TESI DOCTORAL

DYNAMIC SCHEDULING OF PARALLEL APPLICATIONS ON SHARED-MEMORY MULTIPROCESSORS

Autor: XAVIER MARTORELL BOFILL

Abstract

Current parallel shared-memory multiprocessors are complex machines, where a large number of architectural aspects (both of processors and memory) have to be simultaneously addressed in order to achieve high performance. The quick evolution of the parallel machines has been followed by the evolution of the parallel execution environments. Both user and kernel execution levels are the subject of current research works that try to achieve a good cooperation between them and with the hardware. These were the trends we were interested in, when we started the development of this thesis.

The thesis proposes a parallel execution environment, where is possible to integrate new approaches both at user and kernel levels. The main goal is to achieve both efficient and effective support for multi-user parallel processing across a wide range of shared memory parallel computers, from small symmetric multiprocessors to high-end systems with globally shared address space.

The fundamental concept used in this work to achieve the global objective is the cooperation among the different execution levels in a parallel execution environment. The three levels we consider are: compiler/application, user-level execution environment and operating system. Cooperation must be bidirectional, from the application to the user-level environment and operating system and vice-versa.

The kind of work and the experimentation needed to complete the previous goal requires the availability of a whole parallel execution environment, from the operating system to the compiler/application level, including the user-level execution management. This is the reason why one of the main results of this thesis is a complete parallel execution environment (the NANOS execution environment) providing a compiler (the NANOS compiler), a user-level threads library (NthLib), an operating system interface and scheduling mechanisms (the NANOS O.S.) and the definition of how the three levels are going to cooperate.

Our starting point is the Nano-Threads Programming Model. Under NPM, the compiler decomposes the application in a hierarchical graph of parallel tasks. From this graph, the compiler generates parallel code using the services provided by a threads package interface. During code generation, the compiler statically determines the finest granularity of parallel tasks worth to be exploited, having in mind the efficiency of the user-level package implementation. Using the threads package and operating system interfaces at run-time, the program is also able to group tasks in order to use a coarser level of granularity, adapted to the actual system conditions. The hierarchy of parallel tasks allows us to experiment with the exploitation of multiple levels of parallelism.

There are several mappings among the application, run-time and operating system levels which are the key to provide high performance:

- The application is responsible for the mapping of application-level tasks to user-level threads supported by the threads package.
- The user-level threads package is responsible of the mapping of user-level threads to kernel-level threads (virtual processors), provided by the operating system.
- The operating system is responsible of the mapping of kernel-level threads to processors.

The objective of the user-level threads package interface is to support general, multi-level, unstructured and fine-grained parallelization of applications. The compiler can then represent successor and predecessor relationships between the application tasks, including the ability of driving processors to execute different portions of the application. Fine-grain parallelism is supported through mechanisms to efficiently schedule parallel loops.

The objective of the kernel-level interface is to provide a new, highly dynamic, space/time-sharing environment using two levels of kernel scheduling. The kernel offers a set of scheduling policies and the system manager can decide which one is active, considering the workload. The active kernel-level scheduling policy implements the first scheduling level. Low-level management involving individual processors and the specific work they have to perform becomes the second scheduling level. The motivation for these two levels of scheduling is that, in our opinion, the application should be the scheduling target for the first level and the individual processes the scheduling target for the second level. The kernel-level interface gives support to both the application and the user-level threads package. The kernel-level interface allows the application to dynamically request and release processors, indicating the amount of parallelism that it can exploit. The application can also check the number of processors currently allocated and can spawn the parallelism accordingly. The user-level threads package uses the kernel interface to ensure that the application proceeds with an execution as smooth as possible, even when the operating system moves some processors away from the application.

The evaluation of the NANOS parallel execution environment has been done on a Silicon Graphics Origin2000 machine. The evaluation consists of three different types of experiments to evaluate the overhead introduced by the execution environment, the performance obtained running applications in dedicated mode and the performance obtained running parallel workloads. The NANOS and the SGIMP native execution environments are compared in these experiments.

The results obtained can be summarized as follows:

- The NANOS execution environment provides higher functionality, and enables the support of unstructured and multi-level parallelism. The runtime overhead is comparable to the overhead introduced by the SGIMP commercial environment in Origin2000 machines.
- Existing applications can benefit from the exploitation of multiple levels of parallelism. The improvement in performance of these applications ranges from 30% to 65% with respect the single level version.
- The NANOS environment effectively shows better behavior in common heterogeneous workloads consisting of several applications, requesting a different number of processors. The improvement in performance ranges, in this case, from the 10% to a 80% of the achieved throughput, comparing with the same workloads executed in the SGIMP execution environment.

Dedicatòria

Als meus pares.

ACKNOWLEDGEMENTS

A lot of people has made possible this work. I would like to express my more frankly gratitude to all of them for being there, working with me, helping me.

First of all, I want to thank Jesús Labarta and Nacho Navarro for being my advisors during this work. I started working in operating systems and microkernels with Nacho Navarro in my last undergraduate year. After my graduation, I continued working in the operating systems group in the Computer Architecture Department of the Universitat Politècnica de Catalunya. After some time, Jesús Labarta call us for starting the preparation of the NANOS Esprit Project. From Jesús Labarta, I think I've learned to organize things in a practical way. From Nacho, to analyze and abstract new ideas relating them to well-known concepts.

I want to express my gratitude to the rest of colleagues participating in the NANOS project, from the Computer Architecture Department: Eduard Ayguadé, for his work at the compiler level, in the definition of the interface of NthLib (the name was given by him), parallelizing applications, and for his comments about the organization of this document, along the writing. Toni Cortés and Julita Corbalán for their participation in the definition and design of the NANOS operating-system interface. Also to Julita Corbalán for her dedication to parallelize applications. To Marc Gonzàlez and José Oliver ("Oli") for their work in the design and implementation of the OpenMP directives in the NANOS compiler. To Jordi Caubet and Sergi Aldomà for their dedication to the study of the internals of the SGIMP library.

The NANOS project started October 1996. In addition to UPC, the other participants are the University of Patras (Greece), the Consiglio Nazionale delle Ricerche (Naples, Italy) and Pallas (Germany). I would like to thank all our partners for their participation during this work, specially to Dimitris Nikolopoulos and Lefteris Polychronopoulos, for the discussions on the kernel-level framework and scheduling policies and for their dedication during our two work-weeks in Patras in 1997 and 1998. I cannot forget to thank Constantine Polychronopoulos, from the University of Illinois at Urbana-Champaign, for inspiring the NANOS project and provide us with the opportunity to work on this topic. I would also thank to David Craig for his dedication during my two visits to the University of Illinois in 1997 and recently, in 1999.

Thanks to Mateo Valero, the head of the High Performance Computing Group, for his support, and to all my colleagues from the Computer Architecture Department. Specially to the group researching on micro- and exokernels: Ernest Artiaga, Yolanda Becerra, Marisa Gil and Albert Serra. Special thanks also to Toni Juan, my office-mate during all these years, for sharing his research time with me and for the discussions about cache memories. To Susana Moreno, for using the NANOS environment on Alpha machines and for the work she did solving problems with this implementation. To Daniel Ortega and Iván Martel. And to Dolors Royo, Roger Espasa, Marta Jiménez, Pepe González, Gianluca Cornetta and Enric Fontdecaba.

I'm also grateful to the LCAC and CEPBA staffs, specially Oriol Riu ("Uri") for his help during the periods in which I've been intensively doing experiments in *karnak* (Origin2000 system). Without his support for the experiments in dedicated mode, they would have probably taken a longer time to complete.

Finally, but not less important, I would like to thank to my family and friends, for their support along all this time.

This work has been supported by the European Community under the Long Term Research ESPRIT Project E21907 (NANOS) and the Ministry of Education of Spain (CICYT) under contracts TIC97-1445-CE and TIC98-0511, TIC95-0492 and TIC94-0439.

Contents

Chapter 1. Introduction

Abstract

This first chapter introduces the need for efficient and well-tuned parallel execution environments to support the execution of parallel applications in shared memory multiprocessors. It shows the architecture of the current symmetric multiprocessors (SMP) and cache-coherent NUMA machines, and the aspects to consider when developing parallel execution environments. Taking them into consideration, it presents the goals of this thesis, and an overview of the NANOS parallel execution environment. It concludes with the contributions of this work.

Gene Roddenberry Crater on Mars

At the 42nd General Assembly of the International Astronomical Union (IAU), held in The Hague in 1994, a Martian crater was named in honor of Gene Roddenberry.

The Roddenberry crater is located at Martian latitude -49.9 degrees and longitude 4.5 degrees, in Quad MC26SE on Map I-1682. [Quad defines the name of the map on which it appears and "Map" is the USGS number for that map.] Its diameter is approximately 140 km or about 87 miles.

Introduction to "Death of a Neutron Star", Star Trek Voyager, Eric Kotani, Pocket Books, March 1999.

1.1. Parallel architectures and parallel software

Parallel processing increases the computational power of computers ranging from low-end workstations and even personal computers to big mainframes containing a large number of processing elements. Small shared-memory multiprocessor machines are available from a great number of computer builders: Silicon Graphics, SUN Microsystems, Compaq/DEC, Hewlett Packard, Intel, Sequent, Data General, etc. Some of them also build big mainframes. Examples of current shared-memory multiprocessor mainframes are the Origin2000 [72] from SGI, the SUN Enterprise 10000 [25], the Digital AlphaServer [37], etc.

The larger the machines, the more complex they are. Complexity comes partially from the fact that the path from the processors to the memory becomes a bottleneck when more than 10-12 processors are put together. Bad scalability motivates higher memory access latencies and poor performance when the number of processors is increased, due to the memory subsystem bottleneck. Several solutions have been adopted to solve the problem of scalability. They include to improve the path from the processors to the memory, through pipelined memory architectures in bus-based and NUMA computers. As a result, memory accesses have different latencies. The resulting architecture is more complex and difficult to manage to achieve high performance. Along with varying data access latencies, other elements that make parallel processing difficult in current hardware systems are the architecture of the current superscalar processors, the improved synchronization mechanisms and the support for relaxed memory models.

Side by side with the hardware evolution, current system software is also greatly evolving to introduce new and more powerful tools and abstractions for parallel processing. The main functionality of the system software is to allow several applications to use the computer resources (physical processors, memory, etc.). System software, initially based on the operating system abstractions, provided medium cost tools for parallel processing. Later, efficient user-level mechanisms replaced most of the operating system functionality. User-level approaches take advantage of the current microprocessor architectures in a better way because of the low cost of management. Besides, the operating system is responsible for offering the basic abstractions to support parallel execution.

Running parallel applications efficiently in current shared-memory machines requires good designs for the three main levels of operation, namely application, user-level run-time execution environment and operating system. This three levels are widely used along this thesis to serve as the foundation of a parallel execution environment.

At the higher level, the application level, the detection and/or expression of the parallelism useful to be exploited is the key aspect to consider. It is necessary to provide both good mechanisms inside the compiler to detect the parallelism and higher expressiveness in the parallel language for the programmer to rearrange the parallel regions in conformity with the underlying architecture. Current parallelizing environments provide tools both to automatically detect parallelism and to manually express it through language extensions or directives. At this level, the compiler and the application programmer are responsible of generating parallel code which could properly fit to the execution environment where the application is going to be executed. Parallel code generation greatly influences the way the application will be executed. Challenges like exploiting fine-grain parallelism, exploiting multiple levels of parallelism, and improving data locality are easily broken because of an

inappropriate parallel code generation. Each one of these issues can be managed in different ways, depending on the particular research interests. Multiple levels of parallelism can be exploited inside a parallelization environment or using different parallelization environments. The latter happens, for instance, when a parallel application consists of several loop-parallel processes managed through message passing at a higher level.

Fine tuning is very important at the application level. Both the programmer and the compiler have the largest amount of static information about the application (structure, data dependences, algorithm, etc.) compared with the information available at lower levels. At the same time, for the programmer it is easier to manage with information at a higher level than at a low level. On the other hand, the main disadvantage of fine tuning at this level is that any machine dependent arrangement breaks portability to other execution environments. The tuning process often involves calibration of program parameters through experimentation, and this work must be repeated for each parallel application, different input data set or target machine. Fortunately, parallel machines provide several tools to analyze the execution of parallel applications. In this way, fine tuning is affordable for system software developers and experienced users.

In the middle level, the user-level execution environment, the key aspect is the efficient support of the requirements of the application level. The main tasks of the user-level execution environment are creation, management, synchronization and termination of parallelism. Support parallelization as much fine-grain as possible means to reduce the overhead of the user-level execution environment to a minimum. Improving any of the creation, management, joining, aspects becomes important for this reason. Support for multiple levels of parallelism means allowing the exploitation of any kind of parallelism inside the application and at any moment. This also means not making differences among the sequential and parallel entities, in the sense that all of them should be able to spawn parallelism. Achieving data locality at the user level means to follow the indications of the application with respect which processors should access some specific data. Current parallelization environments have a reduced overhead, achieved through fine tuning for each specific architecture.

Finally, at the operating system level, several applications are executed at the same time in the same machine with the operating system intervention. Given that a computer has a limited number of processors, and that several applications are running at the same time, the task of the operating system consists of distributing processors among the running applications in such a way that each application can execute as if the machine were in a dedicated mode. This is not an easy task when the number of processes created by all the applications exceeds the number of physical processors in the machine. Resource sharing is unavoidable and usually prevents achieving a performance comparable to the individual applications execution.

It is commonly accepted that the three levels previously described have to be designed in a complementary way, rather than independently. They must be designed having in mind that from their proper relation the applications executing inside the parallel environment will obtain more benefits when the decisions taken in one level are not the opposite to the ones taken in another level. Several research works have already addressed this issue during the last years [11][33][147][89][75][96][84][5]. In this thesis, we continue working to enforce the cooperation among the different levels as the main goal of this thesis, along with the experimentation with the main aspects which have been commented along this introduction: Fine-grain parallelism, multiple levels of parallelism and data locality.

1.2. Target architectures

Currently, computer builders marketing high-end servers and high performance computer systems are promoting new machines supporting a large number of processors. Some years ago, machines with this characteristic were already the subject of some commercial and research projects such as the Sequent Symmetry multiprocessor system [129], the DASH multiprocessor [74], the Stanford FLASH multiprocessor [71] and the MIT Alewife machine [2]. Due to the promising results of these research projects, along with the increasing availability of high performance microprocessors, the same idea has also been taken by the computer industry and has been encouraged commercially. As a result, during the last decade, a number of different parallel computers have been arising in the commercial market.

1.2.1. Architectural taxonomy

The problem of *scalability*, or how to put side by side a large number of processors, building a machine from which one can obtain good performance, has been attacked differently, depending on the advantages that each builder wanted to obtain. This is one of the main current research subjects related to building parallel machines. Anyway, the number of possible solutions is limited, ranging from loosely coupled systems such as networks of workstations to highly coupled systems like symmetric multiprocessors (SMP's). Solutions can be summarized as follows [130][131][125][48]:

- Network of workstations, consisting of several standalone servers connected over a network. Each server runs its own copy of the operating system. Memory is physically and virtually distributed. All communication relies in message passing.
- Massively parallel processor systems, usually without any shared resource and communicating through a fast specialized interconnection network and message passing. Each processor runs its local copy of the operating system. Memory is physically and virtually distributed.
- SMP nodes, consisting of a limited number of processors, sharing a cache-coherent uniform memory access physical address space and running under one operating system. SMP nodes are based on a global system bus architecture. Resource contention is a factor that limits the number of processors that can be connected to the system bus. SMP nodes can be used to build clustered systems and CC-NUMA SMP nodes.
- Clustered SMP nodes, consisting of several independent nodes, sharing some storage devices. Each SMP node is executing its own copy of the operating system. There is neither global physical nor virtual address space among nodes, although tools could be provided to allow distributed shared memory, usually through mixed hardware/software protocols.
- CC-NUMA nodes (large SMP nodes), consisting of many processors and resources running under one operating system. They provide a cache-coherent non-uniform memory access (CC-NUMA) physical address space.

The work done in this thesis is targeted to the currently existing shared-memory parallel computers. Physically shared memory is provided by both SMP nodes and CC-NUMA nodes. Although they are architecturally different, they both provide a unique physical address space shared among all the processors in the machine. Nowadays, the number of processors supported by single SMP nodes usually ranges from 2 to 64, while CC-NUMA nodes support till 1024 processors. It is expected that these numbers could increase in the future.

In general, SMP and CC-NUMA nodes are of importance because they are well-suited for parallel processing, including computing intensive applications, parallel transaction processing, database management and informational network systems. SMP provides a smooth migration path for applications running on uniprocessors to run on high-performance systems. From the previous topics, this work focuses on improving the parallelism exploitation of computing intensive applications.

1.2.2. Symmetric multiprocessors

A symmetric multiprocessor (SMP) node contains two or more processors, with no master/ slave division of processing. Figure 1 shows an example of a ten processor SMP node. Each processor has equal access to the computing resources of the node. SMP's are architecturally characterized by a direct connection model where all components (processors, memory and input/output) are equidistant and directly connected to one another. Symmetric multiprocessors have uniform memory access times. Theoretically, variation in data access time is a result of the component speed and nothing else.

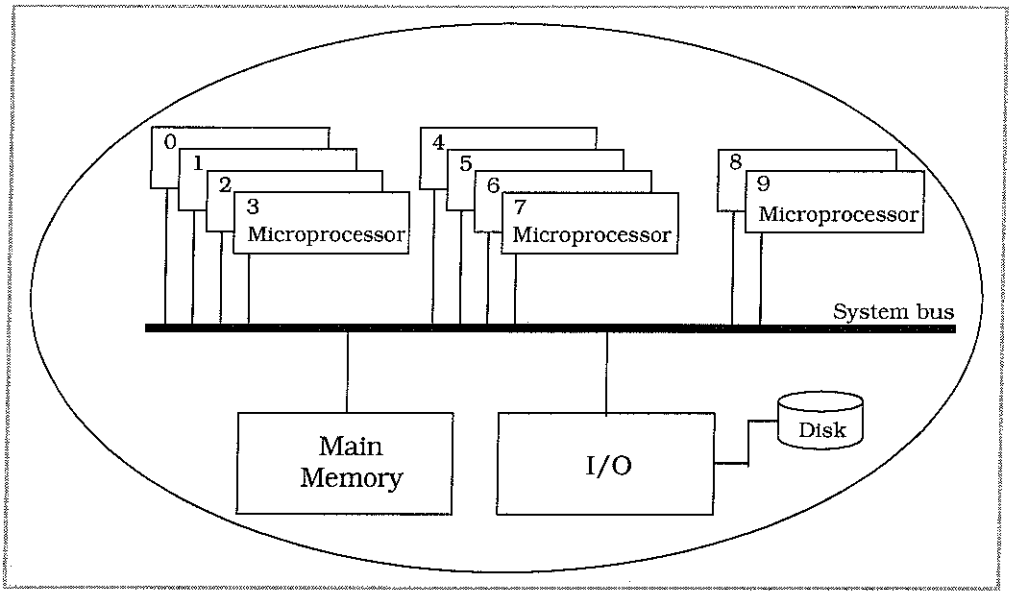

Figure 1: A single SMP node containing ten processors

As shown in Figure 1, SMP's have all components connected to a single system bus, providing a way for a tightly coupled connection among the resources. All data turnovers between any two components have to use the system bus. The bus not only supports data transfers, but also helps the system to provide data coherence. This is related to the cache memories attached to the microprocessors to improve their performance. Each data transfer initiated from a component is snooped by all other components, that can, in this way, modify the state of data blocks inside their cache memory containing the same data referenced.

Improving the performance of the machine is greatly related to the performance of the system bus. There are only two ways of doing the bus faster: Lowering the latency and increasing the bandwidth. Reducing latency requires to reduce the bus clock cycle time. This limits the bus size because of clock signal propagation constraints. By increasing the bandwidth, busses become wider, being able to transfer bigger data units. Most commonly, a combination of both techniques is used, resulting in shorter and wider busses.

Two important factors limit the scalability of SMP nodes. In the first place, these very short busses required for performance limit scalability because, simply, there is not enough room in the bus to attach a large number of components. In the second place, another factor that limits scalability in single SMP node systems is data access contention in the system bus. Contention comes from the fact that the system bus is used for every data transaction between any two components connected to the bus. While the bus is used for a transaction, any other attempt to use the bus will be blocked till the next available bus cycle.

Examples of SMP nodes are the SUN Ultra Enterprise 10000 server (Starfire) [25][123], the Compaq/DEC AlphaServer 8400 and GS140 [37][38] and the HP 9000 T-Class servers [62]. The SUN and Compaq/DEC machines are commented next.

1.2.2.1. The SUN Ultra Enterprise 10000 (Starfire)

The SUN Ultra Enterprise 10000 server supports up to 64 processors and 64 Gbytes of main memory. It is based on the 400 Mhz. Ultra SPARC II processor and the Gigaplane-XB interconnect technology. Each processor comes with a 4 Mb. external secondary cache.

