; and the type of user / kernel interaction. Current operating systems providing scheduling support for multiprocessor architectures inherit several characteristics from their original UNIX versions (System V [9], BSD [73]).

6.2.1. Setting the degree of parallelism

The first aspect concerns the question about who is in charge of setting the degree of parallelism of an application. It is set by the user in gang scheduling, where the application simply creates a number of processes and the operating system must schedule all of them together to allow the application to make progress. Our approach allows the application to set the maximum number of processors it wants to run on and the operating system will allocate the most suitable number of processors taking into account the overall system load. The actual degree of parallelism is set by the operating system. This feature is also included in the Process Control approach [148] where the operating system sets a number in shared memory informing the user of the number of currently running processes inside the application. It has also recently been incorporated in the SGI IRIX 6.5 operating system in the form of a *nestimated* number of processors [32].

In previous versions of the IRIX operating system (IRIX 6.4), the user-level MP library tracks, through an extra process running inside the application, the machine load and whether the application is executing well or not. This means that in fact the degree of parallelism is set by the user-level execution environment. When the library detects that the machine load is too high (over the amount of physical processors) and the application is not running well (it is not receiving CPU time), it automatically shrinks the parallelism to be spawned at the next parallel construct. This is done with no kernel intervention. There is no global view and the decisions can not be accurate enough to achieve a good multiprogramming efficiency. In particular, when the MP library checks the way the virtual processors are executing, it can detect that they are doing well when, in fact they are running the MP library idle threads, waiting for a synchronization. Also, the SGI MP library is in charge of stopping those processes which are not currently given any work by the application. This is done in a tunable way, allowing the application programmer to set the amount of time the processes are going to spend searching for work before stopping themselves. By default, this time is set to 0.25 seconds (or 10,000,000 iterations in a MIPS R10000 processor running at 200 Mhz.). This can be set individually for each application using the MP_BLOCKTIME environment variable. Although the method is valid, two drawbacks are: first, applications with dynamic parallelism are going to keep some processors during the configured time without making useful work; and second, that this parameter should be tuned for each one of the applications running in the system.

Similarly to Process Control and IRIX 6.5, we also allow dynamic processor requests during the lifetime of an application. IRIX 6.4 is supporting dynamic requests by stopping the processes which do not receive work from the application, as stated in the previous paragraph.

6.2.2. The application as the scheduling target

The second important difference among our approach and other approaches is the way parallel applications are managed by the scheduler. In current operating systems, kernel-level events that are relevant to the execution of an application are communicated to each individual process belonging to an application, in such a way that the rest of processes of the same application continue execution ignoring what has happened. It is the case of gang scheduling. When a process belonging to a parallel application is selected to run, other processes are preempted in order to be able to execute the complete application. At this point, gang scheduling is not aware of whether the preempted processes belong to different applications or not. The affected applications are not informed about the preemptions. They are going to suffer a penalty in their execution motivated from the disordered preemptions. Instead, in our approach the scheduling target are the applications as a whole. In this way, the scheduler first selects which applications are allowed to run and, in a second step, they are given some processors, informed of how many are receiving and allowed to run. During execution, processor movements between applications are minimized. And more important, applications are preempted as a whole when the scheduling policy decides not to give processors to an application during the following time quantum.

In the Scheduling Activations approach all the events are provided to the user level by means of upcalls. This means that in this case the application is able to react to the event. The communication mechanism, the upcalls, are usually too heavy-weight to provide good response time. Shared memory is used both in Process Control and First-Class, as well as in IRIX 6.5. Nevertheless, Process Control and First-Class continue assuming that the scheduler target are the individual processes.

The IRIX 6.5 scheduler allows applications to inform the kernel about their processor requirements. As soon as an application sets the requested number of processors, the IRIX kernel assumes that all related processes belonging to the application should be managed as a whole. The IRIX scheduler based on earnings is used to compute the *estimated* number processors that should receive each application. When computed, this value is supplied to the user-level scheduler as well as the corrected value (the *allocated* number of processors) computed having in mind the *requested* number of processors.

6.2.3. Blocking events management

The third important difference between the different approaches is how blocking interferes with kernel scheduling decisions. This is very related to the second difference (in Subsection 6.2.2), but as different designs present different solutions to this problem, we consider it as an important aspect.

Starting with gang scheduling, when a process of a gang-scheduled application blocks in the kernel, the processor immediately tries to execute another ready-to-run process. This process may belong to another gang-scheduled application, reclaiming other processors to run in its turn, and motivating more context switches. The problem here is that, although gang scheduling ensures that all processes of a parallel application start executing together at the beginning of a time quantum, it does not ensure that all the required processors will be available to the application till the end of the time quantum. At this point the application performance is degraded because of synchronization issues. Our approach maintains the number and mapping of processors allocated to the application, similarly to Scheduler Activations [5]. When a processor becomes free due to a blocking event, it is first returned to the application in case it has more work to do with it. Otherwise, it is given to another running application and both applications are informed of this movement.

In IRIX 6.5, instead, when a process of a parallel application blocks, the process is not guaranteed to remain in the application. The kernel can decide to move the processor to another application running in the system. In this case, the application loosing the processor is not informed about this fact until either the process unblocks and the processor returns or the process is marked as a preempted one and the context is supplied to the user level through shared memory.

6.2.4. Synchronization issues

With respect to the synchronization mechanisms used, gang scheduling usually is accompanied with two-level synchronization methods at user-level [148][47][58]. When a spinning process spends a predetermined amount of time to acquire a spin lock without success, the user-level run-time package decides to yield the processor to allow the progress of other processes in the system. As it has already been commented in Subsection 6.1.9, this is a good solution, but is not general enough to perform well in all situations. As the operating system knows nothing about the execution at user level, an application may loose a processor which, in the worst case, may be given to another spinning process. This is the fourth difference with our approach. We are not using two-level synchronization. Instead, every application detects, at safe points, whether any of its virtual processors has been preempted and then it yields one of its currently assigned processors to the preempted virtual processor.

6.2.5. User / kernel interaction

The last significant difference is the amount and quality of the information shared between an application and the operating system. Process Control maintains a counter of processors allocated to the application and informs the application of kernel level events through UNIX signals. Scheduler Activations and First-Class Threads inform using upcalls. Our proposal is to use the most efficient mechanism to inform applications of such events. Processor allocation and blocking are easily communicated through upcalls, offering the new processor to the application. Processor preemption and unblocking are most easily informed through shared memory. For this reason, we maintain a shared memory area between the applications and the operating system containing the information explained in Subsection 5.2.

The approach taken in IRIX 6.5 provides a per-application shared memory area [116][32] similar to our proposal containing the number of processors requested by the application, the number of currently allocated processors, the number of processors estimated by the kernel that could run in the application; also, it contains the register save areas (RSA's) to save/restore the user-level threads state at blocking and preemption points. In this way, both the user and kernel levels can resume user-level threads whose state is in the shared area. Along with this information there is also some control data related indicating whether an RSA is free or busy and to signal preemption status for each kernel-level thread.

Chapter 7. Implementation Issues

Abstract

This chapter highlights the most relevant implementation issues both in user- and kernel-levels. At user-level, we present here the way we have integrated multiple levels of parallelism with efficient work descriptors and processor grouping.

At the kernel-level we show how our proposals for the kernel scheduling framework can be implemented on existing operating systems, using specific tools provided by the current operating system interfaces.

"... moltes vegades a la vida, el problema és identificar el problema."
(Llegit en plena optimització.)

Josep Borrell, Classe de Polítiques Públiques, Universitat Pompeu Fabra. Diari Avui, dissabte, 23 de maig de 1998.

7.1. User-level implementation issues

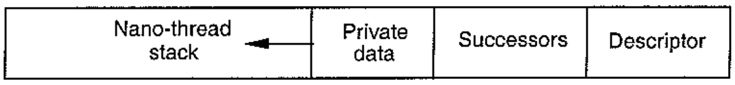
The following subsections present a detailed description of the implementation of the NthLib package as it has been developed on the Silicon Graphics Origin2000 machines, and outlines the different aspects that have been changed in the different ports to other systems. The current implementations in several systems provide most of the following characteristics.

7.1.1. Data structures

The nano-threads library contains several data structures which maintain both information for each nano-thread created and the global information about the user-level execution environment of the nano-threaded application.

7.1.1.1. The nano-thread structure

The nano-thread structure (see Figure 21) contains the nano-thread descriptor, references to the nano-thread successors, nano-thread private data and the nano-thread stack. It is initialized by the `nth_create()` interface described in Subsection 7.1.2.



The contents of the nano-thread descriptor is as follows:

```
struct nth_desc {
     struct nth_desc    *next;
     int                 vp_id;
     void               *sp;
     count_t             npred;
     spin_t              npred_mutex;
     count_t             nsucc;
};
```

Figure 21: Nano-thread structure

The nano-thread descriptor contains the following fields:

- `next`. A pointer to a nano-thread. This field is used to queue the nano-thread in any of the library queues, such as the ready queue.
- `vp_id`. The user-level identifier (0, 1, 2, ...) given by the package to the underlying virtual processor currently executing this nano-thread. It is inherited between nano-threads at context switch time.
- `sp`. The value of the stack pointer is saved in this field when the nano-thread is created or when it is blocked.
- `npred`. The number of predecessors that remain already unresolved for this nano-thread. When it becomes zero, the nano-thread is ready to be executed and can be queued in the ready queue.
- `npred_mutex`. The synchronization variable to access the `npred` field in mutual exclusion.
- `nsucc`. The number of successor nano-threads (representing HTG nodes) depending on this one.

References to successor nano-threads are stored beside the descriptor. At termination, each nano-thread uses this field to update the predecessor counter for their successors. The nano-thread private data area keeps any variable that has to be privatized at nano-thread creation. In Fortran, all loop control variables supplied to a nano-thread have to be privatized because, they are passed by reference by the Fortran compiler. The FIRSTPRIVATE clause, in OpenMP, also uses this feature to pass the initial value of the private variable. The stack is initialized containing the nano-thread arguments in such a way that nano-thread execution can start in the associated C or FORTRAN function. During the nano-thread execution, standard activation frames are allocated in the stack.

Nano-thread structures are allocated using the operating system virtual memory primitives and are never deallocated. Instead, they are reused as described in Section 7.1.2.

7.1.1.2. The nano-threads global data

The nano-threads library maintains a data structure containing global data. It consists of information used during the execution of the nano-threaded application. We have taken care of the placement of each field in memory, avoiding as much as possible any conflicts among them, such as false sharing, fitting each field in its own secondary data cache, when necessary. The important fields in the global structure are:

- The nano-threads ready queue (global and per-processor queues).
- The queue of free nano-thread structures (global and per-processor).
- Stack sizes (total, and for the first and regular nano-threads).
- A reference to the memory shared with the CPU Manager, when present.
- Various counters to take statistics at execution time (e.g., number of allocated structures, current number of created nano-threads and total number of created nano-threads).

The ready queue is a list of nano-threads which allows insertions both in front of it and at the end. New nano-threads created from the start node of a compound node and ready to be executed are usually added to the end of the queue, thus maintaining the same execution order as is given by the application.

After a nano-thread termination, all nano-threads that become ready due to their precedences resolution are enqueued in the ready queue provided by the application at nano-thread creation. An alternative was to execute one of the readied successors in the same processor. We have decided to enqueue it in the ready queue for a double reason: Processor grouping is easily manage in this way and data locality is maintained as the application has indicated.

The queue of free nano-thread structures contains those structures released when nano-threads terminate. This queue is always managed in a LIFO order. In the current implementation, this queue consists of several instances, one for each processor in order to maintain the affinity between the nano-thread structures and the processor where they have been executed.

The nano-threads library and the operating system maintain a shared memory area containing the information related to the processors currently allocated to the application. In such a way, both can accurately know the state of the hardware and software resources: processors allocated, processors stolen, number of nano-threads in the ready queue, etc. When running NthLib on proprietary operating systems (IRIX, Digital Unix, Solaris), we have implemented a reduced interface between the library and the operating system depending on the information that can actually be obtained. Subsection 7.2 describes this interaction in more detail.

7.1.2. Nano-thread creation

Nano-threads are created through the following interface:

- C interface:

```
struct nth_desc * nth_create (void (* func) (...),
                              int npred,
                              int vp,
                              int nsucc,
                              int narg,
                              ... /* nano-thread successors and arguments */ );
```

- FORTRAN interface:

```
struct nth_desc * nthf_create_ (   void (* func) (...),
                                   int * npred,
                                   int * vp,
                                   int * nsucc
                                   nth_argdesc * desc,
                                   int * narg,
                                   ... /* pointers to nano-thread successors and args
*/ );
```

Creation of a nano-thread consists in getting a nano-thread structure from the queue of free structures and initialize it. If the queue is empty, a new structure is allocated using the operating system virtual memory primitives. Typical primitives include `mmap` in IRIX, Digital UNIX, Linux, Solaris and HPUX or `vm_allocate` in Mach.

Next, the starting stack pointer and the number of pending data precedences for the nano-thread are initialized in the descriptor. The successor references are stored beside the descriptor. Nano-thread arguments are copied in the nano-thread stack.

Nano-thread creation is slightly more complicated in FORTRAN because in this language arguments are always passed by reference. In the FORTRAN interface an extra argument (`desc`) is given in order to specify both the size (32/64 bits) and the need for privatization (true/false). We ensure that private arguments are first copied into the nano-thread stack. In this way, the original arguments, belonging to the creator nano-thread, can be afterwards modified without modifying the privatized ones. Finally, the starting nano-thread stack is completed with the correct references to the arguments. Arguments for which privatizing is needed are usually loop control variables, which will be subsequently modified by the creator nano-thread to advance to the next loop iteration. Arguments correctly passed by reference are usually references to global program variables.

If the number of precedences is zero, the nano-thread is automatically enqueued in the ready queue of the processor indicated by the `vp` argument. The `nth_create()` function returns a reference to the newly created nano-thread. The caller is able to supply this nano-thread as the successor for any other nano-thread.

7.1.3. Nano-thread self identification

An executing nano-thread is able to determine which is its nano-thread descriptor using the following interface:

```
struct nth_desc * nth_self ();
```

The `nth_self()` function returns a reference to the currently executing nano-thread. The mechanism used to find the self reference is like the one used in the CThreads package [29] and the OSF/1 operating system [36], based on storing it at a fixed location in the stack and working with stacks aligned to addresses multiple of a power-of-two.

7.1.4. Dependencies control

As we have introduced in Subsection 4.1.2.3, the number of precedences that remain unresolved for a nano-thread can be dynamically increased/decreased when new inner parallelism is started/terminated. The compiler uses the following interface for this purpose:

```
int nth_depsatisfy (struct nth_desc * nth);
void nth_depadd (struct nth_desc * nth, int npred);
```

The function `nth_depsatisfy()` decrements by one the counter of precedences for the supplied nano-thread (`nth`). It returns `true` if the dependence counter reaches zero and `false` otherwise. `Nth_depadd()` is used to increment the number of precedences by the supplied value `ndeps`.

7.1.5. Ready queue management

The ready queue interface allows the application to dynamically generate new work to be executed while spawning parallelism. One global ready queue lets the application to supply nano-threads to the first available processor. Local per-processor ready queues allow to evenly distribute work among processors. Local queues are useful to maintain data locality in parallel loops.

The queue structure maintains a synchronization variable to access it in mutual exclusion and a reference to the first and last elements currently queued. Elements are defined to be nano-threads. Figure 22 shows the structure of the ready queue and its interface. Each queue instance is really allocated to fill an entire cache line to avoid false sharing among the processors.

```
/* Queue structures */
struct nth_single_queue {
    spin_t                  q_mutex;
    struct nth_desc        *q_first;
    struct nth_desc        *q_last;
};

struct nth_ready_queue {
    struct nth_single_queue [MAX_CPUS];
};
```

```
/* Application-level global queue interface */
void nth_to_rq (struct nth_desc * nth);
void nth_to_rq_end (struct nth_desc * nth);

/* Application-level local queue interface */
void nth_to_lrq (int which, struct nth_desc * nth);
void nth_to_lrq_end (int which,
                               struct nth_desc * nth);
/* Dequeue and test for emptiness functions are
internal and they are not visible to the application
level */
```

Figure 22: Ready queue structure and interface

The `nth_to_rq()` and `nth_to_rq_end()` functions insert an element (`nth`) in front of the global ready queue or at the end, respectively. The same functions are provided to operate on the local ready queues. They are `nth_to_lrq()` and `nth_to_lrq_end ()`.

There are internal functions to initialize the ready queue to an empty state, to dequeue the first element in the queue and to test whether the queue is empty. The internal functions `nth_rq_empty()` and `nth_lrq_empty(which)` return TRUE if the corresponding ready queue is empty. A queue is defined to be empty when its `q_first` field is NULL. Waiting for work in the ready queue can be performed by testing the `q_first` field without getting the queue lock, in order to avoid cache ping-pong effects.

The queue of free structures, which is used internally by the nano-threads library uses a similar interface.

7.1.6. Blocking

A nano-thread is able to block while waiting for the termination of other nano-threads. Before blocking, the successor of those nano-threads is setup to be a reference to the blocking nano-thread. The basic blocking primitive is:

```
void nth_block (void);
```

When a nano-thread blocks, it first searches for work to be done, exactly like in a nano-thread termination. If it finds a ready nano-thread, the context switch is performed and the blocking nano-thread simply waits for the termination of its new predecessors. The implemented mechanism ensures a correct behavior in a multiprocessor environment avoiding that early termination of the parallelism activates the nano-thread before blocking has been done. If the blocking primitive does not find any nano-thread to execute, it simply waits for its precedence counter to become zero and continue working.

The blocking interface is intended to be used in sequences like the following:

```
 // add 'N+1' new dependencies to the current nano-thread:
 //    'N' for the new parallelism, '1' for myself.
nth_depadd (nth_self (), N+1);
 // create and queue 'N' new nano-threads passing myself as the successor
for (i=0; i<N; i++) {
      nth = nth_create (func, 0, 1, narg, nth_self (), .../* arguments */...);
      nth_to_rq_end (nth);
}
 // block, waiting for the parallelism to be terminated
nth_block ();
 // continue executing when the parallelism is terminated
```

7.1.7. Portability issues

We have based the implementation of the nano-threads library on top of the Quick Threads (QT) package [68]. This has shown very useful to improve portability. Basic thread creation and context switch are efficiently implemented in the QT package. Nano-thread creation is based on Quick Threads, which allows creation of threads with a varying number of arguments.

Context switch is implemented in two ways in the QT package. First, it is possible to make a standard context switch, which saves the previous state on the current thread stack, changes the stack and restores the new context. This version is used when a nano-thread blocks. It is also very useful to use a simplified context switch when the thread is terminating. In this case, which is the common case in the nano-threads environment, the current context is not saved. Only a switch to the new stack is done and the new context is restored.

Quick Threads also isolates the nano-threads package from several hardware dependent parameters, such as the best size for the representation of integers and pointers (usually 32 or 64 bits), the stack growth (to increasing/decreasing addresses), the best alignment for scalar data and the hardware virtual page size. Porting QT to other architectures is a well-defined task in the sense that a common interface consisting of a few routines has to be supplied. We easily ported the QT package to work on R10000 processors using their 64-bit programming environment and NthLib could be installed with minor changes.

The QT package does not offer mutual exclusion primitives. Next section describes mutual exclusion in the nano-threads library.

7.1.8. Mutual exclusion

Queue and dependence management requires simple and efficient mutual exclusion primitives. In general, the time a thread has to wait for a data structure (the ready queue or the dependence counter of a successor nano-thread) is very short. For this reason, the nano-threads library defines the mutual exclusion interface based on spin locks at user level [58]. This interface is intended to be used internally only.

A lock in the nano-threads library is a memory location which size is hardware dependent. It is defined to be the same as the Quick Threads data type `qt_word_t`:

```
typedef volatile qt_word_t spin_t;
```

The following interface has been implemented to access it:

```
void spin_init (spin_t * spin_var);
void spin_lock (spin_t * spin_var);
void spin_unlock (spin_t * spin_var);
```

The `spin_init()` function initializes the spin lock variable (it sets it to zero). This function is hardware independent.

The `spin_lock()` function is based on *test&set* hardware primitives. In order to avoid a ping-pong effect among the caches of two or more processors spinning in the same lock variable, this primitive tests the contents of the local variable in the local cache first, and does not write to it till the test indicates the lock is zero.

This function is hardware dependent because it has to use an atomic test and set instruction to update the spin lock variable. For example, the MIPS and DEC Alpha implementations are based on a restartable atomic sequence containing a *load linked* and a *store conditional* machine instructions [43][140][61][39][40]. The implementation on the Intel architecture uses the exchange machine instruction (*xchg* [66]) to atomically load the contents of the spin lock variable and store a value of one.

The `spin_unlock()` function consists in resetting the spin lock variable to the initial value (zero). It is hardware independent.

7.1.9. The virtual processors scheduling loop

At initialization time, each virtual processor receives a function and a stack where to start execution at user-level. The starting function simply calls the nano-thread scheduling loop, which is responsible for selecting the next nano-thread to execute. Virtual processors or kernel threads are obtained through the operating system interface. The IRIX operating system provides the `sproc` system call. Linux provides the `clone` system call, very similar to the IRIX `sproc`. Digital UNIX and Mach provide the `thread_create` system call. Solaris and HPUX provide the `_lwp_create` system call.

At the termination of a nano-thread, the scheduling loop is also automatically invoked by the QT package. At this point, the successors of the terminating nano-thread are activated as explained in Subsection 7.1.1.2. Then, it searches for work in the ready queue. It first searches in the local queue of the current processor and, if it is empty, it searches in the global queue.

While performing a context switch from one nano-thread to another, the library transfers the identifier of the virtual processor (`vp_id`) from the old to the new thread. This identifier allows the library to know which are the characteristics and status of the processors assigned to the application. Also, it allows to the application to implement several techniques to improve data locality.

Subsection 7.2 presents the interaction with the operating system which is done also inside the library-level scheduling loop and the Example 7.1 shows the complete virtual processor scheduling algorithm.

7.1.10.Thread fork/join techniques

The main goal of our proposals for implementing efficient thread fork/join in the nano-threads environment is the support for multiple levels of parallelism and processor grouping. In this subsection, we present the implementation of both the GWD and LWD techniques for supplying work to processors and an improved thread joining scheme [85].

7.1.10.1. Forking threads

Forking threads efficiently at the inner-most level of parallelism is based on supplying a work descriptor to the participating processors. The work descriptor consists of a pointer to the function encapsulating the work that has to be executed and its arguments. When the same work descriptor is supplied to a group of processors, each one decides the portion of work that has to execute, based on the arguments, the number of processors working in the group and its own identifier inside the group.

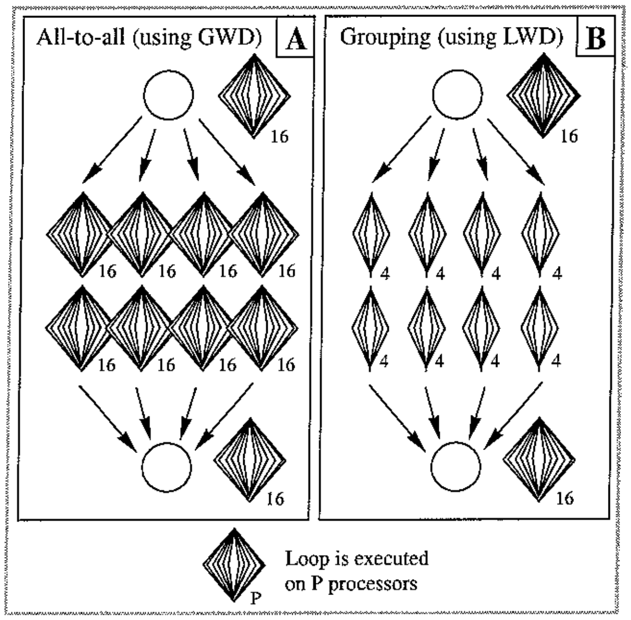


Figure 23: Two alternatives for the exploitation of multiple levels of parallelism

Functionality. The forking techniques are GWD (Global Work Descriptors) and LWD (Local Work Descriptors). Along this subsection, we use the abbreviation WD to refer to both the LWD and GWD techniques. The GWD can be used in spawning the inner-most level in a multi-level parallel application. It supports multiple levels of parallelism because it allows the coexistence of multiple opened parallel constructs, solving the limitation of a single work descriptor found in implementations supporting a single level of parallelism (see Subsection 6.1.9). All processors share a single GWD structure, they all can simultaneously supply work to the GWD and they execute the same work. GWD is expected to perform comparable to existing highly tuned implementations when exploiting a single level of parallelism. Figure

23A shows the execution of a parallel construct consisting of a parallel loop, four independent sections (also containing parallel loops) and another parallel loop. In this case, the master thread executing each section spawns the parallelism associated to the parallel loops inside the section to all the processors. Since the same descriptor is supplied to all the processors for each parallel loop, each processor in the system will execute a chunk of iterations of all the loops (probably always the same, if the compiler exploits loop affinity)

LWD is a more advanced technique that inherits most of the characteristics of the GWD and also allows processor grouping. Using processor grouping, the application is able to drive several processors to work on a independent task or set of tasks. Each group has a master and (possibly) several slave processors. The master processor starts the parallel task and it is in charge of spawning the parallelism encountered inside it. After that, the slave processors cooperate with the master to execute the parallelism. At the end, the master processor waits for the slaves to complete the work. This execution model requires that any processor can supply work to any other processor. There exists one LWD structure for each processor, allowing different work to be supplied to different processors from different parts of the parallel application. In Figure 23B, the master thread executing each section spawns the parallelism associated to the parallel loops inside the section to just a subset of all the processors (in this case, each group of four processors is supplied with a different work descriptor). Due to the definition of these groups, now each processor executes a chunk of iterations for a subset of all the loops inside the parallel sections. However, for loops outside them, all processors cooperate in the execution of the parallel loops. Care should be taken into account about how this change in the structure of the parallelism can influence data locality.

As a result, the LWD overhead can be higher than that of GWD due to the individual supply of work descriptors, but it is expected that limiting the number of processors participating in an inner parallel construct makes worthwhile the exploitation of multiple levels of parallelism.

Implementation. The two WD proposals are implemented as arrays of pointers to work descriptors, behaving as circular lists (see Figure 24, A and B). There is a shared GWD structure and one per-processor LWD structure. Each processor searches for work first in its own LWD and then in the GWD. The size of the WD structures is a multiple of the secondary cache line size and they are aligned to cache line boundaries to avoid false sharing. Both implementations use one-way communication from the master processor to the slave processors. This means that the master processor writes pointers and the slaves read them. After the master processor writes a pointer to a WD location, it takes advantage of having exclusive access to the cache line to also clear a previously used location. This one-way communication mechanism saves several cache misses and invalidations while generating work, thus speeding up part of the critical path of the run-time library. Each processor has its own local index to the WD structures to extract work from them. It knows that there is no work in a WD structure when the location pointed by its index is NULL. The master processor uses a global index (WDP) to store a new pointer in the next available location of the WD structures. The global index is necessary to allow several processors to add work at the same time. Mutual exclusion through load-linked and store conditional instruction sequences is used to update this global index.

For example, in Figure 24A, four work descriptor pointers (shaded area) are currently stored in the GWD, possibly from different parts of the application. Not all processors have executed the same number of descriptors and some of them are extracting work from different locations. Each processor has a local index for work extraction.

Figure 24B presents the implementation of the LWD. It shows four LWD structures for four different processors. In the example, two groups of two processors are already spawned. Each master processor (0 and 2) uses an index associated to each LWD to insert new work. This index is accessed in mutual exclusion to allow several processors to add work to a LWD at the same time. Again, clearing an already used location during work insertion improves the cache behavior. Each processor waits for work in its LWD using a local index. Processors working in a group are consecutive and are identified 0 (the master), 1, 2, and so on.

7.1.10.2. Thread joining

Like many implementations, our proposal for thread joining is based on a distributed structure, to minimize false sharing. However, we add a local sequence number for each processor and a per-processor join values array (PJV) to support multiple levels of parallelism. The per-processor sequence number allows that each individual processor participates in a different number of parallel regions before collaborating again in the same group due to a change in the structure of the parallelism. This is different from the previous implementations, where the sequence number was identifying the currently executing parallel construct.

Figure 24C shows the proposed implementation. When a slave processor in a group terminates its work, it writes its per-processor join sequence number in the Distributed Join Structure (DJS). One cache line stores the sequence numbers of four processors. The master processor has a copy of each thread sequence value in its PJV. It waits for completion of the parallel work by looking at both ends of the DJS. When it detects that the value stored in DJS for one of the two current locations is greater than its local copy, it records the new value in PJV and proceeds to the next processor. Both the DJS and the PJV are allocated in the stack of the master processor in order to allow the existence of several master processors at the same time.

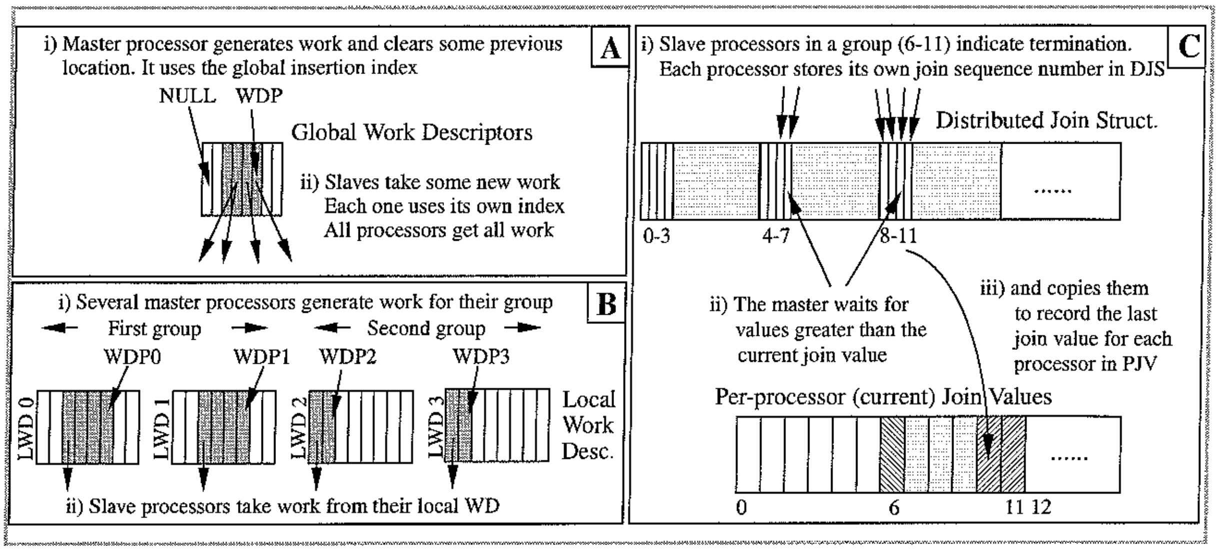

A
i) Master processor generates work and clears some previous location. It uses the global insertion index
NULL
WDP
Global Work Descriptors
ii) Slaves take some new work
Each one uses its own index
All processors get all work
B
i) Several master processors generate work for their group
First group
Second group
WDP0
WDP1
WDP2
WDP3
LWD 0
LWD 1
LWD 2
LWD 3
Local Work Desc.
ii) Slave processors take work from their local WD
C
i) Slave processors in a group (6-11) indicate termination.
Each processor stores its own join sequence number in DJS
Distributed Join Struct.
......
0-3
4-7
8-11
ii) The master waits for values greater than the current join value
iii) and copies them to record the last join value for each processor in PJV
Per-processor (current) Join Values
......
0
6
11 12

Figure 24: Thread fork / join data structures

7.2. Kernel-level implementation issues

We have implemented the kernel-level scheduling framework and scheduling policies described in Chapter 5 in the context of a user-level CPU Manager. This implementation has allowed us to evaluate our approaches in a machine like the Origin2000 system.