The Enterprise 10000 server accommodates a maximum of 16 system boards. Each system board can be configured with up to 4 processors and 4 Gbytes of memory. System boards are connected through the Gigaplane-XB interconnect, a crossbar designed specifically for this machine. It uses a packed switched scheme with separate address and data paths. The reason is that data is usually communicated point-to-point, while addresses have to be distributed simultaneously throughout the system for the snooping protocol. The main structure of the system is shown in Figure 2. Data transfers are done through a 16x16 crossbar allowing communication between any two system boards at the same time. Contention arises when a system board is the origin or the destination for two or more data transfers in the same bus cycle. In this case, only one of the requests can be satisfied and the rest should wait for an available bus cycle. Addresses are communicated to all boards through four independent address busses. Each bus covers 1/4 of the total address space.

With this characteristics, the data crossbar has a latency of 468 ns. The latency of a SUN Ultra 6000, a smaller machine supporting 24 processors is half (216 ns.) The problem is that, in the Enterprise 10000, both local and remote memory references suffer the latency penalty. SUN states that this can be improved in the future [25].

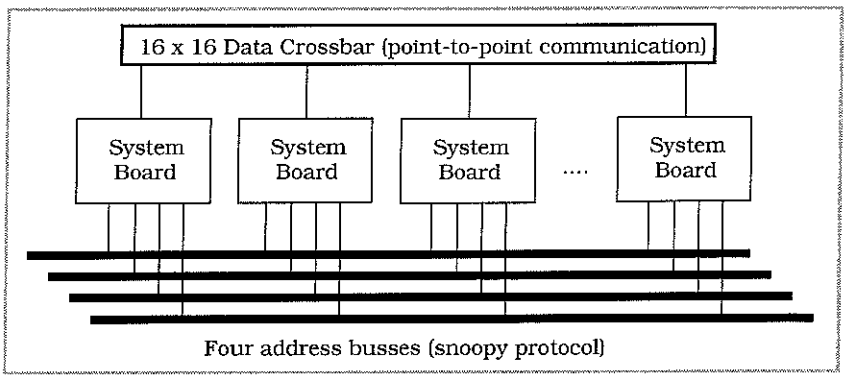

Figure 2: SUN Enterprise 10000 bus architecture

1.2.2.2. Compaq/DEC AlphaServer GS140

The Compaq/DEC AlphaServer GS140 is based on the Alpha 21264 processor. It supports up to 14 processors. Each Alpha processor runs up to 636 Mhz. and has a 4 Mb. external third-level cache. Processors are connected through a system bus supporting a maximum of 28 Gbytes of main memory.

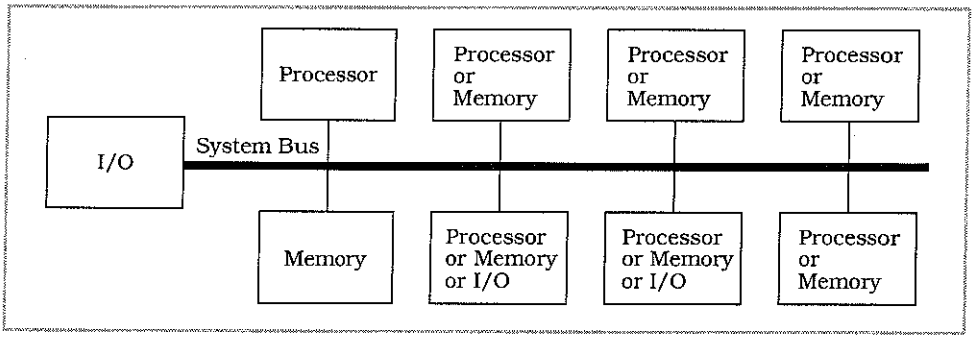

Figure 3: Compaq/DEC bus architecture

Figure 3 shows the structure of this system. The system bus supports the connection of a maximum of 9 system boards. This limitation seems to confirm that the small bus size required for performance is really conditioning the design of the computer. There are three types of system boards: Processor boards, memory boards and I/O boards. A processor board may contain up to two Alpha processors. A memory board can accommodate up to 4 Gbytes of memory. This means that a 14 processor system is limited to 4 Gbytes of memory, due to the maximum of 9 system boards connected to the system bus.

1.2.3. CC-NUMA symmetric multiprocessors (CC-NUMA SMP)

A CC-NUMA SMP node is a cache-coherent non-uniform memory access symmetric multiprocessor. In contrast to a SMP node, in CC-NUMA machines it is assumed that the access time from processors to memory can depend on the location of the originating processor and the location of the memory module accessed. The following could be stated as main objectives of CC-NUMA SMP machines [72]:

- Achieve larger scalability than SMP systems. Currently, this means breaking the frontier at 64 processors provided by the SMP systems.
- Maintain the data coherence among the overall system, as in SMP systems.
- Limit the increment in the cost due to introducing complex hardware (multi-bus alternatives, for instance, are more complex and costly).

Figure 4 shows the structure of a CC-NUMA SMP machine. Several SMP or custom nodes are joined through a fast, low latency, interconnection network. Each node contains a number of processors, memory, local I/O and a fast communication bridge to the central interconnection network. All memory (and usually all I/O) is shared and accessible from all the processors attached to the machine. The topology of the interconnection network greatly varies among different designs and it is the factor that will limit the performance of the overall system. Data coherence must be maintained through a special cache coherence protocol implemented by means of a memory directory located in each node, which tracks data references to remote nodes and references from the local node to remote data.

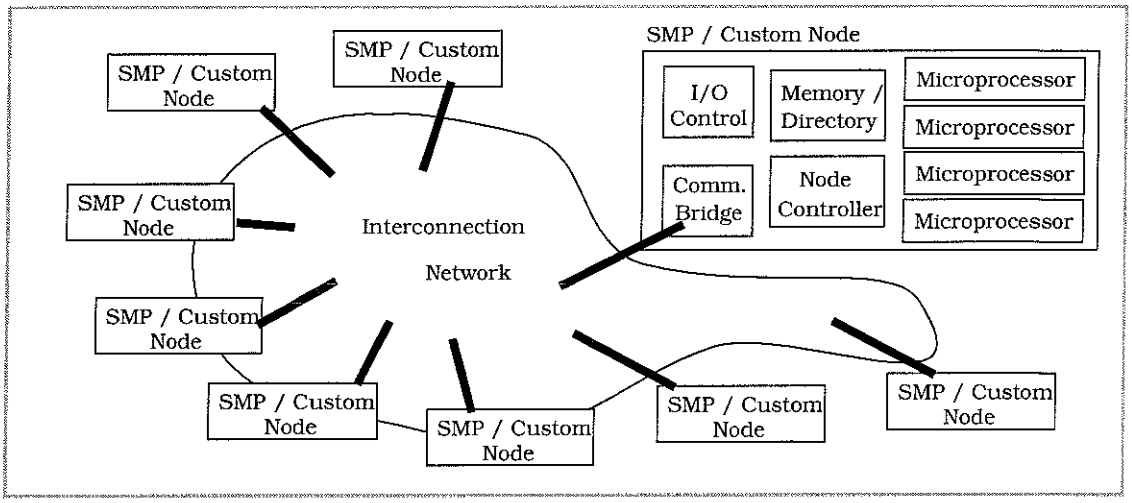

Figure 4: Structure of a CC-NUMA symmetric multiprocessor

Examples of CC-NUMA SMP nodes are the SGI Origin2000 machine [136], the HP 9000 V-Class Enterprise server [63], the Sequent NUMA-Q based servers [131][128] and the Data General AViiON AV 25000 CC-NUMA Server [35][34]. The Origin2000, HP 9000 and Sequent servers are commented next.

1.2.3.1. SGI Origin2000 server

The SGI Origin2000 server supports up to 1024 processors and one Terabyte of main memory. It is based on the 250 Mhz. MIPS R10000 processor (with 4 Mb. of external secondary cache) and a distributed shared memory (DSM) architecture. The DSM architecture implements directory-based memory coherence, removing the broadcast bottleneck that prevents scalability in the snoopy bus-based SMP implementations [72].

The basic Origin2000 node is shown in Figure 5. Each node contains two R10000 processors, with their respective secondary cache memories. The central element in each node is the HUB, which connects both processors to the memory, I/O and the interconnection network (interconnection fabric, in SGI terminology). Each node can accommodate up to 64 Gbytes of main memory and its corresponding directory memory. The global shared address space is distributed among the nodes in slices. Node 0 contains addresses in the lower range 0:N-1, node 1 follows, containing addresses N:2N-1, and so on. The DSM architecture provides global addressability from any processor to all memory and I/O.

The Origin2000 system uses a directory-based memory coherence protocol. Each cache line in memory has an associated directory entry. Directory memory is located near main memory (in the same module, see Figure 5). Each entry contains information about the associated cache line: its system-wide caching state and bit-vectors pointing to caches which have copies of the cache line. Memory can determine which caches need to be involved in a given memory operation in order to maintain coherence. The cache coherence protocol is explained in [72].

Figure 6 shows the structure of an Origin2000 system containing 64 processors. Processor nodes are attached to the interconnection routers, which provide low latency communication. Routers link the HUB inside the basic nodes to the CrayLink Interconnect. Each router has six external full-duplex connections, which are managed internally by a full six-way nonblocking crossbar switch. Machines with 128 processors use meta-routers (routers

connecting routers) to connect four 32-processor groups. Meta-routers are replaced by 5-D hypercubes to reach up to 1024 processors.

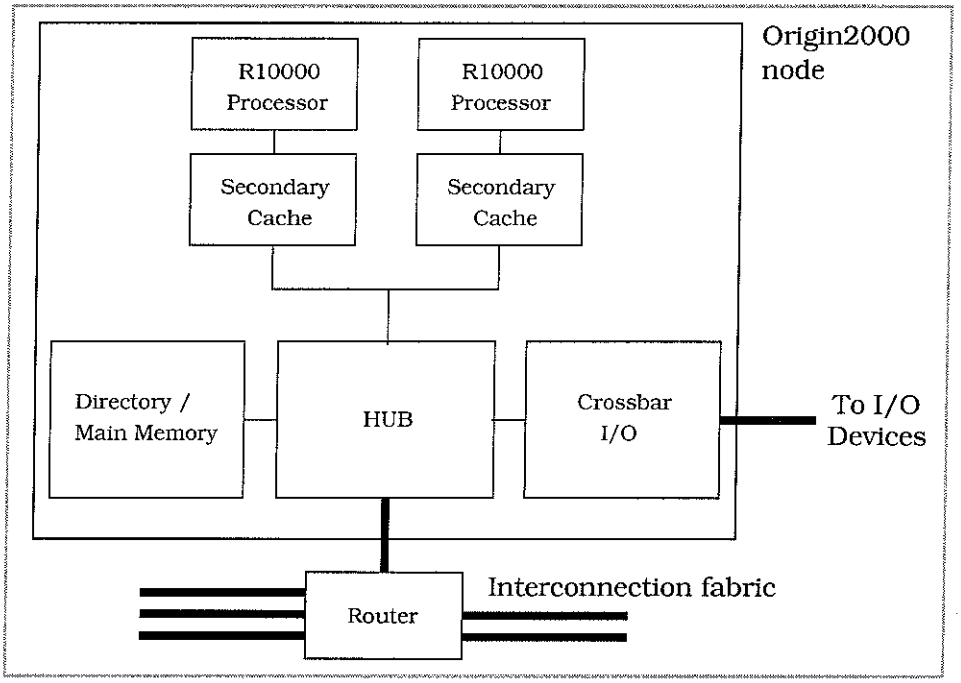

Figure 5: Origin2000 node and external connections

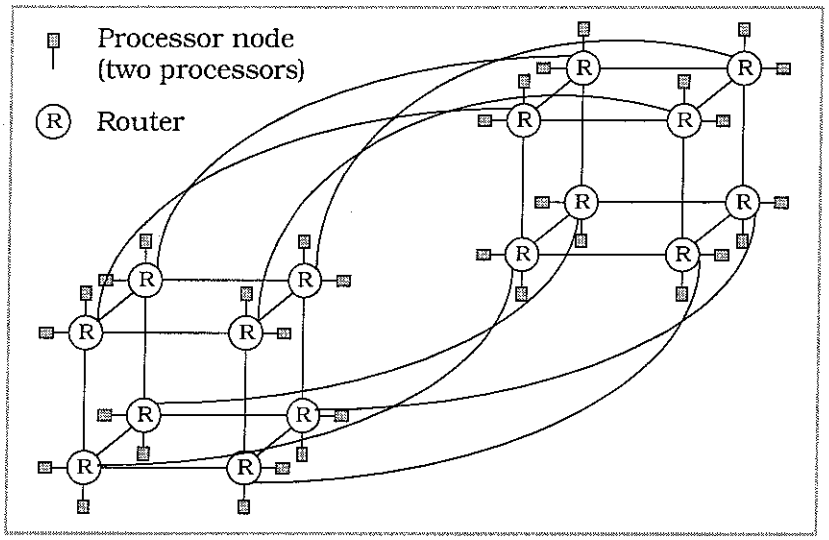

Figure 6: Topology of a 64-processors Origin2000 system

Table 1 shows the average latency of the local and remote memory references in machines with a different number of processors, from 1 to 128. The increment of the latency between 8 and 32 processors is very small due to the use of redundant connection links in routers called express links [72]. It is remarkable than, in a 64 processor machine, the average latency of a remote memory access is only 2.7 times the latency of a local memory access.

In this thesis, we have used a machine with these characteristics (64 processors) for the evaluation of our proposals.

Memory level	Latency (ns.)
L1 cache (195 Mhz. processor)	5.1
L2 cache	56.4
local memory	310
4 cpus remote memory	540
8 cpus remote memory (avg.)	707
16 cpus remote memory (avg.)	726
32 cpus remote memory (avg.)	773
64 cpus remote memory (avg.)	867
128 cpus remote memory (avg.)	945

Table 1: Origin2000 average memory latencies

1.2.3.2. HP 9000 V-Class Enterprise server

The HP 9000 V-Class Enterprise server supports up to 128 processors and 128 Gbytes of main memory. It is based on the 440 Mhz. HPPA-8500 processor and the HP's Scalable Computing Architecture (SCA). The V-Class SCA is a multi-level memory subsystem. The first level (up to 32 Gbytes) consists of a traditional SMP memory; The second level is created by tying first-level memories through a high performance interconnect. The interconnection is named SCA-Hyperlink. The resulting system appears to be as a single globally shared memory multiprocessor system.

The building-block of the HP 9000 V-Class server is shown in Figure 7. It consists on a 8x8 crossbar switch to which 8 four-processor modules and up to 32 Gbytes of main memory can be attached. Each four-processor module has a local I/O module. The connection of memory modules to the crossbar is done through the SCA module, implementing multiple rings connecting building-blocks. SCA allows remote memory accesses to/from other building blocks. Up to four building-blocks can be joined in a 128 processors machine.

The system guarantees cache coherence between multiple building-blocks. Each SCA module contains cache and directory memory (Hyperlink memory). The Hyperlink cache memory contains the remote data referenced by the local building-block. The Hyperlink directory memory contains, for each cache line, the list of building blocks sharing the same information.

In contrast to the Origin2000 system, the HP 9000 V-Class server takes the approach of building a large SMP crossbar-based system (containing up to 32 processors) and take it as the building-block for a larger machine.

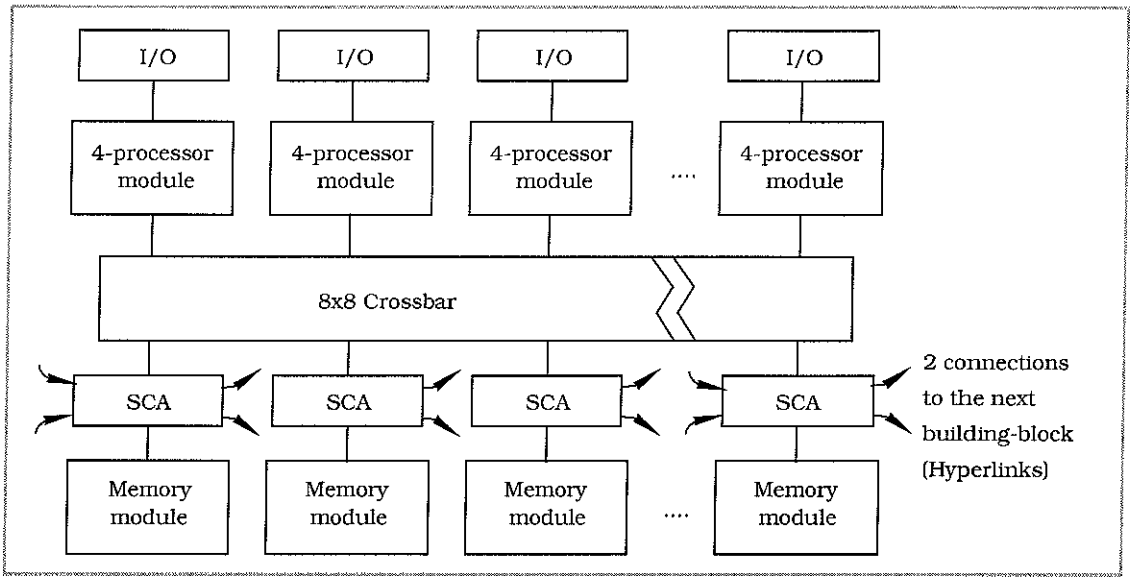

Figure 7: HP 9000 V-Class building-block

1.2.3.3. Sequent NUMA-Q E300 server

The Sequent NUMA-Q E300 server supports up to 128 processors and 128 Gbytes of main memory. It is based on the 400 Mhz. Intel Pentium II Xeon processor and the Scalable Coherent Interface (SCI) [64].

NUMA-Q stands for NUMA based on quads. A quad (see Figure 8) is the basic block for the Sequent server, consisting of four processors (Intel Pentium Pro / Pentium II), connected to an SMP bus, with local memory (up to 4 Gbytes) and local I/O. Basic blocks are attached to the IQ-Link network, which is the Sequent's implementation of the SCI standard. IQ-Link is a unidirectional ring network, running at 1 Gb/s. Figure 8 also shows the resulting system structure. A maximum of 32 quads are supported for a total of 128 processors.

Like in the Origin2000 server, the global shared address space is distributed among the nodes in slices. Data coherence is ensured in hardware inside each quad, like in a SMP machine. The IQ-Link module inside each quad ensures data coherence among quads. It also provides 32 Mb. cache memory for remote data and directory memory to keep track of the state of the local and remote cache lines whose data is in the local quad [81].

Although the high speed network and good local memory latency (250 ns.), remote memory latency is higher than in the SGI Origin2000 machine, reaching 8 times the local latency [80][72].

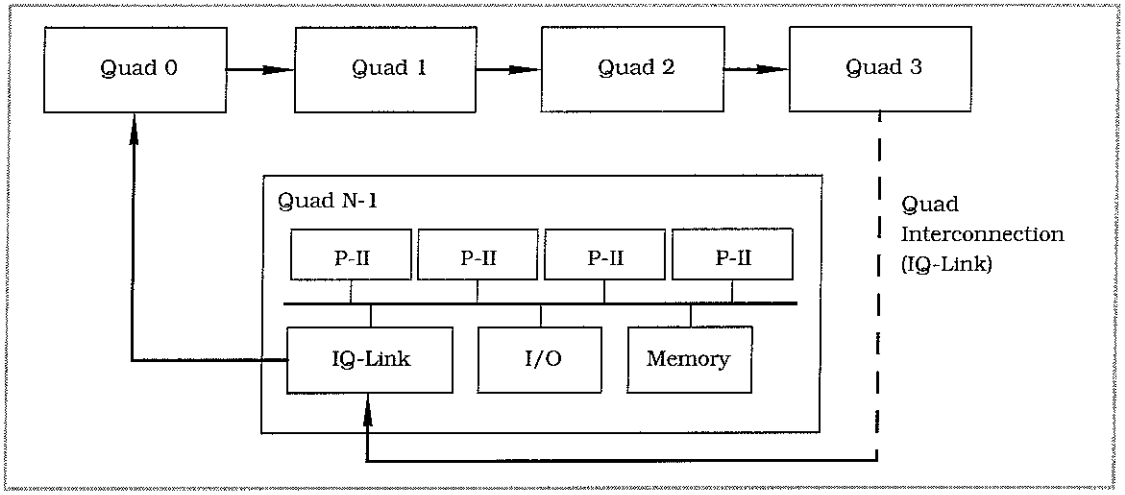

Figure 8: NUMA-Q server topology and Quad structure

1.2.4. Architectural considerations for our proposals

From the previous discussion, we have seen that different implementations of SMP and CC-NUMA machines give pretty different low-level performance results. Scalability is a difficult problem and the different solutions usually introduce a penalty in data access times. Hardware constraints raise a barrier when reaching 12-16 processors in SMP machines.

Current SMP machines try to break this barrier through complex hardware designs, but maintaining the backplane bus and the snoopy cache protocols. The goal is to maintain the memory access times equal from any processor to any memory module at a reasonable cost. Nevertheless, memory contention should also be taken into account. Several processors accessing the same memory module will have to wait for the memory accesses to be served one after the other. Does this mean that equal memory access times are unachievable?

With the design of CC-NUMA machines, architects are implicitly assuming that memory access times do not need to be always the same. This simplifies the hardware and lowers the cost. Nowadays, this has allowed to add up as much as thousands of processors in a single node, running a single copy of the operating system and applications.

From the software point of view, both types of machines are behaving in a very similar way. Different memory accesses can have different latencies. This means that parallel applications that are scaling well till 12-16 processors will suffer a problem similar to the hardware scaling, when executed on 32 and more processors. For instance, an application task spending one millisecond on one processor should ideally take 125 microseconds on 8 processors. Software can easily support this coarse grain distribution of work. By contrast, the same task would spend 15 microseconds on 64 processors. Supporting such fine granularity is very difficult to achieve, taking into account that memory access latencies can reach near one microsecond. This limit in the ability of exploiting fine granularity always exists, although it can be different, depending on the architecture. Application scalability can be achieved by scaling the amount of data that the application is managing.

In this thesis we want to study another alternative. CC-NUMA architectures are based on building-blocks and they further extend the traditional hierarchical view of the physical memory. Parallel applications can adapt to the new hierarchy by exploiting, when possible,

multiple levels of parallelism, and grouping processors to work in more coarse grain distributions of work, even when using more than 12-16 processors.

When we take into account that these machines will be used to run several applications at a time, we realize that the operating-system scheduling mechanisms probably should also be revisited. Operating-system scheduling takes usually the process/thread as the scheduling target. This means that all kernel-level events (processor preemption, blocking, unblocking, etc.) related to a process are communicated to it, without taking care of the parallel application to which the process belongs. This mechanism comes from the uni-processor implementations of the operating systems. Applications can also be considered as the scheduling target. We want to study whether this alternative is feasible for kernel-level scheduling and evaluate the benefits that can carry out.

1.3. Current parallel execution environments

Computer builders provide parallel execution environments along with each parallel computer. A parallel execution environment, from top to bottom, consists of a parallelizing compiler, a user-level threads library and a multiprocessor operating system. The parallelizing compiler either automatically decomposes the application in parallel tasks, or accepts annotated source code, from which it generates parallel code calling the user-level threads library. The user-level threads library supports the execution of the parallel tasks of the application, through efficient fork/join and synchronization mechanisms. The threads library uses the operating system interface to obtain execution resources. The multiprocessor operating system is in charge of managing processors, physical memory and input/output. It manages processor scheduling and memory allocation and placement.

In the following subsections, we introduce some of the commercial and research projects related to supporting parallel execution, introducing the main aspects to consider when designing a parallel execution environment. They are compared with our approaches, later in Chapter 6.

1.3.1. Commercial parallel execution environments

Every commercial parallel execution environment provides a set of parallelizing compilers, usually supporting the C and Fortran languages. For instance, both the MIPS Pro C [132] and Fortran 77 [135] compilers provided by Silicon Graphics are able to automatically extract loop parallelism from sequential applications. This is done with the help of the PCA (Parallel C Analyzer) and PFA (Parallel Fortran Analyzer) tools [134], respectively. Both compilers also accept annotated C and Fortran code. In this case, the programmer can express both parallel loops and parallel sections (data and functional parallelism). Both in automatic and annotated parallelizations, the resulting parallel code calls to the SGI MP library [137][133], the custom threads library provided by SGI to support parallel execution.

Commercial parallel execution environments provide different user-level threads libraries. There are two main types of such libraries: First, the standard libraries such as Pthreads (Posix threads) [65][124]. They are thought to be used by parallel programmers, to build parallel applications, expressing the parallelism by hand. The programmer introduces explicit calls to the threads library to create, manage, terminate and synchronize the parallel application tasks. The Pthreads library is oriented to work with shared memory. There are also standard libraries oriented to message passing, such as PVM [52] and MPI [90][91].

Secondly, there are custom threads libraries (such as the SGI MP library). Such custom libraries are not used directly by programmers, but instead they are used through a parallelizing compiler. Custom thread libraries are highly tuned for execution on top of the parallel architecture. For instance, spawning parallelism in custom libraries is done from a master processor to all the slave processors at a time, instead of supplying work in a one-to-one basis, which it is the case in the Pthreads library. Such fine tuning motivates the use of very simple structures to support the parallelism. Simplicity carries out efficiency, but also there is a lack of functionality with respect to standard libraries. For instance, the SGI MP library forbids to spawn parallelism inside a parallel region. That is allowed in the Pthreads library, which it is not imposing such restriction by allowing that any pthread could spawn a new pthread. In general, it is common that custom libraries do not support the exploitation of multiple levels of parallelism.

The multiprocessor operating system assigns physical processors to application processes (or threads). Different kernel-level scheduling policies are usually provided by the operating system. For instance, the SGI IRIX operating system provides the time-sharing and gang scheduling policies [138] to manage parallel applications. Time-sharing is a priority-based scheduling policy, where processes are scheduled independently of each other. In gang scheduling, instead, processes belonging to the same application are scheduled as a group. In the SGI MP execution environment, the time-sharing policy is used because it is more dynamic than gang scheduling and applications are able to adapt to the number of processors allocated. In fact, each application has a specific thread in charge of controlling the load of the system and whether the application is taking advantage of the allocated processors. This thread serves two purposes. First, in case that thread detects that the load is high and the application is not receiving enough resources, it decides to stop some of the processes of the application, to free some physical processors and reduce the system load. Second, when that thread detects that the load is low and the application can use more processors, it starts some of the processes of the application to take advantage of more physical processors. This is a specific feature of the SGI MP execution environment, not found in other environments.

Other operating systems, such as Digital UNIX, provide the FIFO and round-robin scheduling policies, in addition to time-sharing, to schedule processes/threads on top of processors. SunOS provided a gang scheduling class for lightweight processes (LWPs) [119]. Nevertheless, the parallel execution environments running on top of them are simpler than the SGI one, lacking that kind of communication between the user and kernel levels.

1.3.2. Research projects

Several parallelizing compilers are being developed to generate code to run on top of custom threads libraries. They are research projects oriented to individual application execution. For instance, the SUIF compiler [60] gets sequential Fortran code and automatically generates parallel code to run on a custom library, with no communication with the operating system. The SUIF run-time system supports a single-level of parallelism, in the same way as the SGI MP library does.

Searching for support of multiple levels of parallelism and fine granularity is the main goal of several user-level thread libraries. The Illinois-Intel Multithreading Library (IML [56]) is a user-level threads package supporting nested parallelism and code generation from the Intel Fortran compiler. It runs on PC-compatible Intel multiprocessor machines, on top of the

Windows NT kernel. Communication with the operating system includes a specific interface to get the number of available processors and to stop and resume kernel threads.

COOL [23] is a run-time system supporting parallel object-oriented programs written in the COOL language [24]. It is thought for NUMA machines. The programmer is allowed to express data locality in three different ways, through object, task and processor affinity.

Cilk, Filaments, Concert, Active Threads are thread libraries also providing support to compiler generated code. Communication with the operating system is not considered, in any of these projects. They provide support for multiple levels of parallelism. Data locality is taken into account, providing tools in the library interface to map the application tasks on specific processors. The Cilk language extends C with parallel constructs. The Cilk run-time system [17] is oriented to express parallelism in recursive programs. Filaments [44] can be used directly by the C programmer or from the Sisal functional language. It supports fine-grained iterative and fork/join threads. Concert is a concurrent object-oriented language and run-time system [27] designed to support fine-grain irregular applications which behavior is unknown at compile time. Active Threads [150] is a run-time system supporting code generation for the pSather compiler [93][142]. Threads are grouped in thread bundles, sharing a common scheduler. Bundles facilitate data locality because threads accessing the same data could be assigned to the same physical processors through the bundle.

Projects considering the improvement of communication with the kernel include Process Control, Scheduler Activations, First-Class User-level Threads and Execution Vehicles. Process Control [147][148] introduces the concept of dynamically adapt the parallelism inside applications to the available resources, as indicated by the kernel. The application receives enough information to start and stop processes when needed to adapt to the allocated resources. Scheduler Activations [5] provides to the user level all scheduling events related to the application. The events are receiving a new processor, a processor preemption, thread blocking and thread unblocking. All events are communicated through the upcall mechanism, which sometimes is too costly to allow an efficient communication path between user and kernel levels. First-Class User-Level Threads [84] merges the upcall mechanism with the shared memory. The most aggressive approach is taken in the recent implementation in IRIX6.5 of Execution Vehicles [32]. In this approach, the full context of the kernel-level threads is made available to the user-level execution environment, in such a way that both the kernel and the user levels can resume a preempted thread.

There are other research projects providing threads libraries, such as Quartz [6], FastThreads [4], Presto [14] and SwitchStacks [28] which are interesting for various reasons and have also been studied during the development of this work.

1.4. Software considerations for our proposals

Fine-grain parallelism exploitation can be useful depending on the amount of processors available to the application. From the hardware point of view, when a large number of processors is available, the limits of fine-grain parallelism come from the fact that distributing less work to more processors reduces the amount of data that each processor is accessing, usually increasing the conflicts among the processors. This topic appears in large machines, specially in NUMA architectures. When the amount of conflicts grow, the performance of the

parallel execution decreases. One solution to this problem is to exploit multiple levels of parallelism, when possible. Otherwise, the reduction of the conflicts is only possible by enlarging the size of the data set managed by the application.

From the application point of view, allowing to exploit multiple levels of parallelism means to be able to spawn parallelism at any point during the execution. Some of the current parallel execution environments forbid spawning new parallelism when some parallelism has been already spawned. Such environments forbid multiple levels of parallelism.

In the exploitation of multiple levels of parallelism it is interesting to consider the aspect about how the new parallelism is going to be executed. A first approach consists of using the same processors that are already executing other parallel constructs. Alternatively, imagine that the application can be partitioned along the different parallel constructs. Different processors can be then assigned to work in different portions of the application. Resource distribution is let entirely as an application decision. It can be done in regular or irregular ways, depending on the application structure. For instance, an irregular distribution of processors can assign more processors along the critical path and less to execute other constructs. Application partitioning is an interesting issue because it will potentially reduce the spawning overhead and benefit data locality and working set size.

Supplying accurate information to the operating system about resource needs is also a key aspect. When the application shrinks its parallelism, processors could become quickly available for other applications running in the system, reducing idle time. On the other hand, when an application needs more processors, it is guaranteed that the request will be taken into account in a short enough amount of time. In this way, the application adapts its structure to the available resources.

1.5. Goals of this work

The global objective of this thesis is to achieve *efficient* and *effective* support for multi-user parallel processing across a wide range of shared memory parallel computers, from small symmetric multiprocessors to high-end systems with globally shared address space. This work searches for a complete execution environment for parallel applications. The environment consists of a parallelizing compiler, a specialized user-level threads library, a new operating system interface, new scheduling mechanisms and several system level scheduling policies.

Supporting parallel processing *efficiently* means to achieve both high global system performance and reduced turnaround time for applications running simultaneously on the parallel system. The execution environment is *effective* if it is able to support all the existing applications which are amenable to be parallelized and new ones which can be already designed to be executed in parallel.

The fundamental concept used in this work to achieve the global objective is the cooperation among the different execution levels in a parallel execution environment. The three execution levels which we consider are the compiler/application level, the user-level execution environment and the operating system. Cooperation must be bidirectional, from the application to the user-level environment and operating system and vice-versa.

The kind of work and the experimentation needed to complete the previous goal requires the availability of a whole parallel execution environment, from the operating system to the compiler/application level, and including the user-level execution management. This is the reason why one of the main results of this thesis is a complete parallel execution

environment (the NANOS execution environment) providing a compiler (the NANOS compiler), a user-level threads library (NthLib), an operating system interface and scheduling mechanisms (the NANOS O.S.) and the definition of how the three levels are going to cooperate. Such cooperation can be seen like the path the information follows from the application to the operating system. Most of the development done for the NANOS execution environment has been done as part of the NANOS L.T.R. ESPRIT Project (E-21907) [98][99][100][101][102][103][104], supported by the European Commission [45].

The main objective presented is decomposed below in partial goals, located at the different levels, which have to be obtained first to prove the feasibility of the global one.

1.5.1. Application / compiler level

Supporting effective parallel processing includes the ability to express parallelism in high level languages. This work focuses mainly on FORTRAN applications. The NANOS compiler automatically parallelizes applications based on OpenMP directives [107]. It uses a hierarchical task graph [55] as an internal representation of the parallelism extracted from the application. The compiler generates parallel FORTRAN code from this internal representation.

The development of the NANOS compiler, although has been done outside this thesis, has been directly guided by the design, the experimentation and the results presented in this work.

The goal is to show the feasibility of compiling applications in order to efficiently exploit parallelism in a multiprogrammed environment. This can be further decomposed, in the following issues:

- Search for the limits of fine-grain parallelism. Find which is the limit of fine-grain parallelism in current architectures and what the important aspects to consider are when parallelizing applications at fine granularity levels.
- Allow the exploitation of multiple levels of parallelism, when possible, including both structured parallelism at the loop level as well as task parallelism. Allow experimentation with processor grouping.
- Enforce the cooperation with the lower levels (user-level execution environment and operating system). Communicate information about data locality, and minimum exploitable granularity to the user-level execution environment. Communicate the resource needs of the application to the operating system.
- Facilitate the development of *evolving* and *malleable* [47] applications. Evolving applications dynamically request at any moment the number of processors they need for execution. Malleable applications are able to run on any number of processors, quickly adapting their execution when the number of processors allocated to them changes.

1.5.2. User-level execution environment

Parallel code generated by the NANOS compiler is supported by a specialized threads library (NthLib). This library has been designed as the NANOS compiler target. It can be also used directly by a user/programmer. The goals at this level are:

- Achieve individual application high performance. This means to search for an efficient implementation of the threads library, the basis of the user-level execution environment, without compromising or restricting the achievement of the other goals.

- Support fine-grain parallelization. Provide an execution environment to investigate which are the limits of fine-grain parallelization. Search for techniques to supply work to processors and mechanisms for thread joining, having in mind both UMA and NUMA architectures.
- Support multiple levels of parallelism and processor grouping. Allow the application level to spawn parallelism at any time and drive processors to execute portions of it.
- Offer mechanisms for the application level to control data locality and load balancing.
- Promote efficient cooperation of the user-level execution environment with the application and the operating system levels. Provide the low-level mechanisms for controlling the processors allocated to the application. Detect processor preemptions from the operating system and recover the work to avoid delays in the application.

1.5.3. Operating system level

Parallel execution is supported by the NANOS O.S. The operating system is responsible of managing physical resources and offering them to the applications. This work considers the processor as the main resource to be shared among the executing applications. Memory allocation is considered for NUMA machines. Other issues usually managed by the operating system, such as memory management or input/output are not considered, by now. The goals at this level are:

- Define an operating system interface to support the cooperation between the higher levels (application and user-level execution environment) and the operating system.
- Design a new operating system scheduling framework to allow efficient processor distribution and sharing, considering the application as the scheduling target and providing a smooth kernel-level scheduling in accordance with the applications.
- Enable the design and implementation of kernel-level scheduling policies to distribute the available processors among the executing applications in a dynamic environment.
- Dynamically adapt the parallelism exploited inside each application to the global resource utilization. Processors are moved by the operating system from one application to another to balance the overall system load and applications requirements.
- Consider processor affinity and memory placement in kernel-level scheduling policies, benefiting data locality.

1.6. Description of the complete execution environment

The main structure of the execution environment is presented in Figure 9 and it consists of three basic levels of operation (application/compiler, user-level execution environment and operating system), their interfaces and the possible paths through them. The subject of this thesis consists of the design of both the user and operating system interfaces and the paths connecting them, represented by black arrows in Figure 9. The gray arrows mean paths which are *used* in this work. We have participated in the development of such paths, but they are out of the scope of the work presented in this document.

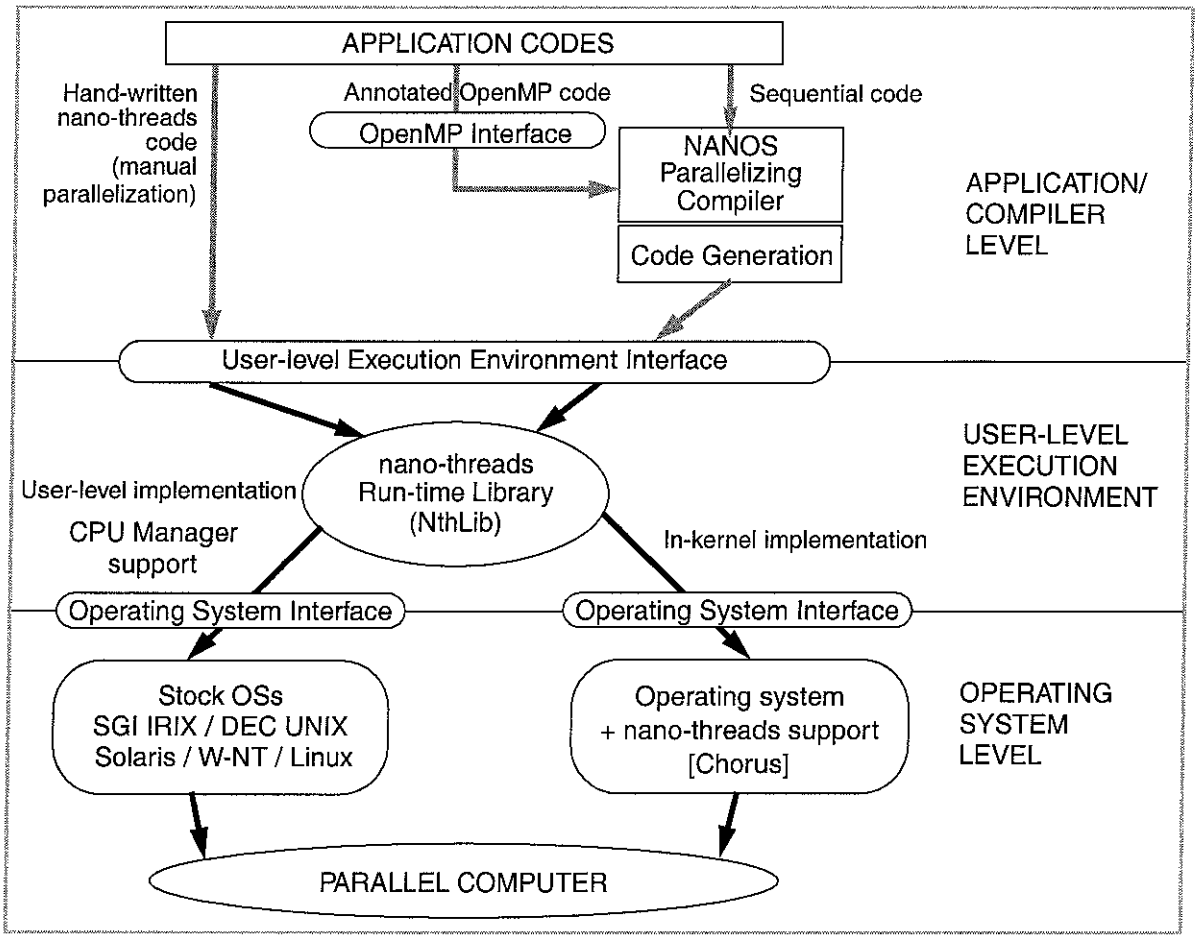

Figure 9: Nano-threads execution environment: main structure

The work done in this thesis uses three different ways of building parallel applications (see the upper part of Figure 9): manual parallelization, parallelization with directives and completely automatic parallelization through the NANOS parallelizing compiler. The NANOS compiler is based on Parafrase-2 [114][115][59], a parallelizing compiler developed at the University of Illinois at Urbana-Champaign. The NANOS compiler includes a graphical interface [20][54] to visualize the structure of parallel applications and add OpenMP directives for parallelization. Along this thesis, a set of (extended) OpenMP directives is used to annotate applications for expressing the parallelism. This is the visible top-level user interface of the parallel execution environment. Each Fortran application is modified introducing the necessary directives, obtaining applications structured following the underlying model. The NANOS compiler parses the directives expressing the parallelism and integrates the information obtained from them to the internal compiler structures. C applications are, by now, manually parallelized using direct calls to the threads library interface.

This thesis defines the two internal interfaces between the three levels of operation: the user-level execution environment interface and the operating system interface. Parallel code is executed on top of the nano-threads library (NthLib, in the middle part of the figure), using the user-level execution environment interface, which gives support to application-level scheduling and efficient thread management. The user-level execution environment interface allows to describe how the computation and data space of an application are structured and the ways in which different execution flows can traverse them. The hierarchical representation of

the computation space has all the information needed to support both structured and unstructured parallelism and dynamic allocation of execution flows (threads) to different partitions of the program, as long as precedences allow it and there are available resources.

At the bottom, the operating system interface enforces the cooperation between the user and kernel levels and allows the application level to adapt to a dynamic processor allocation environment. The operating system interface provides a light-weight communication path between active user applications and the operating system in order to support requests of resources from the user-level execution environment, and also inform it of actual resource allocation and availability. The operating system interface has been designed and completely implemented inside the CHORUS microkernel [121] and partially implemented with the help of a user-level CPU Manager in some stock operating systems.