7.2.1. The user-level CPU Manager

The user-level CPU Manager partially implements the interface presented in Subsection 5.1, establishing the cooperation between the user-level execution environment provided by NthLib and the kernel level. The resulting structure of the execution environment including the CPU Manager is presented in Figure 25.

At initialization time, the CPU manager gets control over the processors of the machine on which it is running by creating one idle thread bound to each physical processor. It also creates the main scheduler thread, which is able to run on any physical processor. Its mission is to apply any of the user-selectable scheduling policies to the application workload.

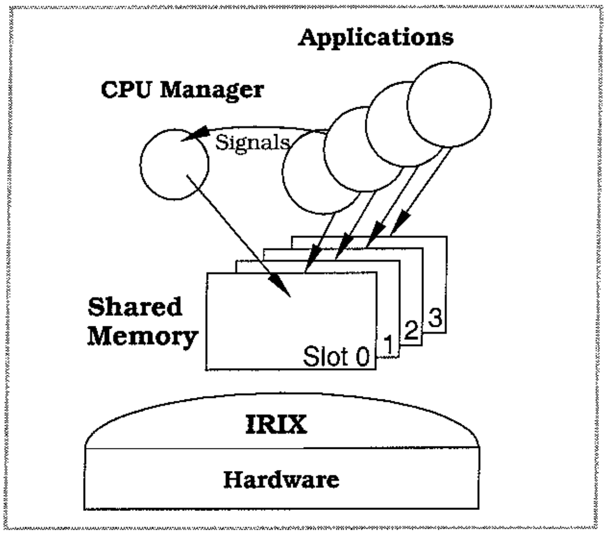


Figure 25: Implementation of the kernel-level scheduling framework on top of the IRIX operating system

The CPU manager establishes a memory area which is shared between the CPU manager and the applications. This area is used to implement, efficiently and with minimal overhead, the proposed kernel interface. As explained in Subsection 5.1.2, this area contains one slot for each application which runs under the control of the CPU Manager. The basic fields in each slot are:

- The number of requested processors (`n_cpus_request`).
- The number of processors currently allocated to the application (`n_cpus_current`).
- The number of processors requested by the CPU manager to be released as soon as possible by the application (`n_cpus_askedfor`)
- The number of processors stolen from the application (`n_cpus_stolen`).
- In addition, each slot contains an array of virtual processors. Virtual processors are numbered in this array and each entry contains the identifier of the kernel thread or process which corresponds to the virtual processor, the physical processor currently assigned to it, if any, and the status of the virtual processor, which can be: *RUNNING* in the application, *FREED* from the application, or *STOLEN* from the application.

Applications are launched by the CPU Manager. Application termination is communicated to the CPU Manager by means of UNIX signals (SIGCHLD). The primitives `cpus_release_self ()` and `cpus_processor_handoff ()` are implemented entirely in the context of NthLib, and communicate the modifications on the status of the application to the CPU Manager through the shared memory area.

Each new application, at initialization time, attaches to an empty slot in the shared memory area and informs the CPU Manager of its arrival using a signal. The signal wakes up the scheduler thread in the CPU Manager, which allocates free processors, if any, to the new application, according to the current scheduling policy. If there are no free processors, either the application is not started or a global scheduling is forced.

Throughout the lifetime of an application, events like processor requests, processor releases and processor stealings are communicated through shared-memory, by using the application slot. The CPU Manager maintains also a private area in memory, which is used for the bookkeeping needed by the scheduling policy. A time quantum is initially set for the CPU Manager. The scheduler thread of the CPU Manager wakes up at the expiration of every time quantum and distributes processors among the current applications applying the desired scheduling policy. In the current implementation, we use a common time quantum of 100 milliseconds between successive invocations of the CPU manager. Due to the difficulties of implementing a processor preemption mechanism after giving the application a grace time to return processors, we have set the grace time to zero. In that way, processors are preempted as soon as the scheduling policy decides the new allocation.

The operating system supplies physical processors with the creation of virtual processors. In the current implementation virtual processors are pre-created from the application. Creation of virtual processors is based on the basic primitives provided by operating systems to create kernel threads inside a shared address space. Examples of these primitives in current operating systems are the `sproc` call in IRIX [31] and the `thread_create` call in Mach [1] and Digital UNIX [42][41][76][77].

The CPU Manager controls the execution of the applications through specific IRIX system calls. The calls `blockproc(pid)` and `unblockproc(pid)` are used to start and stop the execution of individual processes belonging to the applications. These system calls simulate the events of allocating a new processor and stealing a processor, respectively. In other operating systems, a different interface should be used. In the Digital UNIX and Mach operating systems, for instance, the same system calls are `thread_suspend` and `thread_resume`.

The CPU Manager is aware of processor affinity. It uses other specific IRIX system calls to bind processes to physical processors. The system call `sysmp (MP_MUSTRUN_PID, cpu, pid)` is used to bind the process indicated by the `pid` argument to the physical processor number `cpu`. In Digital UNIX and Mach, this call should be replaced by `bind_thread_to_cpu`.

7.2.2. The complete virtual processors scheduling code

NthLib is tightly coupled with the operating system in order to adapt at user-level when any kernel-level scheduling event alters the user-level execution. NthLib uses the kernel interface in order to maintain as many user-level threads running as the number of physical processors assigned by the operating system.

The interaction with the operating system is done mainly by NthLib at every user-level scheduling point, when a nano-thread terminates execution and the package knows that the processor can be preempted safely.

When a nano-thread terminates execution, the NthLib internal function `nth_cleanup` is automatically invoked (QuickThreads manage such invocation). Inside `nth_cleanup` resides all operating system related stuff. The Example 7.1 sketches its functionality.

Example 7.1. User-level scheduling code

```
/*
   Main NthLib scheduling loop (nth_cleanup)
*/
void nth_cleanup (struct nth_desc * nth, int user_return)
{
   /* Satisfy dependences for the successor threads */
   i = 0;
   while (i<nth->nsucc) {
      struct nth_desc * succ = NTH_SUCC(nth, i);
      if (nth_depsatisfy (succ)
         nth_to_lrq (succ->vp_id, succ);
      ++i;
   }

   /* Idle thread starts here */
   while (1) {

      check_os_conditions ();

      /* get the next work-descriptor, if any, and execute it */
      wd = getwd ();
      if (wd!=NULL) {
         schedule (wd);
         continue;
      }

      /* get the next nano-thread from the ready queues and execute it */
      next = getwork ();
         if (next!=NULL)
            nth_dispatch (next);
   }
}

/*
   Auxiliary function to test the O.S. conditions at every scheduling point
*/
void check_os_conditions ()
{
   if (cpus_askedfor () > 0) {
      /* First check whether the operating system is requesting any processors
           to be returned. If so, the current processor is released */

      cpus_release_self ()
   }

   if (cpus_preempted_work () > 0) {
      /* Second, check whether there is any preempted work (kernel thread)
         If so, the current processor is used to recover it in order to
         quickly allow the application to continue execution */

      work = cpus_get_preempted_work ();
      cpus_processor_handoff (work);
   }
}
```

The first thing to do when a nano-thread terminates is to satisfy one dependence for each successor. At this point, the library knows that it has reached a *safe point* where the kernel thread has no useful context from the application point of view. The library takes profit from this situation and makes several tests to check whether the operating system is reclaiming processors to be returned (calling `cpus_askedfor`) or it has already preempted any application work (calling `cpus_preempted_work`). In the first case, the library simply releases the current processor (calling `cpus_release_self`) and updates the current number of processors the application is running on. In the second case, the library transfers the current processor to the preempted work, in order to recover it as soon as possible and let it continue executing application code (this is done calling `cpus_get_preempted_work` and `cpus_processor_handoff`). The number of processors currently assigned does not change. In either case, the current kernel thread is suspended.

Finally, when the operating system conditions have been checked, the `nth_cleanup` function searches for work in the application ready queue and executes it. When there is no work to be done, the operating system conditions are continuously checked in order to release first the processors that are not executing application work, if any.

7.2.3. Implementation of the kernel-level scheduling framework

As it has been stated in Chapter 5, the NANOS operating system level scheduling is application-oriented and takes applications as the scheduling target. In this subsection, we present the implementation of the main abstractions that are used by the operating system scheduling mechanisms to manage and distribute physical processors among the running applications.

7.2.3.1. Data structures

The kernel-level scheduling framework consists of two main data structures: the application slots shared with the applications and the kernel internal structures. They are presented in Figure 26. The left portion of the figure represents the shared memory area, which the kernel shares with all the applications. This area is divided in application slots, one slot for each application. In the right part of the figure, there are the kernel internal data structures consisting of the *work list* structure, the application descriptors and the process descriptors.

With respect the memory shared with the applications, from the kernel point of view, all application slots are readable and writable from any physical processor. The kernel reads data written from applications and writes data to be used by the applications. The kernel does not rely on data written by itself on this memory area, for security reasons. From the applications point of view, each application is able to read and write to its associated slot. The execution of parallel application relies on the information the kernel publishes in this area. The applications can not break the kernel by writing invalid data in this region.

The information available in this area includes, inside each application slot, all the data explained in Subsection 5.2. It is summarized in Table 8, presenting the application-wide information, and in Table 9, presenting the information related to each virtual processor. This information is available to both the kernel the running applications in order to cooperate in the kernel scheduling tasks.

An implementation could find useful to replace the global ready queues inside the kernel by ready queues associated to each application. In this way, each application owns a ready queue where processes/threads are enqueued when they become ready and dequeued for execution. In this way, it is easier for a processor to get work from its preferred application.

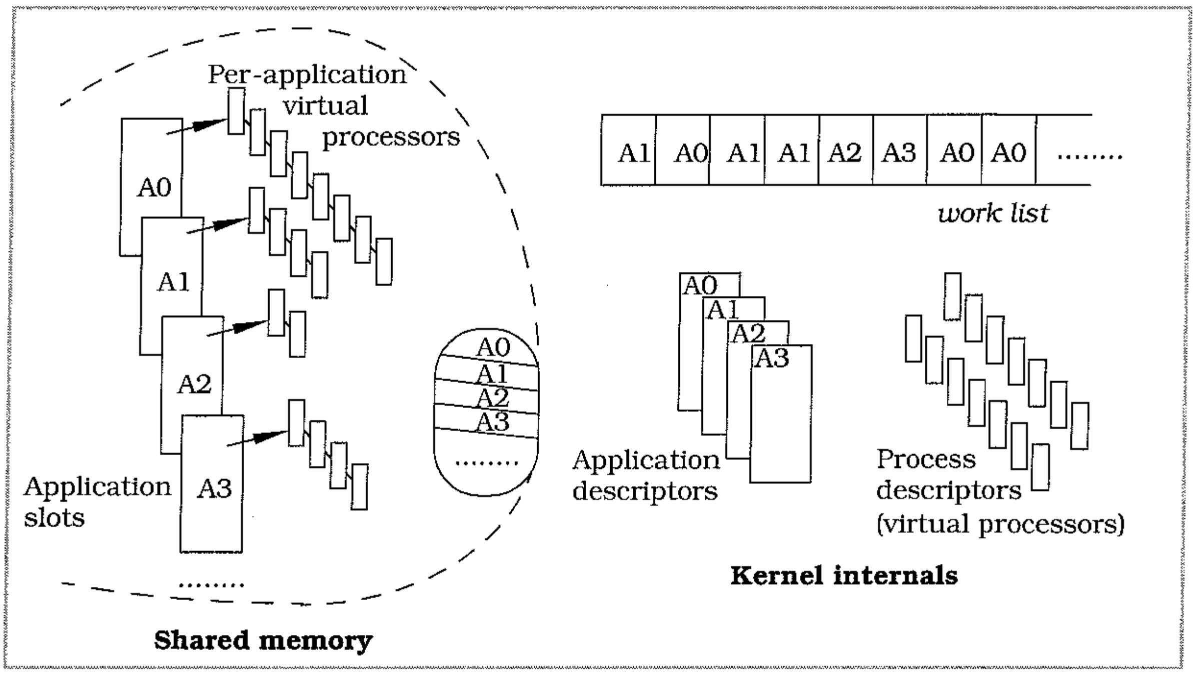


Figure 26: Data structures defined by the kernel scheduling framework

The right portion of Figure 26 represents the information needed inside the kernel (kernel internal structures), which consists of the *work list*, and the application and process descriptors. The *work list* is a list containing a reference to all the applications which are requesting more processors than those allocated to them. Usually, as many references to an application are inserted in this list as processors are needed by the application to complete its request. This list is used to maintain working all the processors that are released by other applications running in the system due to any reason (virtual processor blocking, lack of parallel work inside the application, etc.) Processors released go immediately to visit the *work list* to find a new application where they can go to execute a ready virtual processor.

Data contained in the array of application and process descriptors is simply the information needed by the kernel to manage with applications and processes. This information is mostly replicated in the shared memory area.

Name	Explanation
application_id	The application identifier
n_cpus_requested	The number of processors requested by the application at any given time
n_cpus_current	The number of processors currently allocated
n_cpus_blocked	The number of processors currently blocked
n_cpus_askedfor	The number of processors with preemption warning
n_cpus_preempted	The number of processors currently preempted
thread_info_lock	Lock to access the thread_info array
thread_info [NCPUS]	Array containing the status of each virtual processor (see Table 9)

Table 8: Information shared between the kernel and an application

Name	Explanation
vp_id	The virtual processor identifier
vp_status	The virtual processor status, indicating:
RUNNING	it is running in a processor
FREED	ready, it is available for running
BLOCKED	it is blocked in the kernel
PREEMPTED	it has been preempted and can be recovered
phys_cpu	The physical processor where the virtual processor is running
vp_context	The context of the virtual processor

Table 9: Virtual processor status information (thread_info)

The following subsection presents the main kernel-level algorithms that manage with these data structures.

7.2.3.2. Algorithms

Two main algorithms are involved in the kernel-level scheduling framework. The first one is the algorithm that applies the current scheduling policy to distribute processors among the running applications. The second one deals with processors released from applications and searching for work in the *work list*.

Applying a scheduling policy. In this framework, physical processors are first assigned to an application, and then they choose a virtual processor belonging to that application for execution. Processor assignment to an application is done by the current scheduling policy. The current policy is applied every a certain amount of time (the time quantum), or when a processor needs to apply it to find work to perform. We have selected a time quantum of 100 milliseconds, which is commonly used for scheduling in various operating systems.

Figure 27 presents the resulting algorithm. It shows the main data structures involved and a physical processor while executing the algorithm. The physical processor 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 is going to receive each application for the next time quantum (step `ii)`). The allocation results are communicated to the applications (step `iii)`). To do that, the new numbers about processors allocated (`n_cpus_current`) and processors asked for the kernel to be released (`n_cpus_askedfor`) are stored in the shared memory area for each application. From that point, the application knows that after a grace time, the processors will be stolen by the kernel. Finally, the *work list* structure is used to state which applications are going to receive new processors in the next time quantum (step `iv)`). Two kind of references to applications are stored. First, the kernel stores references to the applications that it has already decided to assign processors. Next, it stores references to applications that *could* receive more processors than those assigned when any processor is released from other applications. For each different new processor, a different reference is added.

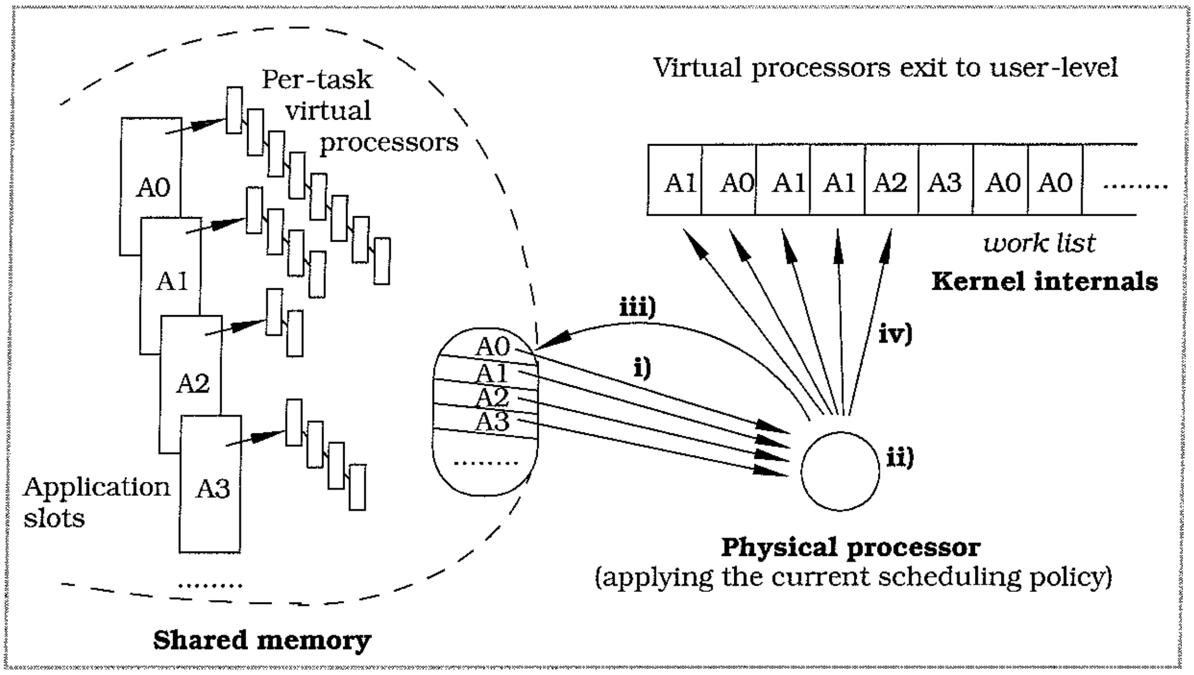

Per-task virtual processors
A0
A1
A2
A3
Application slots
....
Shared memory
A0
A1
A2
A3
....
Virtual processors exit to user-level
A1 A0 A1 A1 A2 A3 A0 A0 ....
work list
Kernel internals
iii)
i)
ii)
iv)
Physical processor
(applying the current scheduling policy)

Figure 27: Algorithm for applying the kernel-level scheduling policies

After applying the previous algorithm, some applications are going to loose processors either because the application detects at user-level the `n_cpus_askedfor` request and calls `cpus_release_self()` or when each physical processor enters the kernel, it detects that it should not be allocated to the current application and the grace time for releasing it has expired. Reaching this point, the processor is ready to apply the algorithm to search for new work to perform. The algorithm is described next.

Searching for work in the *work list*. Any free (idle) processor at kernel-level, or any processor that has detected that should change its allocation to another application, executes the algorithm to search for new work in the *work list*, presented in Figure 28. It first visits the *work list* to find a reference to an executing application which has not all its processor requests satisfied (step `i)`, in the figure). Then, it informs the applications involved in the change of the assignment. If the processor was idle, it has to inform the new application that it is going to start working for it. If the processor was allocated to another application it marks its previous virtual processor as stolen and saves the context in the shared memory area before assigning to the new application. In the figure, this is represented in step `ii)`, which assumes that the processor was assigned to application A0 and it is now assigned to application A1, which corresponds to the first entry in the *work list*. Finally, the processor searches for a released or stolen virtual processor in the new application and exits to user-level, to participate in the application work (step `iii)`, in the figure).

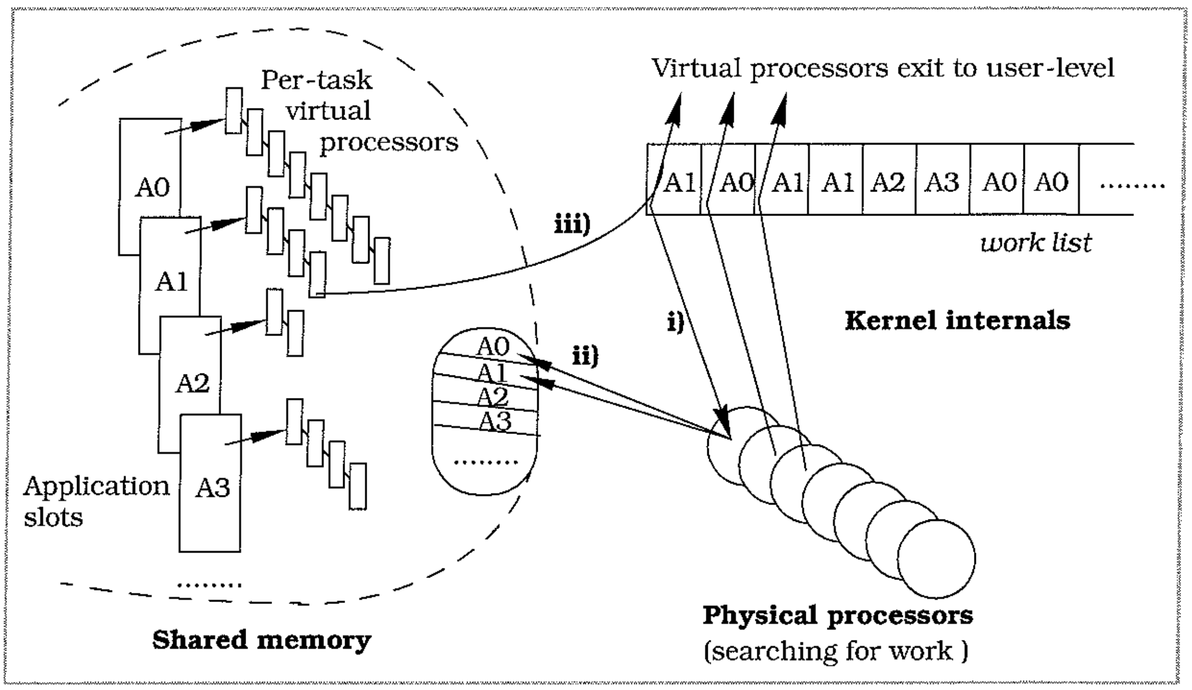

Per-task virtual processors
A0
A1
A2
A3
Application slots
....
Shared memory
iii)
ii)
i)
A0
A1
A2
A3
....
Virtual processors exit to user-level
A1 A0 A1 A1 A2 A3 A0 A0 ....
work list
Kernel internals
Physical processors (searching for work)

Figure 28: Algorithm for searching for new work in other applications

This scheduling framework has been implemented inside a user-level CPU Manager, sharing memory with the running applications for the evaluation of the complete execution environment in the Origin2000 machine.

Chapter 8. Programming Examples

Abstract

In this chapter, we present three interesting parallelizations that we have achieved in this work. In the first place, it shows how the LU decomposition benchmark can be parallelized using nano-thread bursts. The second example presents the annotated version of a portion of the HYDRO2D application, exploiting multiple levels of parallelism, and the resulting parallel code. Finally, the last example shows the structure of the NAS BT application and explains two different approaches for parallelize it.

Frank Poole: "... There's a very good piece of advise I've often found useful: 'Never attribute to malevolence what is merely due to incompetence.' ..."

"3001, The Final Odyssey", Arthur C. Clarke, Ballantine Books, New York, March 1997.

8.1. Hand coded benchmarks: LU decomposition

Along the development of this thesis, a number of hand-coded benchmarks have been useful to help in the design and test of the NthLib threads library. Among them, we present in more detail the LU benchmark because it is a good example of use of the application level scheduling techniques based on nano-threads bursts.

The LU benchmark computes the LU decomposition of a matrix. The Example 8.1-(1) presents the sequential and parallel codes. The LU sequential code consists of a main sequential loop (*k* loop, in the example), which iterates over the columns of the matrix. Inside, there are two parallel loops. The first one iterates over the elements of the current columns, dividing them by the element in the diagonal. This is a small cost loop with respect to the second one. The second loop is double nested and it iterates over the elements remaining between *k+1* and *N*, both in rows and columns.

Example 8.1-(1). LU decomposition

Sequential code.

```
double A[N][N];

void lu ()
{
 for (k=0; k<N; k++) {
    for (i=k+1; i<N; i++) {
       A[i][k] /=  A[k][k];
    }
    for (i=k+1; i<N; i++) {
       for (j=k+1; j<N; j++) {
          A[i][j] -= A[i][k]*A[k][j];
       }
    }
 }
}
```

Parallel code. (continued on Example 8.1-(2))

```
double A[N][N];

void lu ()
{
 int k;

 for (k=0; k<N; k++) {
   parallel_lu_loop_1 (k, N, A);
   parallel_lu_loop_2 (k, N, A);
 }
}
```

The parallel version has been hand-coded following the conventions decided during the design phase of NthLib. Both loops have been replaced by a call to a function which spawns the parallelism (*parallel_lu_loop_1* and *parallel_lu_loop_2*). Both functions receive the same arguments: The current iteration (k), the size of the matrix and a reference to the matrix. These functions are the same, except that each one creates nano-threads on functions implementing different loop bodies (*lu_loop_1* and *lu_loop_2*, respectively). Example 8.1-(2) also shows the function *parallel_lu_loop_2* (see next page). This function creates nano-thread bursts (with `nthf_burst_create`, `nthf_block` and `nthf_burst_wait`, in lines 8, 27 and 43) as are shown in Figure 14a (see Chapter 3). Observe that after creating the nano-thread controlling the barrier synchronization, the loop in line 9 iterates over nano-thread bursts. In each iteration, the scheduler nano-thread checks the number of processors allocated to the application (line 10). This avoids to generate work on processors preempted by the operating system. It also allows to use newly allocated processors for the actual burst, adapting to the system conditions. The conditional in line 18 checks whether enough iterations remain to spawn a complete burst. If so, the scheduler generates the first part of the current burst, creating as many nano-threads as processors and supplying them for execution (lines 20-22). Next, it creates the dispatcher nano-thread (line 23) and the second part of the burst (lines 24-26). Then it blocks, waiting for the execution of the dispatcher nano-thread. When this nano-thread has been executed, this means that at least one processor has terminated its work and the next burst is generated. Lines 29 to 41 generate a partial burst for the last loop iterations. Along the loop execution all nano-threads created receive the nano-thread controlling the barrier synchronization as their successor, and as such, its number of precedences is incremented accordingly (lines 19 and 34).

Lines 46 to 57 contain the function *lu_loop_2*, encapsulating the loop body. This function is executed by all nano-threads. It receives as arguments the iterations they have to execute (*min* and *max*), the current iteration *k*, the dimension of the matrix and a reference to the matrix.

8.2. Applications

From the six applications used for evaluation in this thesis, there are two (SPEC HYDRO2D and NAS BT) that benefit from exploiting multiple levels of parallelism. We show now their internal structure and how they have been parallelized [85].

8.2.1. SPEC 95 HYDRO2D

The HYDRO2D application solves the hydrodynamical Navier Stokes equations to compute galactical jets. It offers two levels of parallelism worth to be exploited. Several subroutines can take advantage of being split in parallel sections, at an outer level. At an inner level, each one of the parallel sections usually contain parallel loops sometimes encapsulated in other subroutines. As an example, Figure 29 shows the structure of subroutine ADVNCE (in the center of the figure) and related subroutines. This set of subroutines is called repeatedly during the execution of HYDRO2D for each timestep. As can be observed, subroutine ADVNCE starts with a parallel loop (in BBF) and then three parallel sections can be spawned, each one containing different subroutine calls (CORIF, STAGF1 and STAGF2). CORIF contains a parallel loop. STAGF1 and STAGF2 are showed in the right side of Figure 29. Each one contains four parallel sections containing parallel loops. Subroutine ADVNCE continues calling subroutines TRANS1/TRANS2. Their internal structure is presented in the left side of the figure. Finally, subroutine ADVNCE calls four times the subroutine FCT, each time working with different data, so the calls are independent. Inside FCT, loop level parallelism is exploited again. Only the execution of the subroutine FCT accounts for the 61% of the total execution time of HYDRO2D application. This is of importance because we are taking advantage of multiple levels of parallelism in a significant portion of the application and we are obtaining good results (see Subsection 9.3.4, where the HYDRO2D application is evaluated).