A set of scheduling strategies have to be designed at each level in order to attain the goals of this work. These strategies define the allocation of the entities available at each level to the entities available in the next lower level (see Figure 10). At the application level, the compiler has to identify sequences of instructions with granularity coarse enough to amortize the management overhead. These sequences (or application tasks) map to user-level threads at run-time following the scheduling policies defined by the user-level execution environment. The user-level environment allows the application to map work to specific virtual processors offered by the operating system level or to simply let the work available to execute on all the processors available. At the operating system level, the scheduling decisions take care of assigning virtual processors (kernel threads) to the available physical processors. It manages also the kernel-level scheduling events relevant to the application and implements the kernel-level scheduling policies.

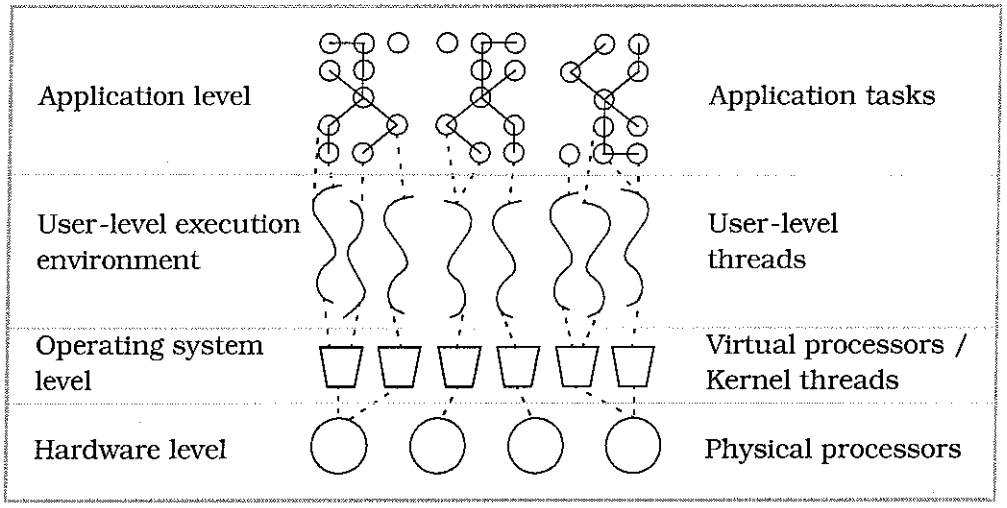

Figure 10: Mapping entities across the different execution levels

1.7. Contributions of this thesis

The main contribution of this thesis is to design and implement the Nano-Threads Programming Model on a complete user- and kernel-level parallel execution environment, resulting in an efficient product, competitive with widely-used parallel execution environments. Cooperation between the user and kernel levels is the basis to achieve high performance.

As in many other sections along this thesis, the contributions provided can be classified by the level of operation where they are located (application/compiler, user-level execution environment and operating system). Special emphasis is done to highlight the aspects that improve cooperation among the levels.

1.7.1. Contributions at application level

The contributions at application level are the following:

- Mapping the hierarchical task graph structure (the internal compiler representation of an application) to a user-level execution environment interface, allowing the execution of applications decomposed in an hierarchy of parallel tasks.
- Allowing the application level to know about virtual processors, making the master/ slaves scheme more general and allowing the application to decide which processors work as masters and which as slaves inside the hierarchy of parallel tasks.
- Merging the support of multiple levels of parallelism with data locality issues, resulting in the proposal for establishing processor groups at application level. Within this approach the application is able to restrict the number of processors participating in a specific parallel construct, dedicating the remaining processors to other tasks.
- Mapping well-known application level scheduling algorithms (dynamic, guided self-scheduling, etc.) to the Nano-Threads Programming Model, using the support from the user-level execution environment.

1.7.2. Contributions at the user-level execution environment

The contributions at the user-level execution environment are:

- Define a user-level threads library interface to support parallelization following the Nano-Threads Programming Model.
- Efficiently support multiple levels of parallelism, achieving the integration, in the same execution environment, of different mechanisms for providing high functionality to the higher levels of parallelism and high performance to the inner-most level. Provide nano-threads, giving full functionality, and work descriptors, offering limited functionality/ high performance.
- Support processor grouping, allowing the application to supply work to specific processors, setting the master and slaves for each parallel construct.
- Support thread bursts, avoiding the need to spawn all the parallelism at a time and allowing to check the number of resources allocated between bursts, adapting the parallelism.
- Provide an execution environment highly dynamic, enforcing the adaptability to the available resources and enabling an efficient execution on a general-purpose multitasking system by means of integrated cooperation with the operating system.

1.7.3. Contributions at the operating system level

The contributions at the operating system level are:

- Consider the application as the scheduling target, instead of independent processes or threads. Physical processors are assigned to applications and remain assigned till the application decides to free them or the current scheduling policy decides to allocate them to another application.

- Design an operating system scheduling framework in which all applications requests are considered globally for taking the decision of how many processors allocate to each application.
- Allow the operating system scheduling policy to decide to which applications allocate free processors in the near future, in case some processors become out of work.
- Enforce the communication and cooperation with the higher levels, defining an efficient interface between kernel and user levels, allowing the applications to dynamically request and release processors. Integrate the best proposals of the previous works. Export the kernel-level scheduling events to the user-level using the most efficient way.
- Provide a user-level design and implementation of the proposals through a CPU Manager, enforcing portability to different operating systems.

1.8. Thesis structure

This document is organized in 10 chapters. Chapter 2 presents the complete definition of the Nano-Threads Programming Model, including the elements required from the user-level execution environment, the operating system. It presents also the programming language we have selected to express the parallelism. From our point of view, at the same level of importance, Chapter 3 discusses the relationship that we establish among the three levels of operation (application, user-level execution environment and operating system) and how the entities inside each level of operation are mapped on the entities of the next lower level.

Chapter 4 describes the user-level execution environment interface, which can be used either for code generation from a compiler or directly by the programmer. Chapter 5 presents the extensions to the operating system interface to support the Nano-Threads Programming Model.

After presenting all the proposals of this thesis, Chapter 6 discusses the comparison with the previous and related work and Chapter 7 highlights what we consider the most important aspects of the implementation of the NANOS parallel execution environment.

Chapter 8 shows some examples of use of the NANOS parallelizing environment, both directly from a programming language such as C and using OpenMP Fortran directives. OpenMP extensions are used to express multiple levels of parallelism and processor groups in two well-studied applications.

Chapter 9 presents the evaluation of the complete NANOS parallel execution environment as has been implemented in the Origin2000 machine, starting with the evaluation of the overhead introduced by the user-level threads library, continuing with the evaluation of the performance obtained in individual parallel applications and terminating with the evaluation of several workloads. Along the evaluation, the NANOS parallel execution environment is compared with the SGI MP execution environment.

Finally, Chapter 10 contains the conclusions of this thesis and the work planned for the future.

Chapter 2. Programming Model

Abstract

This chapter presents the Nano-Threads Programming Model (NPM). Following this model, parallel applications are decomposed in tasks and represented through a Hierarchical Task Graph (HTG) structure. The way an application is decomposed enables the exploitation of fine-grain and multi-level parallelism. This chapter highlights the requirements that NPM needs from the user-level execution environment and the operating system. Finally, it introduces the programming language we have used to parallelize applications.

"El mar és com és, un gran amic enigmàtic de caràcter desigual. Moltes de les seves coses, no és necessari entendre-les."

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

2.1. The Nano-Threads Programming Model (NPM)

The complete programming model defined in this work is based on the Nano-Threads Programming Model (NPM) [112]. The reason for using this model is that it provides a means for detecting, representing and exploiting both fine-grain and multiple levels of parallelism from existing applications.

The Nano-Threads Programming Model was first introduced in [112][116] to provide highly optimized light-weight threads. It has been further developed in [92][127]. As the model defines, the *parallelizing compiler* identifies the maximum parallelism contained in the application through *data* and *control* dependence analysis and generates an intermediate representation of the parallel application taking the form of a *Hierarchical Task Graph* (HTG) [92][112]. From the HTG, the compiler generates parallel code using the services provided by a threads package interface. During code generation, the compiler statically determines the finest granularity of parallel tasks worth to be exploited having in mind the efficiency of the user-level package implementation. Using the threads package and operating system interfaces at run-time, the program is also able to group tasks in order to use a coarser level of granularity, adapted to the actual system conditions.

2.1.1. The Hierarchical Task Graph (HTG)

The hierarchical task graph (HTG) is a graph composed of *simple* and *compound* nodes. Simple nodes contain sets of operations that need to be executed sequentially or do not represent enough work to be executed in parallel. Compound nodes encapsulate a new level of the hierarchy containing more complex computations. They are composed of simple and compound nodes. Typical compound nodes correspond to loops and complex blocks of code containing parallel sections. Compound nodes contain two special control nodes: the *start* and the *stop* nodes. The start node is the entry point for the compound node and manages the execution of the internal nodes; the stop node is the exit point that manages and signals to the successors the completion of the corresponding compound node. Nodes at the same level of the hierarchy in the HTG are connected by directed arcs if there are data or control dependences between them. A node y is *data dependent* on node x if they access the same data and at least one of them modifies the data. A node y is *control dependent* on node x if the execution of y is decided by the execution of node x. Node x is the *source* of the dependence and node y is the *sink*. Dependences impose a partial order on the execution of the nodes.

Figure 11 shows a sample program and the HTG structure derived by the compiler. The HTG has three levels of nodes: at the outer level of the hierarchy, there is a single compound node representing the whole program. At the next level, nodes *PROG* and *END* are the start and stop nodes, respectively, of the main compound node. Inside the main compound node, one can find four compound nodes representing the three procedure invocations (*z()*, *g()* and *h ()*) and the *DO* loop. The node representing the loop contains three simple nodes (representing the assignment statements to arrays A and B, and the evaluation of the condition in the *IF* statement and three compound nodes (representing the *THEN* and *ELSE* bodies and the invocation of function *f*). Dependences are represented by directed edges. Solid edges represent data dependences. Dashed edges represent control dependences. For instance, the compound node representing the invocation of procedure *h()* is data dependent on nodes

representing the invocation of *z()* and the *DO* loop and it is control dependent on the start node *PROG*. Both the *THEN* and *ELSE* bodies are control dependent on node *IF*.

We define a *task* as a collection of HTG nodes at the same level of the hierarchy that have enough granularity to be executed as a user-level thread. The compiler decides the finest granularity for the tasks taking into account the overhead of the user-level execution environment. For instance, in Figure 11, nodes *A*, *B* and *f()* have been joined in a task. Dependences between nodes of the HTG define *precedence relations* between the corresponding tasks. These precedence relations establish a *predecessor/successor* relationship between the tasks, that must be preserved in their parallel execution. The compiler generates a function for each task in such a way that the code generated for the application can be seen as an executable representation of its HTG.

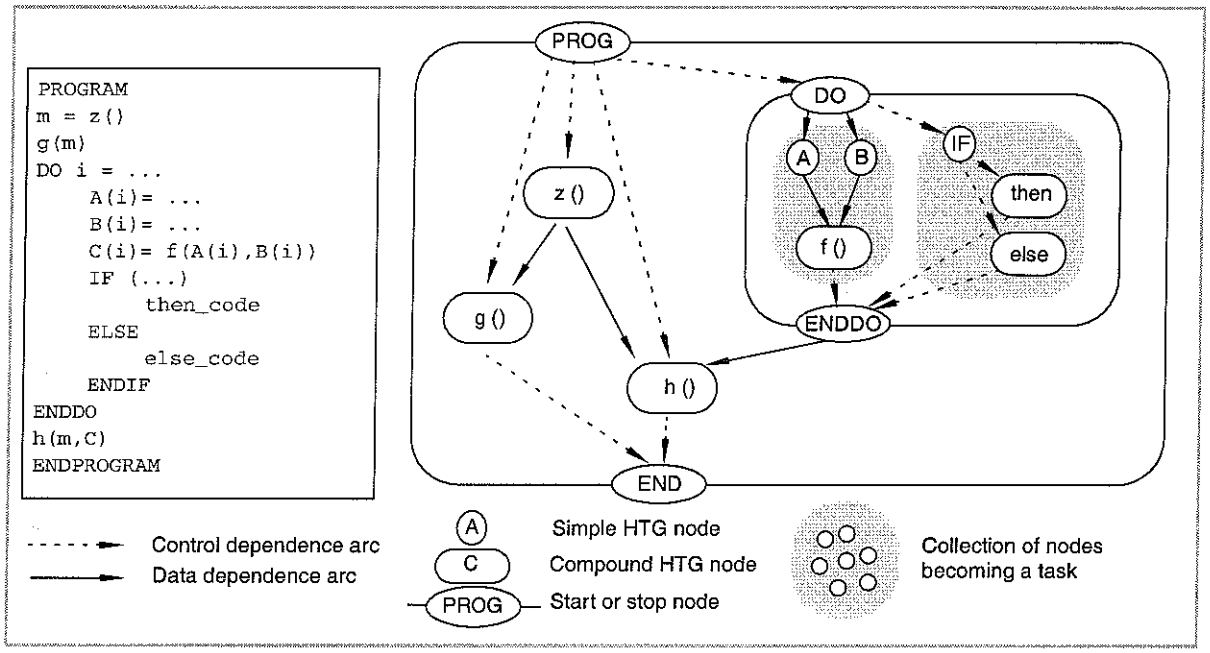

Figure 11: Sample program and its associated HTG

2.1.2. The HTG execution mechanism

The execution of the parallel program consists of the execution, in some order, of the functions associated to the HTG tasks. The execution order must ensure that all the precedences for a task are satisfied before its associated function is executed. A *nano-thread* corresponds to an instantiation of an HTG task in the form of an independent user-level control flow.

A task can be instantiated as a nano-thread when at least one control dependence is resolved by a predecessor task. The user-level execution environment offers nano-threads to instantiate the application tasks as user-level threads. One nano-thread instantiates one or more tasks, depending on the specific level of the hierarchy and the current availability of processors running in the application. It is possible that a nano-thread executes several tasks sequentially in case of lacking processors. The nano-thread instantiating a task may not be ready to be executed as some data dependences may be still unresolved. As soon as all data dependences for it have been satisfied, the nano-thread is ready for execution. The user-level execution environment offers a user-level ready queue to keep all those nano-threads waiting for available processors.

The execution of an HTG begins with the main node being prepared for execution and inserted in the ready queue as a nano-thread. Virtual processors assigned to an application use auto-scheduling [92][110] to execute a nano-thread selected from the ready queue. Auto-scheduling means that processors search for work independently from each other.

A compound node is a candidate to create new parallelism. The execution of a compound node begins at its *start* node. The function generated for the start node is a control function, which is in charge of the application level scheduling of the tasks enclosed in the compound node. This function evaluates, at run-time, the availability of resources and the parallelism it could generate based on the precedences among its internal tasks. Using that information, it decides whether to create nano-threads for its internal tasks or to execute himself the computations of the internal tasks, avoiding to manage too many contexts and reducing the associated overhead. In case it decides to spawn parallelism, the precedences observed among the internal tasks of the compound node are represented as predecessor/ successor relations among nano-threads. For this reason, nano-threads have to be created from the bottom to the top. For instance, the four nodes inside the main compound node of Figure 11, are created starting with *h()* and *g()* and continuing with *z()*, and the *DO* loop.

The function associated with the *stop* node of a compound node has a double purpose as defined by NPM. On one hand, it is in charge of satisfying the precedence relations of the successor nodes, allowing them to execute, when all their precedences have been satisfied. On the other hand, it can check the execution conditions looking at the information given by the operating system level. This is used to detect, and correct when necessary, at user-level, any kernel-level scheduling event influencing the execution of the application, such as processor preemptions.

In the following two sections, the requirements imposed by NPM into the user-level execution environment and into the operating system are examined.

2.2. Requirements set by NPM on the user-level environment

The Nano-Threads Programming Model introduces a specific parallelization scheme, and a great coordination among the different levels of operation, both aspects different from other parallelization models. The supporting user-level run-time environment should take the differences into account. In this subsection, we consider the ways the model can be supported by a user-level execution environment. Such environment is influenced by the application and the operating-system levels, and it has to provide solutions to the requirements of both.

2.2.1. User-level resource identification

All thread packages provide some means of thread self identification. Knowing thread identifiers is usually enough to work with a traditional threads package, where the application level is not aware of the processors where the parallel work is executed, such as in Pthreads.

Knowing thread identifiers only is not enough to support the NPM. The reason is that the application is more aware than in other parallel environments of its own structure (represented through the HTG) and the mapping of such structure to the processors provided by the operating system. As a result, the application level wants to identify which user-level thread is executing a specific portion of the parallel work. Also, the application wants to be able to map such parallel work to one of the virtual processors offered by the operating system.

The user-level execution environment has to provide the basic tools to allow the application to correctly map its parallel structure to the available processors. Tools include naming of virtual processors, along with each processor status, whether each processor is available or not and the reason why (e.g., it has been preempted, it is blocked, etc.).

Processor and thread identification is commented in more detail in Subsection 3.3.1.

2.2.2. Spawning parallelism through ready queues

NPM does not define any scheme for parallelism spawning (work generation) from the application level. Any work generation scheme (static, dynamic, guided self-scheduling, etc.) can be appropriately mapped to NPM to allow the application to generate work to processors in the way that favors data locality, load balancing or both. This means that the run-time execution environment should support the standard and well-known application-level scheduling algorithms [83]. Scheduling algorithms on NPM are explained in more detail in Subsection 3.2.2.

In addition to the support of application-level scheduling algorithms, the user-level execution environment has to support a mechanism to represent predecessor/successor relationships between application tasks. Representing precedences between tasks is accomplished by specifying the tasks which have to be executed after a given one, at creation time. This feature is explained in more detail in Subsection 3.3.3.

Supporting a single level of parallelism can be achieved through a work descriptor located in a known memory area, from which all processors get the work to execute. For multiple levels of parallelism, data structures supporting the description of parallel work have to allow that several processors generate work at the same time. The descriptions of different tasks are going to co-exist to be executed by the same or different processors. A common data structure useful to support several task descriptions co-existing at the same time is a queue. The user-level execution environment has to offer ready queues to keep work ready to do, but having no processor on which to execute.

Work spawning can be done in a global way, in the sense that any available processor can get its work from a global pool (a global ready queue can be used for this purpose). Or otherwise, work can be supplied to a specific processor or group of processors, introducing the need for local work generation and virtual processor identification at application-level. As a result, both global and per-processor local ready queues are required by NPM and should be offered by the user-level execution environment.

2.2.3. Waiting for work

The user-level execution environment is responsible of managing the processors when there is no parallel work to execute, and the application decides not to free the processors (e.g., when the application is executing in a short sequential section). While the application does not spawn parallelism, the user-level execution environment maintains processors actively waiting for work. This is performed by means of an idle function. Each potentially available processor has the option of invoke the idle function when it does not have work to do. Global and per-processor local ready queues have to be periodically examined by idle functions.

In addition, the operating system may claim some of the processors executing in the application to be returned to the operating system. Idle functions also check for the operating

system conditions and they answer accordingly. This is an extra functionality supported by the idle functions in a dynamic resource allocation environment, like the one proposed by NPM.

Idle functions should work in a *safely* way, that is without getting any mutual exclusion. This is important because this behavior makes easy to release a processor from the idle function code. Also, idle functions can be safely preempted at any point, without compromising the execution of other threads, due to synchronization issues. For these reasons, we say that the idle functions work is a *safe* point for preemption.

The complete functionality of the idle functions inside the user-level execution environment is presented in Subsection 7.1.9.

2.2.4. Multiple levels of parallelism

Supporting multiple levels of parallelism is one of the most complex features included in the user-level execution environment. Next subsections sketch how data belonging to different levels of parallelism is available and it can be accessed and a way to achieve efficiency when supporting multiple levels of parallelism.

2.2.4.1. Local address spaces

Supporting multiple levels of parallelism requires that the execution at each level maintains a set of local variables which can be used not only by the current execution level, but also by all the enclosed levels. So, such set of variables has to be maintained during the execution of the most internal levels. Several alternatives were studied in order to find the best choice having in mind all possible aspects of the problem:

- Local variables could be statically allocated in the data segment. This solution does not allow to spawn any parallelism because the variables can not be replicated for each thread. Old Fortran compilers allocate local variables in the data segment. New compilers do that in the local stack. Parallelizing compilers need to allocate variables in the local stack.
- Local variables could be allocated in the heap. This means an extra memory management for thread address spaces, using implementations similar to malloc/free, but with the extra of having to know how many threads are accessing each local address space and with the addition of some kind of garbage collection to release unused memory.
- Another solution consists of using a cactus stack. Using cactus stacks, the local variables are allocated in the stack of the parent thread. They are accessed through a static link, supplied from the parent to the child threads. The drawback with this alternative is that the compiler must be aware of the cactus stack structure and it has to generate special code to access local data belonging to the parent threads [92].
- Local variables could also be allocated in the stack of the executing thread (the parent of the parallelism), as in the cactus stack solution and all references to local variables needed by the children threads can be passed to them as parameters.