Example 8.1-(2). LU decomposition (continued)

```
void parallel_lu_loop_2 (int k, int n, double A[N][N])
{
 int vp;
 int i = k+1;
 struct nth_desc * nth_end_chunk;
 struct nth_desc * nth;

 nth_end_chunk = nth_burst_create (1);
 while (i<n) {
    int lkthreads = nth_cpus_current ();
    unsigned int first = lkthreads;
    unsigned int last = lkthreads;
    unsigned int dim = (n-(k+1)) / (lkthreads*4);
    int diml;
    if (dim==0) dim = 1;
    diml = n - dim*(first+last);

    if (i<diml) {
       nth_depadd (nth_end_chunk, first+last);
       for (vp=0; vp<first; vp++, i+=dim) {
          nth = nth_create_1s (lu_loop_2, 0, vp, nth_end_chunk, 5, i, i+dim, k, n, A);
       }
       nth_dispatcher_create (nth_self ());
       for (vp=0; vp<last; vp++, i+=dim) {
          nth = nth_create_1s (lu_loop_2, 0, vp, nth_end_chunk, 5, i, i+dim, k, n, A);
       }
       nth_block ();
    }
    else {
       diml = n-i;
       first = diml / dim;
       diml %= dim;
       if (diml>0 && first==0) first = 1, dim = 0;
       nth_depadd (nth_end_chunk, first);
       nth = nth_create_1s (lu_loop_2, 0, 0, nth_end_chunk, 5, i, i+dim+diml, k, n, A);
       i += dim+diml;
       for (vp=1; vp<first; vp++, i+=dim) {
          nth = nth_create_1s (lu_loop_2, 0, vp%lkthreads, nth_end_chunk,
                                5, i, i+dim, k, n, A);
       }
    }
 }
 nth_burst_wait (nth_end_chunk);
}

void lu_loop_2 (int min, int max, int k, int n, double A[N][N])
{
 int i = min;
 while (i<max) {
    int j = k+1;
    while (j<n) {
       A[i][j] -= A[i][k] * A[k][j];
       ++j;
    }
    ++i;
 }
}
```

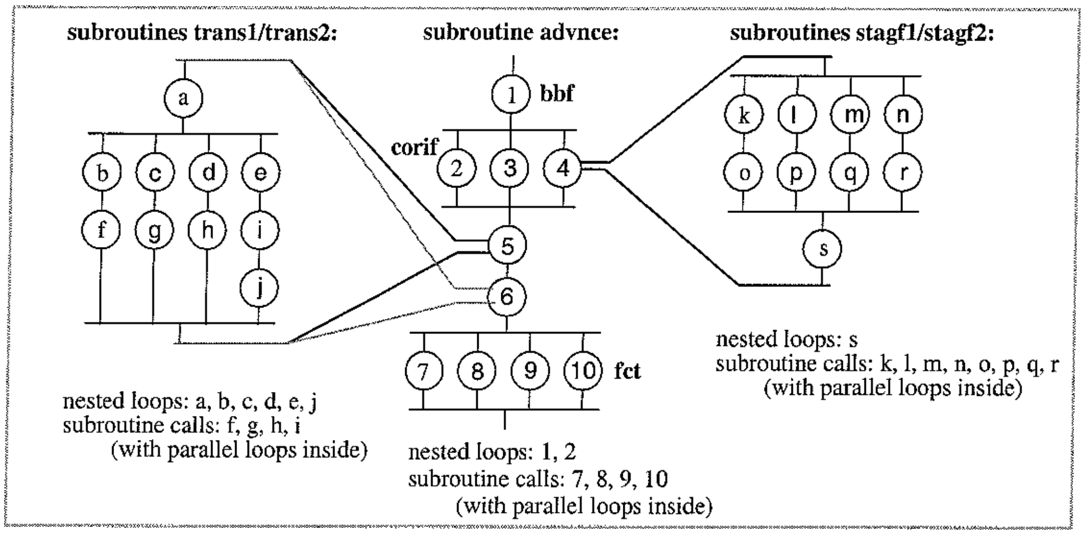

subroutines trans1/trans2:
subroutine advnce:
subroutines stagf1/stagf2:
a
b
c
d
e
f
g
h
i
j
1
bbf
corif
2
3
4
5
6
7
8
9
10
fct
k
l
m
n
o
p
q
r
s
nested loops: a, b, c, d, e, j
subroutine calls: f, g, h, i
(with parallel loops inside)
nested loops: 1, 2
subroutine calls: 7, 8, 9, 10
(with parallel loops inside)
nested loops: s
subroutine calls: k, l, m, n, o, p, q, r
(with parallel loops inside)

Figure 29: Structure of subroutine ADVNCE and opportunities for parallelization

An important property of the structure of the subroutine ADVNCE is that along the different parallel sections, the same data is accessed. For instance, nodes labeled k and o (in STAGF1), nodes b and f (from TRANS1) and node 7 (in ADVNCE), all work on matrices named RON and RO. Nodes 1, p, c, g and 8 work on matrices ENN and EN, and so on. This means that driving a processor to work on these nodes will benefit data locality. Besides, this is an opportunity to distribute processors to execute different sections, thus enlarging the working set of each processor in each parallel section where it participates. The result is that subroutine ADVNCE can establish four groups of processors, each group executing along one of the parallel sections.

The execution environment proposed by OpenMP talks about *teams* of processors, as a concept related to orphan DO / SECTIONS directives. When a processor reaches an orphan directive, either a previously PARALLEL directive has defined the team of processors to work in parallel or the parallelism has not been spawned and, then, the team consists of only one processor. We generalize this idea to allow that inside a parallel region more parallelism could be exploited. The Example 8.2-(1) shows the original code of a portion of the subroutines ADVNCE and FCT. Subroutine ADVNCE calls four times to subroutine FCT with different data to work. Subroutine FCT contains a series of parallel loops, like the one shown in the figure.

Processor groups or *teams* are established using a few extensions to OpenMP. The annotated application resulting after directive insertion is shown in the Example 8.2-(2). The number of processors is got from the operating system interface using the `nthf_cpus_current` primitive (line 24). The number of processors got here will be used along all the parallel section to generate parallelism and in the work sharing constructs to distribute work. The threads library is aware of ensuring execution of at least *ncpus* virtual processors during the execution of the parallel section. The number of independent parallel sections is computed in line 25 as the minimum value between the number of available processors and the maximum number of sections (four, in this case). After that, the parallelism is spawned. Each parallel section is mapped onto a specific processor using the ONTO clause on the SECTION directives. The expression supplied evaluates to the identifier of the virtual processor where the section should be executed. In case 8 processors are available (ncpus=8), the sections are started in virtual processors 0, 2, 4 and 6. The interface of the subroutine FCT has been modified to receive the group configuration in order to correctly spawn the inner level of parallelism. For instance, from the first section point of view, only two processors are available and they are numbered 0 and 1. The second section will use processors 2 and 3, and so on.

Example 8.2-(1). HYDRO2D original code

```
      SUBROUTINE ADVNCE
      ...
      CALL FCT   ( RON , RON , RO )
      CALL FCT   ( ENN , ENN , EN )
      CALL FCT   ( GZN , GZN , GZ )
      CALL FCT   ( GRN , GRN , GR )
      ...
      END

      SUBROUTINE FCT ( UNEW, UTRA, UOLD )
      ...
      DO 100  J = 1,NQ
      DO 100  I = 1,MQ1
         EK =  DT / DZ(I)
         DK =  DC( EK , VZ1(I,J) )
         AK =  AC( EK , VZ1(I,J) )
         DZ1(I,J) =  ( DK + VC1(I,J) )  *  ( UOLD(I+1,J) - UOLD(I,J) )
         AZ1(I,J) =    AK               *  ( UTRA(I+1,J) - UTRA(I,J) )
  100 CONTINUE
         ...
      END
```

With respect the subroutine FCT, shown in Example 8.2-(2), the parallel loop is specified using a PARALLEL DO directive, with WDSTATIC scheduling for high performance of the inner-level of parallelism, PRIVATE variables EK, DK and AK and the CPUS and RELATIVE extended clauses are used. The CPUS clause specifies an expression indicating that NCPUS/NSECT processors should be used in this parallel loop. Assuming eight processors were available in ADVNCE, NCPUS/NSECT=8/4 = 2 processors should be used in this loop. The next question is *which* processors should be used. Using the RELATIVE clause, the programmer specifies that the processor reaching the parallel loop is the processor with the lowest identifier to participate in its execution. This processor becomes the master of the group. It will execute the loop as processor 0 and NCPUS/NSECT-1 processors with higher identifier, numbered consecutively will help in the execution as slave processors. In the running example, assuming 8 available processors, the loop inside the first section (FCT (RON...)) is executed using processors 0 and 1, which is what was previously planned. Not shown here, there is the possibility of specifying that the work supply should be done in the virtual processors identified by their absolute number. This would be done using the ABSOLUTE clause, instead of RELATIVE.

For completion, the Example 8.2-(3) shows the parallel code generated by the NANOS compiler. Lines 59 to 69 in subroutine ADVNCE are the code executed by the scheduler nano-thread to create the four parallel sections and supply them to the appropriate processors. After creation, the scheduler nano-thread blocks, waiting for the termination of all sections, and the virtual processor goes to execute the first parallel section, along with the slave processors. Each nano-thread created executes a subroutine containing a call to the subroutine FCT, passing the appropriate arguments for each section (lines 73 to 76). Inside the subroutine FCT, the code for spawning the second (inner-most) level of parallelism uses work-descriptors for performance (lines 80 to 89), supplied individually to build the groups of processors. Notice here how the RELATIVE clause causes the generation of the code in lines 80-81 to determine in which virtual processor the code is running. The virtual processor identifier obtained is set as the master of the new parallel region and it is used as the starting point to supply the new parallelism.

Example 8.2-(2). HYDRO2D annotated code (extended OpenMP)

```
      SUBROUTINE ADVNCE
      ...
      NCPUS = nthf_cpus_current ()
      NSECT = min(4, NCPUS)
C$OMP PARALLEL
C$OMP SECTIONS
C$OMP SECTION ONTO(mod(0,NSECT)*(NCPUS/NSECT))
      CALL FCT   ( RON , RON , RO , NCPUS, NSECT)
C$OMP SECTION ONTO(mod(1,NSECT)*(NCPUS/NSECT))
      CALL FCT   ( ENN , ENN , EN , NCPUS, NSECT)
C$OMP SECTION ONTO(mod(2,NSECT)*(NCPUS/NSECT))
      CALL FCT   ( GZN , GZN , GZ , NCPUS, NSECT)
C$OMP SECTION ONTO(mod(3,NSECT)*(NCPUS/NSECT))
      CALL FCT   ( GRN , GRN , GR , NCPUS, NSECT)
C$OMP END SECTIONS
C$OMP END PARALLEL
      ...
      END

      SUBROUTINE FCT ( UNEW, UTRA, UOLD , NCPUS, NSECT)
      ...
C$OMP PARALLEL DO SCHEDULE(WDSTATIC) PRIVATE(EK,DK,AK)
C$OMP& CPUS(NCPUS/NSECT) RELATIVE
      DO 100  J = 1,NQ
      DO 100  I = 1,MQ1
         EK =  DT / DZ(I)
         DK =  DC( EK , VZ1(I,J) )
         AK =  AC( EK , VZ1(I,J) )
         DZ1(I,J) =  ( DK + VC1(I,J) )  *  ( UOLD(I+1,J) - UOLD(I,J) )
         AZ1(I,J) =    AK               *  ( UTRA(I+1,J) - UTRA(I,J) )
  100 CONTINUE
      ...
      END
```

The inner-most level of parallelism starts executing subroutine fct_loop_024 (line 93), which contains the body of the parallel loop. This subroutine receives as arguments, among other, the identifier of the virtual processor where it is running on (nth_me_024), the number of processors participating in the parallel loop (nth_nprocs_024) and the identifier of the master processor for the current parallelism (nth_firstcpu_024). With these arguments, the current processor obtains the portion of the loop that should execute (lines 96 to 104). Then, it enters the execution of the loop body (lines 105 to 113). Other arguments passed to this function are variables that the compiler has found in the loop body and that are neither declared as PARAMETER nor residing in COMMON blocks.

The evaluation of the execution of the HYDRO2D application is shown in Chapter 9, where the results of the multi-level parallelization are compared with those of the single-level version.

Example 8.2-(3). HYDRO2D parallel code (calling NthLib)

```
      SUBROUTINE advnce
      ...
      ncpus = nthf_cpus_current()
      nsect = min(4,ncpus)
      CALL nthf_depadd(nthf_self(),05)
      nth_mask = 0
      nth = nthf_create_1s(advnce_s_031,0,mod(3,nsect) * (ncpus / nsect),
                           nthf_self(),nth_mask,2,ncpus,nsect)
      nth = nthf_create_1s(advnce_s_030,0,mod(2,nsect) * (ncpus / nsect),
                           nthf_self(),nth_mask,2,ncpus,nsect)
      nth = nthf_create_1s(advnce_s_029,0,mod(1,nsect) * (ncpus / nsect),
                           nthf_self(),nth_mask,2,ncpus,nsect)
      nth = nthf_create_1s(advnce_s_028,0,mod(0,nsect) * (ncpus / nsect),
                           nthf_self(),nth_mask,2,ncpus,nsect)
      CALL nthf_block()
      ...
      END

      SUBROUTINE advnce_s_028(ncpus,nsect)
      ...
      CALL fct(ron,ron,ro,ncpus,nsect)
      END

      SUBROUTINE fct(unew,utra,uold,ncpus,nsect)
      ...
      nth_selfv_024 = nthf_self()
      nth_cpuv_024 = nthf_cpu(nth_selfv_024)
      nth_nprocs_024 = ncpus / nsect
      CALL nthf_wdcreate(nth_wdesc_024,fct_loop_024,nth_selfv_024,07,nth_nprocs_024,
                         nth_cpuv_024,az1,utra,uold,vc1,dz1)
      CALL nthf_depadd(nth_selfv_024,nth_nprocs_024 + 1)
      DO nth_p_024 = nth_cpuv_024,nth_nprocs_024 - 1 + nth_cpuv_024
         CALL nthf_wdsupply(nth_p_024,nth_wdesc_024)
      END DO
      CALL nthf_endsupply(nth_selfv_024)
      ...
      END

      SUBROUTINE fct_loop_024(nth_me_024,nth_nprocs_024,
                              nth_firstcpu_024,az1,utra,uold,vc1,dz1)
      ...
      nth_lme_024 = nth_me_024 - nth_firstcpu_024
      nth_bottom_024 = 1
      nth_top_024 = nq
      nth_niter_024 = nth_top_024 - nth_bottom_024 + 1
      nth_rest_024 = mod(nth_niter_024,nth_nprocs_024)
      nth_chunk_024 = nth_niter_024 / nth_nprocs_024
      nth_down_024 = min(nth_lme_024,nth_rest_024)+nth_bottom_024+nth_chunk_024*nth_lme_024
      nth_balance_024 = nth_lme_024 .LT. nth_rest_024
      nth_up_024 = nth_down_024 + nth_chunk_024 + nth_balance_024 - 1
      DO j = nth_down_024,nth_up_024
         DO 100 i = 1,mq1
            ek = dt / dz(i)
            dk = dc(ek,vz1(i,j))
            ak = ac(ek,vz1(i,j))
            dz1(i,j) = (dk + vc1(i,j)) * (uold(i + 1,j) - uold(i,j))
            az1(i,j) = ak * (utra(i + 1,j) - utra(i,j))
 100  CONTINUE
      END DO
      END
```

8.2.2. NAS BT

The NAS BT benchmark [10] solves three sets of uncoupled block tridiagonal systems of equations, first in the *x*, then in the *y* and finally in the *z* direction. Each block contains 5x5 elements. These systems arise in many CFD applications.

Figure 30 shows the structure of the application. An iterative loop sequentially calls to routines *compute_rhs*, *x_solve*, *y_solve* and *z_solve*. The dependences in these routines determine which loops can be parallelized. For instance, *x_solve* carries the dependence in the first dimension being the loops that traverse the second and third dimension completely parallel; similarly, *y_solve* carries the dependence in the second dimension and *z_solve* in the third dimension. A possible strategy would be to parallelize the loop that traverses the third dimension in routines *x_solve* and *y_solve* and parallelize the loop that traverses the second dimension in routine *z_solve*. Although this parallelization strategy implies totally parallel loops, it suffers from the data movement overhead (transposition) that occurs when going from *y_solve* to *z_solve* and back again.

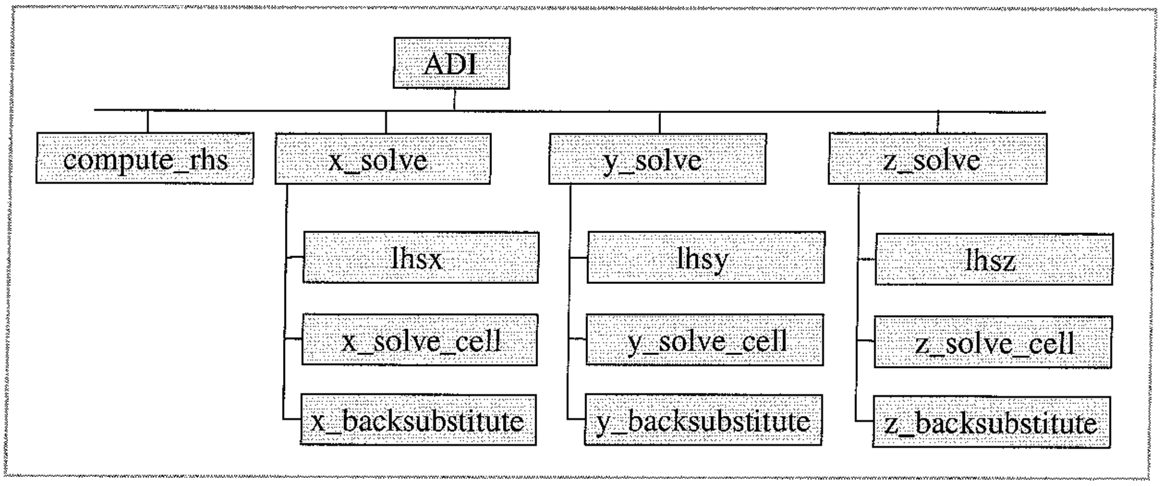


Figure 30: Internal structure of the NAS BT application

The data movement overhead of the transposition can be avoided if the third dimension is also parallelized in *z_solve*; this requires the use of the OpenMP ORDERED clause and directive that forces the sequential execution of the distributed loop iterations. In order to allow a pipelined execution of the ORDERED dimension, loop blocking is applied. In this way, a chunk of iterations in processor *p+1* is executed when the same chunk of iterations finishes its execution in processor *p*. Figure 31 shows the data distribution among processors and the resulting execution model for the one-dimensional parallelization in the *z_solve* routine. Although this introduces the overhead of blocking and synchronization, the overlap of different chunks in different processors results in an improved performance.

When the number of iterations is small to fed a large number of processors, two dimensions are worth to be parallelized. In order to avoid data movement, our strategy parallelizes the second and third dimension in all the routines. This implies that two dimensions are executed in parallel in *x_solve*, but one of the two dimensions parallelized are executed in an ORDERED way in both the *y_solve* and *z_solve* routines. Figure 32 shows the data distribution performed among processors and the resulting execution model for the multidimensional parallelization in the *z_solve* routine.

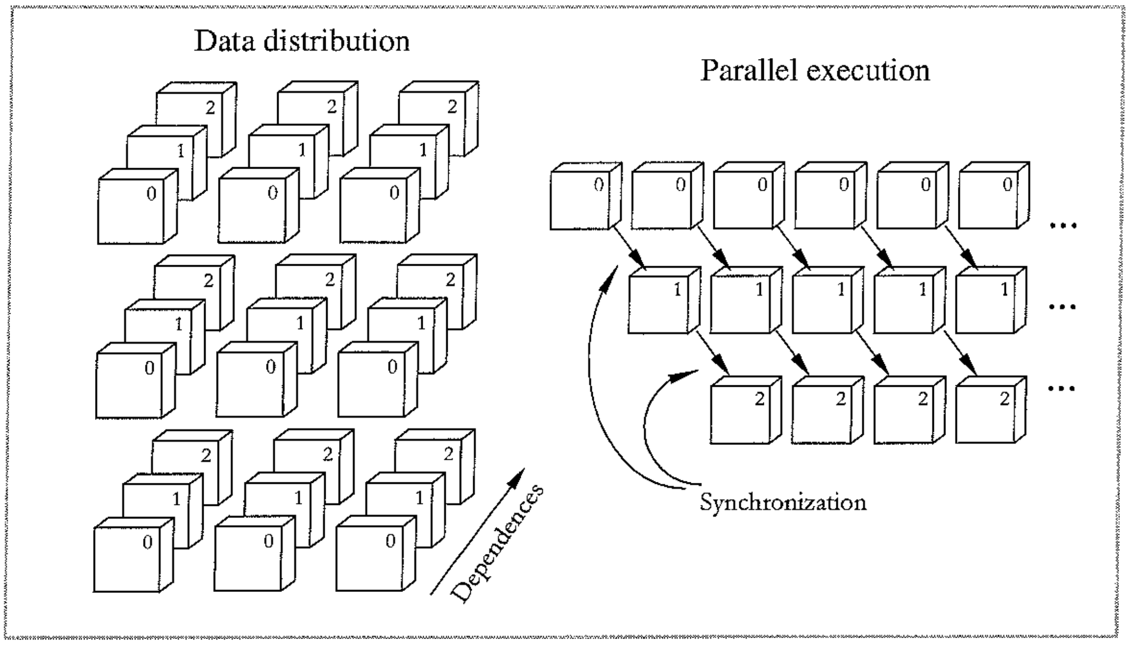


Figure 31: One-dimensional data distribution and pipelined ORDERED execution

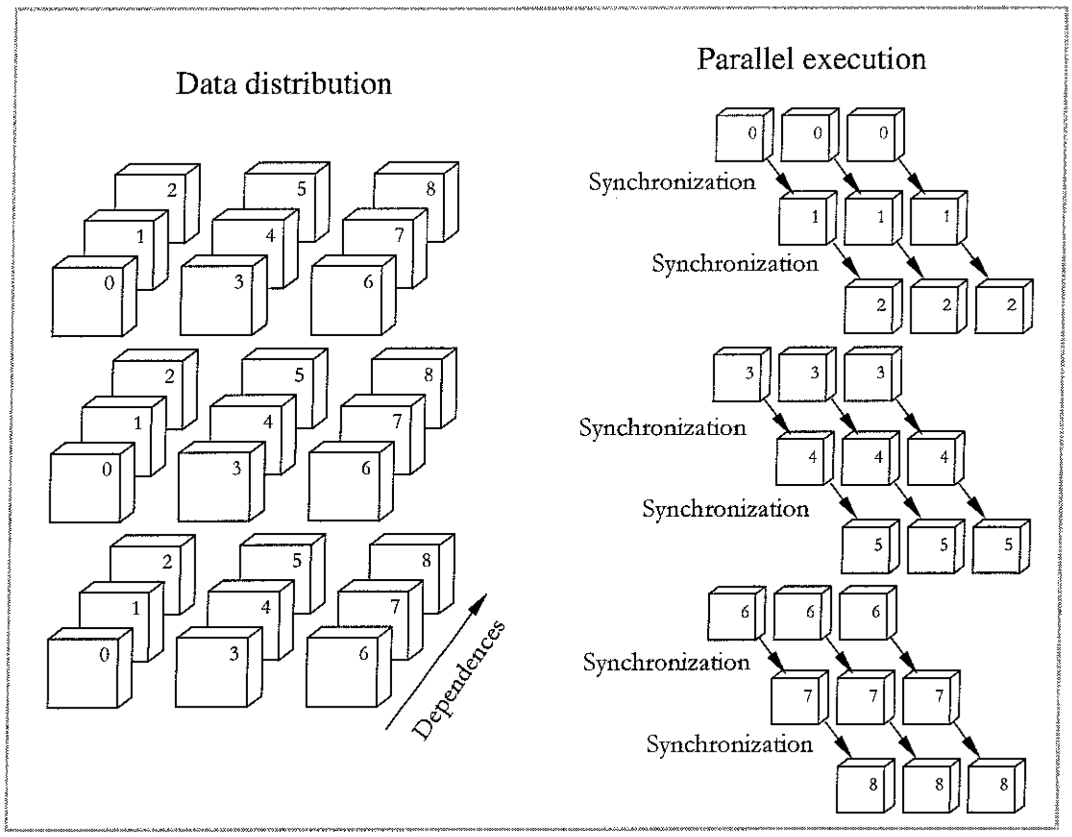


Figure 32: Two-dimensional data distribution and pipelined ORDERED execution

The evaluation of the execution of the NAS BT application is shown in Chapter 9, where the results of the multi-level parallelization are compared with those of the single-level version.

Chapter 9. Evaluation

Abstract

This chapter presents the performance evaluation of the NANOS parallel execution environment. The performance evaluation is carried on a Silicon Graphics Origin2000 computer. Along the evaluation, the main contributions of this thesis are validated. The results are compared with the performance of the native parallel environment shipped with the Origin machine.

Molins de Rei - Platja de Gavà (by bike)

	Dist. (km)	Avg. (km/h)	Accum. time (h:mm:ss)
Molins de Rei	0		
Torrelletes	12.92	19.7	0:39:15
Begues	16.02	16.4	0:58:31
Gavà, platja	33.45	19.4	1:43:16
Gavà	39.22	-	-
Pic de Sant Climent	46.47	-	-
Sant Boi	47.76	19.7	2:25:22
Molins de Rei	58.07	20.0	2:54:08

9.1. Performance evaluation environment

The design and implementation of the NANOS parallel execution environment has been carried on a Silicon Graphics Origin2000 machine [139][72]. The machine is owned by the European Center for Parallelism of Barcelona (CEPBA [46]). It has sixty-four R10000 MIPS processors [61] 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. The CC-NUMA architecture of the machine is explained in the Subsection 1.2.3.1.

The Origin2000 computer is running the Silicon Graphics IRIX 6.5 operating system, shipped by Silicon Graphics. The operating system provides low level tools for analyzing the performance of parallel applications, including support for collecting the performance counters of the R10000 processors or tracing the system calls invoked by the processes. Silicon Graphics provides also tools to parallelize applications and run them in parallel.

All the benchmarks and applications used for evaluation in this chapter have been compiled to run on both the SGI MP and NANOS environments. We have used the native MIPSpro F77 to parallelize the programs to run in the SGI MP environment. Input source programs were previously annotated with standard OpenMP directives. The SGI MP environment has been described in Chapter 6 (Subsection 6.1.9).

We have used the NANOS compiler to generate parallel code to run on the NANOS execution environment. The input source files to the NANOS compiler have been annotated with the extended OpenMP directives proposed in this work. The NANOS compiler generates an intermediate Fortran file, containing calls to NthLib. This intermediate file is then compiled through the native MIPSpro F77 compiler and linked to NthLib.

Compilation of all the benchmarks and applications in both environments has been done using the same command line options for the native MIPSpro F77 compiler: -64 -mips4 -r10000 -Ofast=ip27 -LNO:prefetch_ahead=1.

The following sections present the results of the performance evaluation and comparison of both environments. The evaluation is done through the following types of experiments:

- Evaluation of the overhead introduced by the user-level execution environment on the parallel applications.
- Evaluation of the performance of individual applications on a dedicated machine.
- Evaluation of workload performance.

9.2. Evaluation of the user-level execution environment overhead

During the development of the nano-threads library, we have evaluated the overhead of thread management and we have found the major hardware/software issues that usually hurt performance.

The key is that it is not enough to ensure that the thread primitives are efficient on their own. Thread management can be done in a very simple way and the specific thread routines can be individually very efficient. But the important question is how they perform during a parallel execution, when they are interacting with each other.

It is more important to reduce the primitives interaction than to try to reduce the number of instructions that any primitive contains. For example, using a counter in shared memory seems the best way to implement thread joining due to the simplicity of the code. But this is usually not the case, specially in NUMA machines. We are going to show that implementing thread creation/work supply and thread joining with more complex structures, but taking into account the way the memory accesses are performed, we can achieve better performance than with simpler data structures.

For this reason, in this subsection, we present the evaluation of the individual library services, and also the evaluation of the performance of the primitives executed in a parallel environment using a synthetic benchmark (*overhead*), which is specially designed to measure the fork/join performance.

9.2.1. NthLib primitives overhead

An initial evaluation of the overhead introduced by the NthLib primitives was done in a previous implementation in the Intel architecture. Results are presented in [88]. The overhead introduced by NthLib in the MIPS architecture is presented in this subsection. The overhead is measured as the execution time taken by the most common NthLib primitives and code sequences. All the measurements done in this subsection have been obtained using the memory-mapped free-running hardware counter provided by the Origin2000 hardware [31]. The clock resolution is as precise as 800 nanoseconds.

Table 10 presents the execution time (in microseconds) taken by four of the most-used NthLib primitives. Two of the four primitives are related to nano-thread management (`nthf_create_1s` and `nthf_depadd`) and the other two are related to work descriptors (`nthf_wdcreate` and `nthf_wdsupply`). Nano-thread and work descriptor creation is done with four user arguments. The primitives are evaluated running inside a micro-benchmark, in which we have introduced the probes needed to obtain the execution time of the primitives. The micro-benchmark is used to measure the execution time of the different primitives, each time it spawns parallelism. It has been run on 1 to 8 processors to demonstrate the effect of having several processors running at the same time, and interacting with each other, on the performance of the primitives. For each experiment, the table presents the minimum and maximum execution times, obtained in different executions.

In Table 10, we can observe that the execution time of the `nthf_create_1s` primitive increases when the number of processors running in the benchmark is greater. This is because when supplying work to one processor, all data fits in the cache memory of the processor and the execution is efficient. Instead, when nano-threads are created to be supplied to other processors, the cache lines initialized at creation time are later accessed from another processor to execute the thread. And, when the nano-thread structure is reused to create another nano-thread, the same cache lines (residing now in the cache of the remote processor) are reclaimed again from the processor creating the nano-thread. This ping-pong effect causes the increment in the execution time. Depending on how far is the remote processor, the primitive takes more or less time, ranging from 4.8 to 10.4 microseconds when running the experiment in 8 processors. The minimum time correspond to executions where the two processors involved in the nano-thread creation are close. Maximum times correspond to a larger distance between the processors involved. The execution time of this primitive can be taken as a reference of the `nthf_burst_create` and `nthf_dispatcher_create` primitives, which consist of a single `nthf_create_1s`.

The same happens to the `nthf_depadd` primitive. When running on one processor, the primitive is very efficient initializing the precedence counter of the calling nano-thread. But when several processors have previously accessed the counter for writing, they leave the modified value in a remote cache and the primitive spends more time waiting for the coherence mechanism in the Origin2000 to get the value from there. The execution time reaches a top limit at 4 microseconds.

Compare the previous results with the next two primitives. The primitive `nthf_wdcreate` only packs its arguments into a work descriptor. Initializing the descriptor and copying four arguments takes a maximum of 1.6 microseconds. And the primitive `nthf_wdsupply` only stores a pointer to a descriptor in a work descriptor array of pointers after computing the next free location. Although its execution time depends on the number of processors, it is at least three times faster than `nthf_create_1s` under all conditions.

NthLib primitives	Processors	Execution time (in us.)
nthf_create_1s (4 user arguments)	1	1.6 - 2.4
	2	4.8 - 8.8
	4	4.8 - 9.6
	8	4.8 -10.4
nthf_depadd	1	< 0.8
	2	2.4 - 3.2
	4	3.2 - 4.0
	8	3.2 - 4.0
nthf_wdcreate (4 user arguments)	1	< 0.8
	2	< 0.8
	4	0.8 - 1.6
	8	0.8 - 1.6
nthf_wdsupply	1	< 0.8
	2	0.8 - 1.6
	4	1.6 - 2.4
	8	2.4 - 3.2

Table 10: NthLib primitives evaluation

The execution time of the two most common code sequences for supply work to processors are presented in Table 11. In the first place, the calling sequence for spawning parallelism using nano-threads consists of a call to `nthf_cpus_current` to determine the number of processors currently allocated, a call to `nthf_depadd` to initialize the precedence counter and as many calls to `nthf_create_1s` as needed to supply work to the given number of processors. Execution times quickly grow to a hundred microseconds when running on 8 processors. This sequence is shown in the Example 4.5 (lines 11 to 31), presented in Chapter 4.

On the other hand, the calling sequence for spawning parallelism using work descriptors consists of the same calls to `nthf_cpus_current`, and `nthf_depadd`, plus a call to `nthf_wdcreate` and as many calls to `nthf_wdsupply` as the number of processors assigned. This code sequence is shown in the Example 4.7 (lines 10 to 18), presented in Chapter 4. Observe that, again, the execution times of the work descriptor sequence is three times faster than the nano-threads one. The important difference here is that while when using nano-threads a new local address space (stack) is allocated and initialized, when using work descriptors the work is efficiently represented by a work descriptor and supplied to the processors without the need of nano-thread creation.

NthLib common creation sequences	Processors	Execution time (us.)
Spawning parallelism using nano-threads	1	2.4 - 4.8
	2	18.4 - 24.0
	4	38.4 - 58.4
	8	74.4 - 111.2
Spawning parallelism using work descriptors	1	1.6 - 2.4
	2	7.2 - 9.6
	4	12.0 - 14.4
	8	28.8 - 32.2

Table 11: Evaluation of common creation sequences

9.2.2. NthLib fork/join overhead

In order to evaluate the overhead of the fork/join mechanisms used in NthLib, we have implemented a specific benchmark which stresses the thread creation and joining primitives. Using this test, we have measured all the thread creation techniques (nano-threads and work descriptors) and the joining mechanisms (shared counter and the joining distributed structure). Experiments presented here include the evaluation of one and two levels of parallelism.

9.2.2.1. Evaluation of the one-level parallelism overhead

Parallel applications are usually parallelized using one-level of parallelism. For such situations, NthLib allows to use both work descriptors and nano-threads. The motivation is that, on one side, for each fine-grain work sharing constructs you should use work descriptors in order to attain good performance. On the other side, nano-threads allow to use advanced scheduling techniques at user-level. In cases where the amount of work is enough, such scheduling techniques can improve performance.

The overhead benchmark (see Figure 33) consists of a main loop which performs 1000 iterations. Each iteration of the main loop spawns and joins parallelism on the desired processors, from 1 to 64. The work supplied to each processor (subroutine *cost*) is an unoptimized empty loop doing a number of iterations. As much as twelve experiments have been done, varying the cost of the inner-most loop, from 1024 to a million and a half iterations. From them, we have selected six representatives, labeled #0, #2, #4, #6, #8 and #9. For each experiment, Table 12 presents the cost of the work which would be performed by each processor in absence of overhead, measured both in microseconds and number of iterations per processor, from a real fine-grain parallelism to a more coarse grain parallelism. For instance, when running on 4 processors, the benchmark #2 spends 24.8 microseconds to perform 4096 iterations of the inner loop (1024 iterations per processor). Some of the numbers of iterations to perform have been selected to be divisible by 24 and 48 processors.

```
      PROGRAM overhead
      integer iter, i
      integer N,M
      integer cost

      READ (*, 10) N             ! 1000
      READ (*, 10) M             ! 64, 192
      READ (*, 10) cost          ! 16 - 2048
10    FORMAT (I)

      do iter = 1, N
C$OMP PARALLEL DO SCHEDULE(WDSTATIC) LOCAL(I) SHARED(cost
         do i = 1, M
            call work (cost)
         enddo
      enddo
      END
```

Figure 33: Source code of the *overhead* benchmark

Microseconds/ Iter. per proc.	**Experiment number/ Overhead (number of empty iterations)**					
Processors	**#0/ 1024**	**#2/ 4096**	**#4/ 12288**	**#6/ 49152**	**#8/ 196608**	**#9/ 393216**
1	24.8/ 1024	99.2/ 4096	297/ 12288	1188/ 49152	4755/ 196608	9510/ 393216
4	6.20/ 256	24.8/ 1024	74.5/ 3072	297/ 12288	1188/ 49152	2377/ 98304
8	3.10/ 128	12.4/ 512	37.2/ 1536	149/ 6144	594/ 24576	1188/ 49152
16	1.55/ 64	6.20/ 256	18.6/ 768	74.5/ 3072	297/ 12288	594/ 24576
32	0.77/ 32	3.10/ 128	9.30/ 384	37.2/ 1536	149/ 6144	297/ 12288
64	0.39/ 16	1.55/ 64	4.65/ 192	18.6/ 768	74.5/ 3072	149/ 6144

Table 12: Microseconds and iterations per processor spent by each processor in the *overhead* benchmark

Figures 34 - 36 show the performance of the overhead benchmark in each one of the experiments, including the overhead introduced by the parallel execution environment. Each one of the figures is associated with a column in Table 12. The figures present the execution time taken by the benchmark when executed sequentially and using from 1 to 64 processors. Six different versions of the benchmark are presented in each figure. SEQx is the sequential version in each case, given as a reference. MP-FOPx and MP-SHMx are SGI-MP versions. In the MP versions, work supply is done always through a global descriptor. Thread joining, instead, can be implemented using atomic operations in memory (FOP's) or a shared memory distributed joining structure (SHM). The NANOS environment is evaluated through the three ways of generating work implemented. NNTH-GWDx and NNTH-LWDx are the versions using local-supplied and global-supplied work descriptors in the NANOS environment. LWD and GWD use a distributed joining structure for joining threads. NNTH-NTHx is the version using nano-threads and ready queues. NTH joins threads through the shared dependence counter.

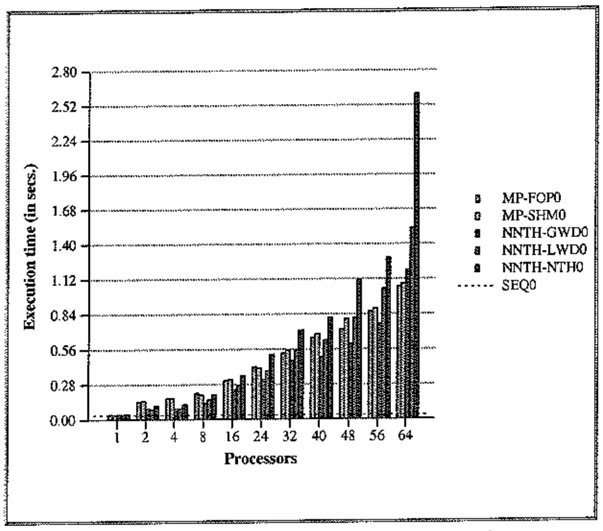


Figure 34A: Overhead execution time (in secs.) - 25 us., 1024 iterations

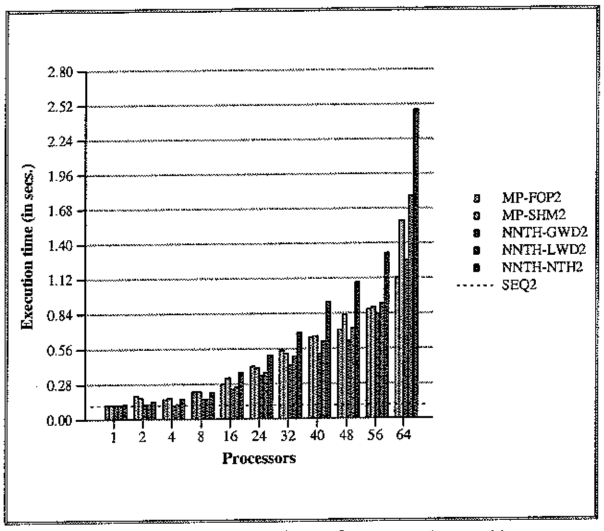


Figure 34B: Overhead execution time (in secs.) - 100 us., 4096 iterations

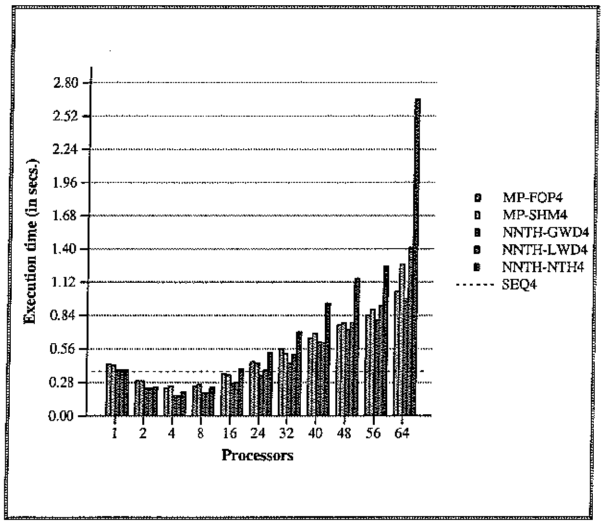
The experiment shown in Figure 34A distributes 1024 iterations among the participating processors. This experiment is really fine-grained. Only 16 iterations are given to each processor when running on 64 processors. The sequential execution time (SEQ0) is very small, as much as 0.036 seconds. All parallel implementations running on 1 processor show some overhead. In the case of the SGI-MP library, MP-FOP0 and MP-SHM perform slightly worse in one processor (they take 0.040 seconds) because of the extra code of checking the number of processors available and deciding to proceed sequentially. The efficient NNTH implementations (LWD0 and GWD0) take around 0.040 seconds also, spawning parallelism on one processor. The NTH0 experiment spends more time (0.043 seconds) due to the creation of the nano-threads.

The different implementations evolve in a similar way when more processors are used. Nearly all the execution time shown in the plot is due to the overhead of the run-time execution environment. For this reason, the graph does not show any speedup in the execution time. The plot shows that the overhead increases linearly with the number of processors. The NTH implementation is the exception when using 64 processors. Usually, the SGI-MP implementations suffer from more overhead when running in less processors (2-8), probably because the SGI-MP library is tuned for working with a large number of processors. From 32 processors and above, the FOP implementation beats the SHM one. This agrees with the goal of the FOP implementation which is to ensure a good synchronization behavior when running in a large number of processors.

The NNTH implementations have different behavior when the number of processors increases. The GWD technique performs usually as well as the MP versions. The LWD technique shows slightly more overhead than the previous commented implementations. The direct cause for this behavior is the way the work is provided to the processors, one at a time, instead of through a global work descriptor. Supplying the work in this way enlarges the critical path of the application each time the master processor needs to generate work. Even with this drawback, LWD behaves comparable to the MP library. Both the GWD and LWD techniques are validated in this way. The NTH implementation, based on nano-threads, shows higher overhead, when using 16 processors and more, due to the conflicts motivated by the implementation based on ready queues and the joining counter. The overhead of nano-threads is twice the overhead of any of the other implementations, when running in 64 processors. This means that nano-threads, as actually implemented, are not supporting efficiently such fine-grain parallelism.

Figure 34B shows the performance when the total number of iterations is multiplied by four (reaching 4096). In this case, all techniques show that the overhead when running in 2 and 4 processors is large enough to compensate the benefits of having additional processors. As a result, a slight loss of performance is observed. This means that when the work supplied to a processor is below 25 microseconds (or 1024 empty iterations), it is not possible for the run-time execution environment to hide the time to spawn parallelism.

When using 8 and more processors, the overhead exceeds the benefits of spawning parallelism and the execution time starts rising. GWD2 and LWD2 are also comparable to the MP implementations. At 16 processors and more, the NTH implementation presents more overhead than others. At 64 processors, the FOP and GWD implementations are comparable and also are the SHM and LWD techniques. Only NTH shows again an overhead nearly doubling that of its counterparts. Observe that the NNTH-NTH0 bar at 64 processors is taller than the NNTH-NTH2 bar. This indicates that most of the execution time is due to the management of parallelism. This fact is possible because of the noticeable standard deviation of this experiments.



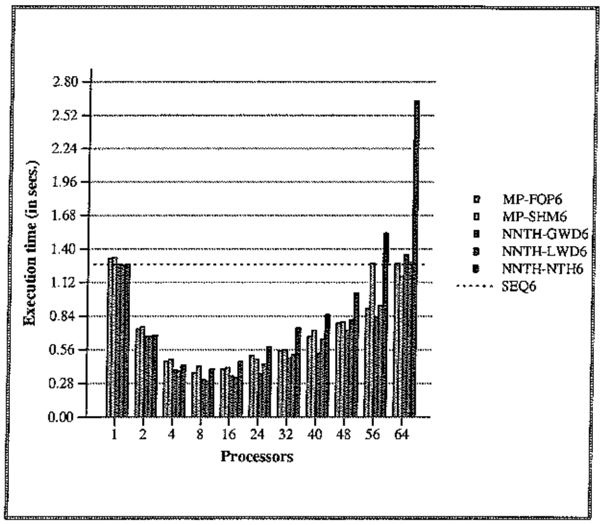


Figure 35A: Overhead execution time (in secs.) - 0.3 ms., 12288 iterations

Figure 35B: Overhead execution time (in secs.) - 1.2 ms., 49152 iterations

Figures 35A-B show the performance of the overhead benchmark when distributing 12288 and 49152 iterations among the participating processors. The work spawned when running in one processor is 0.3 and 1.2 milliseconds, respectively. Typical applications have parallel constructs of this size. This graphs show that this work sizes can be efficiently executed in no more than 8 processors. Executing in 16 processors will provide nearly the same improvement than executing with 8, although the number of processors is doubled. Observe that, in these experiments, the performance of NTH4 and NTH6 is already comparable to that of GWDx, FOPx and SHMx till the point where no more improvement is achieved (between 8 and 16 processors). Showing a behavior comparable to the SGI-MP environment till this point using nano-threads is interesting because it allows to exploit the same amount of parallelism that the SGI-MP environment, but providing (and being able to exploit) more functionality.

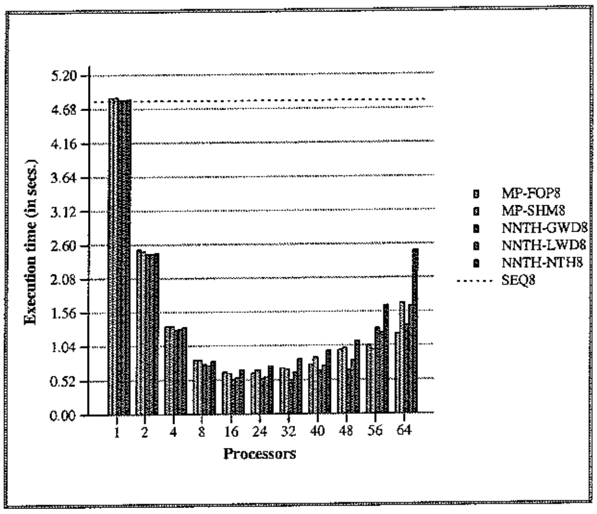


Figure 36A: Overhead execution time (in secs.) - 4.7 ms., 196608 iterations

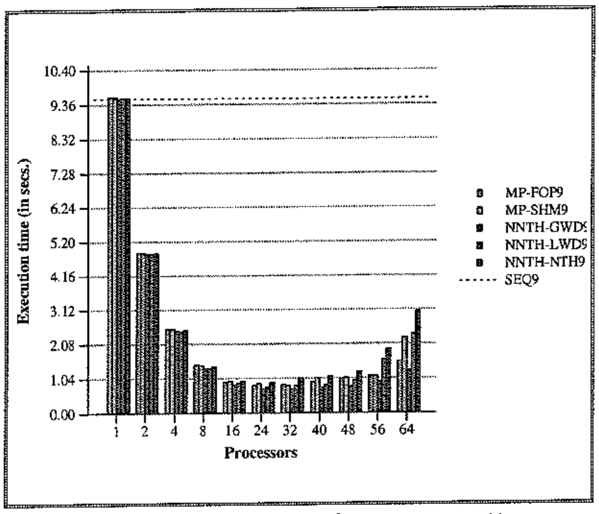


Figure 36B: Overhead execution time (in secs.) - 9.5 ms., 393216 iterations

Figures 36A-B are examples of coarse grain applications, in which the work size ranges from 4 to 10 milliseconds, when executed on 1 processor. Nevertheless, the parallelization becomes fine grain when distributed across 32 or 64 processors (becoming as small as cents of microseconds). Again, the comparison of the different techniques shows that they behave similarly. As a result, applications containing parallel tasks on the order of milliseconds can exploit parallelism efficiently using 16-24 processors. Beyond that number, we think that this is the point where applications can exploit multiple levels of parallelism. The nano-threads implementation can be used to spawn the higher levels of parallelism, while the work descriptor approaches will be used for the inner level.

To complete the study of the different techniques we present also a plot in Figure 37, showing the cache behavior in the set of experiments running on 32 processors.

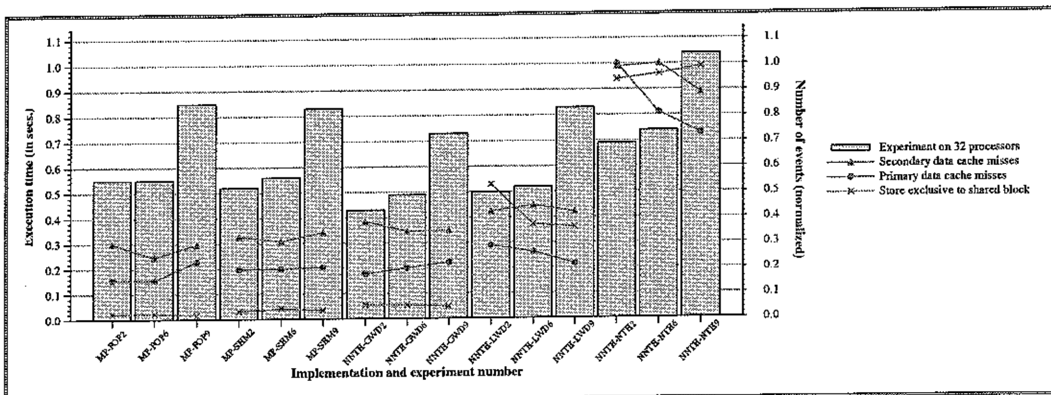


Figure 37: Cache misses, external interventions and invalidations related to execution time in the overhead benchmark running on 32 processors

Figure 37 compares, with respect the cache behavior, the five techniques for work generation (FOP, SHM, GWD, LWD and NTH). Each bar represents the execution time of the benchmark running on 32 processors, running the experiment indicated in the associated label. For instance, NNTH-LWD2 corresponds to the experiment #2 (which performs a total of 4096 iterations) using the LWD technique. Along with the execution times of the benchmark, the plot also presents the normalized numbers of primary and secondary data cache misses and the normalized number of store operations requiring exclusive access to a shared cache line. These events have been collected using the R10000 hardware event counters, through the *perfex* analysis tool, provided by SGI.

Observe that for the FOP, SHM and GWD techniques, the events presented behave the same. For the LWD technique, the number of stores that reclaim exclusive access to a cache line increases. This is because of the way the work is supplied to processors, one to one. The store event is caused each time the master processor generates work, getting exclusive access to the cache line where it stores the pointer to the work descriptor. The cache line is, then, read by the destination slave processor, which requests shared access to the line. Thus, one cache line per processor exchanges twice its status each time work is generated. Finally, for the nano-threads implementation (NTH) all the events increase. This is due to the extra functionality of allocating a nano-thread for each processor, initialize and supply them. The important thing here is that the number of events is under control, so that nano-threads can be effectively used for the outer levels of parallelism.

```
      PROGRAM overhead2levels
      integer iter, i
      integer N,M
      integer cost

      READ (*, 10) N        ! 1000
      READ (*, 10) M        ! 64, 192
      READ (*, 10) cost     ! 16 - 2048
10    FORMAT (I)

      do iter = 1, N

         ncpus = nthf_cpus_actual ()
         nsect = min (4, ncpus)

C$OMP PARALLEL SECTIONS
C$OMP SECTION ONTO (mod(0,nsect)*(ncpus/nsect))
C$OMP PARALLEL DO LOCAL(I) SHARED(cost)
C$OMP& SCHEDULE(WDSTATIC)
C$OMP& CPUS(ncpus/nsect) RELATIVE
         do i = 1, M/nsect
            call work (cost)
         enddo

C$OMP SECTION ONTO (mod(1,nsect)*(ncpus/nsect))
C$OMP PARALLEL DO LOCAL(I) SHARED(cost)
C$OMP& SCHEDULE(WDSTATIC)
C$OMP& CPUS(ncpus/nsect) RELATIVE
         do i = 1, M/nsect
            call work (cost)
         enddo

C$OMP SECTION ONTO (mod(2,nsect)*(ncpus/nsect))
C$OMP PARALLEL DO LOCAL(I) SHARED(cost)
C$OMP& SCHEDULE(WDSTATIC)
C$OMP& CPUS(ncpus/nsect) RELATIVE
         do i = 1, M/nsect
            call work (cost)
         enddo

C$OMP SECTION ONTO (mod(3,nsect)*(ncpus/nsect))
C$OMP PARALLEL DO LOCAL(I) SHARED(cost)
C$OMP& SCHEDULE(WDSTATIC)
C$OMP& CPUS(ncpus/nsect) RELATIVE
         do i = 1, M/nsect
            call work (cost)
         enddo

      enddo
      END
```

Figure 38: Source code of the *overhead2levels* benchmark

9.2.2.2. Evaluation of the two-level of parallelism overhead

NthLib is going to be used by applications exploiting multiple levels of parallelism. For this reason we also want to measure the overhead introduced by the library when the application spawns multiple levels of parallelism. From our experience, the most common situation will be to exploit two levels of parallelism: A first level consisting of parallel sections and a second, inner level, consisting of parallel loops. This is the case we have modelled with the overhead2levels benchmark, whose code is shown in Figure 38. The code presented consists of 4 parallel sections containing parallel loops, a common structure found in real applications (HYDRO2D, TURB3D, see Chapter 8). The amount of work done by the two-levels overhead benchmark is the same that the work done by the one-level version (Subsection 9.2.2.1). For this reason, we can compare the overhead introduced by NthLib in the two versions.

Figures 39A and B show a comparison between the LWD- and NTH-based single-level versions of the overhead benchmark against the following versions of the overhead2levels benchmark: NNTH-2L4x corresponds to the code presented in Figure 38, where 4 parallel sections are spawned at the outer level. NNTH-2L16x and NNTH-2L32x corresponds, respectively, to experiments spawning 16 and 32 parallel sections at the outer level. Executions are done using from 1 to 64 processors and the execution time is presented. The overhead of the two-level parallelizations depends on the number of processors used and the amount of work supplied. In Figure 34A, for a small number of processors (1-8) and a limited number of parallel sections (experiment NNTH-2L4-2), the overhead is comparable with the single-level versions. Instead, the overhead of spawning the outer level on 16 or 32 processors is noticeable (experiments NNTH-2L16-2 and NNTH-2L32-2). This is because those experiments create 16 or 32 nano-threads for the parallel sections and they have to be executed on the limited number of processors available. With that overhead, the execution time of the two-levels benchmark more than doubles the execution time of the single-level benchmark. When the number of processors increase, the overhead is not so noticeable. This means that for applications with fine-grain two-level parallelism, the performance on a small number of processors will be worse than when using a single-level, but they could take advantage of the multiple levels when using a larger number of processors.

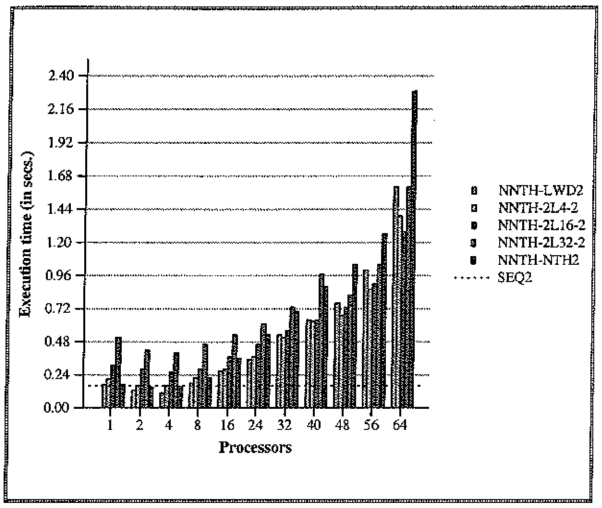


Figure 39A: Overhead execution time (in secs.) using two levels of parallelism (100 us., 4096 iterations)

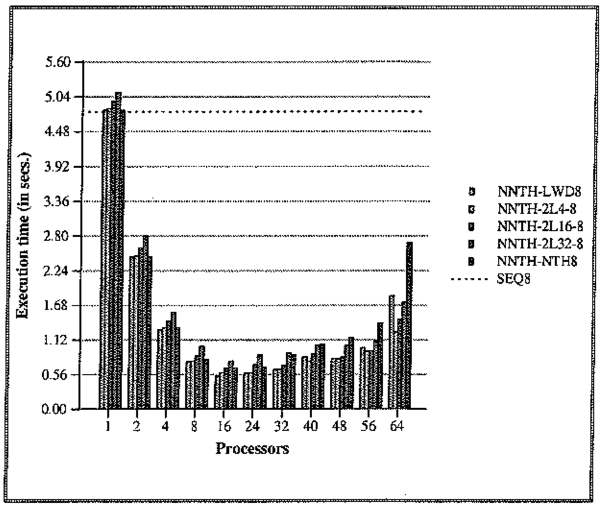


Figure 39B: Overhead execution time (in secs.) using two levels of parallelism (4.7 ms., 196608 iterations)

Figure 39B shows that, when the amount of work increases, the overhead on a small number of processors does not reach 5%. On the other hand, when the number of processors is large (8 and above), the overhead of the two-level parallelization is less than the parallelization using nano-threads (NNTH-NTH8). This is a good result because it shows that the approach of having an efficient way of work supply based on work descriptors is useful for the two-levels parallelization.

9.2.3. Conclusions of the overhead evaluation

From the previous evaluation of the overhead of the NthLib, we conclude that the nano-threads primitives are well tuned for the underlying architecture and the mechanisms for thread spawning and joining are at least as efficient as the mechanisms used inside the SGI MP library.

Nano-thread creation and enqueuing is the most costly operation, its cost being, in general, below 10 microseconds. This experiment has been run in a 64 processor machine where the average latency for a remote memory access is around 800 ns. This means that the number of remote memory accesses in a nano-thread creation is very limited. Probably, this point requires further investigation to determine whether the actual number of remote accesses can be reduced.

The work descriptors primitives, on the other hand, are really more efficient than the nano-threads primitives. This is because the functionality they are providing is limited compared with the nano-threads functionality. The simplicity of the work descriptors primitives reduces the number of remote accesses and obtains a performance which is at least three times better than nano-threads.

With respect the fork/join experiments, they show that the overhead introduced by the NANOS environment is, most of times, comparable to the overhead introduced by the SGI MP environment. The fork/join experiments put under stress the mechanisms designed and implemented in this work. The results show that, even providing an extra functionality, as is the support of multiple levels of parallelism and processor grouping, the mechanisms perform comparable to the SGI MP ones.

A more specific experiment to evaluate the impact of these mechanisms on the memory performance show also that the extra functionality is the cause of a slight increment on the memory traffic.

After that, the next section, which shows the evaluation of the performance of individual applications running in the NANOS environment, will clarify whether the mechanisms proposed are useful for real parallel execution.

9.3. Evaluation of individual application performance

Following, we present the benchmarks and applications we have used to validate the design and implementation of our user-level proposals (Subsections 9.3.1 and 9.3.2). Two benchmarks and six applications are used to compare the user-level environment when running in a dedicated Origin2000 machine (Subsections 9.3.3 and 9.3.4).

9.3.1. Benchmarks

Two synthetic benchmarks are used to evaluate some of the application-level scheduling algorithms presented in Subsection 3.2.2. Such algorithms are implemented using nano-threads.

Synthetic Jacobi Iteration. The Jacobi Iteration benchmark resolves a linear system of equations. It uses a matrix of 2000 rows by 2000 columns and spends 400 iterations.

Synthetic LU matrix decomposition. This benchmark computes the LU decomposition of a matrix. It has been presented in Subsection 8.1. The matrix size is set to 2000 rows by 2000 columns.

9.3.2. Applications

The set of applications used to validate the approach presented in this thesis are from the SPEC FP 95 benchmarks [122] (five applications) and the NAS Parallel Benchmarks [10] (one application).

SPEC 95 SWIM application. The SWIM application solves the system of shallow water equations using finite difference approximations on a N1xN2 grid (usually 512x512). SWIM offers loop level parallelism within a set of procedures (CALC1, CALC2 and CALC3), that are executed repeatedly. The execution of these routines accounts for the 99% of the total execution time. The application is useful to evaluate the overhead introduced by the user-level threads library and the parallelization process done by the compiler.

SPEC 95 TOMCATV application. The TOMCATV benchmark is a mesh generation program. It uses a mesh of 512x512. The execution of TOMCATV has two phases: First, the application reads a large file during a few seconds; And then, it enters the computation phase. TOMCATV consists of a single function, containing the input statements to read the input file and a sequential loop with several coarse-grained parallel loops inside, to perform the computation. It offers a single level of loop parallelism. This application is useful to evaluate the overhead introduced by the user-level threads library and the parallelization process done by the compiler.

LTOMCATV application. The LTOMCATV application is a SPEC95 TOMCATV application enclosed inside an outer sequential loop. This allows us to have an application requesting a different number of processors during its execution. LTOMCATV spends ten iterations of the TOMCATV application. It requests one processor each time it starts reading the input file. Then, it requests the number of processors indicated by the user to execute the computation phase. This application is used for the workload evaluation in Subsection 9.4.

SPEC 95 TURB3D application. The TURB3D program is used for simulating an isotropic, homogeneous turbulence in a cube with periodic boundary conditions in a 3-dimensional space. It solves the Navier-Stokes equations using a pseudo-spectral method. Leapfrog-Crank-Nicolson scheme is used for time stepping. The application offers several levels of parallelism. At an outer level, it offers section-level parallelism that computes the same function (FFT) over different arrays. At the inner level, the execution of the subroutine FFT offers two levels of parallelism, both consisting of parallel loops. Which one of the two inner levels can be exploited is a matter of how much fine-grain they are, and how much is the overhead introduced by the user-level threads package.

SPEC 95 SU2COR application. The SU2COR benchmark computes, using the Monte-Carlo method, the masses of elementary particles in the framework of the Quark-Gluon theory. This application offers two levels of parallelism. The outer level consists of up to four parallel sections. The inner level contains parallel loops.

SPEC 95 HYDRO2D application. The HYDRO2D application solves the hydrodynamical Navier Stokes equations to compute galactical jets. It offers two levels of parallelism worth to be exploited. At an outer level, section level parallelism can be exploited as several (equal and different) procedures are called on different data. For instance, the execution of the subroutine FCT accounts for the 61% of the total execution time. In the inner level, each invocation of this subroutine also offers loop-level parallelism. The application seems to be useful to prove how a well balanced allocation of processors to the different levels of parallelism can result in a better processor utilization. See Chapter 8 for a complete description of the parallelization of this application.

NPB BT application. The BT benchmark solves three sets of uncoupled systems of equations, each being block tridiagonal with 5x5 element blocks. These systems arise in many CFD applications. See Chapter 8 for a complete description of the parallelization of this application.

9.3.3. Benchmark evaluation

The evaluation of the LU and Jacobi benchmarks is presented in the following subsections.

Synthetic Jacobi Iteration benchmark. Figure 40 shows the results obtained in the execution of the Jacobi benchmark. The figure presents the execution time (in seconds) of various versions of the application, from 1 to 64 processors. The sequential execution time is 108 seconds (labeled SEQ, dotted line) and has been obtained executing the sequential version of the benchmark. Parallel versions of this benchmark execute in the NANOS environment and use regular nano-threads for spawning parallelism. In all versions, data locality is maintained across Jacobi iterations. This means that in all iterations, the same data (a slice of the matrix) is distributed to (and accessed by) the same processor.