We adopted the last solution. This solution makes the user-level threads package to provide primitives for thread creation with a variable number of arguments. As it is seen along this thesis, the overhead introduced by argument passing is very small and can be assumed for the benefits obtained by the new functionalities provided. Another advantage of argument passing is that each thread can access variables in a standard way, and any compilation back-end can be used to generate the machine language code from the parallelized Fortran/C source code, relying on the efficient local variables allocation mechanism used by the standard

compilation back-ends. This scheme avoids the need for dynamic allocation of address spaces from the heap [92]. An extra heap management does not agree with our design decisions. The library already handles a heap from where it allocates memory for nano-thread stacks. Our goal is to avoid two different memory managements in the nano-threads package.

As a consequence of allocating local variables in the parent thread stack, the parent thread should be blocked during the execution of the children, in case it can not proceed with the execution. In this way, as a side effect of taking this solution, the parent and successor HTG nodes have been joined in one nano-thread with a blocking primitive, thus maintaining the nano-thread stack where data resides.

Taking the previous discussion into account, in order to execute the application, nano-threads access two kinds of application data using standard mechanisms:

- Global application data, as defined by the application programmer through the high level language (Fortran or C), resides in the global shared data segment.
- Function and subroutine arguments and local data and nano-thread local data, defined respectively by the programmer through the high level language (Fortran, C)[1] and by the parallelizing compiler; in our implementation, they reside in the nano-thread stack. We call this set of data the address space of a nano-thread.

Nano-thread code access application global data using the same addressing modes than in a standard sequential program. If access to global data has to be done in a certain order, a data dependence arc will force to execute one nano-thread before the other. The HTG definition and the nano-thread sequence control mechanism make unnecessary mutual exclusion between nano-threads. Only inside the nano-threads library mutual exclusion is necessary to access shared internal library data.

Allocating the local address space of a compound node in the stack of the start node may require to maintain it when the start node terminates, till the stop node is executed. To see that, consider the execution of the compound node shown in Figure 12. The nano-thread instantiating the start node allocates in its stack the local variables of the function. To do it, the compiler uses the standard local variable allocation mechanism (initializing the frame pointer and decrementing the stack pointer). Then, the start node nano-thread instantiates the body of the routine, launching the internal nodes nano-threads (*seq_i*). As an option, any of the parallel sequences could be joined with the start and stop nodes to build a larger task to be executed as a nano-thread. This is indicated in the figure through the shaded area.

Every internal node nano-thread has to receive a pointer to a local variable in order to be able to reference it. In addition, the start node nano-thread would have to wait for the termination of the internal nodes before destroying the local variables of the function. The address space destruction consists in the standard deallocation of local variables (restoring the stack pointer from the frame pointer). This operation corresponds to the stop node. As a result, the code for the start and stop nodes has to be joined in one function using a block operation, in between, to wait for the rest of the parallel sequences.

1. Standard FORTRAN-77 subroutine local data declarations are allocated in the data segment, not in the stack. FORTRAN parallelizing compilers (and C compilers) use the stack to allocate subroutine local data.

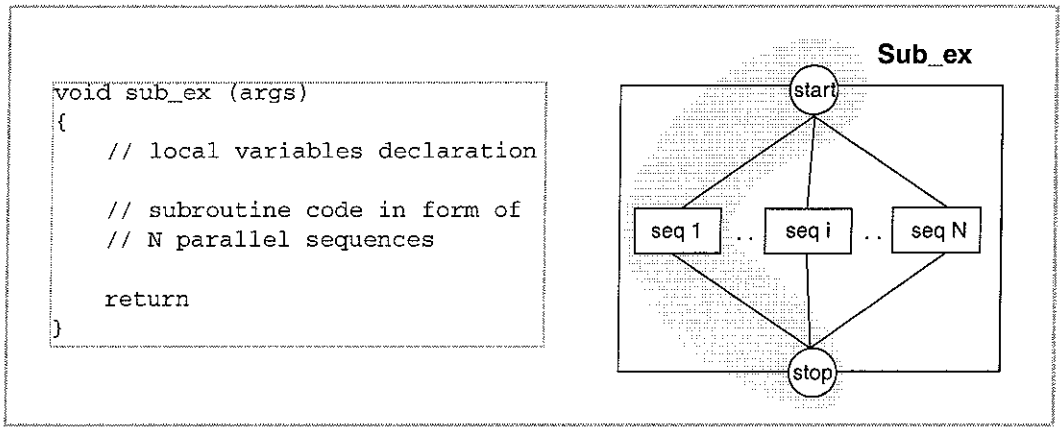

Figure 12: Compound node associated to a subroutine.

2.2.4.2. Support for fine-grain parallelism

Nano-threads provide the basis for supporting multiple levels of parallelism. Several parts of the application can proceed in parallel while spawning more parallelism. Each part of the application is depending on the nano-thread which started it, working with its own local address space, independent from the other parts of the application.

Nevertheless, nano-thread creation and management imposes more overhead than the strictly necessary when only one level of parallelism is exploited, or even when the inner-most level of parallelism is exploited. Overhead comes from the fact that each nano-thread is defining an address space. This is necessary for the outer levels of parallelism, but not for the inner-most level.

In order to reduce the overhead of spawning parallelism in the inner-most level, a different mechanism can be provided. Resembling the parallel execution environments which provide a single level of parallelism, this mechanism is based on work descriptors. A work descriptor is a data structure describing the work to be done in parallel. Work descriptors are created before spawning the parallelism and they are supplied to the executing processors. Each processor gets a pointer to the work descriptor and starts the execution of a portion of the work described. The exact portion depends on the self identification. This mechanism is thought for the inner-most level of parallelism.

This is the way we propose for integrating, in an efficient way, multiple levels of parallelism in a single environment. Outer levels of parallelism are spawned using threads providing an address space. The inner-most level can be spawned more efficiently using a work descriptor. Idle functions search for both work descriptors and new threads to execute. When a work descriptor is found, it is executed simply by calling the associated function. When a new thread is found, the idle thread makes a context switch to execute it.

2.3. Requirements set by NPM on the operating system

The operating system is in charge of distributing processors among the executing applications. The NPM imposes that all changes in processor allocations done by the operating system and relevant to the applications have to be communicated to the applications [118]. In this section, the requirements for applications adaptability and operating-system scheduling policies are presented.

2.3.1. Application adaptability to the available resources

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

With respect the first aspect, the nano-thread starting the execution of a compound node takes the decision whether to proceed in parallel or to execute itself (sequentially) the computations of the internal nodes. The operating system has to provide some interface to allow the application to check which the number of processors available for spawning parallelism. Checking this number just before spawning parallelism, the application ensures that it is going to use all the processors allocated to it.

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

The third aspect, applications being able to adapt to processor preemptions, requires also some help from the operating system. The operating system moves processors from one application to another following some scheduling policy. The requirement from the application point of view is that preemptions do no occur. As this is usually not possible, the run-time execution environment may help to provide such a feeling, by recovering preemptions. A good solution from the operating system point of view is, on one hand, to provide some mechanism to reduce preemptions at a minimum. And at the other hand, to provide a complete interface for preemption recovery. This is explained in Chapter 5.

2.3.2. Operating system scheduling policies

Kernel-level scheduling usually also consists of a set of kernel-level scheduling policies which the operating system applies to distribute processors to applications. Several kernel-level scheduling policies help in achieving good results in performance inside the NPM.

At any time, there is a current active scheduling policy, applied to all applications running in the system. The active policy can be dynamically changed without incurring any overhead to the running applications. Applications only notice the different performance results obtained from the processor allocation decisions taken by the policy newly established. Different application workloads can benefit from different policies [33][75].

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

characteristics, such as a NUMA memory subsystem can also be taken into account at that point.

The benefit of looking at applications as a whole is that processors know where to look for work first (the application where they are assigned to). In case the application has no work to perform, its cooperation with the operating system makes it to release some processors, which will search for work in other applications. The scheduling policies implemented and evaluated in this work are presented in Subsection 5.3.

2.4. Programming language

The Nano-Threads Programming Model is a good way of representing the parallelism found inside applications, but sometimes the compiler finds code structures that are difficult to analyze and parallelize. With the purpose of increasing the expressiveness given to programmers, we use an extended set of OpenMP directives [107] for expressing parallelism in Fortran.

2.4.1. Goals of the OpenMP directives

Expressing parallelism at application level is usually a hard work for application programmers. Sequential applications can be parallelized by hand after a short/long period of study of the application and the parallel execution environment, depending on the complexity and size of the application. Fortunately, compilers provide some help, by allowing programmers to annotate source code with directives. Directives in Fortran and pragmas in C are the standard way for expressing parallelism at application level.

The main goal of the current OpenMP directives proposal is to provide a powerful standard method for expressing parallelism in Fortran applications written in a sequential form. Most applications are first written sequentially and later parallelized. For this reason it is very important that the tools for expressing parallelism in sequential codes become more and more expressive, easy to use and portable.

OpenMP directives are more powerful than previous directive standards (for instance, than Parallel Computing Forum - PCF - directives). New clauses have been added and there is also the possibility of expressing multiple levels of parallelism. This feature is very important for using these directives in conjunction with the NPM.

2.4.2. Expressing parallelism in OpenMP

The sample program presented in Figure 11 could be parallelized using OpenMP as is presented in Figure 13, where three levels of parallelism are expressed, although not all the relationships between parallel tasks are represented. For instance, the HTG structure shows that tasks *g()* and *h()* can be executed in parallel and this is not possible in the OpenMP version. In this thesis, we are using, along with the standard OpenMP directives, some extensions to provide the programmer with more expressiveness. Extensions include the expression of processor groups and the ability to set predecessor/successor relationships among several parallel sections.

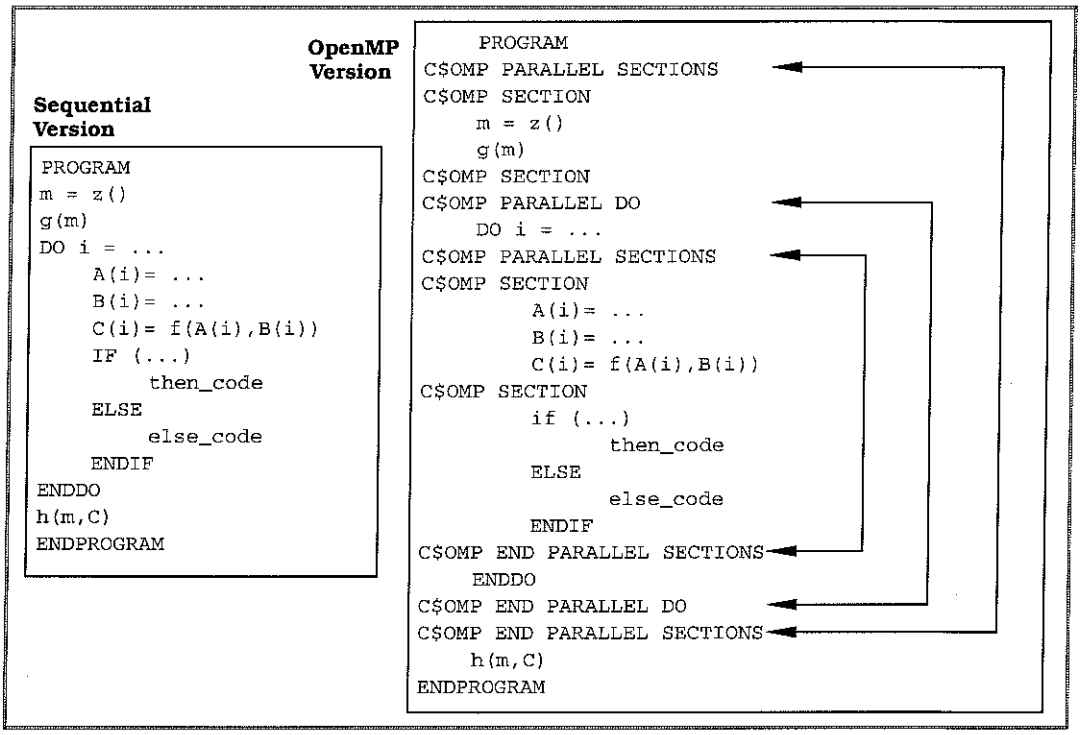

Figure 13: Sample program parallelized using OpenMP directives

When expressing multiple levels of parallelism, the programmer wants to control how to distribute the parallel tasks among the participating processors, establishing groups of processors. This is the motivation of the first extension to the OpenMP directives. Each time parallelism is spawned, the programmer can express in which processor(s) the new parallelism should be executed (see Subsections 4.2.1.2 and 8.2.1).

The second extension to OpenMP allows the programmer to set the predecessor/ successor relationships among several parallel sections, allowing that some of them can be executed in parallel, but sequentially with respect to others. This approach can reduce the amount of unbalance that several parallel sections could have among them.

Using the OpenMP directives in this work, we have achieved the following goals:

- Easy the task of parallelizing applications. Parallelizing applications with directives is easier than code them by hand to run in parallel.
- Offer a standard parallelization environment. OpenMP directives are being impulsed by the OpenMP organization and they are a powerful standard proposal for expressing parallelism.
- Enforce portability. Applications parallelized on the NPM can be run in other OpenMP - based execution environments. Applications parallelized using OpenMP can be run on the NPM.
- Use some extensions to the original OpenMP proposal, providing support for them from the NANOS user-level execution environment and test their usefulness to express and exploit parallelism.

Chapter 3. Scheduling

Abstract

This work considers that a complete parallel execution environment can be divided into three levels of operation (application, user-level execution environment and operating system). This division establishes three changes for mapping the entities of one level to the entities of the next level. This is commonly known as scheduling.

The application level needs to be able to map application tasks to user-level threads. The user-level environment must offer efficient user-level threads supporting multiple levels of parallelism and mapped to the virtual processors offered by the operating system. The operating system has to map virtual processors to physical processors and communicate all application-related scheduling events to the user-level.

"'Excuse me' - Barnaby said - 'but this isn't the Academy. And a student's thesis is a long way from a workable plan.'"

"Descent", Star Trek, The Next Generation, Pocket Books, Oct. 1993.

3.1. Scheduling levels

The Nano-Threads Programming Model (presented in Chapter 2) is the starting point to achieve the goals of this thesis, that is, to efficiently support both global-system and individual- application high performance in shared-memory parallel systems, achieving a good cooperation among the different levels of execution.

In Chapter 2 we have introduced the requirements needed for each one of the three levels of operation, namely application, user-level run-time environment and operating system. In this chapter, we focus in the cooperation established among them. There are several mappings among application, run-time and operating system levels which are the key to provide high performance:

- The application is responsible for the mapping of application-level tasks to user-level threads supported by the threads package.
- The user-level threads package is responsible of the mapping of user-level threads to kernel-level threads (virtual processors), provided by the operating system.
- The operating system is responsible of the mapping of kernel-level threads to processors.

How to establish the mappings between the entities used in the different levels of operation is usually known as application-, run-time- and (operating-system) kernel-level scheduling. The subject of this chapter is the design of the scheduling techniques at the three levels and the cooperation established among them.

3.2. Application-level scheduling

After the NANOS compiler has decomposed the application in tasks and built the HTG, each task (equivalent to a node in the HTG) can be executed in parallel with all other tasks with which there has no dependence relation. Direct execution of the HTG structure *as is*, following the paths defined by the precedences between nodes, would provide the largest amount of parallelism. Assuming there would be no parallelism management overhead and assuming infinite resources for parallel execution, direct execution would provide the best performance.

Anyway, coming to the actual world, parallelism management overhead is noticeable and the number of processors available is limited. The latter is solved through the kernel level scheduling, which distributes processors among the applications. Management overhead is the reason for introducing application level scheduling. The user-level execution environment can also use scheduling to map application generated threads to operating-system virtual processors.

A first step in application-level scheduling, which we consider already done when executing a parallel application, is to make the HTG as coarse grained as possible. This means that the compiler joins both dependent nodes and nodes too small to make worthwhile the spawning of parallelism. The potential overhead introduced by the parallelization is limited in this way. So, the applications executed on top of the NPM are already tuned for parallel execution, avoiding spawning parallelism for tasks too small. The NANOS compiler is in charge of this rearrangement of the HTG structure, before generating the parallel code.

A second step consists of allowing the application to decide the amount of parallelism to spawn with respect the available processors (see Subsection 3.2.1). In the third place, depending on the structure of the parallelism, the application wants to use different work

generation schemes to provide work to the processors. Such schemes are well-known scheduling policies such as static, interleaved, guided, trapezoid, etc. (the mapping of such scheduling policies to NPM is presented in Subsection 3.2.2). In the fourth place, maintaining locality is a key aspect to achieve high performance (see Subsection 3.2.3). In the fifth place, the application wants to express unstructured parallelism through the predecessor/successor relations among parallel regions of code (Subsection 3.2.4). Finally, there is the question about whether multiple levels of parallelism and processor grouping are affordable (This issue is discussed in Subsections 3.2.5 and 3.2.6).

3.2.1. Deciding the amount of parallelism

The application can determine, during run-time and from the interface with the user-level execution environment, the number of processors it is running on and decide the amount of parallelism to spawn. Depending on the conditions of the actual execution environment, the application can actually spawn all the parallelism available at a time, only spawn a portion of it, or even decide to proceed sequentially.

The amount of parallelism spawned at a given time is independent of the application-level scheduling scheme used. Alternatives provided to applications range from spawning all the parallelism at a time in a static way till spawning a portion of the total amount of parallelism in a dynamic way. In the first case, the work is distributed as evenly as possible among the available processors at a time. Processors execute their corresponding part of the work till each one finishes and signals termination. This is usually a good alternative for small parallel regions, in which the overhead of parallelism should be maintained as small as possible. Optionally, each portion of the work can be assigned to a specific processor (under application control) to maintain data locality as much as possible. The disadvantages of this static approach are that load balancing is not possible, as the work is distributed once, with no option to reconsider the decision. Also, in case new processors are allocated to the application during the execution of the parallelism, they can not participate in the parallel work again because the initial decision can not be reconsidered.

In the second case, spawning a portion of the parallelism in a dynamic way, all work will be available to all processors and they will pick it up from a global structure. Although this approach prevents the preserval of data locality, it favors load balancing. Also, it allows to reconsider the decision of spawning parallelism, taking again into account whether the number of processors allocated by the operating system has changed from the last decision point.

Other approaches and scheduling algorithms for spawning parallelism are discussed in Subsection 3.2.2. This is also related to the requirements needed for the user-level run-time execution environment (presented in Section 2.2). What usually happens is that when an application spawns all the parallelism at a time, it uses the most efficient way (work descriptors) to represent the work to be done. Instead, dynamic spawning of the parallelism must paid an extra overhead to allow the expression of the later continuation of the spawning operation, which is done through the creation of nano-threads.

Static approaches are usually better when dealing with fine-grain parallelism because of its reduced overhead. Also, it is used in small machines, where the number of processors allocated to an application is not going to change by a great amount. On the other hand, as the parallelism becomes coarser or the parallel environment offers more processors, the dynamic approaches can be more suitable. In any case, in current machines data locality is a precious issue. Usually, preserving data locality (Subsection 3.2.3) is more important than allowing

dynamic application-level scheduling [83]. This is the reason why we have given enforced support for the static approaches.

3.2.2. Application-level scheduling policies

Orthogonally to the decision of how much parallelism to spawn at any moment, there is the question of how to distribute work among the processors. The policies proposed in [83] for loop scheduling have been taken into account to provide the proper support for application-level scheduling.

The set of application-level scheduling policies which we have tested on NPM is composed by the following:

- Static; The application distributes work among the available processors as evenly as possible. Chunks of N/P consecutive iterations are assigned to each processor, where N is the amount of iterations which is distributed at this time, and P is the number of available processors at this time. N may be the complete set of iterations or vary during execution. In case N is the complete set of iterations, the decision is taken once. Otherwise, during the execution of the entire loop, the decision of spawning is evaluated several times. Locality is enforced by assigning the same data to the same processors, being the application which controls the assignment of iterations to processors.
- Static with chunk=C, also known as Interleave; The application distributes work among the available processors in chunks of C consecutive iterations. Each processor has to execute N/C chunks. Again, the decision of spawning can be taken just once, in case all loop iterations are considered at a time, or several times if only a portion of the complete set of iterations is used at any time. This scheme is useful for triangular loops, where the amount of work per iteration greatly varies from iteration to iteration. Locality is also enforced by assigning the same data to the same processors, under application control.
- Dynamic with chunk=C; The dynamic scheduling policy sets a global work descriptor from which all processors pick up work. Each processor gets C iterations at a time, executes them and tries to get C iterations more, and so on, till the work is exhausted. This is useful for unbalanced loops, where there is no previous knowledge about the amount of work to be done in each iteration. Data locality is not preserved because the processors get work randomly.
- Guided self-scheduling (GSS [113]); GSS begins assigning large chunks of iterations. Each processor receives as much work as the number of remaining iterations divided by the number of allocated processors (N/P). Following this formula, chunks quickly decrease in size till all the work has been generated.
- Trapezoid self-scheduling; Trapezoid starts with chunks smaller than GSS, searching for a compromise to avoid giving too large chunks. It assigns first N/2P iterations to each processor. The size of the chunk decreases slower than in GSS (at a rate of N/(8*P*P)).
- Adaptable-size chunking [86]; This algorithm is explicitly thought to be used in a dynamic environment, where processors quickly move from a one application to another. It generates work using two chunk sizes. Processors receiving the larger chunk size are assumed to be stable in the application, so are going to work with small overhead. Processors receiving small chunks are candidates to leave from the application first when the operating system requests to do so.