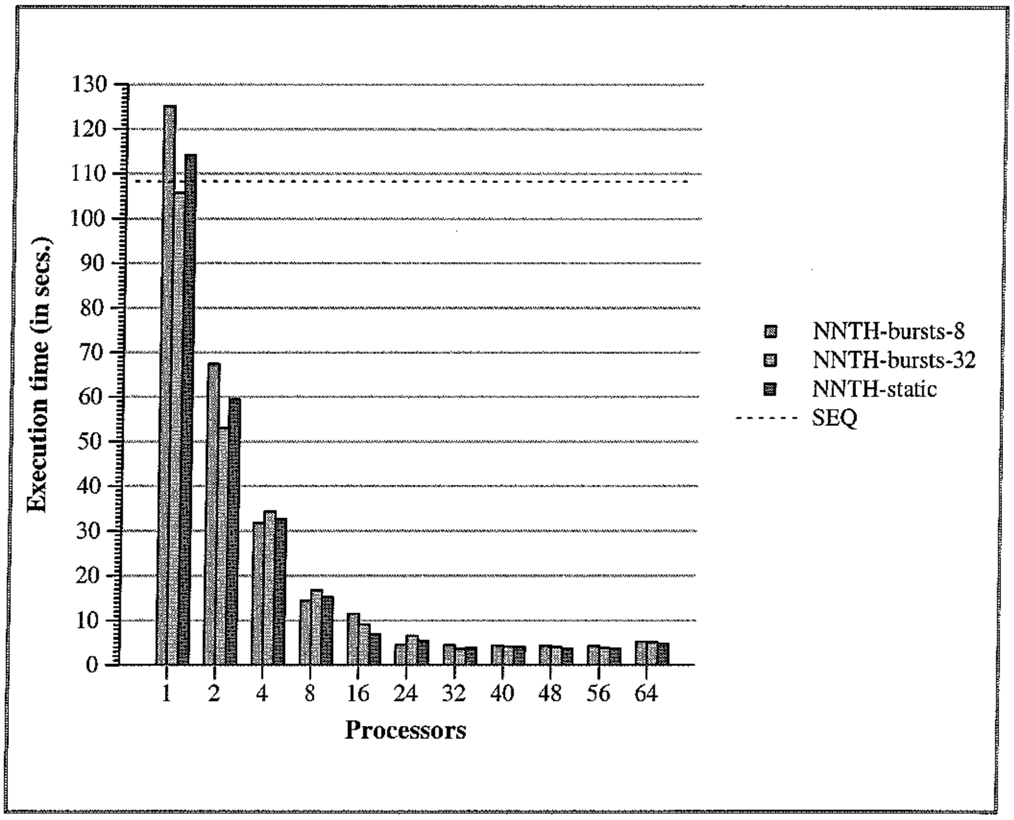


Figure 40: Execution times obtained for the Jacobi benchmark

Bars labelled NNTH-bursts-8 and NNTH-bursts-32 show the execution time of the parallelization based on bursts. In these versions, each time the execution reaches a parallel region, the work is only partially spawned and the remaining work is saved for later spawning. The experiment NNTH-bursts-8 distributes work in chunks of 8 iterations of the parallel loop for each processor. The experiment NNTH-bursts-32 uses a chunk size of 32 iterations. Finally, bar labeled NNTH-static shows the results when the iterations are distributed N/P for each processor. In this case, the work is distributed once, as evenly as possible, for each parallel loop.

Results show that the burst-based parallelizations perform as well as the static approach. This means that, when data locality is maintained, the extra overhead of managing bursts during the execution of the benchmark is hidden by the benefits of running in parallel.

Synthetic LU matrix decomposition benchmark. Figure 41 shows the execution time of three versions of this benchmark, running from 1 to 64 processors. The sequential execution time (SEQ, dotted line) is 155 seconds. It corresponds to the execution of the sequential version of the LU benchmark.

Bar labeled NNTH-bursts corresponds to the parallelization explained in Chapter 8 (Subsection 8.1). In this version, each time the execution reaches a parallel region, the work is partially spawned using bursts. The chunk size to be used is computed each time the parallelism is spawned. In the figure, this is compared with the traditional static approach (bar labelled NNTH-static). In this case, the work is distributed once for each parallel region. Both versions are based on nano-threads.

In this benchmark, as the LU decomposition of the matrix proceeds, data locality is slightly disturbed because the parallel loop is smaller than in the previous iteration. From the comparison, we extract that using bursts when data locality is not completely guaranteed may hurt performance even when the static approach performs well. This is because of the extra overhead of burst management. Usually, this kind of problems could be solved by applying more aggressive techniques for data locality.

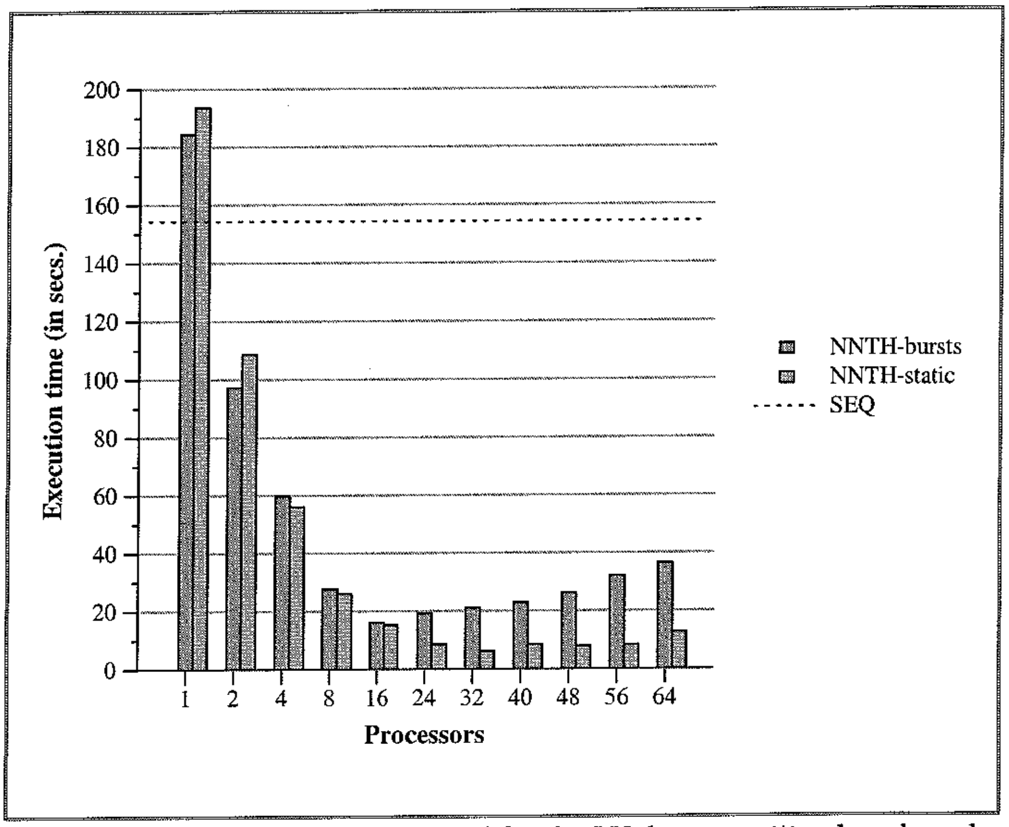


Figure 41: Execution times obtained for the LU decomposition benchmark

9.3.4. Evaluation of individual applications

The execution of the six applications presented in Subsection 9.3.2 is now evaluated. Figures 42 to 50 present several experiments for each application. In general, the experiments are labelled as follows: The dotted line labelled SEQ represents the execution time of the sequential version of the application, given as a reference. MP-FOP and MP-SHM are two different experiments running on top of the MP library. The difference between them is the mechanism used for joining threads. FOP stands for the hardware-based atomic memory operations and SHM for a distributed joining structure in shared memory. The labels NNTH-GWD, NNTH-LWD and NNTH-NTH stand for single level parallelizations using, global work descriptors, local work descriptors and nano-threads respectively. NNTH-1LVL stands for a single-level parallelization on the NANOS environment, based on local work descriptors. And NNTH-MLV corresponds to multi-level parallelizations on the NANOS environment, spawning the outer levels using nano-threads and the inner-most level using local work descriptors.

SPEC 95 SWIM application. Figure 42 shows the execution times of the SWIM application when running, from 1 to 64 processors, in the MP and NANOS environments. The sequential time (SEQ) is indicated through the dotted line, as a reference. The figure presents two different experiments running on the MP library, and three different experiments running on the NANOS environment. The MP library based experiments show no differences in performance, except when running in one processor. This means that, for applications with coarse granularity loops, the distributed joining mechanism implemented on shared memory offers the same performance than the mechanism based on the Origin atomic memory operations (FOP's). The usual unbalance existing in parallel regions avoids the situation where all processors are trying to join with the master at the same time and the memory subsystem does not become a bottleneck. When running in one processor, we have detected that the MP-SHM version of that application suffers from cache effects that increase the execution time.

The experiments on the NANOS environment when running using the GWD and LWD techniques show the same cache effects on 1 to 4 processors. When the size of the application data fits in the caches of the processors (from 8 processors and above), there is no difference between the different implementations. The efficient LWD and GWD implementations behave the same that the MP library based implementations. Only the nano-thread based implementation (NTH) shows the overhead of the extra management of local address spaces and ready queues.

From the plot, the best number of processors to execute SWIM is 24. The absolute minimum execution time is reached when using 32 processors, but the difference with respect the execution on 24 processors is of several cents of a second. As a conclusion, the performance is better in 24 processors. The FOP and SHM versions reach an execution time of 5.21 and 5.25 seconds, respectively. LWD downs to 4.47 seconds. The results show that within coarse-grained applications, spawning parallelism from the master to the slaves in a one-to-one way is performing very good.

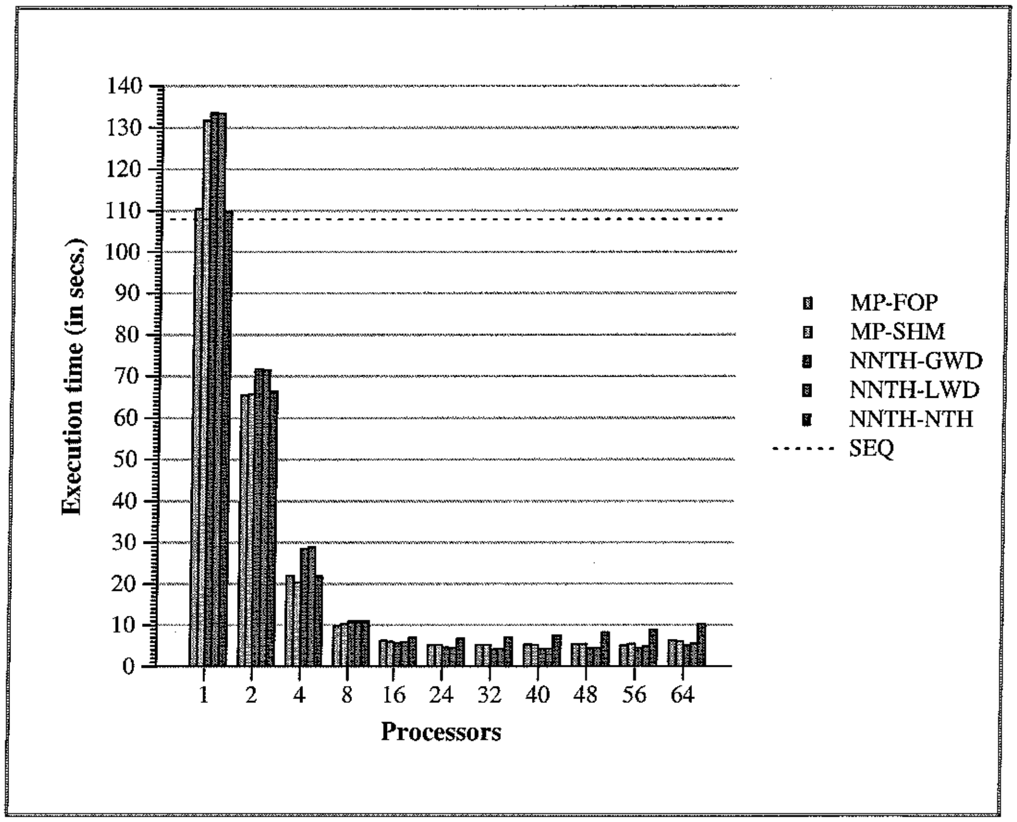


Figure 42: SPEC95 SWIM execution times

SPEC 95 TOMCATV application. Figure 43 shows the execution times of the TOMCATV application when running, from 1 to 64 processors, in the MP and NANOS environments. The sequential time (SEQ) is indicated through the dotted line, as a reference. The figure presents two different experiments running on the MP library, and three different experiments running on the NANOS environment. In TOMCATV, all the experiments show the same behavior. When running from 1 to 4 processors the application does not fit in the secondary cache of the participating processors. For this reason, the execution time of the application is increased in some executions due to cache effects. The standard deviation of these experiments is larger than the standard deviation of the experiments running on more than 4 processors.

From the plot, TOMCATV seems to achieve good speedup when running up to 16 processors only. This is because the execution time of the sequential phase (input/output phase) is around five seconds. The execution time of the parallel computation phase is 3.5 seconds on 16 processors and 3.1 seconds of 24 processors, which is a gain already noticeable. In this case, the differences between the MP and NANOS environments are minimal.

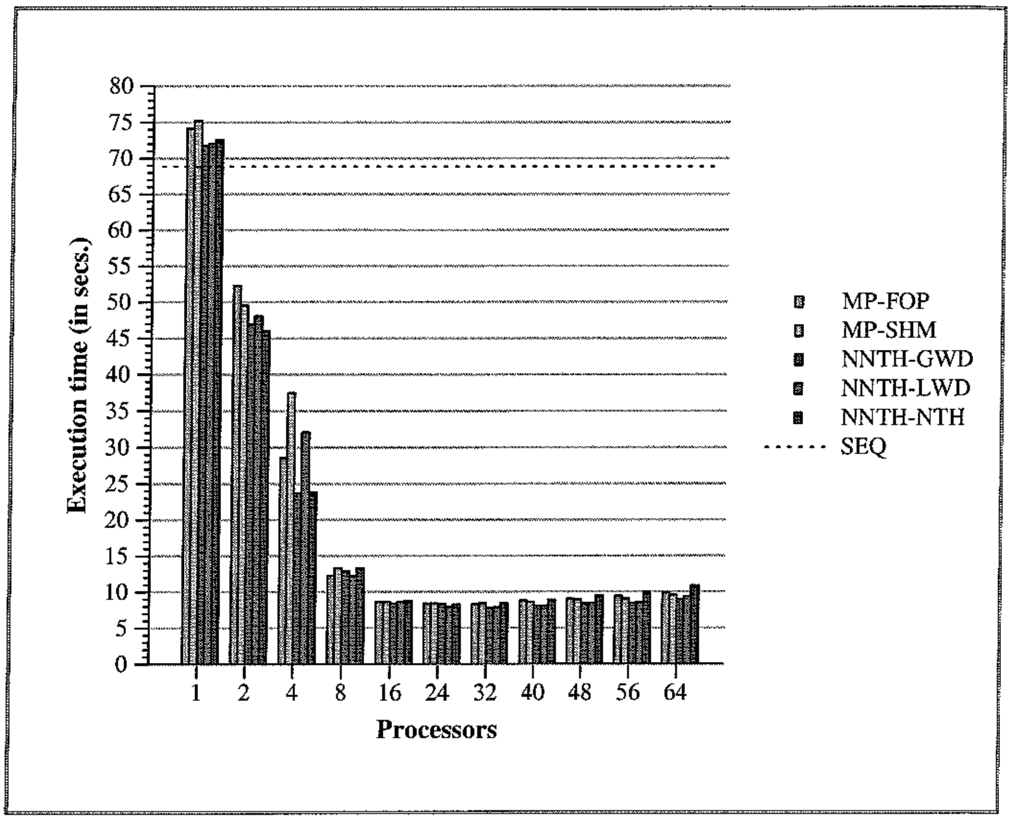


Figure 43: SPEC95 TOMCATV execution times

SPEC 95 TURB3D application. Figures 44 and 45 show the internal structure and the evaluation of the TURB3D application [7]. Figure 45 presents the execution time of the sequential TURB3D application (SEQ, represented through the dotted line), as a reference. Along with it. the figure also presents the execution time of the MP-SHM and NNTH-1LVL experiments, which are equivalent single-level parallelizations, exploiting the parallel loops only, running one top of the MP and NANOS environments, respectively. There are no significant differences among them. The best number of processors for spawning parallelism in TURB3D is 24, where the execution time reaches 30.5 seconds, or an speedup of 9.5.

The figure also presents the evaluation of two multiple level parallelizations. The first one (NNTH-MLV-MIX) corresponds to the spawning of the parallel sections numbered 1-18 and 20-31 (see Figure 44), along with the parallel loops inside them and inside nodes 19 and 32. In this parallelization, data computed by sections 13 to 18 is merged in node 19, which gets the contribution of the parallel sections and computes new values. And the same happens between data computed in node 19 and parallel sections in nodes 20 to 25 and between nodes 26 to 31 and node 32. This mixing of data causes an increment of the memory traffic, which avoids getting good performance compared with the loop-level parallelization, in which this merging does not occur because each parallel loop accesses the same portion of the data.

The experiment labelled NNTH-MLV consists of changing the parallelization of the nodes 19 and 32. The goal is to maintain data locality in these parallel loops. Although algoritmically it is possible to modify both loops to maintain data locality through loop partitioning, the results are not good because a number of operations should be replicated in each new parallel section and the extra work is too large to get better speedup than the single level parallelization.

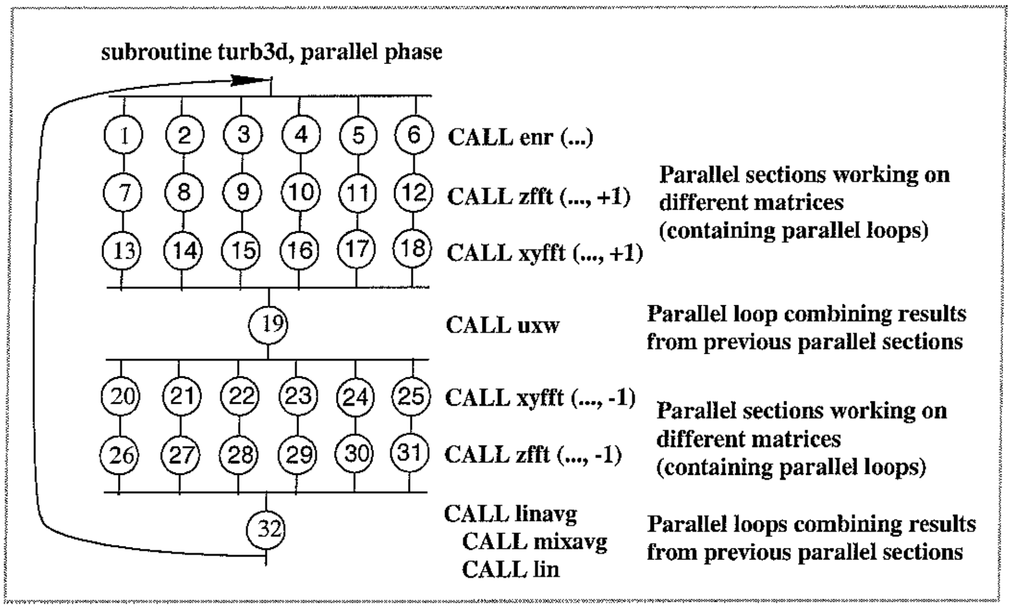

subroutine turb3d, parallel phase
CALL enr (...)
CALL zfft (..., +1)
CALL xyfft (..., +1)
Parallel sections working on different matrices (containing parallel loops)
CALL uxw
Parallel loop combining results from previous parallel sections
CALL xyfft (..., -1)
CALL zfft (..., -1)
Parallel sections working on different matrices (containing parallel loops)
CALL linavg
CALL mixavg
CALL lin
Parallel loops combining results from previous parallel sections

Figure 44: Structure of the TURB3D application

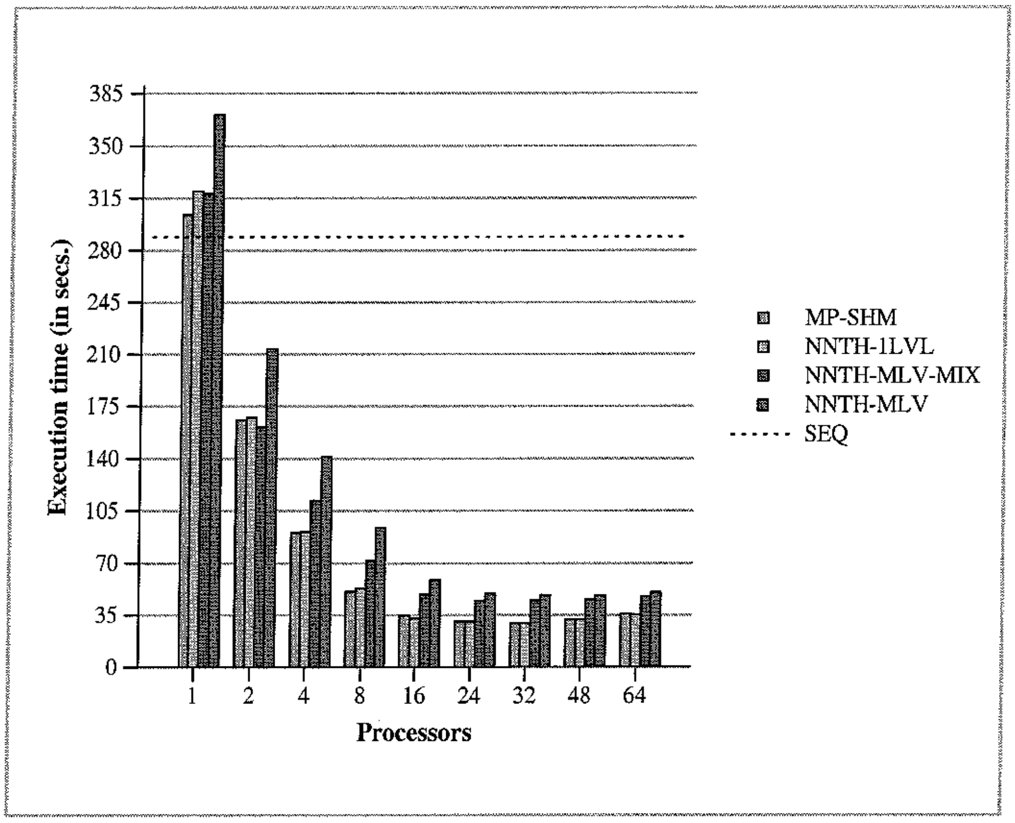


Figure 45: SPEC95 TURB3D execution times

SPEC 95 SU2COR application. Figure 46 shows the evaluation of the SU2COR application. It presents the execution time of the sequential version of the application (SEQ, dotted line), given as a reference. In this case, the experiments labelled MP-SHM and NNTH-1LVL are the equivalent parallelizations of SU2COR running on top of the MP and NANOS environment. The SU2COR application has several fine-grained loops, which make the parallelization based on local work descriptors to reflect more overhead than the MP based parallelization. On the other hand, the current multi-level parallelization (NNTH-MLV) is not taking advantage of data locality, and is not obtaining better performance than the single level one.

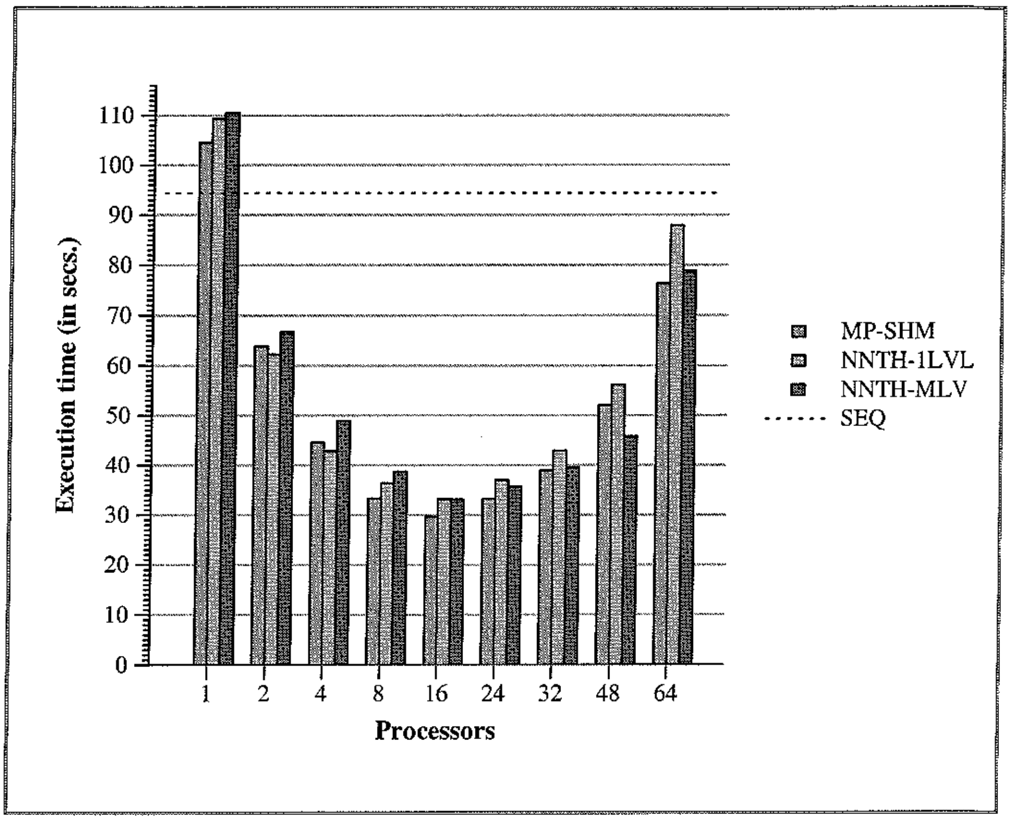


Figure 46: SPEC95 SU2COR execution times

SPEC 95 HYDRO2D application. Figures 47 and 48 show the execution time and speedup of several experiments running the HYDRO2D application [85]. The structure and parallelization of the HYDRO2D application is presented in Chapter 8. Four different versions of the Hydro2D benchmark have been executed using the reference input file, as provided in the SPEC benchmarks. Sequential execution time is 154.71 seconds, which agrees with the 154 seconds reported in the SPEC benchmark CFP95 summaries [122]. This is represented through the dotted line (SEQ) in Figure 47. The experiments labelled MP-SHM and NNTH-1LVL are equivalent parallelizations of the application running on the MP and NANOS environments, respectively. This application contains several parallel loops which are fine-grained (below 100 us.). This motivates that the NNTH-1LVL version (based on local work descriptors) reflects more overhead than the MP version. This is also the reason to expect better performance in the NNTH-MLV experiments. Inside the multi-level parallelization, such small loops are partitioned using less processors (more specifically, 1/4 of the processors used in the single level version). Two effects benefit the execution of the multi-level version: First, the spawning time for each parallel region is reduced and, seconds, the different parallel regions can proceed independently, achieving a higher load balancing. This can be seen in Figure 48, looking at the speedup of the multi-level version when running on 32 and more processors. The single level versions of the application stop getting speedup when running with 16 processors and the multi-level version continues obtaining good performance.

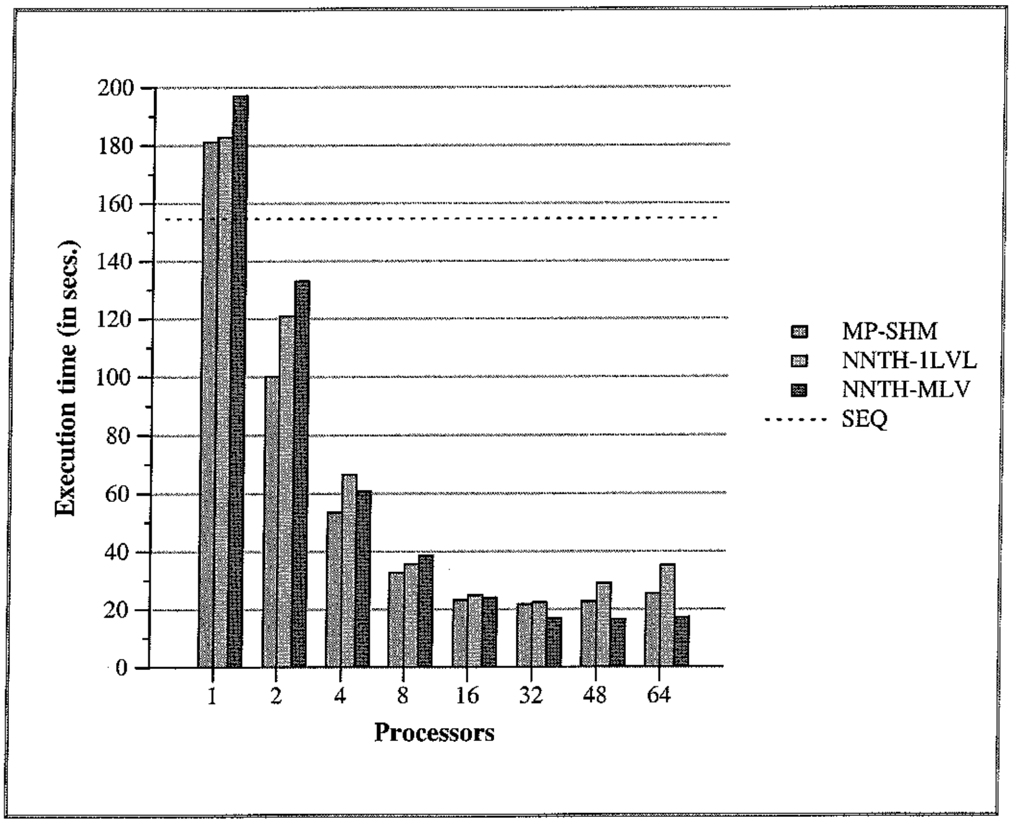


Figure 47: SPEC95 HYDRO2D execution times

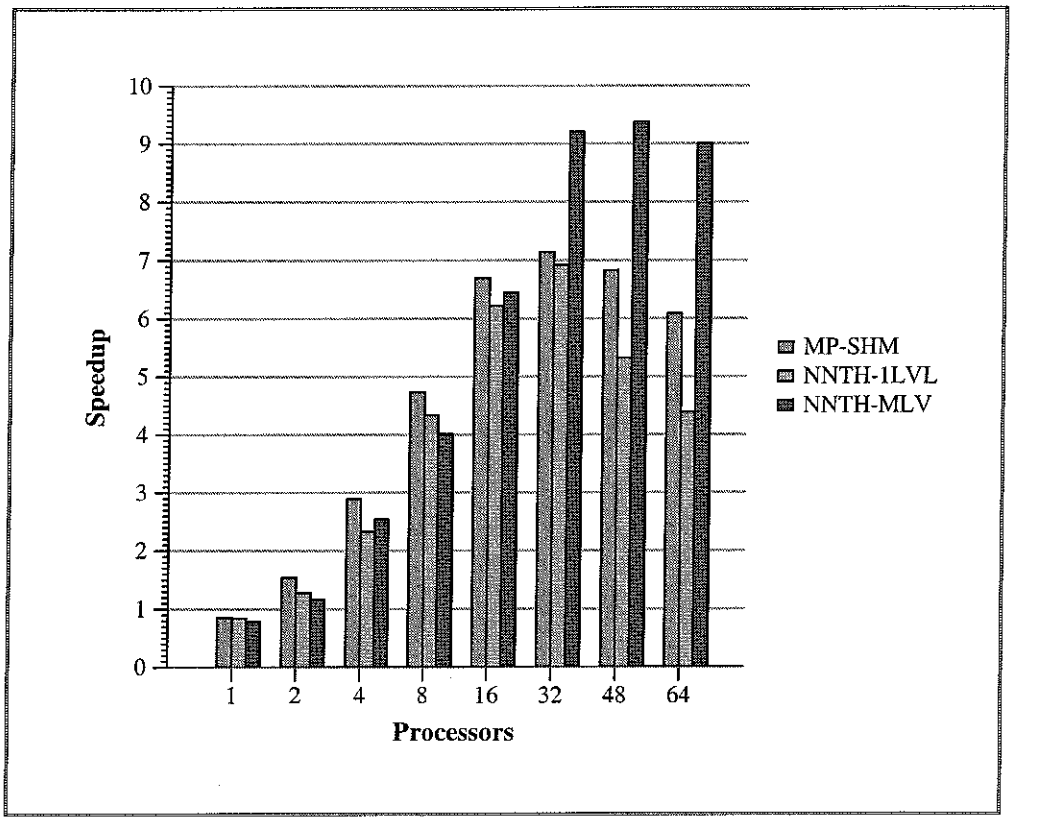


Figure 48: SPEC95 HYDRO2D speedup

A fair comparison can be established between the single (NNTH-1LVL) and multi-level (NNTH-MLV) versions. As a result, for up to 8 processors, the overhead of the two-level parallelization causes an increment in the execution time below 10%. In this application, four different processor groups are established at the outer parallelization level, so at least 8 processors are required to effectively exploit multiple levels of parallelism. Four groups of 2 processors give an speedup of 4.0 compared to the 4.5 achieved by the one-level version running in 8 processors. When 16 processors (four groups of 4 processors) are used, nearly the same speedup is achieved by both NNTH versions. When using more than 16 processors, the one-level version is unable to scale, while the multi-level version scales till a speedup of 9.3 on 48 processors and giving an improvement of 30% in 32 processors with respect the single level version.