To avoid taking the decision of spawning parallelism only once, all policies are able to be combined with factoring, also from [83], to execute the parallelism in bursts. Before

generating each burst, the application has the opportunity of checking the number of available processors and decide whether to execute in parallel or not and compute the parameters for the scheduling policy in use to spawn parallelism next time. Figure 14 shows the behavior of the interleave and adaptable size chunking approaches when using bursts.

Figure 14a shows how work is generated when using an interleaved scheme and nano-thread bursts. For each processor (3 in the example) a first chunk containing two iterations is generated. After that, a dispatcher nano-thread is enqueued and the burst is completed supplying a second chunk for each processor. All nano-threads, including the dispatcher are enqueued in the global ready queue. When the first processor finishes executing, it picks up the dispatcher, thus generating more work to be done (the second burst), and continues working on the second part of the first burst. The goal here is to maintain the processors working with enough work generated without generating all the parallelism in a fine-grain manner at a time, which would spend a large amount of memory.

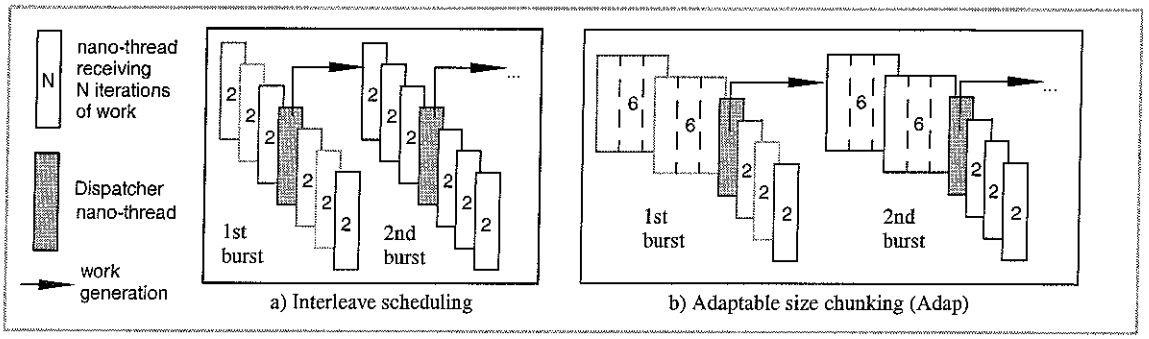

Figure 14: Work generation using burst-based scheduling algorithms

Figure 14b shows the bursts technique applied to a different scheme for application level scheduling. In this case, large chunks are merged with small chunks to reduce the overhead introduced through spawning parallelism. The idea is also to maintain at least one processor executing a small chunk of iterations to be able to quickly adapt to changes in processor allocation introduced by the operating system.

3.2.3. Locality issues

The application level relies on the services provided by the run-time library to spawn and manage parallel tasks. While designing a new run-time library, one has to think of which the needs of parallel applications are, having in mind the physical execution environment (hardware) where they are going to execute.

Our target machines range from small symmetric multiprocessors to production machines containing a large number of processors. A key aspect to consider is how much the application should be involved in the assignment of parallel tasks to processors. We have determined that it is very important to allow the application to guide the run-time library about where to execute its tasks. This is because the application can be aware of how the data access patterns are for each parallel construct. This issue is more important on NUMA architectures. The mechanism provided to the application consists of allowing work generation on specific processors.

Also, the compiler can help in achieving locality by generating code for each processor which correctly accesses the same data across different parallel loops. Sometimes,

the code inside parallel loops can be slightly modified to align the data accesses done by the processors with the data accesses done by the previous loops.

3.2.4. Dependent parallel regions of code

As soon as an application is represented through an HTG structure, the parallelism detected by the compiler can be highly unstructured in the sense that inside every node several dependant regions can be spawned in parallel if the supporting threads library allows to represent the precedences among them. Parallelization through directives is not usually expressive enough to allow such an unstructured representation.

Figure 15 shows an example consisting of a parallel application structure (Figure 15a), in which there are eight regions of code which are dependent as indicated in the figure (region 3 depends on region 2, region 5 depends on region 4 and so on). This parallel structure can be parallelized using two main approaches. The first one is used in traditional parallelization environments (Figure 15b). It uses a barrier synchronization to control the end of the four first parallel regions and signal the starting of the four second parallel regions. When the parallel execution shows some load unbalance among the different parallel regions, this approach sums the unbalances along the largest path (consisting of regions 4 and 3, in the figure).

The second approach, which is supported in the NANOS environment, is to allow the application to express the precedences among regions. Observe in Figure 15c how this approach is able to mitigate the effect of load unbalance due to the removal of the barrier synchronization involving all processors. Instead, when the execution of a parallel region terminates, the application automatically starts executing the dependent one. In general, the load unbalance is reduced with respect the traditional approach. The application can express any precedence relation among parallel regions. The extended OpenMP directives used for expressing such precedences are presented in Chapter 4.

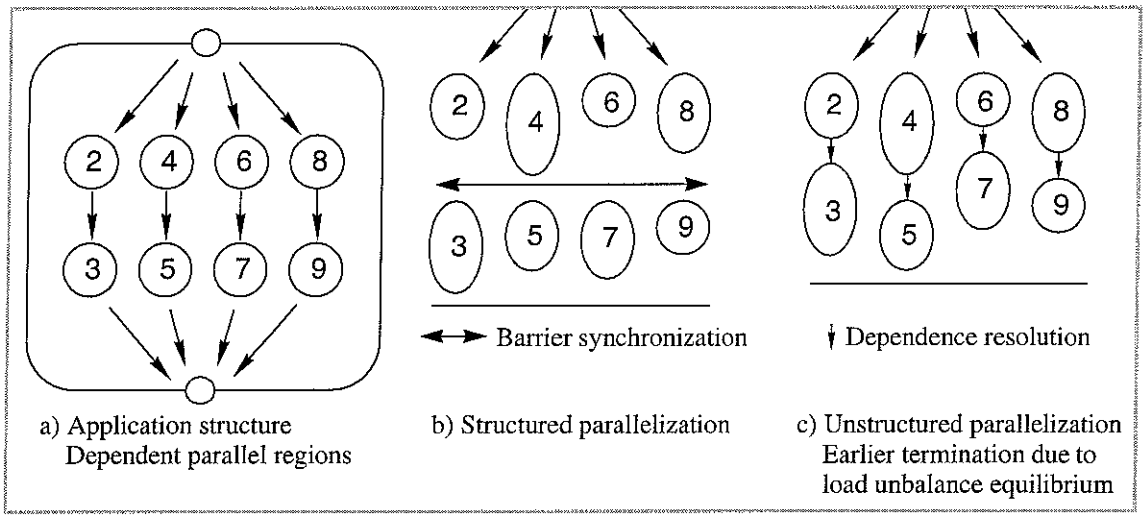

Figure 15: Structured vs. unstructured parallelization

3.2.5. Multiple-levels of parallelism

Supporting the execution of the hierarchical parallelism represented by the HTG structure is equivalent to provide multiple levels of parallelism. In this work, we have taken advantage of having the applications decomposed in a hierarchical way, to provide support for exploiting multiple levels of parallelism.

Figure 16 shows the structure of a parallel application which exhibits two levels of parallelism. At an outer level, regions 2, 4, 6 and 8 are independent among each other. The same happens with regions 3, 5, 7 and 9. Among them, region 3 depends on region 2 and so on. At an inner level, each one of the regions contains a parallel loop.

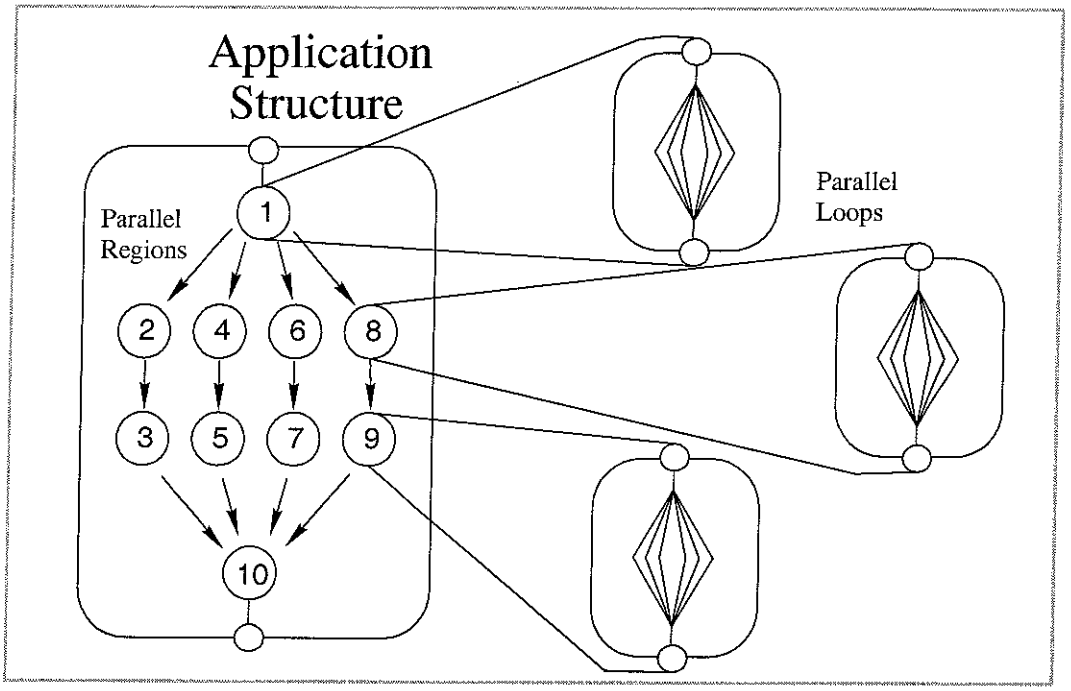

Figure 16: Application structure allowing multiple levels of parallelism

Figure 17 shows the differences between the exploitation of single and multiple levels of parallelism and the different approaches that can be taken in each case. When exploiting a single level of parallelism only, either the outer or the inner level of parallelism can be exploited, but not both. Figure 17a shows the result of exploiting the outer level of parallelism in the application represented in Figure 16, assuming that 16 processors are available. The outer level consists of two groups of 4 parallel regions. The parallelism is spawned in such a way that one processor executes one parallel region. If the run-time execution environment forbids spawning further levels of parallelism, no other parallelism can be exploited. In particular, parallel loops inside each one of the parallel regions have to be executed sequentially. As a result, only 4 processors are used in the outer level approach.

Figure 17b shows the parallelization of the inner level. This is usually the approach taken by most of programmers and parallelizing compilers because it is easy to detect parallelism in loops and it is the most commonly supported parallelism exploitation. This second approach allows to use all 16 available processors. Also, it is going to exhibit the best performance when the parallel loops are large enough. There are, although, locality issues that should be taken into account. For instance, in the example of Figure 17b, it seems better to

execute the loop in the way it is shown (regions 1, 2, 3, 4, and so on), instead of executing regions in the following order: 1, 2, 4, 6, 8, 3, 5, 7 and 9. This is because, if there is a dependence from region 2 to region 3, it is probable that a portion of the data used in region 2 is also used in region 3. So, it is better to execute region 3 immediately after region 2 than execute regions 4, 6, and 8 in between, polluting the cache memory of the processors before executing region 3.

Figure 17c shows a first approach for multi-level parallelization (all-to-all). Four parallel regions (2, 4, 6 and 8) are first spawned. The processor executing each region finds a parallel loop inside and spawns further parallelism. All parallel loops are executed by all processors (16, in the example). This means that each processor is going to receive the same portion of the parallel loop than in Figure 17b approach. The difference is that the work is supplied as soon as the parallel regions are spawned. This early spawning of parallelism provides, at a time, four times more work than in Figure 17b approach. This is a good solution to reduce the effect of load unbalancing because as soon as a processor terminates with the iteration of a parallel loop, it can proceed executing the next loop. The same execution can be achieved by eliminating the implicit barrier synchronization at the end of parallel loops in traditional parallelization environments.

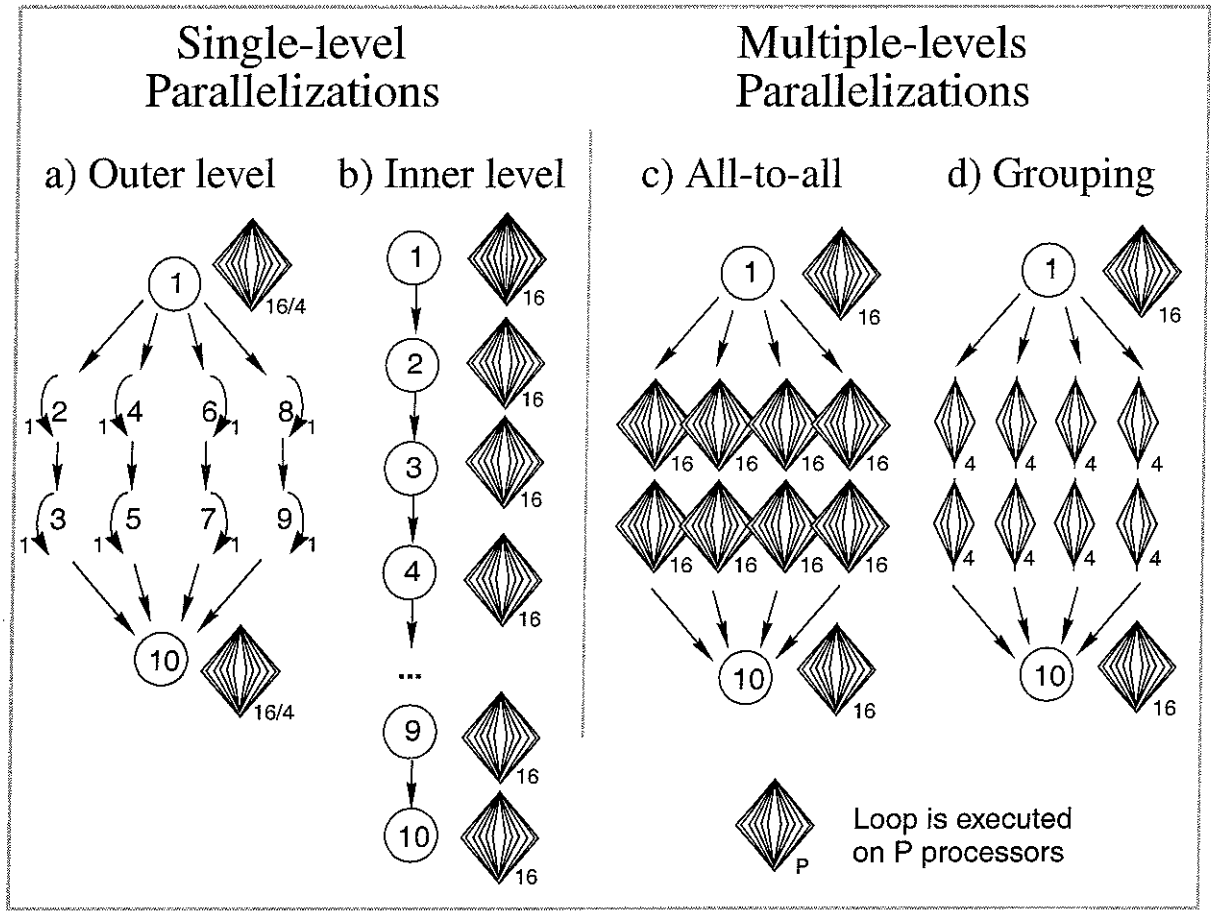

Figure 17: Single-level vs. multiple-level parallelization

3.2.6. Processor grouping

When allowing to spawn multiple levels of parallelism, the execution environment can provide tools to drive processors to execute at different parts of the application. It is very different to make all processors to participate in the execution of all parallel tasks, like in a single-level parallel execution, from guiding each processor to execute the desired tasks.

Processor grouping is the method we have selected at application level to guide processors to execute some (but not all) of the active parallel tasks. Figure 17d shows that approach (grouping) that can be taken when supporting multiple levels of parallelism. It consists of driving processors to independently execute a portion of the application. In the example, all parallel loops inside parallel regions are executed using 4 processors, for a total of 16 processors. A first group of 4 processors is used to execute regions 2 and 3, another group for regions 4 and 5, an so on. The advantage is that the inner level of parallelism is distributed among less processors than in the all-to-all approach. The working-set size assigned to each processor is the same than in the *b* and *c* approaches, but the number of different data structures used for each processor is smaller. In general, processor grouping makes larger the working set each processor takes from a parallel task, thus improving locality and reducing false sharing among the processors. This is of importance when the amount of work inside the inner level is big enough to be exploited using 4/8 processors, but it is too small to be exploited on more processors [106].

The method we use to drive processors consists of establishing the processor groups at every spawning point. All spawning points at an outer level of parallelism are able to drive some selected processors to each one of the parallel tasks, building a *group* of processors. One of the processors assigned to each task executes the task. It is the *group master* for the task it has started executing on. Other processors inside the same task are *group slaves*. Then, independently inside every processor group, each time the group master finds an inner parallel task, it spawns the new parallelism over the group slaves (sometimes we also call them as its *friends*; OpenMP uses *team*, instead).

3.3. Run-time library level scheduling

The purpose of the run-time execution environment is to coordinate the application needs with what the operating system offers. The application wants to execute tasks and the operating system offers virtual processors to execute them. The relation between a run-time library giving support to NPM and the applications has three important aspects to consider. First, the run-time library has to allow the application to map tasks to specific virtual processors (usually for data locality purposes). This can be done by exporting virtual processor identifiers to the application level (see Subsection 3.3.1). Second, the application also wants to stock several tasks at a time for execution. The library can support this feature through ready queues (see Subsection 3.3.2). And third, the application wants to express predecessor/successor relations between parallel tasks (explained in Subsection 3.3.3).

The relation of a run-time library supporting NPM and the operating system is based on the operating system interface providing information about the current status of processor allocation. The run-time library is in charge of tracking the status of each one of the virtual processors and decide low level processor movements to benefit the execution of the application. This will be explained in Section 3.4.

3.3.1. Resource identification and scheduling

The run-time library provides tools for virtual processor identification (also commented in Subsection 2.2.1). Virtual processors are identified by consecutive numbers starting from zero to the number of processors available in the machine minus 1; Gaps are not allowed. A decision here is to ensure that when the operating system informs an application that it has allocated P physical processors, the application automatically knows that those processors are represented by virtual processors numbered from 0 to P-1. No translation is required to know which virtual processors are available. This assumption can be broken when the operating system assigns/preempts processors. In these situations, the threads library is in charge of the situation, fixing a consecutive numbering of virtual processors and being aware of providing physical processors to all virtual processors, while the application terminates the currently spawned parallelism.

Using the virtual processor identifiers, the application indicates the mapping of nano-threads to virtual processors. After the mapping is set, it is maintained and the threads package ensures that virtual processors having work to do are eventually mapped on a physical processor. This means that the scheduling performed by the operating system may break the mapping between physical processors and nano-threads, but not between virtual processors and nano-threads. Checking often the number of allocated processors is a good way of minimizing the number of times that such mapping is hurt by the operating system scheduling.

3.3.2. Mapping application tasks to virtual processors

The application level instantiates its parallel tasks using nano-threads. Nano-threads are mapped to virtual processors labeling them with the virtual processor identifier. This is usually done at thread creation; it can also be done sooner or later after thread creation; or even, it can be done in a random way. The purpose of explicitly mapping tasks to virtual processors is to achieve data locality. Instead, allowing a random mapping searches for improving the load balancing.

The run-time execution environment offers a ready queue of ready-to-run nano-threads. The structure of the ready queue is as follows:

- A global ready queue serves the purpose of load balancing. All virtual processors search for work in the global ready queue. Nano-threads enqueued in the global ready queue are picked up at random by a processor and executed in it.
- Local per-processor ready queues are oriented to support data locality. One virtual processor searches for work in each local ready queue. For this reason, the local ready queues are identified with the identifier of the associated virtual processor. Nano-threads labeled with a virtual processor identifier are always enqueued in the matching ready queue.

Different implementations of the ready queues can be provided to the application level in order to provide simple/extended functionality without compromising performance when the extra functionality is not needed.

3.3.3. Precedence driven execution

From the application structure given by the HTG, application tasks are instantiated as nano-threads. Nano-threads are a run-time representation of the structure of the HTG through the expression of predecessors and successors among them. Figure 18 presents a sample HTG

annotated to show which characteristics are associated to each task when it is instantiated by a nano-thread. Nano-threads provide the address space for the execution of the HTG node. They maintain the predecessor/successor relationships and can be labeled to execute in specific virtual processors.