NAS BT application. Figures 49 and 50 show the execution time and speedup of the NPB APPBT application [85]. We have selected to run the experiments with the small version (class W) of the BT application because achieving good results in small applications is important, so that this means the overhead introduced by the run-time environment is small. All versions of the BT application have been compiled with -O3 compilation option instead of -Ofast=ip27 because this is the option used in the standard compilation of the NAS benchmarks in SGI machines. We have observed that compiling with the -O3 option the application shows better performance than with -Ofast=ip27. Due to the parallelization scheme and the application class, the application can be executed on as much as 24 processors.

In Figure 49, the execution time of the sequential version of the application is showed (SEQ, dotted line). It is higher than the execution time of the parallelized versions running on one processor. This is because of some optimizations done in the parallel versions, which improve data locality. The MP-SHM is the single-level loop parallelization version, running on the MP environment. The NNTH-1LVL experiment is the one-dimensional pipelined parallelization running on top of the NANOS environments (see Chapter 8). This version shows higher overhead when running on a small number of processors, but it shows an improvement with respect the MP-SHM version when using 4 and more processors.

The NNTH-MLV version corresponds to the two-dimensional pipelined parallelization achieved through two-levels of parallelism (as shown in Chapter 8). Comparing the results in the nano-threads environment, the results on a small number of processors show that the multi-level version is worse than the one level version. This is due to the extra overhead introduced by the multi-level version. However, when using more than 4 processors, the multiple-level version achieves higher speedup (see Figure 50), reaching 14.5 on 24 processors. The gain in the speedup reaches 65% with respect to the one dimensional parallelization.

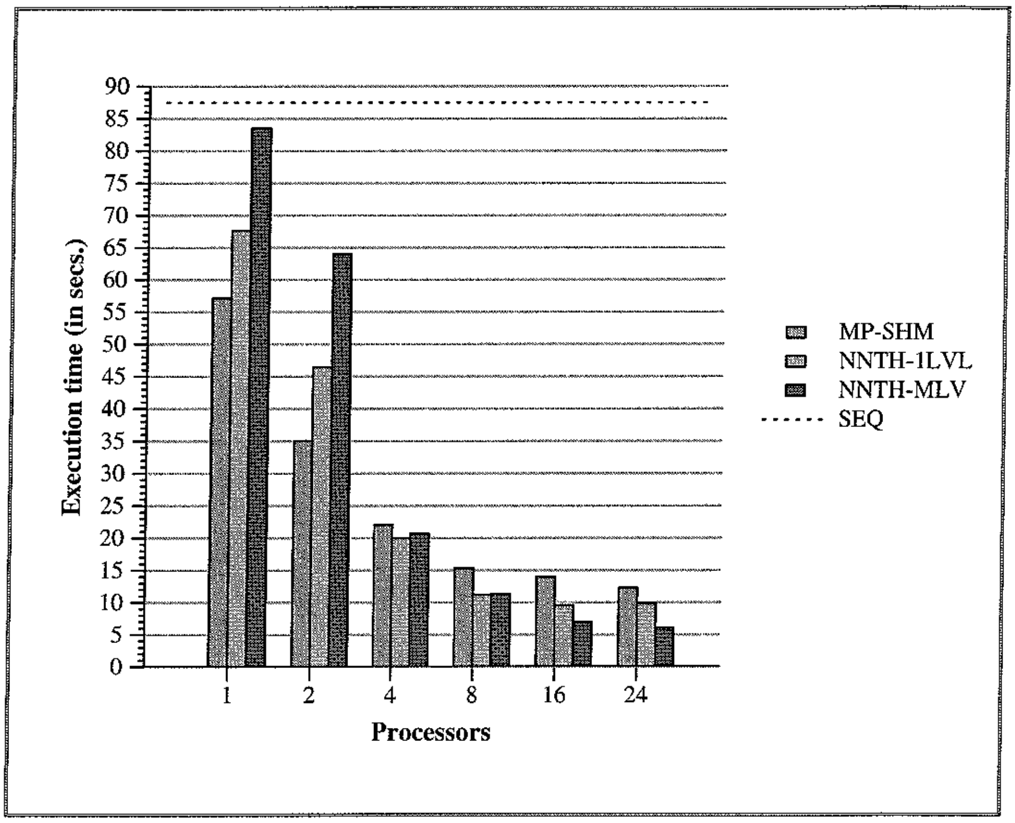


Figure 49: NAS BT execution times

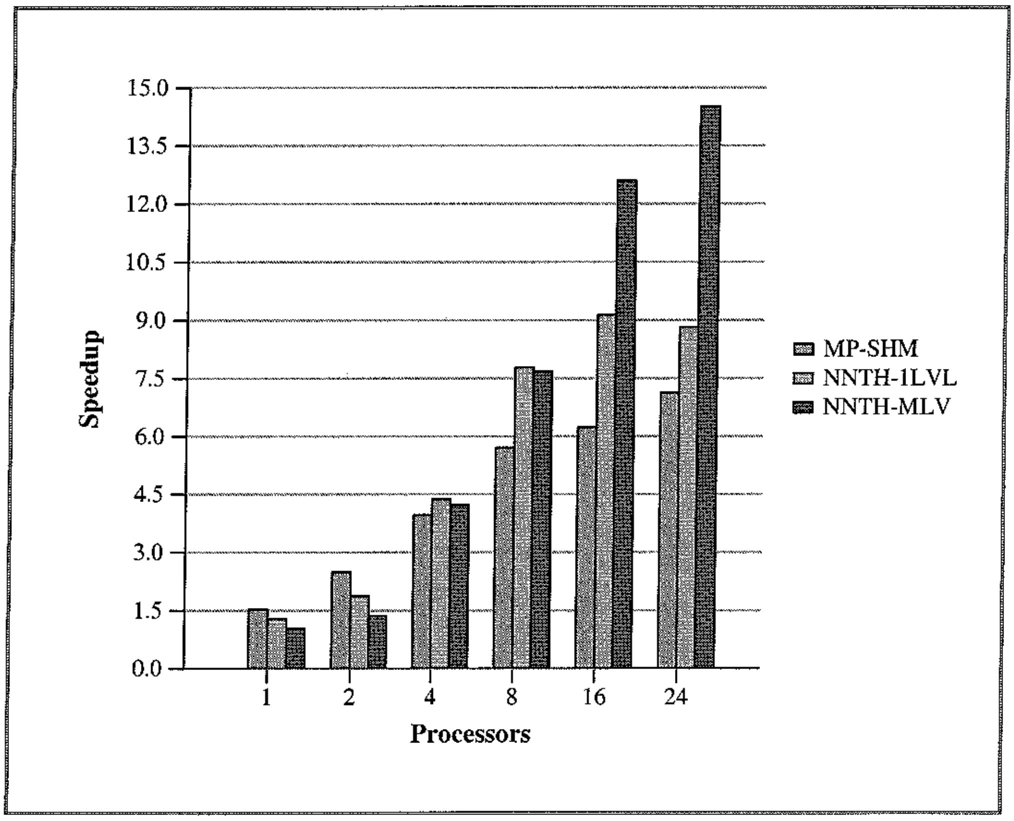


Figure 50: NAS BT speedup

9.3.5. Conclusions of the evaluation of individual applications

From the previous evaluation of several benchmarks and applications running in a dedicated NANOS environment, we conclude that our proposed environment performs as well as the native SGI MP environment for applications when exploiting a single level of parallelism, and that there exist some applications that effectively take advantage of the exploitation of multiple levels of parallelism.

From the evaluation of the LU and Jacobi benchmarks, using nano-thread bursts, we learn that application-level scheduling based on nano-thread bursts performs comparable to the static approaches, when data locality is not an issue. Otherwise, when data locality is broken due to the algorithm, as happens in the execution of the LU benchmark, the overhead introduced by the burst-based approach is noticeable when increasing the number of processors.

From the evaluation of two applications exploiting a single level of parallelism (SPEC95 SWIM and TOMCATV), we conclude that the performance of the parallelization based on work descriptors perform as well as the native SGI MP library environment. From these experiments, we detect that the cache behavior can influence both the LWD and GWD techniques when running in a small number of processors. This will be a subject of further study after the termination of this work. It is remarkable than, when running on 8 and more processors, the behavior on both environments is very similar. Although above 24 or 32 processors, the applications do not obtain further speedup, the interesting point is that the performance is not degrading when using GWD or LWD. This is important because it means that applications containing both fine-grain and coarse grain parallel loops, such as the SWIM, can be executed on a large number of processors, ensuring that the execution environment will not degrade performance when executing the small loops. With respect the evaluation of nano-threads, instead, we observe that the nano-threads behavior is degrading when the number of processors increase.

From the evaluation of four applications exploiting multiple levels of parallelism we conclude that achieving high performance is possible, but it can be limited by the structure of the applications. TURB3D and SU2COR do not benefit from the exploitation of multiple levels of parallelism because of the large amount of data exchanged among the processors. Instead, in the HYDR2D and the NAS APPT BT applications, exploiting multiple levels of parallelism benefits the execution of the applications when running in 16 and more processors. Achieving data locality in applications exploiting multiple levels of parallelism is an open issue, to be studied after the termination of this work.

9.4. Evaluation of workload performance

In the following subsections, the execution of four different workloads is evaluated in the MP and NANOS environments. Each different workload is used to highlight some of the characteristics of the two execution environments. The workloads are built using the applications explained in Subsection 9.3.2 and evaluated individually in Subsection 9.3.4. The only exception is the LTOMCATV application. The LTOMCATV application is a dynamic version of the TOMCATV application. In LTOMCATV, a TOMCATV application has been enclosed inside an outer sequential loop performing ten iterations. For each outer iteration, it requests one processor for the input data phase and it requests the processors indicated by the user for the computation phase.

All workloads consist of several (possibly different) applications, executed in parallel, requesting a number of processors. All applications start at the same time. When one of the applications terminates, another instance of the same application, with the same processor request is launched automatically. All instances start executing with one processor and they request for more processors when spawning the parallelism for the first time. The LTOMCATV releases *P-1* processors each time it enters a sequential phase.

The resulting workload execution is visualized through the Paraver tool [109], representing time in the *x* axis and applications in the *y* axis (see Figure 51, as an example). 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. Throughout each horizontal line, the points where the lines change their color are also marked using flags, allowing the black and white printing of these images. For example, in Figure 51, nine complete instances of the *appl 3* are executed, while only two instances of the *appl 6* are completely executed in the same amount of time. In this kind of figures, we will count the number of instances executed from the starting of a specific instance of an application (shown at the left-hand side), till the termination of another specific instance of an application (at the right-hand side).

Two versions of the workloads are executed. The first one is the MP-SHM version of the applications running on top of the SGI MP Library and the IRIX operating system as shipped by Silicon Graphics, with the dynamic adaptation to the number of available processors activated, as is by default (the OMP_DYNAMIC environment variable is set to TRUE). And the second version is the NNTH-LWD version of the applications, exploiting multiple levels of parallelism in the HYDRO2D and BT cases, running on top of NthLib and the CPU Manager, using the user and kernel interfaces presented in Chapters 4 and 5, respectively.

In the following subsections, we present the different workloads and their evaluation. The first workload (wl1) uses the dynamic LTOMCATV application to find which is the correct fine-tuning of the MP library for it. Other applications need not to be fine-tuned because they proceed in parallel most of time during their execution.

After determining the correct block time value for the LTOMCATV application, the second workload (wl2) shows that fine-tuning is not enough to achieve a smooth execution for all the applications running in the system. In particular, applications requesting a larger number of processors than others can suffer from synchronization problems, resulting in higher execution times.

The third and fourth workloads consists of a larger number of applications. In the third workload (wl3), applications can request a limited number of processors (the limit is set to eight processors). Research and production centers usually limit the number of processors that each application can request in order to avoid overloading the system. In the case presented here, one application is allowed to execute with a larger number of processors to evaluate also its execution.

In the fourth workload (wl4), all the applications are executed using the number of processors at which they achieve better speedup during an individual execution in a dedicated machine. Results show that this is the case where the NANOS environment outperforms significantly the SGIMP environment.

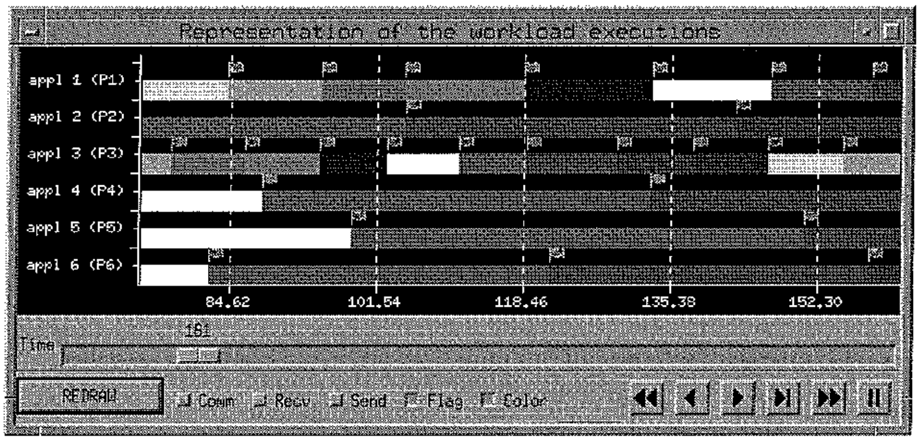


Figure 51: Example of the representation of a workload execution

9.4.1. Fine-tuned workload execution

The objective of the first workload (wl1) is to show that the SGIMP environment must be fine-tuned to achieve better performance. Fine tuning is not easy because it depends on the application. The workload wl1 consists of five LTOMCATV applications requesting 16 processors each. Table 13 shows the workload composition. It requests a maximum of 80 processors, in a 64 processors machine. Fine tuning is achieved through the block time parameter, which allows processes to release the physical processor when no parallel work is available for execution (Subsection 6.2.1 explains in more detail this parameter).

Application	Requested Processors
LTOMCATV	1, 16
LTOMCATV	1, 16
LTOMCATV	1, 16
LTOMCATV	1, 16
LTOMCATV	1, 16
TOTAL requests	**80**

Table 13: Wl1 workload composition

Three different experiments are done with this workload. The first one (Figure 52, window named SGIMP(1)) corresponds to the standard SGIMP environment, with the default block time (10,000,000 iterations). In the second one (window SGIMP(2)) the block time has been reduced to 200,000 iterations. This number has been selected after testing the workload with different blocking times ranging from 50,000 to 10,000,000 iterations. The third one (Figure 52, window NNTH-Cluster) is the execution of the workload on the NANOS environment with the Cluster scheduling policy. All windows use the same scale for the visualization. They present the workload execution from the point where the second instance of each application starts.

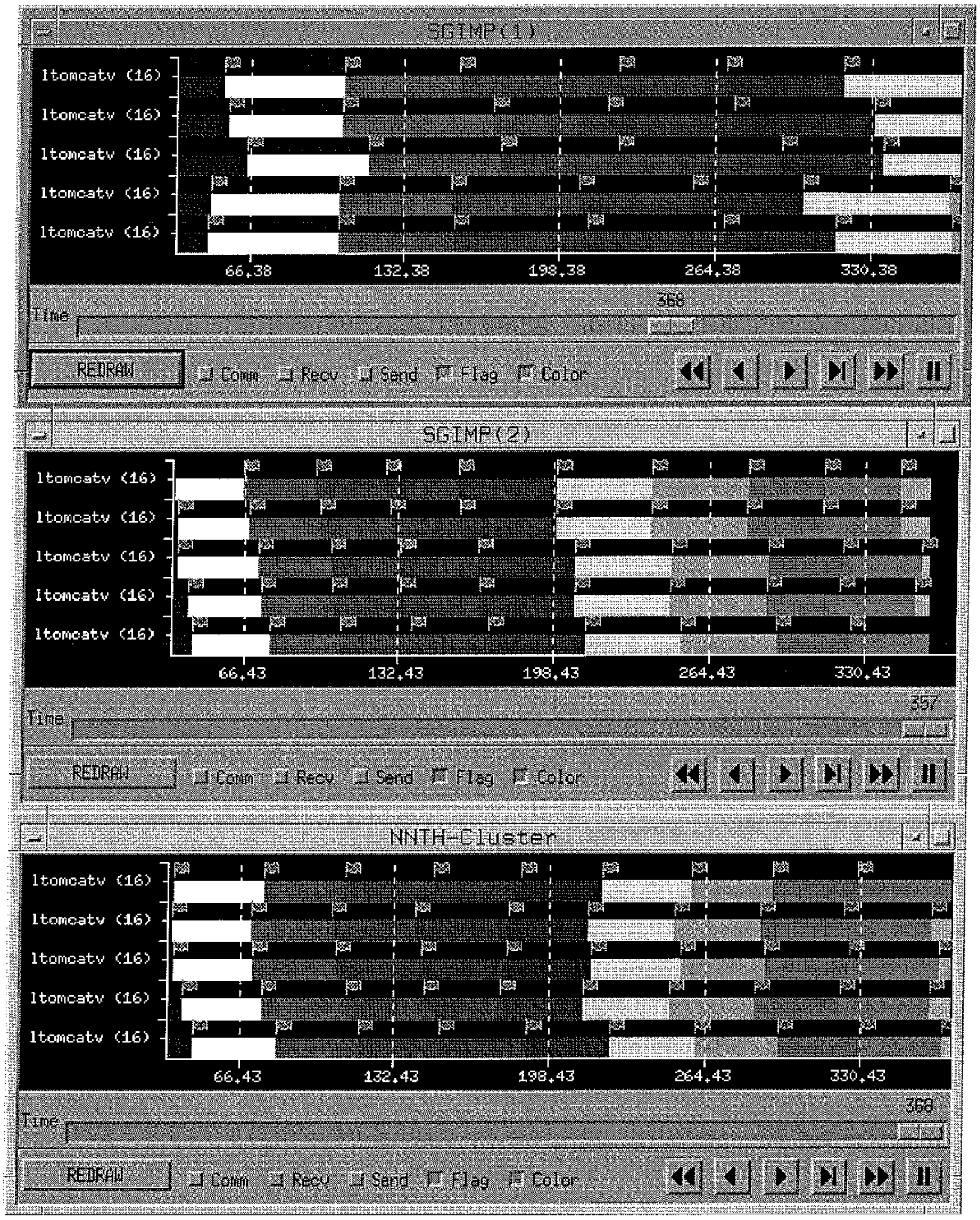


Figure 52: W11 workload execution on the SGI-MP and NANOS environments

It can be observed that, while the standard SGIMP environment is able to execute around six complete instances of the applications, both the fine-tuned SGIMP and NANOS environments are executing as much as nine instances of each application. This result indicates that it is good for the global workload performance that dynamic applications can release processors when they are executing in a sequential section. Either the execution environment or the application itself can be aware of releasing the processors. The SGIMP environment is in charge of that and the block time parameter indicates how much time to wait from the point where processors become idle till they will be blocked. In the case of the NANOS environment, it is the application which sets the number of requested processors to one and releases the processors.

Comparing the SGIMP(2) and NNTH-Cluster windows, we determine that the best results are obtained for the SGIMP fine-tuned execution. The ninth instance of each application in the SGIMP environment terminates earlier than the corresponding instance in the NANOS environment.

Table 14 shows the numeric evaluation of the wl1 workload, also reflecting the previous conclusion. The table presents the number of complete instances executes of each application, the average execution time and its standard deviation. In the numbers presented in Table 14 we observe that fine tuning in the SGIMP environment greatly influences the execution time of the applications, from around 53 seconds to 35 seconds. The execution time of the applications running in the NANOS environment is only slightly higher (36 seconds). Observe also that the standard deviation of the execution times of the different instances are significantly greater in both SGIMP executions with respect the results given by the Cluster policy. This is a first important result for the execution in our environment, which will be a observed consistently in the evaluation of the following workloads. Users will observe a smoother kernel-level scheduling when running in the NANOS environment than when running in the SGIMP environment, resulting in a more predictable execution.

Application	**Number of complete instances / Average execution time (in secs.) / Standard deviation**								
	SGIMP (1)			**SGIMP (2)**			**Cluster**		
LTOMCATV	6	53	7.1	9	35	4.5	8	36	1.5
LTOMCATV	5	55	6.2	9	34	4.5	9	36	1.7
LTOMCATV	5	54	8.6	9	35	4.0	9	36	1.0
LTOMCATV	6	52	5.3	9	34	4.5	9	35	1.6
LTOMCATV	6	53	4.1	8	35	4.3	9	35	0.6

Table 14: Results of the evaluation of the wl1 workload

9.4.2. Benefits of user/kernel cooperation

The objective of the second workload (wl2) is to show that, even when applying fine tuning, not all applications are going to take advantage of the processors allocated to them. To demonstrate that, we have changed three of the LTOMCATV applications in wl1 for two SWIM applications requesting 32 and 24 processors, respectively. Table 15 shows the resulting workload. From the evaluation of the individual applications, presented in Subsection 9.3.4, SWIM obtains the same execution time when using 32 and 24 processors. This means that working inside the workload, both SWIM applications should also take the same amount of execution time. That will demonstrate a fair distribution of processors.

Application	Requested Processors
LTOMCATV	1, 16
LTOMCATV	1, 16
SWIM	32
SWIM	24
TOTAL requests	**58-88**

Table 15: Wl2 workload composition

Figure 53 shows the execution of the wl2 workload on the two different environments: SGIMP and NANOS. The LTOMCATV application is tuned as in the previous workload, in the SGIMP environment. Three different scheduling policies are presented in the NANOS environment: Equipartition, Cluster and Batch. Numeric results for the wl2 workload are presented in Table 16.

With respect the SGIMP environment (window named SGIMP in Figure 53), it can be observed that the execution of the SWIM application requesting 32 processors is delayed with respect the SWIM application requesting 24 processors. All instances of the SWIM application requesting 32 processors suffer from higher execution times. The average of their execution time is 8.2 seconds, compared with the 6.1 seconds taken in average by the SWIM application requesting 24 processors (see Table 16). Although the LTOMCATV applications are releasing the processors during their sequential phase and the mechanism providing dynamic adaptation to the number of available processors is activated, the kernel-level scheduling policy is not able to ensure that the same processes belonging to each application are executed. This is the cause of synchronization problems, noticeable in the SWIM application that requests 32 processors, delaying its execution.

The previous result can be compared with the effect of applying the Equipartition policy (window named NNTH-Equip in Figure 53). In this case, Equipartition decides to give 16 processors (= 64 processors / 4 applications) to each application, regardless of the number of processors they are requesting. The resulting execution times for the SWIM applications are very similar and lower than the execution times achieved by the SGIMP environment. The difference with the SGIMP environment is that, in this case, the number of ready processes is limited to 64. This avoids any synchronization problem, but not all processors are used at all time because the LTOMCATV applications are releasing 15 processors each, during their sequential I/O section. The Equipartition policy does not assign these processors to other applications.

The Cluster policy (window named NNTH-Cluster in Figure 53) solves the Equipartition drawback. Each time a LTOMCATV application releases the processors to execute the sequential section, the processors are given to the SWIM applications. Also, when the LTOMCATV applications again requests the processors, they are reassigned. The use of the NANOS kernel interface results in improved execution. With this approach the execution times of the applications receiving the processors are even better (from 7 to 6.5 seconds, or an average improvement of 7%, with respect the Equipartition). As the Cluster policy assigns processors in clusters of 4, both SWIM applications receives the processors released as evenly as possible.

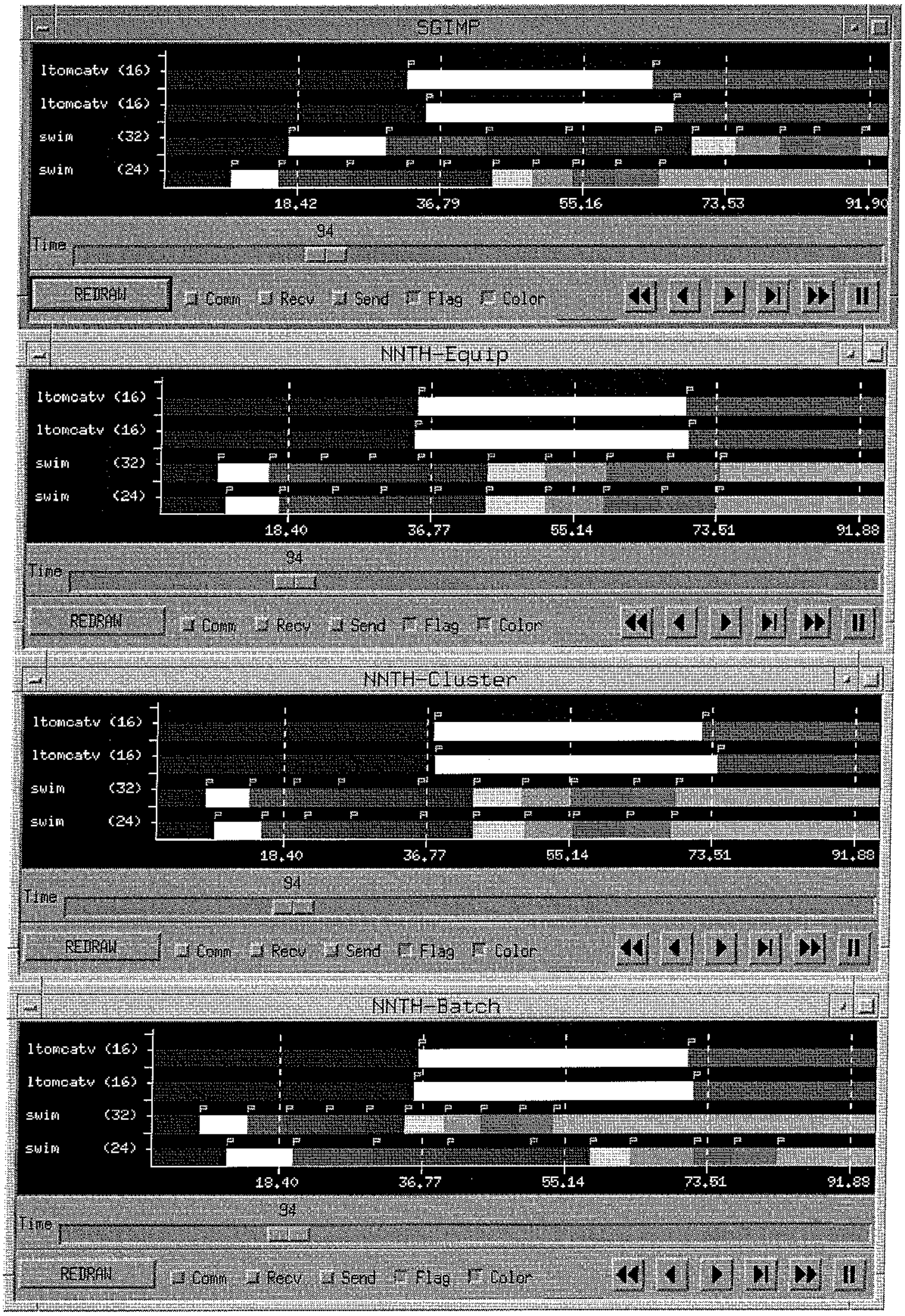


Figure 53: Wl2 workload execution on the SGIMP and NANOS environments

In Figure 53 we present also the execution of the same workload under the Batch policy (window named NNTH-Batch). This policy has a very predictable behavior and it is useful to see how the processors released by some applications (LTOMCATV) can be used by another ones. Under the Batch policy, physical processors are usually assigned to the LTOMCATV applications (32) and to the SWIM requesting 32, for a total of 64 processors allocated. Each processor released by the LTOMCATV applications is automatically assigned to the SWIM requesting 24 processors. The resulting environment for this last application is that it executes only when the LTOMCATV applications release processors. When all applications are running, the second SWIM spends 9.7 seconds. When the first SWIM terminates, the average execution time of the second one goes down to 6.0, giving a total average of 7.8 seconds, as reported in Table 16.

With respect the standard deviations showed by the execution time of the different instances, observe in Table 16 that both the Equipartition and the Cluster policies achieve standard deviations significantly lower than the SGIMP environment.