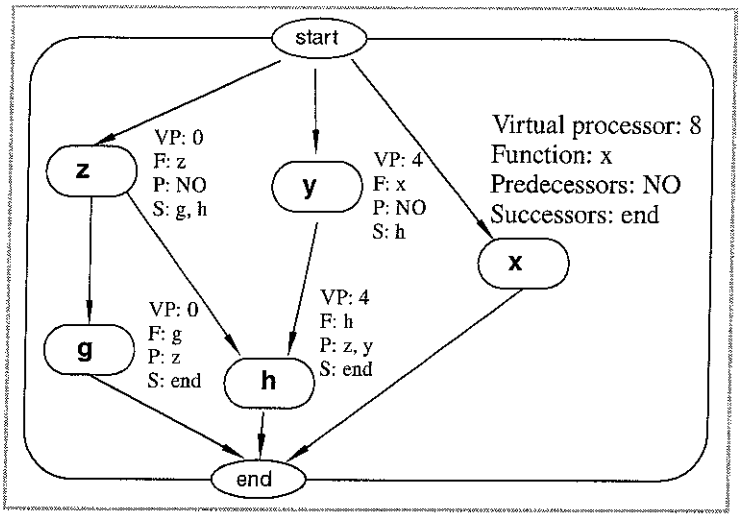

Figure 18: Sample HTG, nano-threads and precedence relations

For instance, in Figure 18, the nano-threads instantiating nodes *g*, *h* and *x* are created first, setting their successor to the *end* node. Node *y* can be instantiated as soon as *h* is. And node *z*, after *g* and *h* are instantiated. Instantiation always proceeds from the bottom of the HTG to the top. The node *h* has two predecessors (*z* and *y*). This means that as soon as nodes *z* and *y* have been executed, nano-thread instantiating node *h* can be enqueued in the ready queue for execution. When a node finishes execution, it has to signal its successors in case they have to be enqueued. In the figure, node *z* signals both nodes *g* and *h* because they are its successors.

When a nano-thread should be enqueued for execution, the run-time library takes into account whether the nano-thread is labeled with a virtual processor identifier (*VP*). When it is the case, the nano-thread is enqueued in the local ready queue of the given virtual processor. Otherwise, it is enqueued in the global ready queue. In the example of Figure 18, node *z* is enqueued at the very beginning to be executed in virtual processor 0 (vp: 0), node *y* in virtual processor 4 and node *x* in virtual processor 8. Nodes *g* and *h* are labeled for execution in virtual processors 0 and 4, respectively. This is useful to maintain data locality, when the application knows, for instance, that nodes *z* and *g* access a large amount of shared data.

3.4. Kernel-level scheduling

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

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

the system. The role of the operating system in processor scheduling becomes important when the load of the machine is high, so that a number of physical processors must be shared by a larger number of virtual processors. This work concentrates in providing new techniques and mechanisms for supporting well-known and new scheduling policies. The scheduling mechanisms are designed to enforce a close cooperation among the application, the run-time execution environment and the operating system. Cooperation is based on information shared across the different levels.

3.4.1. Sharing information with the upper levels

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

The information includes the number of processors on which the application wants to run at any moment and the number of processors currently allocated by the operating system to the application. From the number of requested processors, the operating system simply decides, in a first step, how many processors to allocate to each application. Processors are then moved, in a second step, from one application to another. It is possible that between the two steps, some time passes to allow the application to release the processors to be moved voluntarily. This functionality is designed to avoid as much as possible the preemption of running processes by the operating system.

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

3.4.2. Synchronization and processor preemptions

Each time the application needs to do some operation dependent on the number of running processors, it uses the number of processors allocated provided by the operating system. This ensures that, at least during a short amount of time, such processors are available (in average, during half a scheduling period or quantum). This means that, most of times, the processors are not going to lose a synchronization, so they are not going to delay the whole application execution.

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

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

3.4.3. The application as the scheduling target

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

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

The scheduling policies that can be applied by the NANOS operating system range from the well-known equipartition, batch or round-robin policies to other kind of policies that can make more use of the information about processors request.

3.4.4. Processor affinity

Processor affinity is an important issue to consider in kernel-level scheduling because of the different access latencies that have cached, local and remote memory locations. Cache memory is always of importance, both in SMP and CC-NUMA machines [146][94][141]. When a processor runs inside an application, the processor caches are filled with data which is usually accessed several times. Moving processors from one application to another causes a total or partial cache corruption. Processor affinity is useful for the cases where partial cache corruption occurs to take advantage of the data remaining in the cache when the processor is allocated again to the same application.

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

Scheduling at operating system level uses two levels of affinity. In a first step, a processor is assigned to an application where it has run before. In a second step, inside an application, a processor is assigned to a virtual processor where it run before, if any.

3.5. Complete interaction between the three levels of operation

As a result of the previous discussion about scheduling among the three levels of operation, the resulting environment behaves as follows:

- Each application dynamically informs the user-level execution environment and the operating system about its requirements (number of processors) reflecting the actual degree of parallelism that the application wants to exploit at user-level.

- The operating system distributes processors at least at fixed time slices, taking this information into account. It can also redistribute processors using other events, such as changes in processor requests.
- The application is informed about the operating-system allocation decisions and tries to match the parallelism that it generates to the assigned number of processors.
- The user-level execution environment is in charge of ensuring that the parallelism spawned by the application will execute as smoothly as possible, even when the operating system reallocates processors.

In more detail, when it is time for the operating system to reallocate processors, it applies the current scheduling policy, deciding how many processors is going to receive each application in the next time slice. As a result of this decision, some applications are going to loose processors. Then, it asks these applications for processors to be freed, optionally giving them a certain amount of time (the *grace time* [84][152]) to answer to the request by releasing the processors. If an application does not answer to the request in time, or when the grace time for that application is zero, the operating system will forcefully claim back processors through preemption, and inform the application. When some work has been preempted, the application always readapts at user-level when a running virtual processor reaches a *safe* point (see Subsection 2.2.3), by yielding the associated physical processor to a preempted process. A safe point is a user-level dispatching point, where the virtual processor knows that the application and runtime synchronization constraints are satisfied. Ensuring that all preempted virtual processors are stopped at safe points is very important in order to avoid preemption inside critical sections. To try to drive preempted virtual processors to safe points is critical to avoid situations where virtual processors are preempted while holding a user-level lock.

Processors moving to another application are allocated to them. The applications receiving processors are informed in such a way that the processors can participate either in the current parallel execution, if there is work available to perform, or as soon as the application checks the number of processors allocated from the operating system.

Chapter 4. User-level Interface & Functionality

Abstract

In this chapter we present the user-level interfaces designed and used in this thesis. First, we present the design of the user-level threads library (NthLib) interface and functionality that supports the Nano-Threads Programming Model and adapts to the available resources. The NthLib interface is used directly by the NANOS Compiler to generate parallel code. It can also be used to hand-code parallel applications, although this is usually a hard task.

Then, we present the OpenMP directives and extensions we have used to parallelize at source level the applications presented in chapters 8 and 9.

"Posa la teva mirada en el camí del cim, però no t'oblidis de mirar-te els peus. El darrer pas depèn del primer. No et pensis que ja hi has arribat, perquè veus el cim. Pensa en els peus, assegura el pròxim pas, però que això no et distregui de l'ideal més elevat. El primer pas depèn de l'últim."

Un admirador de la muntanya.

4.1. Run-time library

NthLib (the NANOS user-level threads library [87][88]) is a fundamental part of this thesis because it joins the upper part of the parallel execution environment (application/compiler level) with the lower part (operating system level), enforcing the cooperation between all three levels of operation.

In this section, we present the design decisions adopted for the development of NthLib, along with a complete description of its interface and external functionality. Chapter 7 sketches its implementation.

4.1.1. Design decisions

The nano-threads package is designed to support general, multi-level, unstructured and fine-grained parallelization of applications. As the parallelization is performed through a compiler, the package interface has been specifically designed to provide the functionality needed by the compiler. The current package interface consists of a basic interface plus some extensions. The basic interface allows the compiler to generate new nano-threads setting their successors and predecessors, controlling dependencies and queueing them in the ready queue. Extensions to the basic interface introduce mechanisms to efficiently schedule parallel loops. The interface allows that the code generated for an application could be seen as an executable representation of the application HTG.

The first design decision is to use standard compilation back-ends to generate the executable code. The code generated for each node is embedded into a standard C or Fortran function and standard activation frames are created following the same conventions used in sequential programs. Each nano-thread can access the application global data and some local private data. Global data is accessed in the same way the sequential programs do. Data dependences between nano-threads guarantee a correct data access order. Local data resides in the thread stack and it is also accessed using standard mechanisms. The library also provides a mechanism to allow threads to block, giving their children access to the parent local variables.

The second design decision, in accordance with NPM, is to create, at a specific point, the appropriate amount of parallelism that can be efficiently executed with the available resources. For large parallel nodes this implies that the decision of spawning parallelism is taken several times, correctly adapting the number of threads created to the resources currently allocated by the operating system. This establishes a difference between our nano-threads implementation and other thread implementations, such as Cilk [17], COOL [23], Filaments [44], Concert [27] in which the decision of spawning parallelism is taken once - and irremediably - for an entire section of parallel code. The main goal of our limited creation decision is to maintain the ready queue with enough work to be performed in the future, avoiding whenever possible that it becomes empty, but without wasting memory to represent a lot of threads and also to be able to dynamically adapt to resource changes. Our tests show that trying to feed an average of six user-level threads for each physical processor is enough to provide the correct amount of parallelism in most of the applications. When resources are enough, the application can spawn the finest granularity nano-threads. In this way, it keeps the ability to quickly respond to the processor reallocation mechanism of the operating system. When resources are scarce, the application can generate less nano-threads and slightly more coarse-grained in order to reduce the user-level scheduling overhead.

The third design decision is to reduce the memory management overhead to a minimum. We want to avoid separated management of thread descriptors, thread arguments and local variables. As at least one structure containing the thread stack is necessary to execute the thread, we use such structure to enclose all those data. The (relative large) size of the nano-thread structure is not a problem in our implementation, as applications can control the amount of threads they create at the same time. We are also examining memory management techniques that exploit affinity between the thread stacks and the processors [105].

The fourth design decision consists on achieving an explicit and compact representation of the precedence relations among threads that can be dynamically build and updated at run-time depending on the system conditions. Output data dependencies are represented through successor nano-threads, and HTG nodes at the same level of the hierarchy are created in reverse order to correctly setup the successors for each (predecessor) node. Several successors can be specified for each nano-thread. A per-thread counter represents the remaining unresolved input data dependencies for each nano-thread [12]. The counter is initialized at thread creation. Every time a predecessor terminates, the counter is decremented. When the counter reaches zero, the nano-thread is ready for execution. Due to several optimizations done while creating parallelism it may occur that new nano-threads can dynamically be added as predecessors of a previously created end node. In this case, the counter of unresolved dependencies is therefore incremented by the amount of new predecessor nano-threads.

The fifth design decision is to include, in the library, mechanisms to allow good load balancing and exploitation of data locality. For this reason, we include both one global and per-processor local ready queues, respectively. The global queue provides a means to generate work to be performed by any processor and, therefore, to obtain a good load balancing. The data affinity allowed by local queues is very important to achieve a good performance. We have also experimented with hierarchical ready queues and processor grouping in order to allow applications to drive processors where they could be more useful [106].

The last design decision is to allow the compiler to select between the full functionality or a simpler and more efficient form of spawning parallelism. This decision motivated the introduction of work descriptors, a data structure used to represent parallel work to be done and to supply it to the allocated processors.

4.1.2. User-level NthLib interface

Tables 2 and 3 present the nano-threads user-level interface for Fortran and C, respectively. In the following description of the interface, the Fortran primitives are used. Differences with the C interface are highlighted when necessary. Along the description, the sample HTG represented in Figure 19 is used. Left part of Figure 19 presents an HTG composed of a starting and an ending sequential parts (nodes labeled `initial` and `final`) and six parallel sections of code. The sections are not completely parallel, but instead they have several dependences among them, indicated by the arrows between the nodes. The right part of Figure 19 represents the HTG of each one of the parallel sections, consisting of a parallel loop. Thus, two levels of parallelism can be exploited in this HTG.

NthLib (FORTRAN) interface	Functionality

void nthf_package_init_ (
struct nth_package_args * nthargs,
void (* nthf_main) (...),
int * narg, ...);
 	
struct nth_args {
int max_processors;
int requested_processors;
int initial_stack_size;
int stack_size;
};
	Package initialization

struct nth_desc * nthf_create_ (void (* nth_func) (...), int * npred, int * vp_id, int * nsucc, nth_argdesc * argdesc, int * narg, ... /* list of successor nano-threads */ ... /* list of arguments */); struct nth_desc * nthf_create_1s_ (void (* nth_func) (...), int * npred, int * vp_id, struct nth_desc * succ, nth_argdesc * argdesc, int * narg, ... /* list of arguments */); typedef unsigned long nth_argdesc;	Nano-thread creation nthf_create is the general primitive nthf_create_1s is a simplified primitive, accepting a single successor nano-thread only
struct nth_desc * nthf_burst_create_ (int * npred);	Nano-thread burst starting point
void nthf_dispatcher_create_ (struct nth_desc ** nth);	Application-level scheduler
struct nth_desc * nthf_self_ (void); int nthf_cpuv_ (struct nth_desc * nth);	Nano-thread self identification Virtual processor where nano-thread is executing
int nthf_depsatisfy_ (struct nth_desc ** nth);	Dependence resolution
void nthf_depadd_ (struct nth_desc ** nth, int * npred);	Dependence addition
void nthf_to_rq_ (struct nth_desc ** nth); void nthf_to_rq_end_ (struct nth_desc ** nth);	Global ready queue management
void nthf_to_lrq_ (int * which, struct nth_desc ** nth); void nthf_to_lrq_end_ (int * which, struct nth_desc ** nth);	Local ready queue management
int nthf_block_ (void);	Nano-thread blocking

Table 2: User-level NthLib interface for Fortran

NthLib (FORTRAN) interface	Functionality
void nthf_burst_wait_ (struct nth_desc ** nth_burst);	Nano-thread burst termination control
void nthf_wdcreate_ (struct work_desc * wd, void (* func) (...), struct nth_desc ** succ, int * narg, ...);	Work descriptor initialization
void nthf_gwdsupply_ (struct work_desc * wd, struct nth_desc * succ); void nthf_wdsupply_ (int * vp_id, struct work_desc * wd);	Global/local work descriptor supply
void nthf_endsupply_ (struct nth_desc * succ);	Nano-thread blocking, waiting for work descriptor termination

Table 2: User-level NthLib interface for Fortran

NthLib (C) interface	Functionality
void nth_package_init_ (struct nth_args * nth_args, void (* nth_main) (...), int narg, ...);	Package initialization
struct nth_desc * nth_create (void (* nth_func) (...), int npred, int vp_id, int nsucc, int narg, ... /* list of successor nano-threads */ ... /* list of arguments */); struct nth_desc * nth_create_1s (void (* nth_func) (...), int npred, int vp_id, struct nth_desc * succ, int narg, ... /* list of arguments */);	Nano-thread creation
struct nth_desc * nth_burst_create (int npred);	Nano-thread burst starting point
void nth_dispatcher_create (struct nth_desc * nth);	Application-level scheduler
struct nth_desc * nth_self (void); int nthf_cpuv (struct nth_desc * nth);	Nano-thread self identification Virtual processor where nano-thread is executing
int nth_depsatisfy (struct nth_desc * nth);	Dependence resolution
void nth_depadd (struct nth_desc * nth, int npred);	Dependence addition
void nth_to_rq (struct nth_desc * nth); void nth_to_rq_end (struct nth_desc * nth);	Global ready queue management
void nth_to_lrq (int which, struct nth_desc * nth); void nth_to_lrq_end (int which, struct nth_desc * nth);	Local ready queue management

Table 3: User-level NthLib interface for C

NthLib (C) interface	Functionality
int nth_block (void);	Nano-thread blocking
void nth_burst_wait (struct nth_desc * nth_burst);	Nano-thread burst termination control
void nth_wdcreate (struct work_desc * wd, void (* nth_func) (...), struct nth_desc * succ, int narg, ... /* arguments */);	Work descriptor initialization
void nth_gwdsupply (struct work_desc * wd struct nth_desc * succ); void nth_wdsupply (int vp_id, struct work_desc * wd);	Global/local work descriptor supply
void nth_endsupply (struct nth_desc * succ);	Nano-thread blocking, waiting for work descriptor termination

Table 3: User-level NthLib interface for C

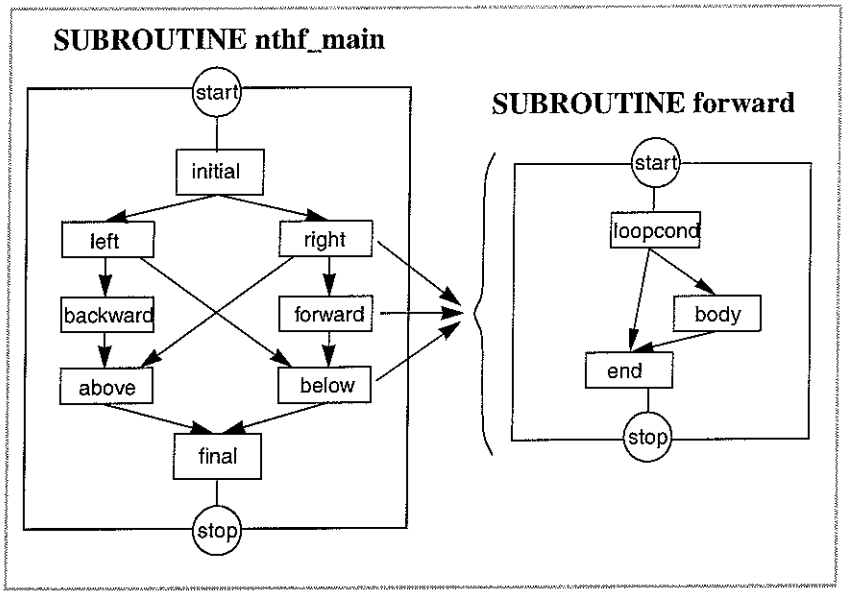

Figure 19: Example HTG used along the description of the NthLib primitives

4.1.2.1. NthLib package initialization

The NthLib package is initialized using the `nthf_package_init` procedure (see Example 4.1). The primitive receives as arguments, a reference to a structure containing several run-time variables (see the `nth_args` structure in Table 2), the function which will be started by the first nano-thread (`nthf_main`), the number of arguments and the arguments which is receiving the `nthf_main` function.

The structure `nth_args` includes the arguments used by NthLib to setup the execution environment. The field `max_processors` indicates the maximum number of processors the application will be able to use. The field `requested_processors` contains the number of processors requested automatically by NthLib before calling `nthf_main`. Later on, the application can dynamically request more or less processors, with a maximum of `max_processors`. The field `initial_stack_size` should be setup with the size (expressed in

virtual memory pages) of the stack that should be allocated by the very first nano-thread. The field `stack_size` should contain the stack size (also in pages) allocated for the regular nano-threads. Usually, the stack size of the first nano-thread should be greater than the regular nano-threads stack size. This is because usually the first nano-thread stack contains a large amount of variables, belonging to the Fortran main program, while the regular nano-threads do not need the same amount of space.

Example 4.1. Using `nthf_package_init`

```
PROGRAM nanos
      INTEGER N
      INTEGER nthargs (4)
...
      nthargs(1) = 16   ! max_processors
      nthargs(2) = 1    ! requested
      nthargs(3) = 1024 ! first_stack_size
      nthargs(4) = 4    ! stack_size
      CALL nthf_package_init(nthargs, 1, N)
END
```

```
SUBROUTINE nthf_main (N)
   INTEGER N
   DOUBLE PRECISION A(N,N)
   ...
   ...! Start parallelism here
   ...
END
```

4.1.2.2. Nano-thread creation

Nano-threads are created using the `nthf_create` call (see Example 4.2). Creating a nano-thread means, first, to instantiate a task of the HTG, setting the relationship of the task with its predecessors and successors; Second, it means to establish a new user-level context to execute the task; and third, it means to setup a hint for the processor where the task should be executed.

Example 4.2. Using `nthf_create`

```
SUBROUTINE nthf_main (N)
      INTEGER N
      DOUBLE PRECISION A(N,N)
      EXTERNAL final
      INTEGER*8 mask, nth
...
      nth_mask = 0   ! Arguments by reference
      nth = nthf_create (final,     ! Function to call
                         2,         ! Number of precedences
                         0,         ! Processor for enqueuing
                         0,         ! Number of successors
                         nth_mask   ! Argument description
                         2, A, N)   ! Number of arguments and arguments
...
END
```

```
SUBROUTINE final (A, N)
   INTEGER N
   DOUBLE PRECISION A(N,N)
...
END
```

During nano-thread creation, information is provided to set the relationship of the task instantiated with its predecessors and successors, establishing a run-time representation of the HTG structure of the application. This is done through the `npred` and the varying size `list of successors` (see Table 2). The `npred` argument tells NthLib the number of predecessors which are given this nano-thread as a successor. Only when the number of predecessors becomes zero, meaning that all the precedences have been satisfied, the nanothread can be executed on a processor. After the nano-thread execution, NthLib is in charge of decrementing the counter of precedences of its successors (given in the `list of successors`), and set them ready for execution in case the number of precedences reach zero.