Application	**Number of complete instances / Average execution time (in secs.) / Standard deviation**											
	SGIMP			**Equip**			**Cluster**			**Batch**		
LTOMCATV	1	31	-	1	35	-	1	34	-	1	35	-
LTOMCATV	1	32	-	1	35	-	1	36	-	1	36	-
SWIM (32)	9	8.2	3.3	9	7.1	0.9	9	6.7	1.4	9	5.0	0.4
SWIM (24)	9	6.1	1.3	9	7.0	0.4	9	6.5	1.0	9	7.8	2.1

Table 16: Results of the evaluation of the wl2 workload

9.4.3. Workload with limited requests

The goal of the third workload (wl3) is to evaluate what happens when a larger number of applications is executed at the same time. The number of processors requested for each application is decreased with respect the requests in the previous workloads, in order to avoid overloading the machine. This comes from a policy used in a number of research and production centers [46], where the maximum number of requested processors is explicitly limited by the system administrators to avoid overloading the parallel computer. For the execution of this workload we consider that the number of processors is limited to 8, except for one application, which has been allowed to run on 16 processors. The complete workload description is shown in Table 17. This workload is requesting a maximum of 72 processors, not far away from the 64 processors available in the machine. This suggests that there will not be large synchronization problems during the execution of the applications.

Application	**Processors requested**
BT	8
BT	4
LTOMCATV	8
LTOMCATV	4
SWIM	8
SWIM	4
HYDRO2D	16
HYDRO2D	8
SU2COR	4
TURB3D	8
TOTAL requests	**57-72**

Table 17: Wl3 workload composition

Figure 54 shows the execution of the wl3 workload in the SGIMP and NANOS environments. Both LTOMCATV applications are tuned as in the previous workloads, using a block time of 200,000 iterations, for the SGIMP environment. The workload has also been executed with the standard block time for all the applications and no differences in performance have been observed, probably because only the LTOMCATV applications are able to release processors and the number of processors involved (10) is smaller than in the previous workloads. The Cluster policy is used in the NANOS environment.

The execution on the SGIMP environment (window SGIMP in Figure 54) shows that, in general, the instances of the same application requesting more processors receive a larger number of them along the execution, and their execution time is smaller than the execution time requesting less processors. Nevertheless, this is not the case of the HYDRO2D application. The instances of HYDRO2D requesting 16 processors run worse than the instances requesting eight processors. Again, this confirms that synchronization issues among a large number of processes are degrading the execution of the applications. And this problem usually appears between 8 and 16 processors in the SGIMP execution environment.

Instead, the execution of the applications in the NANOS environment (window NNTH-Cluster in Figure 54) proceeds slightly better and the applications requesting more processors can take advantage of the them, including the HYDRO2D application requesting 16 processors.

Table 18 shows the numeric evaluation of the wl3 workload. The NANOS environment is able to execute more instances of the BT(8), LTOMCATV(4), SWIM(8) and HYDRO2D(16) applications. Instead, the SGIMP environment executes more instances of the SWIM(4) and HYDRO2D(8), meaning that both environments perform in a similar way. The interesting differences raise in the execution time taken by the HYDRO2D(16) application in the SGIMP environment (118 seconds) compared with the execution time obtained by in the NANOS environment (41 seconds). Also, and as it has been commented in the previous workloads, the standard deviation obtained in the execution of all the applications is consistently lower in the NANOS environment.

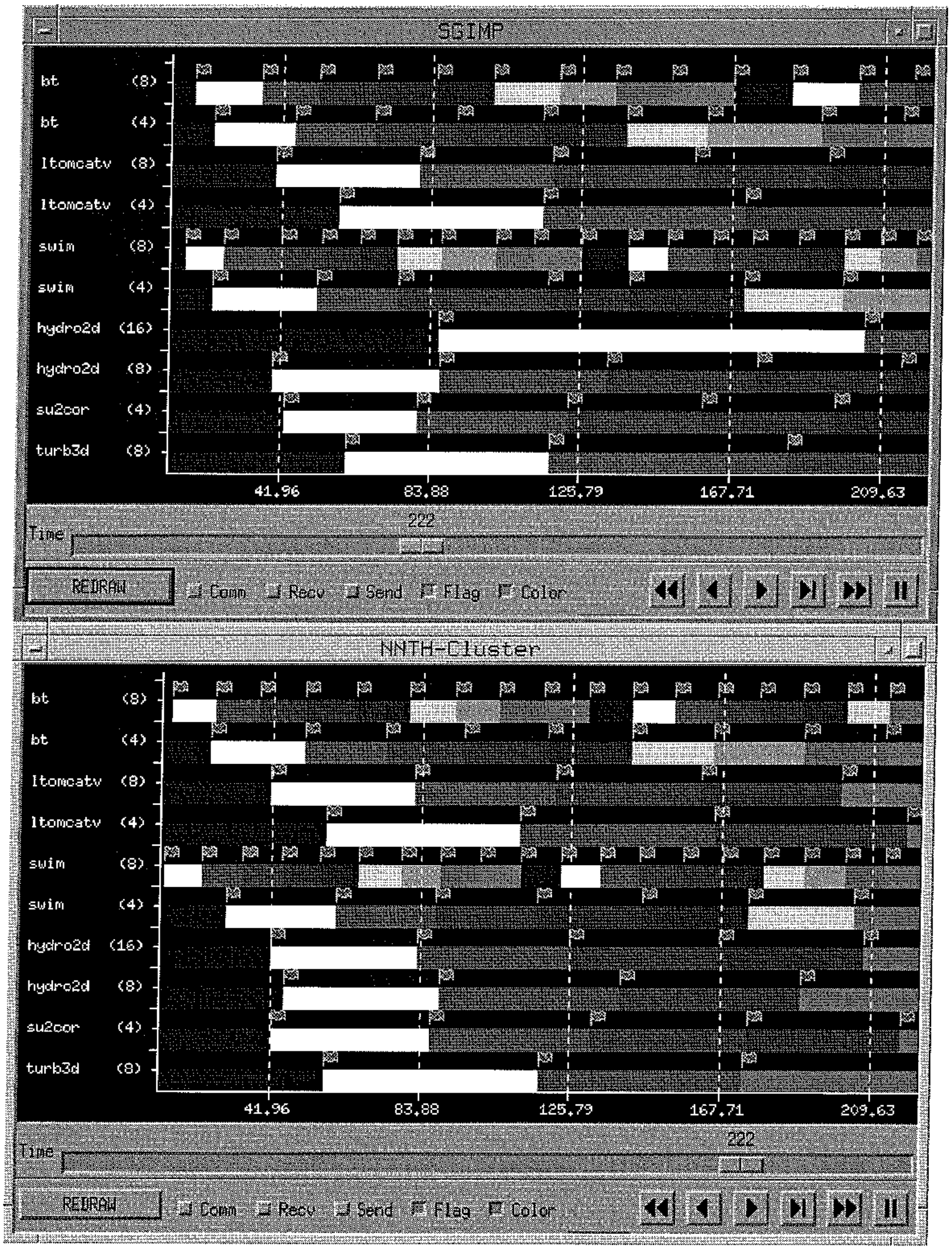


Figure 54: W13 execution on the SGIMP and NANOS environments

Application	Number of complete instances / Average execution time (in secs.) / Standard deviation					
	SGIMP			Cluster		
BT (8)	12	17	1.2	16	13	0.9
BT (4)	8	24	2.9	8	24	1.3
LTOMCATV (8)	4	39	1.4	4	40	0.6
LTOMCATV (4)	2	57	0.7	3	54	0.4
SWIM (8)	17	12	1.8	18	11	0.4
SWIM (4)	7	29	6.1	6	29	0.7
HYDRO2D (16)	1	118	-	4	41	1.0
HYDRO2D (8)	4	44	3.0	3	48	2.1
SU2COR (4)	4	38	2.3	4	44	1.0
TURB3D (8)	2	62	4.3	2	59	1.8

Table 18: Wl3 workload evaluation

The analysis of this workload is completed with two plots presenting the mapping of applications (consisting of several processes) to physical processors as it has been decided by each scheduling policy. Figure 55 (window SGI-IRIX-MP) shows the mapping done by the IRIX 6.5 kernel level scheduling and Figure 56 (window NNTH-CLUSTER) shows the mapping done by the NANOS Cluster policy. These figures present execution time in the *x* axis and each one of the 64 physical processors in the *y* axis. An horizontal line for each processor is displayed, possibly using different colors along the execution time. All processes belonging to the same application are displayed using the same color in the different physical processors. A change in the color of an horizontal line means that a different process of a different application has been scheduled in the corresponding processor. In addition, a vertical yellow line is displayed to indicate a process migrating from one processor to another.

As can be observed by comparing both figures, the execution in the NANOS environment is smoother. The number of times a processor decides to change the process it is executing is smaller than in the SGIMP environment and the number of process migrations is reduced. The reduction of the process migrations goes from 1,300 migrations in the SGIMP environment to 500 in the NANOS environment, achieving greater processor affinity. This also benefits to achieve global workload performance.

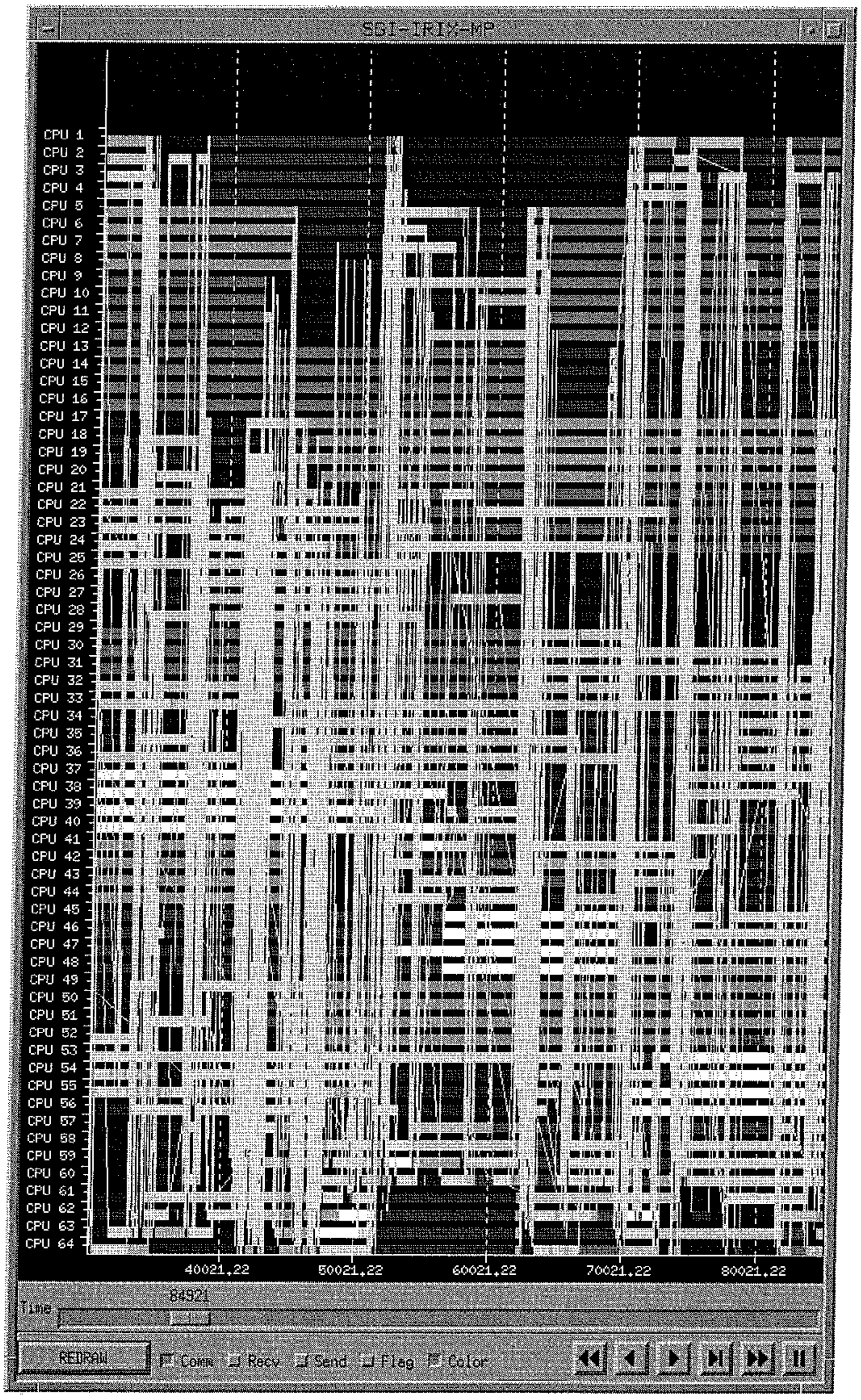


Figure 55: SGIMP kernel level scheduling for the wl3 workload

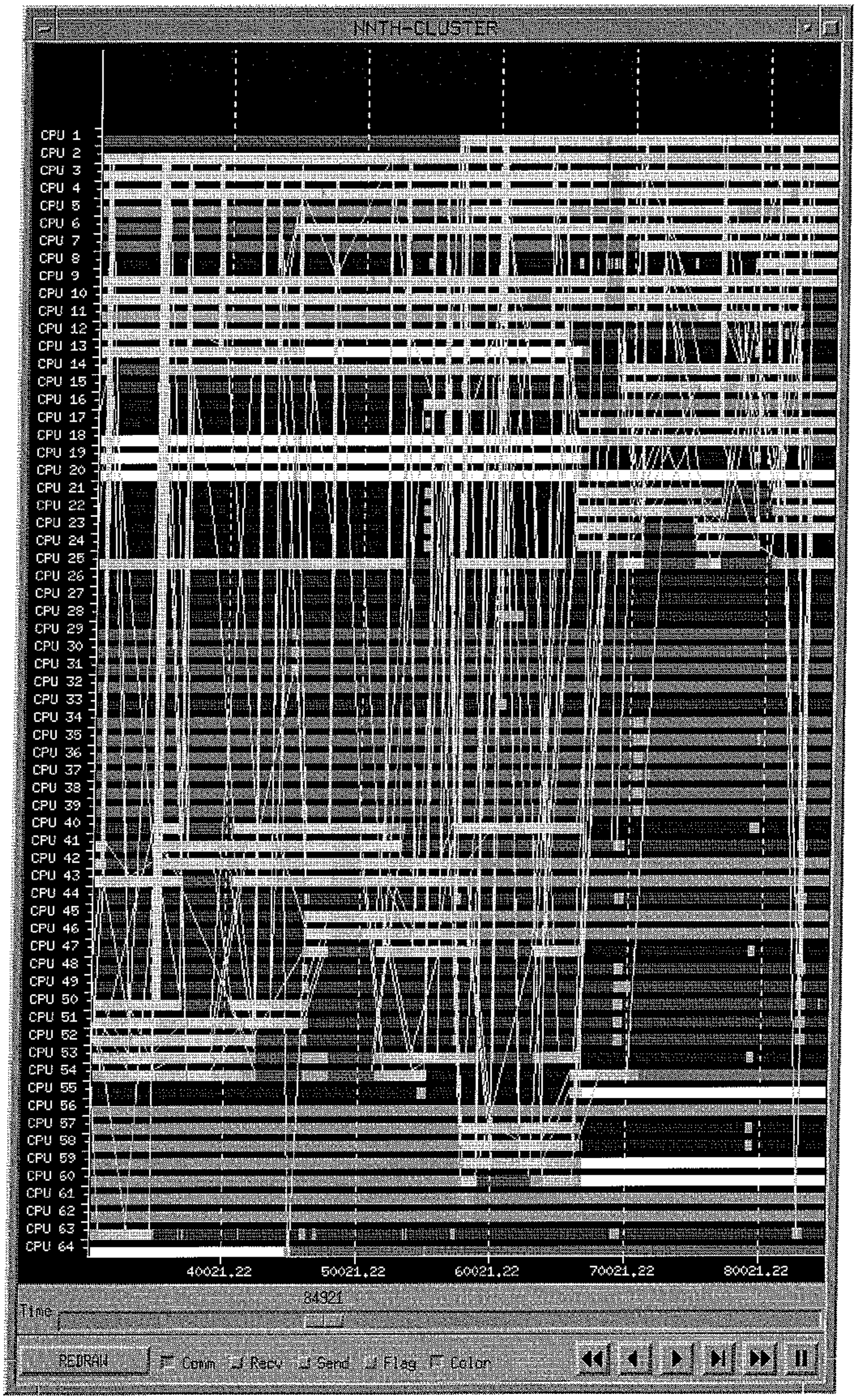


Figure 56: NANOS kernel level scheduling for the wl3 workload

9.4.4. Workload with larger requests

The objective of the fourth workload (wl4) is to show which is the behavior of the execution of a workload consisting of 6 different applications, each one requesting a number of processors (from 8 to 24). The workload composition is presented in Table 19 and the evaluation is presented in Figures 57-60 and in Table 20. Each application is requesting a number of processors in which its execution as an individual application achieves good speedup. The load of the system while running this workload ranges from 81 to 96 processors, on a 64 processor machine, depending on the current phase of the LTOMCATV application. The workload represents a common system load for this kind of machine.

Application	Processors requested
BT	16
LTOMCATV	1, 16
SWIM	16
TURB3D	24
HYDRO2D	16
SU2COR	8
TOTAL requests	**81-96**

Table 19: Wl4 workload composition

Figure 57 shows the behavior of the workload running in the MP environment, from the point where the second instance of the BT application is launched (left-most flag) till the termination of the eleventh instance of the BT application (right-most flag). During this period of time, the machine is heavy loaded with 81 to 96 processors requested and the scheduling frameworks and synchronization issues in the respective environments are stressed.

Figures 58 to 60 show the execution of the same workload in the NANOS environment, using the same time scale, and running under the Equipartition, Round-robin and Cluster policies. It can be observed that the BT and SWIM applications are clearly benefited in the NANOS environment. The LTOMCATV, TURB3D and HYDRO2D show smaller execution times also.

The good behavior obtained from the Equipartition, Round-robin and Cluster policies indicates that the NANOS scheduling framework is well designed to achieve high performance when running parallel workloads with a high degree of parallelism, outperforming the results obtained from the IRIXMP environment. Further research to determine which can be an optimal kernel-level scheduling policy for the NANOS environment is currently under development [117], and falls off the scope of this thesis.

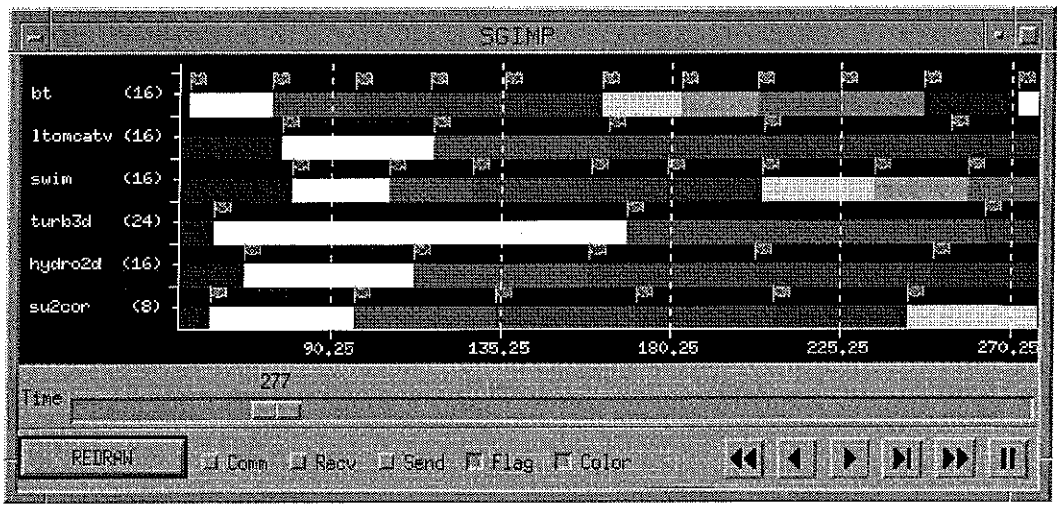


Figure 57: Wl4 workload execution on the SGIMP environment

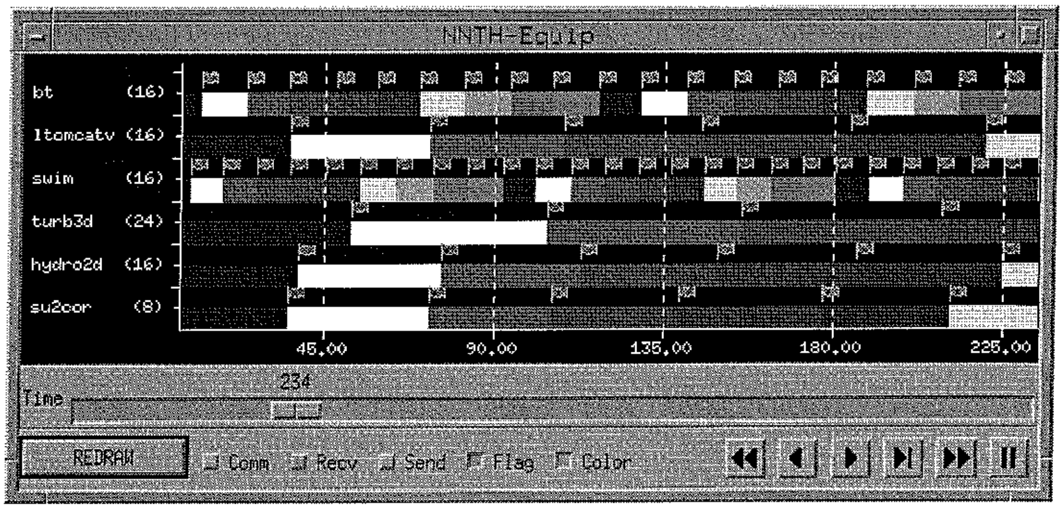


Figure 58: Wl4 workload execution on the NANOS environment (Equipartition policy)

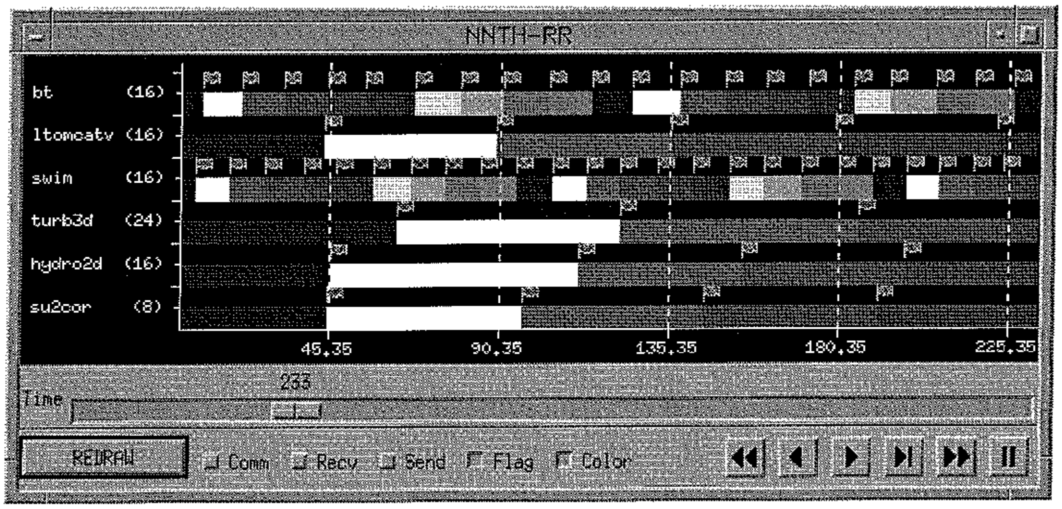


Figure 59: Wl4 workload execution on the NANOS environment (Round-Robin policy)

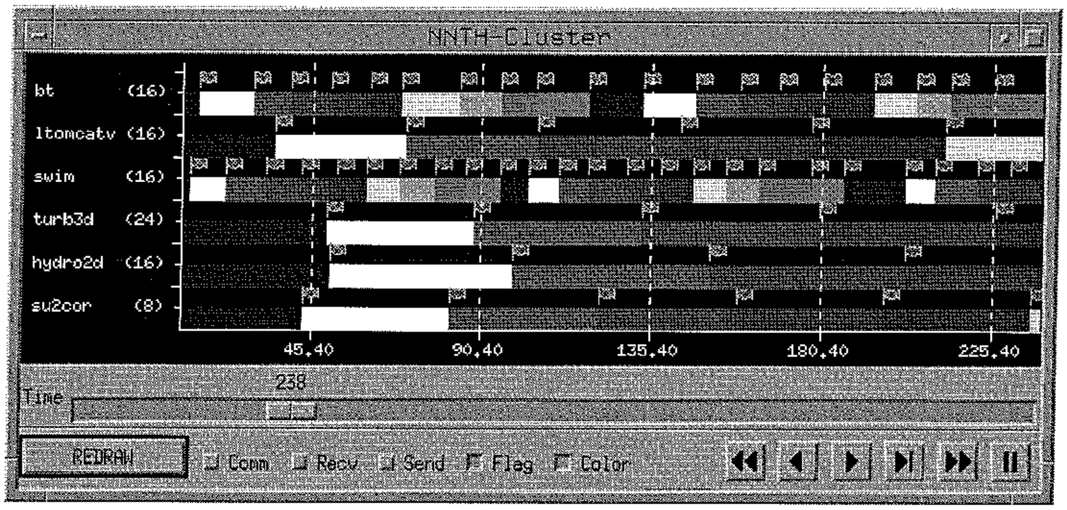


Figure 60: Wl4 workload execution on the NANOS environment (Cluster policy)

Table 20 shows the number of complete instances of each application executed, and the average execution time and standard deviation obtained for the different applications, for each one of the policies under evaluation. The number of instances executed for all the applications is greater when the workload runs on the NANOS environment than in the SGIMP environment. The benefits in the execution of the applications are also observed in that the standard deviation of the execution time of each instance is smaller when running in the NANOS environment. From the data presented, we conclude also that the SGIMP environment penalizes the applications which try to use more processors. This is because of synchronization is more difficult to achieve in these applications. For example, only two instances of the TURB3D are executed in the SGIMP environment, while the Equipartition and Cluster policies execute up to three and four complete instances, respectively. Both policies benefit from the NANOS scheduling framework, where the movements of processors between applications are communicated to the application and the user-level execution environment helps in solving the preemptions.

Application	**Number of complete instances / Average execution time (in secs.) / Standard deviation**											
	SGIMP			**Equip**			**RR**			**Cluster**		
BT	10	22	1.8	18	12	0.5	19	11	0.9	18	12	2.0
LTOMCATV	4	44	3.8	5	37	1.4	4	45	1.4	5	36	1.2
SWIM	7	26	3.8	24	9.0	0.3	23	9.3	0.6	23	9.4	2.0
TURB3D	2	102	7	3	52	0.8	2	61	2.1	4	44	3.0
HYDRO2D	4	46	1.4	5	37	0.9	3	51	10	3	51	1.6
SU2COR	5	37	1.0	5	35	2.3	3	49	2.4	5	39	1.0

Table 20: Results of the evaluation of the wl4 workload

In this workload, the Round-robin policy shows a flaw compared to both the Equipartition and Cluster policies. For example, it executes only two instances of the TURB3D application, due to the large number of context switches between applications caused by this policy. Context-switching applications at every time quantum breaks processor affinity and hurts the performance of the applications. From that fact, we conclude that a specific policy, designed for the Nano-Threads Programming Model, is of great importance.

9.4.5. Conclusions of the workload evaluation

After presenting the evaluation of four different workloads consisting of different combinations of applications, in this subsection we summarize the results obtained in them and we briefly extract some conclusions.

Table 21 shows the throughput and average response times obtained in each one of the workloads, for both the SGIMP and NANOS environments. The Cluster policy is used when collecting the results in the NANOS environment. The throughput is computed by counting all the complete instances shown in the corresponding figures. The average response time is computed as the weighted mean of the different average execution times, following the formula:

$$Average\ response\ time = \frac{\sum_{i=1}^{nappls} avgtime_i \cdot instances_i}{Throughput}$$

Workload	Ref. to figure		Throughput		Average response time (in seconds)	
	SGIMP	NNTH-Cluster	SGIMP	NNTH-Cluster	SGIMP	NNTH-Cluster
Wl1	Fig. 52	Fig. 52	44	44	34.5	35.6
Wl2	Fig. 53	Fig. 53	20	20	9.6	9.4
Wl3	Fig. 54	Fig. 54	61	68	26.9	24.9
Wl4	Fig. 57	Fig. 60	32	58	36.0	19.6

Table 21: Throughput and average response time (in s.) obtained in the workload execution

Comparing the results obtained from the different workloads, we conclude that the NANOS execution environment is able to achieve at least the same throughput and similar average response times for the two first workloads. These workloads have been used to tune the execution of the applications in the SGIMP environment.

After that, and more important, the NANOS environment effectively shows better behavior in common heterogeneous workloads consisting of several applications, requesting a different number of processors. In the third workload (wl3), the throughput achieved by the NANOS environment is 10% better than in the SGIMP environment and the average response time is 8% better. In addition, when the applications are allowed to exploit parallelism, requesting the number of processors which provide good speedup (workload wl4), the throughput is increased by 80% and the average execution falls to nearly half in the NANOS environment.

The NANOS execution environment reduces the need of fine-tuning of individual applications, providing an environment which is already tuned for the applications, as they are partitioned and parallelized by the compiler. The resulting execution is smoother, providing lower standard deviations in the execution time of several instances of the same application. The user will observe more predictability in the execution time of its applications. Finally, but not least important, we have observed that the SGIMP environment is unable to guarantee a constant allocation of processors to each application, resulting in a problem similar to a priority inversion, when applications requesting more processors run slower than applications requesting less processors. This problem is solved for applications running in the NANOS environment.

Chapter 10. Conclusions and Future Work

Abstract

This chapter presents the conclusions obtained from this thesis and the work we plan to do in the future to continue the research on parallel execution environments.

Barça - Brasil 2 - 2
... while writing the conclusions...
April 28, 1999

10.1. Conclusions of this thesis

After working during the last years in the development of this thesis, one has to think about which the initial goals were and which degree of achievement has been reached for them. The initial global objective of providing an efficient and effective support for multi-user parallel processing in shared-memory multiprocessors has been achieved. The design and development of the NANOS execution environment presented in this thesis demonstrates that point. The NANOS environment is efficient. It provides similar performance than the SGIMP parallel execution environment. By using an standard way of expressing the parallelism, we also achieve the effectivity claimed in the global objective. Existing and new applications coded in a standard way run on the provided environment. In addition, the NANOS environment integrates several topics currently under research and not present in commercial environments. Applications benefit from exploiting multiple levels of parallelism and the improved cooperation with the operating system. The main objective was decomposed in the introduction of this work in several sub-objectives. They are now checked and the results obtained are explained.

10.1.1. Supporting the Nano-Threads Programming Model

The work done in this thesis takes the Nano-Threads Programming Model as a basis for the development of the NANOS parallel execution environment. The NPM introduces a hierarchical decomposition of the applications in parallel tasks, through the Hierarchical Task Graph (HTG) structure. The approach taken translates each parallel task in a parallel function, to be executed through a user-level thread (nano-thread). Tasks are related through precedences in the HTG. Nano-threads have been designed to support the representation of the HTG, including precedences among them. Compound tasks contain further parallelism to be exploited. Nano-threads instantiating compound tasks are allowed to spawn further parallelism. The resulting environment supports the execution of arbitrarily unstructured and deep HTG structures.

During this work, it has been of great importance to understand the kind of HTG structures that usually represent parallel applications and how the compound tasks are usually organized, in order to provide the correct support for multiple levels of parallelism. This understanding has allowed us to design the NthLib interface to map general HTG structures to parallel code.

A contribution of this thesis deals with the mapping of different application-level scheduling policies to the NPM. They include the static, dynamic, guided self-scheduling and trapezoid scheduling policies. In addition, our proposal is the adaptive-size chunking scheduling policy. All the dynamic policies can also be combined with factoring (based on nano-thread bursts), which allows us to provide high adaptivity to an environment where the number of processors allocated dynamically changes.

10.1.2. Multiple levels of parallelism and processor grouping

This work demonstrates that providing efficient exploitation of multiple levels of parallelism is possible and makes important contributions both in the design and implementation phases of execution environments supporting this feature.

Providing local address spaces. Local address spaces are needed to support privatization of variables at the outer levels of parallelism. In our approach, local variables to be used by the inner levels of parallelism are allocated in the current nano-thread stack. They are always passed as parameters to these further levels. They can be passed by reference or through privatization. The nano-thread creation primitives support this local address space management. After the evaluation of this mechanism, the conclusion is that passing this extra number of parameters compared with other parallel execution environments is not introducing a noticeable overhead.

Processor grouping. Merging multiple levels of parallelism with data-locality issues, results in the proposal for processor groups. When allowing to spawn multiple levels of parallelism, the application decides which processors spawn the parallelism (act as masters) and which ones are merely executing work (act as slaves). The application reflects its HTG structure in the generation of the parallelism, giving more importance to some parts of the parallel execution by assigning more processors to them.

The execution environment provided by NthLib provides tools to drive processors to execute at different parts of the application as a processor group. The advantage of this approach is that each processor accesses a larger portion of the data structures involved in the parallel computation, reducing the conflicts between processors, such as false sharing.

Implementation issues. The implementation of the support for multiple levels of parallelism is based on local ready queues. Each processor owns a local queue and extracts work from it. All processors can generate work on all queues. Each queue acts as a buffer to save the work that the application has assigned to the associated processor. The contribution of this thesis is that local queues are also used to establish processor groups when the application wants to use them.

10.1.3. Fine-grain parallelism

In this thesis, we have been looking for the limits in the exploitation of fine-grain parallelism in shared-memory multiprocessors. Current SMP and CC-NUMA architectures allow to break the barrier at 12-16 processors and machines containing a larger number of processors are available. The larger the machines, the more difficult to achieve high performance, taking advantage of a large number of processors, in individual applications due to the complexity of finding a good data distribution. The techniques for spawning and joining parallelism developed inside this work support a granularity as fine as other parallel execution environments, providing, at the same time, higher functionality.

Not only the approach of multiple levels of parallelism has been extended to support processor grouping, but also a contribution of this work is the integration of two fork/join techniques in the same execution environment. The first mechanism, based on nano-threads, is to be used by the outer levels of parallelism. It provides all the functionality needed for supporting multiple levels of parallelism. The second mechanism, based on work descriptors, is to be used by the inner-most level of parallelism. It provides high performance and limited functionality. Combining both mechanisms has been the key to allow the exploitation of fine grain parallelism and achieve high performance when exploiting multiple levels of parallelism.

10.1.4. Cooperation between user and kernel levels

The proposals done in this thesis for user-kernel cooperation have been proven to be useful to provide a smooth kernel-level scheduling, in accordance with the needs of each application. Joining several techniques for informing the applications about kernel-level scheduling decisions and providing a small amount of time to the applications to answer to the changes (the grace time), the execution is improved.

Malleable applications. Applications coded in the NANOS environment tend to become malleable. They request a number of processors for working, possibly reconsidering it during execution. And they adapt to the kernel conditions, by running on the number of processors the kernel allocates to them. The evaluation done in the thesis demonstrates that this kind of applications facilitates kernel-level scheduling, improving the throughput of the system.

Kernel-level scheduling framework. This thesis proposes a new kernel-level scheduling framework, where the application becomes the scheduling target. This means that processors are allocated to applications and try to select work form that preferred application first. Only in case the application releases the processor, it is allowed to change its preferred application.

A contribution of this thesis is that the scheduling policy can decide, not only the allocation changes for the current quantum, but it can also provide a *work list* of applications which are going to receive processors during the current quantum in case other applications release processors. For this reason, changing the preferred application of a processor is done in two different situations: When the operating system decides a new reallocation of processors to applications, or when a processor is released from an application and finds work in the *work list*.

The evaluation of the execution of application workloads demonstrates that the number of process migrations and context switches is reduced when using our kernel scheduling framework, compared with the results obtained in a commercial system.

10.1.5. Conclusions taken from the evaluation

We have evaluated the NANOS execution environment through the evaluation of the user and kernel levels. The experiments were classified as follows:

- Evaluation of the overhead introduced by the user-level execution environment on the parallel applications.
- Evaluation of the performance of individual applications on a dedicated machine.
- Evaluation of workload performance.

From the evaluation of the overhead introduced by the user-level execution environment in the parallel applications, we conclude that the number of remote memory accesses is limiting the performance of the nano-threads primitives. Nevertheless, the overhead is low enough to allow a good exploitation of multiple levels of parallelism, due to the integration of the mechanisms based on work descriptors.

From the evaluation of individual applications, running in a dedicated mode, we conclude that the NANOS environment is comparable to the SGIMP environment when exploiting a single level of parallelism. In some cases, the results are even slightly better. When exploiting multiple levels of parallelism, the evaluation using complete applications demonstrates that the integration of two different mechanisms for spawning parallelism (based on nano-threads and work descriptors, respectively) achieves better performance than exploiting a single level. In this way, applications can take advantage of being executed on a larger number of processors.

Finally, from the evaluation of different workloads running in both the SGIMP and NANOS environments, we conclude that the NANOS environment effectively shows better behavior in common heterogeneous workloads consisting of several applications, requesting a different number of processors. When the applications are allowed to exploit a high degree of parallelism, requesting the number of processors which provide good speedup, the throughput is increased by 80% and the average execution falls to nearly half in the NANOS environment, compared with the SGIMP environment running the same applications.

The NANOS execution environment reduces the need for fine-tuning of individual applications, providing an environment which is already tuned for the applications, as they are partitioned and parallelized by the compiler. The resulting execution is smoother, providing lower standard deviations in the execution time of several instances of the same application. The user will observe more predictability in the execution time of its applications. In addition, the design of the kernel scheduling framework is validated observing that the NANOS environment guarantees that requesting more processors ensures a faster execution.

10.2. Future work

Several issues remain opened and can be further developed as a continuation of this thesis. They are outlined in the following subsections.

Extensions to the OpenMP directives. Along this thesis we have used a set of extensions to the OpenMP directives. These extensions are part of another work, currently under development. We are providing a complete parallel execution environment on top of which new proposals of directives can be easily designed, implemented and tested.

Programmers, when parallelizing applications, search for improved expressiveness and easy of use. The goal is to continue developing directive extensions, enriching the expressiveness of the current ones. The main open question is how the groups of processors are defined when using these directives. This is important to help programmers to exploit multiple levels of parallelism in applications.

Improvements in the definition of the directives can lead to changes in the user-level execution environment interface. The NthLib interface proposed in this thesis is opened to improvements to better adapt to code generation from different proposals with respect directives.

Take advantage of the work descriptor experience. The development of two different, and integrated, mechanisms (nano-threads and work descriptors) for supporting multiple levels of parallelism has been one of the approaches taken in this thesis which has been carried out with more discussions during its development. Although we finally decided to proceed with the distinction, it is not clear that both mechanisms can not be joined, providing a common interface. The important question is whether the resulting mechanism could offer the same functionality than nano-threads and the same performance than work descriptors. We plan to continue studying how this can be achieved, re-designing and evaluating the new proposals.

Introducing automatic load balancing mechanisms. In this work, data locality has been given more importance than load balancing. For irregular applications, where the amount of work can not be distributed evenly among the processors because the amount of work is not known when spawning parallelism, load balancing is of importance. Dynamic Bisectioning Scheduling (DBS) has already been implemented inside NthLib and it is currently under evaluation as an open issue, to be completed in the near future.

Improving the user-kernel cooperation. Although we think we are providing a good and well-tuned mechanism for cooperation between the user and kernel levels, an open question is whether the applications can help the kernel-level scheduling. The goal is to allow each individual application to dynamically detect at run-time which is the number of processors from which it can obtain better performance.

Within this approach, the application itself is able to decide how many processors to request, given the computed performance and the maximum number of processors offered by the operating system. Again, this open issue will improve the interface between the user and kernel levels for better individual and workload performance.

Kernel-level framework. The evaluation of the proposals at the kernel level will be completed in the future. We want to evaluate the benefits which can be obtained from scheduling in advance, taking into account the information supplied by the applications, and planning the processor allocations for the future to avoid abrupt changes in the allocation of processors. This will allow to extensively apply the grace time to allow the applications to completely avoid processor preemptions.

Kernel-level scheduling. Another open question is the search for a more specific scheduling policy, specifically designed for the Nano-Threads Programming Model, which can improve the results of application workloads. The idea here is to offer two level scheduling policies, where the machine is first partitioned at the higher level, using Dynamic Space Sharing Scheduling (DSS), among the executing applications taking into consideration the processor requirements of each application, as well as the overall system workload. At the lower level, Selective Scheduling is applied in order to promote processor affinity. This issue has already provided good results [118][117], within the work done with the High Performance Information Systems Laboratory in the University of Patras (Greece).

Distributed memory architectures. After having worked with SMP and CC-NUMA machines, an open line consists of taking advantage of the experience to port a similar parallel execution environment to distributed memory machines, including clusters of SMP's, and networks of workstations. We think that the notion of group of processors can be very useful to provide high performance to applications running on such machines, with the ability of exploiting multiple levels of parallelism.

Real applications. Another open issue consists of getting real applications, study them using the experience got from this work and determine whether they can take advantage of any of the achievements of this thesis. The main goal would be to evaluate how many applications can take profit from the exploitation of multiple levels of parallelism. As a result, the guidelines for the exploitation of such kind of parallelism would be clarified for programmers to use it.

Porting to other architectures / operating systems. The NANOS execution environment is being ported to other architectures and operating systems. It has already been tested on the Alpha AXP architecture/Digital UNIX, in SPARC/Solaris and in Pentium/Linux. Further portings include Pentium/Windows NT.

Influence commercial execution environments. Some of the ideas discussed and implemented in our prototype can influence commercial operating systems and parallelizing environments. In this direction, we have recently visited Silicon Graphics and Kuck & Associates, Inc. (KAI). Our environment has been presented to them and they have expressed some interest in being in touch to continue sharing points of view. For example, we have started to experiment on running SGIMP-parallelized applications on top of the NANOS environment. SGI commercial compilers are able to get optimum performance from the SGI architecture and our execution environment can exploit multiple levels of parallelism. Another experience is to modify the dynamic adaptability of the SGI applications to the suggested number of processors by using our global reallocation policies.

References

[1] M. Acceta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M.Young, "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the USENIX Conference, July 1986.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine: Architecture and Performance", Proceedings of the 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995.

[3] S. Aldomà, "Mecanismes en el Kernel d'un Sistema Operatiu pel Control de l'Execució d'Aplicacions Paral.leles", M.S. Thesis, Departament d'Arquitectura de Computadors - UPC, February 1999, written in Catalan.

[4] T. E. Anderson, "FastThreads User's Manual", Department of Computer Science and Engineering, University of Washington, January 1990.

[5] 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.

[6] T. E. Anderson, E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", Department of Computer Science and Engineering, University of Washington, September 1989.

[7] 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 Politecnica de Catalunya, Technical Report: UPC-DAC-1998-48, November 1998.

[8] 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. Language and Compilers for Parallel Computing (LCPC), Minneapolis, USA, August 1997.

[9] M. Bach, "The Design of the UNIX Operating System", Prentice-Hall, 1986.

[10] 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.

[11] J. Barton, N. Bitar, "A Scalable Multi-Discipline Multiple-Processor Scheduling Framework for IRIX", Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing", April 1995.

[12] C. J. Beckmann, "Hardware and Software for Functional and Fine Grain Parallelism", Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1993.

[13] C. J. Beckmann, C. D. Polychronopoulos, "Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs", CSRD Technical Report 1207, August 1992.

[14] B. Bershad, E. Lazowska, H. Levy, "Presto: A System for Object-oriented Parallel Programming", Software - Practice and Experience, vol. 18, no. 8, pp. 713-732, 1988.

[15] D. L. Black, "Scheduling and Resource Management Techniques for Multiprocessors", Ph.D. Thesis, CMU-CS-90-152, Carnegie Mellon University, July 1990.

[16] R.D. Blumofe, “Managing Storage for Multithreading Computations”, M.S. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou, “Cilk: An Efficient Multithreaded Runtime System”, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ‘95), Santa Barbara, California, July 19-21, 1995.

[18] J. Boykin, D. Kirschen, A. Langerman, S. LoVerso, “Programming Under Mach”, Addison-Wesley Publishing Company, 1993.

[19] D.R. Butenhof, “Programming with POSIX threads”, Professional Computing Series, Addison-Wesley, ISBN 0-201-63392-2, May 1997.

[20] C.R. Calidonna, M. Giordano, M. Mango Furnari, “A Graphic Parallelizing Environment for User-Compiler Interaction”, Proceedings of the 13th ACM International Conference on Supercomputing (ICS’99), Rhodes, Greece, June 1999.

[21] J. Caubet, “Paral.lelització Automàtica i Serveis de l’Entorn del Sistema”, M.S. Thesis, Departament d’Arquitectura de Computadors - UPC, February 1999, written in Catalan.

[22] R. Chandra, “COOL User Manual”, Computer Systems Laboratory, Stanford University, November 1992.

[23] R. Chandra, A. Gupta and J. L. Hennessy, “Data Locality and Load Balancing in COOL”, Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (PPoPP), pp. 249-259, May 1993.

[24] R. Chandra, A. Gupta, J. L. Hennessy, “Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language”, Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, February 1992.

[25] A. Charlesworth, “STARFIRE: Extending the SMP Envelope”, IEEE Micro, Jan/Feb 1998.

[26] A. Chien, U. Reddy, “ICC++ Language Definition”, Concurrent Systems Architecture Group Memo, http://www-csag.cs.uiuc.edu/, 1995.

[27] A. Chien, V. Karamcheti, J. Plevyak, “The Concert System - Compiler and Runtime Support for Efficient Fine-grained Concurrent Object-oriented Programs”, Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. UIUCDCS-R-93-1815, 1993.

[28] J. Chow, W. L. Harrison III, “Switch Stacks: A Scheme for Microtasking Nested Parallel Loops”, Center for Supercomputing Research and Development (CSRD), University of Illinois at Urbana-Champaign, 1990.

[29] E. C. Cooper and R. P. Draves, “CThreads”, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1998.

[30] D. Cortessi, A. Evans, W. Ferguson and J. Hartman, “Topics in IRIX Programming”, Doc. num. 007-2478-004, Silicon Graphics, Inc., http://techpubs.sgi.com, 1996.

[31] D. Cortessi, A. Evans, W. Ferguson and J. Hartman, “Topics in IRIX Programming”, Doc. num. 007-2478-006, Silicon Graphics, Inc., http://techpubs.sgi.com, 1998.

[32] 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.

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

[34] Data General Corporation, "AViiON AV 25000 ccNUMA Server", White paper 012-005167-00, http://www.dg.com/aviion/html/av_25000_enterprise_server.html, October 1998.

[35] Data General Corporation, "Data General's NUMALiiNE Technology: The Foundation for the AV 25000 Server", White paper 012-005119-01, http://www.dg.com/about/html/av25000_foundation.html, October 1998.

[36] J. M. Denham, P. Long, J. A. Woodward, "DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation", Digital Technical Journal, vol. 6, no. 3, Summer 1994.

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

[38] Digital Equipment Corporation / Compaq Computer Corporation, "AlphaServer GS60/GS140 and 8200/8400 Systems", technical summary, http://www.digital.com/alphaserver/products.html, 1999.

[39] Digital Equipment Corporation / Compaq Computer Corporation, "Digital UNIX: Assembly Language Programmer's Guide", Maynard, Massachusetts, March 1996.

[40] Digital Equipment Corporation / Compaq Computer Corporation, "Digital UNIX: Calling Standard for Alpha Systems", Maynard, Massachusetts, March 1996.

[41] Digital Equipment Corporation / Compaq Computer Corporation, "Digital UNIX: Guide to DECthreads", Maynard, Massachusetts, December 1997.

[42] Digital Equipment Corporation / Compaq Computer Corporation, "Digital UNIX: Programmer's Guide", Maynard, Massachusetts, March 1996.

[43] J.H. Edmondson, P. Rubinfeld, R. Preston, V. Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor", IEEE Micro, 15(2), pp. 33-43, April 1995.

[44] D. R. Engler, G. R. Andrews and D. K. Lowenthal, "Filaments: Efficient Support for Fine-Grain Parallelism", Technical Report 93-13a, Department of Computer Science, University of Arizona, Tucson, 1993.

[45] ESPRIT, European Union Information Technologies Programme, http://www.cordis.lu/esprit/home.html.

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

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

[48] M. Fillo, R.B. Gillet, "Architecture and Implementation of MEMORY CHANNEL 2", Digital Technical Journal, Vol. 9 No. 1, 1997.

[49] I. Foster, B. Avalani, A. Choudhary, M. Xu, "A Compilation System that Integrates High Performance Fortran and Fortran M", Proceedings of the Scalable High Performance Computing Conference, Knoxville (TN), May 1994.

[50] I. Foster, D. R. Kohr, R. Krishnaiyer, A. Choudhary, "Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface", Proceedings of the Supercomputing '96, November 1996.

[51] V. W. Freeh, G. R. Andrews, "fsc: A Sisal Compiler for Both Distributed- and Shared-Memory Machines", Technical Report 95-01, Department of Computer Science, University of Arizona, Tucson, February 1995.

[52] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, V. Sunderam, "PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing", MIT Press, 1994.

[53] M. Gil, X. Martorell, N. Navarro, "eXc: Scheduler Activations on Mach 3.0", Proceedings of the Seventh IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems (PDCS '95), Washington D.C., October 1995.

[54] M. Giordano, M. Mango Furnari, "HTG-DT 5.0 User Manual", TR 131/98/IC, Istituto di Cibernetica CNR, Arco Felice, Naples, Italy, March 1999.

[55] M. Girkar and C. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs", IEEE Transactions on Parallel and Distributed Systems, vol. 3(2), pp. 166-178, March 1992.

[56] M. Girkar, M.R. Haghighat, P. Grey, H. Saito, N. Stavrakos, C.D. Polychronopoulos, "Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems", Intel Technology Journal, Q1 issue, February 1998.

[57] T. Gross, D. O'Halloran and J. Subhlok, "Task Parallelism in a High Performance Fortran Framework", IEEE Parallel and Distributed Technology, vol. 2, no. 3, Fall 1994.

[58] 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.

[59] M. R. Haghighat, C. D. Polychronopoulos, "Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs", Proceedings of the Sixth Annual Languages and Compilers for Parallelism Workshop, 1993.

[60] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, December 1996.

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

[62] Hewlett Packard, "HP 9000 T-Class Servers", http://www.datacentersolutions.hp.com/2_1_20_index.html, 1999.

[63] Hewlett Packard, "HP 9000 V-Class V2500 Enterprise Server", White paper, http://www.datacentersolutions.hp.com/v2500_wp.html, 1999.

[64] IEEE Computer Society, "IEEE Standard for Scalable Coherent Interface (SCI)", IEEE Std 1596-1992, New York, August 1993.

[65] IEEE Computer Society, "POSIX System Application Program Interface: Threads Extension [C Language] POSIX 1003.4a Draft 8". Available from the IEEE Standards Department.

[66] Intel Corporation, "Pentium Pro Family Developer's Manual", December 1995.

[67] V. Karamcheti, "Run-Time Techniques for Dynamic Multithreaded Computations", Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1998.

[68] D. Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", University of Washington, Technical Report UWCSE 93-05-06, 1993.

[69] C. H. Koelbel, D. B. Loveman, R.S. Schreiber, G. L. Steele, M. E. Zosel, "The High Performance Fortran Handbook", Scientific Programming, 1994.

[70] C. Koppe, "Sleeping Threads: A Kernel Mechanism for Support of Efficient User Level Threads, Proceedings of the Seventh IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems (PDCS '95), Washington D.C., October 1995.

[71] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K.Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy, "The Stanford FLASH Multiprocessor", Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, April 1994.

[72] 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.

[73] S. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3 BSD UNIX Operating System", Addison-Wesley, 1989.

[74] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta and J. Hennessy, "The DASH Prototype: Implementation and Performance", Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

[75] 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.

[76] K. Loepere, "OSF Mach Kernel Interface", Open Software Foundation and Carnegie Mellon University, April 1993.

[77] K. Loepere, "OSF Mach Kernel Principles", Open Software Foundation and Carnegie Mellon University, April 1993.

[78] K. Loepere, "OSF Mach Server Library Interfaces", Open Software Foundation and Carnegie Mellon University, April 1993.

[79] K. Loepere, "OSF Mach Server Writer's Guide", Open Software Foundation and Carnegie Mellon University, April 1993.

[80] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace", Proc. of the 23rd Annual International Symposium on Computer Architecture, Philadelphia (USA), May 22-24, 1996.

[81] T.D. Lovett, R.M. Clapp, R.J. Safranek, "NUMA-Q: An SCI-based Enterprise Server", White paper, Sequent Computer Systems, Inc., 1996.

[82] D. K. Lowenthal, D. R. Engler, "Performance Experiments for the Filaments Package", Technical Report 93-26, Department of Computer Science, University of Arizona, Tucson, September 1993.

[83] E.P. Markatos, T. J. LeBlanch, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors", Proceedings of the Supercomputing '92, pp. 104-113, 1992.

[84] 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.

[85] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. Gonzàlez and J. Labarta, "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors", Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99), Rhodes, Greece, June 1999.

[86] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, "Analysis of Several Scheduling Algorithms under the Nano-Threads Programming Model", Proceedings of the International Parallel Processing Symposium (IPPS '97), Geneve, Switzerland, April 1997.

[87] 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.

[88] 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.

[89] 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.

[90] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard", The International Journal of Supercomputer Applications and High Performance Computing 8", 1994.

[91] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface", University of Tennessee, Knoxville, July 1997.

[92] J. E. Moreira, "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors", Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1995.

[93] S. Murer, J.A. Feldman, C.C. Lim, M.M. Seidel, "pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation", International Computer Science Institute, University of California at Berkeley, Technical Report 93-028, December 1993.

[94] 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.

[95] 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.

[96] V. K. Naik, S. K. Setia, M. S. Squillante, “Processor Allocation in Multiprogrammed Distributed Memory Parallel Computer Systems”, IBM Research Report RC 20239, October 1995.

[97] V. K. Naik, S. K. Setia, M. S. Squillante, “Scheduling of Large Scientific Applications on Distributed Memory Multiprocessor Systems”, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM Press, 1994.

[98] NANOS Consortium, “M1.D1: Nano-Threads: Programming Model Specification”, NANOS project deliverable, http://www.ac.upc.es/nanos, July 1997.

[99] NANOS Consortium, “M1.D2: Environment and Benchmark Selection”, NANOS project deliverable, http://www.ac.upc.es/nanos, December 1997.

[100] NANOS Consortium, “M2.D1: Port of Chorus”, NANOS project deliverable, http://www.ac.upc.es/nanos, April 1998.

[101] NANOS Consortium, “M2.D2: n-RTL Implementation”, NANOS project deliverable, http://www.ac.upc.es/nanos, April 1998.

[102] NANOS Consortium, “M2.D3: OS Implementation, Port to Chorus”, NANOS project deliverable, http://www.ac.upc.es/nanos, April 1998.

[103] NANOS Consortium, “M2.D4: RTL Calls Generator”, NANOS project deliverable, http://www.ac.upc.es/nanos, April 1998.

[104] NANOS Consortium, “M2.D5: Manual Parallelization”, NANOS project deliverable, http://www.ac.upc.es/nanos, April 1998.

[105] D. Nikolopoulos, E. D. Polychronopoulos, T. S. Papatheodorou, “Efficient Runtime Thread Management for the Nano-Threads Programming Model”, Proceedings of the IPPS/SPDP’98 Workshop on Runtime Systems for Parallel Programming, Lecture Notes in Computer Science Vol. 1388, Jose Rolim (Ed.), pp. 183-194, Orlando, Florida, USA, March/April 1998.

[106] D. Nikolopoulos, E. D. Polychronopoulos, T. S. Papatheodorou, “Enhancing the Performance of Autoscheduling in Distributed Shared Memory Multiprocessors”, Proceedings of the 4th International Euro-Par Conference, LNCS vol. 1470, pp. 491-501, Southampton, UK, September 1998.

[107] OpenMP Organization, “Fortran Language Specification, v 1.0”, www.openmp.org/openmp/mp-documents/fspec.ps, October 1997.

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

[109] 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.

[110] C. D. Polychronopoulos, “Control Flow and Data Flow Come Together”, Technical Report, Center for Supercomputing R&D, University of Illinois, November 1990.

[111] C. D. Polychronopoulos, “Multiprocessing versus Multiprogramming”, Proceedings of the International Conference on Parallel Processing, St. Charles IL, August 1989.

[112] C. Polychronopoulos, “*nano*threads: Compiler driven multithreading”, In 4th. International Workshop on Compilers for Parallel Computing, 1993.

[113] C.D. Polychronopoulos, D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", IEEE Transactions on Computers, C-36(12), December 1987.

[114] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung and D. A. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Proceedings of the 1989 International Conference on Parallel Processing (ICPP), St. Charles, Illinois, 1989.

[115] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, D. A. Schouten, "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran", Languages and Compilers for Parallel Computing, MIT Press, 1990.

[116] C. Polychronopoulos, N. Bitar and S. Kleiman, "nanoThreads: A User-Level Threads Architecture", Technical Report, Computer Science Research and Development, University of Illinois at Urbana-Champaign, CSRD TR-1297, 1993.

[117] 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.

[118] E.D. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. S. Papatheodorou, 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.

[119] M.L.Powell, S.R. Kleiman, S. Barton, D.Shah, D. Stein, M. Weeks, "SunOS Multi-thread Architecture", Proceedings of the USENIX Winter '91 Conference, Dallas, Texas, 1991.

[120] S. Ramaswamy, "Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations", Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1996.

[121] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, "Overview of the Chorus Distributed Operating System", Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures, April 1992.

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

[123] SUN Microsystems, "The Ultra Enterprise 10000 Server", Technical White Paper, 1997.

[124] SUN Microsystems, Inc., "Pthreads and Solaris Threads: A Comparison of two user level threads APIs", SunSoft, Revision A, May 1994.

[125] SUN Microsystems, Inc., "The SUN Enterprise Cluster Architecture", Technical White Paper, October 1997.

[126] H. Saito, "Multithreading Runtime Support for Loop and Functional Parallelism", M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1998.

[127] D. A. Schouten, "Efficient Scheduling of Parallel Tasks in a Multiprogrammed Environment", Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1995.

[128] Sequent Computer Systems, Inc., "NUMA-Q 2000 Model E300. Specification Summary", http://www.sequent.com/products/highend_srv/xeon_spec11_6.pdf, PD-1159, October 1998.

[129] Sequent Computer Systems, Inc., "Optimal Machine Architectures for Parallel Query Scalability: Sequent's Scalable Data Interconnect (SDI) Clusters", Whitepaper, PD-1131, May 1996.

[130] Sequent Computer Systems, Inc., "Sequent's NUMA-Q Architecture", Whitepaper, http://www.sequent.com/products/highend_srv/Numa_Arch_wp.pdf, PD-1124, June 1997.

[131] Sequent Computer Systems, Inc., "Sequent's NUMA-Q SMP Architecture", Whitepaper, http://www.sequent.com/products/highend_srv/NUMA_SMP_REV.pdf, PD-1134, June 1997.

[132] Silicon Graphics Computer Systems (SGI), "MIPSpro C and C++ Pragmas", Doc. num. 007-3587-001, http://techpubs.sgi.com, 1998.

[133] 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.

[134] Silicon Graphics Computer Systems (SGI), "MIPSpro Auto-Parallelizing Option Programmer's Guide", Doc. num. 007-3572-002, http://techpubs.sgi.com, 1998.

[135] Silicon Graphics Computer Systems (SGI), "MIPSpro Fortran 77 Programmer's Guide", Doc. num. 007-2361-006, http://techpubs.sgi.com, 1998.

[136] Silicon Graphics Computer Systems (SGI), "Origin and Onyx2 Theory of Operations Manual", Doc. num. 007-3439-002, http://techpubs.sgi.com, 1997.

[137] Silicon Graphics Computer Systems (SGI), "Origin2000 and Onyx2 Performance Tuning and Optimization Guide", Doc. num. 007-3430-002, http://techpubs.sgi.com, 1998.

[138] Silicon Graphics Computer Systems (SGI), "REACT Real-Time Programmer's Guide", Doc. num. 007-2499-006, http://techpubs.sgi.com, 1998.

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

[140] R. L. Sites, "Alpha Architecture Reference Manual", Digital Press, 1992, ISBN 1-55558-098-X.

[141] 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.

[142] D. Stoutamire, "pSather 1.0 Manual", International Computer Science Institute, University of California at Berkeley, Technical Report 95-058, October 1995.

[143] B. Stroustrup, "The C++ Programming Language", 2nd. Ed. Reading, MA., Addison-Wesley, 1991.

[144] The Portland Group Inc., "PGF77 Workstation Reference Manual", part num: 2310-990-990-0297, http://www.pgroup.com, 1997.

[145] The Portland Group Inc., "PGF77 Workstation User's Guide", part num: 2300-990-888-0297, http://www.pgroup.com, 1997.

[146] 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.

[147] 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.

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

[149] D. B. Wagner, B. G. Calder, "Portable, Efficient Futures", Department of Computer Science, University of Colorado at Boulder, Technical Report CU-CS-609-92, August 1992.

[150] B. Weissman, "Active Threads: an Extensible and Portable Light-Weight Thread System", International Computer Science Institute, University of California at Berkeley, Technical Report 97-036, September 1997.

[151] B. Weissman, "Performance Counters and State Sharing Annotations: a Unified Approach to Thread Locality", Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), 1998.

[152] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1990.