For the newly created nano-thread, the new context consists of a pointer to a function to be executed (`nth_func`) and a varying number of arguments supplied to that function (`list of arguments`). A variable number of arguments is necessary because, depending on the

amount of variables used by the nano-thread, a different number of arguments have to be supplied. The number of arguments supplied to the nano-thread is indicated by `narg`. Each argument is described through the `argdesc` bitmap, indicating whether the argument is 64- or 32-bit size and whether it should be privatized for the new nano-thread or passing a reference to the original value is enough. Two bits in `argdesc` describe each argument. In the `nth_create` C interface, the mask is not necessary because the arguments can be passed by value, if necessary.

During nano-thread creation, a hint can be provided about the processor on which to execute it. This is done through the `vp_id` argument. In case the `vp_id` indicates a valid processor and `npred` is zero, the nano-thread is enqueued immediately for execution. In case the `npred` argument is greater than zero, the nano-thread is simply marked to be queued in the given processor. In case the given `vp_id` is -1, the application is in charge of supplying it to a processor.

The `nthf_create` function returns a reference to a nano-thread descriptor, in case the application wants to be aware of its enqueuing to the ready queue.

The primitives `nthf_burst_create` and `nthf_dispatcher_create` are forms of nano-thread creation, designed specifically for burst management. They are described in Chapter 8.

4.1.2.3. Nano-thread basic management

Nano-thread self identification. An executing nano-thread can obtain a reference to its own nano-thread descriptor using the `nthf_self` function. This is useful to perform different operations on the currently executing nano-thread.

Precedence management. The initial number of precedences set to a nano-thread can be dynamically modified both to add new precedences in case new inner parallelism is spawned or to satisfy dependences. Dependence resolution is used when the application can determine that a nano-thread can be activated earlier than expected. This usually happens due to the evaluation of conditional sentences. Adding dependences is performed through the `nthf_depadd` interface and dependence removal through `nthf_depsatisfy`.

It is legal that the currently executing nano-thread add a number of dependences to itself before spawning parallelism and wait for its termination. In this case, one extra precedence, representing the thread itself, must be added to the total amount of parallelism spawned.

Nano-thread blocking. Although nano-threads are run-to-completion threads, usually it is useful to join the nano-threads instantiating the start and stop nodes of a compound node to save the context between the start and stop nodes. The local variables declared at the scope of the start node are maintained till the execution of the stop node and can be accessed by the parallelism spawned inside. Nano-threads are blocked using the `nthf_block` primitive.

Example 4.3 shows how the HTG represented in the left half of Figure 19 is translated to executable code. The code generated to spawn parallelism starts adding a number of precedences to the current nano-thread (line 2). In the example, the current nano-thread adds three precedences to itself because it is going to wait for two of the parallel sections (named `above` and `below`) and itself. The nano-threads are then spawned from the bottom to the top of the HTG (line 4 to line 14). The order is imposed by the precedences expressed by the HTG structure. One or two successors are supplied to each nano-thread to represent the successor

relations also present in the HTG structure. The precedence counter of each nano-thread is setup indicating the amount of predecessors it has. For instance, the nano-thread instantiating the `right` task is created with a precedence counter of 0 because it is going to start execution immediately and two successors (`above` and `forward` sections, see line 13). It receives two arguments (`A` and `N`) and it is supplied to processor 0 to start execution. In line 14, the `left` task is supplied to processor 1 to start execution in parallel with the `right` task. The nano-thread spawning the parallelism blocks in line 15, waiting for all the parallelism to terminate. At this point, the processor is allowed to execute any work that it finds available.

Example 4.3. Nano-thread basic management

```
      SUBROUTINE nthf_main(N)
      INTEGER N
      DOUBLE PRECISION A(N,N)
      EXTERNAL final
      INTEGER*8 nth_mask
      INTEGER*8 self, nth_below, nth_above, nth_backward, nth_forward, nth_right, nth_left
      ...
1     self = nthf_self()                    ! Get reference to self
2     CALL nthf_depadd(self, 3)             ! Predecessors: above, below and self
3     nth_mask = 0                          ! Arguments by reference
4     nth_below = nthf_create(n_below1, 2,  ! Two precedences
5                             -1,           ! No hint for processor
6                             1, nth_mask,  ! Number of successors and argument mask
7                             2,            ! Number of arguments
8                             self,         ! Successor
9                             A, N)         ! Arguments
10    nth_above =    nthf_create(n_above1,2,-1,1,nth_mask,2,self,A,N)
11    nth_forward =  nthf_create(n_forward1,1,-1,1,nth_mask,2,nth_below,A,N)
12    nth_backward = nthf_create(n_backward1,1,-1,1,nth_mask,2,self,A,N)
13    nth_right =    nthf_create(n_right1,0,0,2,nth_mask,2,nth_above,nth_forward,A,N)
14    nth_left =     nthf_create(n_left1,0,1,2,nth_mask,2,nth_below,nth_backward,A,N)
15    CALL nthf_block()
      ...
      END
```

4.1.2.4. Ready queue management

Nano-threads can be enqueued in different ready queues. The application is in charge of it, in case it does not supply a valid `vp_id` number at nano-thread creation. Nano-thread dequeue from ready queues for execution is usually done inside NthLib only.

The primitives `nthf_to_rq` and `nthf_to_rq_end` allow to enqueue nano-threads for execution in the global ready queue (at the head or at the end, respectively). The primitives `nthf_to_lrq` and `nthf_to_lrq_end` allow to enqueue nano-threads at the head or at the end of the specified local ready queue.

Example 4.4 shows how `nth_right` is enqueued in the global ready queue (line 4) and `nth_left` is enqueued in the local queue of virtual processor number 1 (line 10).

Example 4.4. Ready queue management

```
      SUBROUTINE nthf_main
      ...
1     nth_right = nthf_create(n_right1,0,-1, ! No precedences, and no hint for enqueuing
2                   2,nth_mask,2,nth_above,
3                   nth_forward,A,N)
4     nthf_to_rq_end(nth_right)              ! Enqueuing at the global ready queue
5     nth_left = nthf_create(n_left1,0,-1,
6                   2,nth_mask,2,nth_below,
7                   nth_forward,A,N)
8     ...                                    ! Perform other tasks before enqueuing and
9                                            ! Enqueue at local ready queue of
10    nthf_to_lrq_end(1,nth_left)            !     processor number 1
      ...
      END
```

Example 4.5. Static application level scheduling

```
1     SUBROUTINE forward(a,n)
2     INTEGER *8 nth_mask_01
3     INTEGER nth_rest_01, nth_p_01, nth_chunk_01, nth_down_01, nth_up_01
4     INTEGER *8 self, nth_01
5     INTEGER nth_bottom_01, nth_top_01, nth_nprocs_01
6     EXTERNAL forward_loop_01
7     INTEGER n
8     DOUBLE PRECISION a(n,n)
9     INTEGER i
10    self = nthf_self()
11    nth_nprocs_01 = nthf_cpus_current()
12    nth_bottom_01 = 1
13    nth_top_01 = n
14    nth_chunk_01 = (nth_top_01 - nth_bottom_01 + 1) / nth_nprocs_01
15    nth_rest_01 = abs(mod(nth_top_01 - nth_bottom_01 + 1,nth_nprocs_01))
16    IF (nth_chunk_01 .EQ. 0) nth_nprocs_01 = nth_rest_01
17    CALL nthf_depadd(self,nth_nprocs_01 + 1)
18    nth_mask_01 = 011
19    nth_down_01 = nth_bottom_01
20    DO nth_p_01 = 0,nth_rest_01 - 1
21       nth_up_01 = nth_chunk_01 + nth_down_01
22       nth_01 = nthf_create(forward_loop_01,0,nth_p_01,1,nth_mask_01,04,
23                            self,a,n,nth_down_01,nth_up_01)
24       nth_down_01 = nth_down_01 + nth_chunk_01 + 1
25    END DO
26    DO nth_p_01 = nth_rest_01,nth_nprocs_01 - 1
27       nth_up_01 = nth_down_01 + nth_chunk_01 - 1
28       nth_01 = nthf_create(forward_loop_01,0,nth_p_01,1,nth_mask_01,04,
29                            self,a,n,nth_down_01,nth_up_01)
30       nth_down_01 = nth_chunk_01 + nth_down_01
31    END DO
32    CALL nthf_block()
33    END
34
35    SUBROUTINE forward_loop_01(a,n,nth_min,nth_max)
36    INTEGER i
37    DOUBLE PRECISION a(n,n)
38    INTEGER n, nth_min, nth_max
39
40    DO i = nth_min,nth_max
41       ! loop body(a,n)
42    END DO
43    END
```

Original OpenMP code.

```
      SUBROUTINE forward(a,n)
      ...
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO i = 1, N
         !loop body(a,n)
      ENDDO
C$OMP END PARALLEL DO
      ...
      END
```

4.1.2.5. Application level scheduling and nano-thread bursts

Code generation for different application level scheduling is achieved by means of different code structures. Example 4.5 shows the code generated for the parallel loop shown in the right portion of Figure 19. It shows the code generated for the STATIC loop scheduling expressed through the directives also shown in the example (at the bottom right corner of the figure). See Section 4.2, for a description of the OpenMP directives and extensions used in this work.

In the code of the Example 4.5, line 10 obtains a reference to the nano-thread currently executing for later use. Line 11 calls the operating system interface (defined in this work, see Chapter 5) to obtain the number of processors allocated to the application. After knowing how many processors are available, lines 12 to 16 are used to compute whether the total amount of iterations is divisible between the number of processors. If so, all processors receive the same number of iterations; otherwise, `nth_rest` processors are receiving one iteration more than the others, thus distributing the iterations as evenly as possible. The following DO loops distribute the work between the processors, creating one nano-thread for each processor. Nano-threads are created and enqueued in the ready queue of the participating processors (ranging from `0` to `nth_nprocs_01-1`, see lines 22, 23, 28 and 29). The number of precedences supplied to each nano-thread is zero because they can start execution immediately. After generating the work, the nano-thread blocks itself (line 32). Observe that the same mechanism used in the Example 4.3 is used to ensure a correct blocking; in line 17, `nthf_depadd` is used to add a number of precedences to the current nano-thread, indicating the amount of parallelism spawned plus itself.

When the amount of work to do inside a parallel loop is large enough, the loop can be further partitioned in small chunks. This is also going to favor that the application can be able to control the generation of parallelism inside the loop taking into account the number of processors assigned to it along the execution of the loop. Nano-threads are then generated in a *burst* mode, similar to the factoring technique [83]. In each burst, a number of regular nano-threads matching the number of processors plus a dispatcher nano-thread are created. The regular nano-threads execute a portion of the total iteration space. The dispatcher nano-thread is in charge of controlling the end of the current burst and starting the next. When the dispatcher is executed, it decides to schedule again a number of iterations on the available processors. The process is repeated till all the loop iterations are exhausted. Different scheduling algorithms (from [83]) have been mapped to the burst generation style.

Example 4.6 shows the `forward` loop implemented with the burst scheme applied to an interleave scheduling policy. The OpenMP directives used are shown at right, in the figure.

Lines 11 to 15 setup the bottom and top limits of the parallel loop along with the chunk size, the description of the arguments to be supplied to the nano-threads and get a reference to the current nano-thread. From this point, the current nano-thread becomes the scheduling nano-thread for this loop. In line 16, it creates a special nano-thread (the barrier nano-thread) whose purpose is to implement the barrier synchronization at the end of the parallel loop. Line 17 starts the loop which controls the creation of the nano-thread bursts. Inside, the number of available processors is read (line 18) and a first burst is started (lines 19 to 25). Observe that the successor of all nano-threads created here is set to the nano-thread barrier. This means that the scheduling nano-thread becomes free to continue executing. When the first burst is already created, the scheduling nano-thread creates the dispatcher nano-thread (line 26). This sets the dispatcher as a precedence of the scheduling nano-thread, in such a way that the scheduling nano-thread can block, waiting for the execution of the dispatcher (line 27).

When the dispatcher is executed it simply wakes up the scheduling nano-thread to spawn the next burst. This process is continued till all the loop iterations have been scheduled. Observe that each time a new burst is created, the number of allocated processors is taken into account to schedule the correct amount of iterations.

Example 4.6. Nano-thread bursts

```
1    SUBROUTINE forward(a,n)
2    INTEGER *8 nth_mask_01
3    INTEGER nth_rest_01, nth_p_01, nth_chunk_01, nth_down_01, nth_up_01
4    INTEGER nth_bottom_01, nth_top_01, nth_nprocs_01
5    INTEGER *8 self, nth_01
6    EXTERNAL forward_loop_01
7    INTEGER n
8    DOUBLE PRECISION a(n,n)
9    INTEGER i
10   ...
11   nth_chunk_01 = 16
12   nth_bottom_01 = 1
13   nth_top_01 = N
14   nth_mask_01 = 011
15   self = nthf_self()
16   nth_end_chunk_01 = nthf_burst_create(1)
17   WHILE (nth_bottom_01 .LT. nth_top_01)
18      nth_nprocs_01 = nthf_cpus_current()
19      CALL nthf_depadd(nth_end_chunk, nth_nprocs_01)
20      DO nth_p_01 = 0, nth_nprocs_01 - 1
21         nth_down_01 = nth_bottom_01
22         nth_up_01 = MIN(nth_down_01 + nth_chunk_01 - 1,top_01)
23         nth = nthf_create(forward_loop_01,0,nth_p_01,1,nth_mask_01,04,
24                           nth_end_chunk_01,a,n,nth_down_01,nth_up_01)
25      ENDDO
26      CALL nthf_dispatcher_create()
27      CALL nthf_block()
28   END WHILE
29   CALL nthf_burst_wait(nth_end_chunk_01)
30   ...
31   END
```

Original OpenMP code.

```
     SUBROUTINE forward(a,n)
     ...
C$OMP PARALLEL DO SCHEDULE(INTERLEAVE,16)
     DO i = 1, N
        !loop body(a,n)
     ENDDO
C$OMP END PARALLEL DO
     ...
     END
```

4.1.2.6. Loop scheduling using work descriptors

When the compiler determines that the amount of work in a parallel loop is small, it can decide to spawn it using work descriptors, instead of nano-threads. Work descriptors provide higher performance than nano-threads, specially when managing small loops, although the former offer limited functionality. In particular, work descriptors can only be used at the inner-most level of parallelism. Fortunately, loops at the inner-most level of parallelism are usually the ones that offer the smallest amount of work, so both facts occur at the same time.

Example 4.7 shows the code generated by the NANOS compiler when using work descriptors along with the original OpenMP code (see Section 4.2 for the explanation of these directives and extensions). The structure of the code is similar to the nano-threads version, but it has three remarkable differences: First, all work supplied to the processors is represented through a specific structure, called the *work descriptor*. Instead of providing independent arguments to each processor, as is the case when using nano-threads, the same arguments are provided using the work descriptor structure. Second, work descriptors are thought for simpler and more efficient code generation than nano-threads. Only the STATIC and INTERLEAVE scheduling algorithms are supported. It is supposed that the loop is so small that no other scheduling algorithms can provide high performance. Specifically, any DYNAMIC algorithm,

in which the processors should compete to get the iterations to perform, is going to pay a too high synchronization penalty to offer high performance. Third, the iteration space distributed to each processor is computed by the processor itself, instead of being computed by the scheduling code. This helps in the performance improvement because it reduces the time to schedule the loop iterations.

Example 4.7. Parallel loop using work descriptors

Original OpenMP code.

```
      SUBROUTINE forward(a,n)
      ...
C$OMP PARALLEL DO SCHEDULE(WDSTATIC)
      DO i = 1, N
         !loop body(a,n)
      ENDDO
C$OMP END PARALLEL DO
      ...
      END
```

```
1    SUBROUTINE forward(a,n)
2    INTEGER nth_cpuv_01
3    INTEGER *8 nth_wdesc_01(0:9)
4    INTEGER *8 nth_selfv_01
5    INTEGER nth_bottom_01, nth_top_01
6    INTEGER nth_nprocs_01
7    EXTERNAL forward_loop_01
8    INTEGER n
9    DOUBLE PRECISION a(n,n)
10   nth_selfv_01 = nthf_self()
11   nth_cpuv_01 = nthf_cpu(nth_selfv_01)
12   nth_nprocs_01 = nthf_cpus_current()
13   CALL nthf_wdcreate(nth_wdesc_01,forward_loop_01,nth_selfv_01,04,
14                      nth_nprocs_01,nth_cpuv_01,a,n)
15   CALL nthf_depadd(nth_selfv_01,nth_nprocs_01 + 1)
16   DO nth_p_01 = nth_cpuv_01,nth_nprocs_01 - 1 + nth_cpuv_01
17      CALL nthf_wdsupply(nth_p_01,nth_wdesc_01)
18   END DO
19   CALL nthf_endsupply(nth_selfv_01)
20   END
21
22   SUBROUTINE forward_loop_01(nth_me_01,nth_nprocs_01,nth_firstcpu_01,a,n)
23   INTEGER nth_me_01, nth_nprocs_01, nth_firstcpu_01
24   DOUBLE PRECISION a(n,n)
25   INTEGER n
26   INTEGER nth_lme_01, nth_balance_01, nth_niter_01, nth_step_01, nth_rest_01
27   INTEGER nth_chunk_01, nth_down_01
28   INTEGER nth_up_01, nth_bottom_01, nth_top_01, i
29   nth_lme_01 = nth_me_01 - nth_firstcpu_01
30   nth_bottom_01 = 1
31   nth_top_01 = n
32   nth_niter_01 = nth_top_01 - nth_bottom_01 + 1
33   nth_rest_01 = mod(nth_niter_01,nth_nprocs_01)
34   nth_chunk_01 = nth_niter_01 / nth_nprocs_01
35   nth_down_01 = min(nth_lme_01,nth_rest_01)+nth_bottom_01+nth_chunk_01*nth_lme_01
36   nth_balance_01 = nth_lme_01 .LT. nth_rest_01
37   nth_up_01 = nth_down_01 + nth_chunk_01 + nth_balance_01 - 1
38   DO i = nth_down_01,nth_up_01
39      ! loop body(a,n)
40   END DO
41   END
```

Lines 10 to 19 in Example 4.7 contain the scheduling code of the `forward` parallel loop using a STATIC scheduling policy and work descriptors. After getting the processor where the current nano-thread is executing (line 11) and the number of processors available for executing the loop (line 12), the work descriptor is created in line 13. The primitive `nthf_wdcreate` receives a reference to the memory area where the work descriptor should be created (usually in the nano-thread stack), the function that should be executed, the successor nano-thread (usually itself), the number of arguments and the specific arguments supplied for the execution of this loop. After that, the work descriptor can be supplied to specific processors using the primitive `nthf_wdsupply`. In this example, the loop is executed on all available

processors (the number given by the `nthf_cpus_current` primitive). Code generation for processor grouping is similar and it is commented extensively in Chapter 8. The scheduling code terminates calling the `nthf_endsupply` primitive (line 19), which is the equivalent of nth_block when using work descriptors. Calling this function, the master processor goes to execute its portion of the loop, if any, and waits for the parallelism to terminate.

The loop body procedure starts at line 22. The code computes which iterations has to perform the current thread (lines 29 to 37), based on the relative thread identifier (in `nth_lme_01`). The loop body follows in lines 38 to 40.

4.2. Compiler directives

The description of the OpenMP directives used in this thesis to parallelize applications is presented in this section.

4.2.1. Description of the OpenMP directives

The NANOS compiler parses the following OpenMP standard directives and extensions to build / refine the HTG representing the application.

4.2.1.1. Standard OpenMP directives

The subset of the OpenMP directives and clauses that have been used is summarized in Tables 4 and 5. A parallel loop in OpenMP is defined by the directive pair PARALLEL DO / END PARALLEL DO, containing the parallel loop. This directive specifies that each working processor can execute separate iterations of the enclosed loop. The SCHEDULE (mode, chunk) clause (see Table 5) is used to specify the way the compiler schedules iterations from the loop among the participating processors (mode=STATIC[,CHUNK] | DYNAMIC | GUIDED). Directive END PARALLEL DO is optional.

The PARALLEL SECTIONS directive allows the user to parcel out code into a set of independent sections. Directive SECTION is used to identify the beginning of each section. All sections can run completely in parallel.

A more general parallel region in OpenMP is defined by the directive pair PARALLEL / END PARALLEL. It establishes the boundary where new parallelism is created. All code between PARALLEL and END PARALLEL directives is executed by all the processors allocated to the application. Inside, the specific tasks that should be executed by only one processor are expressed by means of a set of work-sharing constructs. There are three different types of work-sharing constructs: the first one is oriented to the parallel execution of loops. The second one expresses parallelism among several independent sections of code. The third one is related to restrict the execution to only one processor. The result is as several parallel loops/sections had been encapsulated in one PARALLEL / END PARALLEL construct, possibly including some sequential parts. This saves the creation of parallelism at each work-sharing construct, thus reducing the overhead.

The first work-sharing construct is started through a DO directive, which specifies that each processor can execute separate iterations of the enclosed loop. A NOWAIT clause at the END DO directive means that the processors reaching the end of the loop can continue execution without waiting for the other processors to terminate the loop execution. If nothing is specified at the end directive, a barrier synchronization is assumed.

The second work-sharing construct is started through a SECTIONS directive, which specifies that each one of the enclosed SECTION can be executed in parallel by a different processor. A NOWAIT clause can be used to omit the implicit barrier at the synchronization at the END SECTIONS directive.

The third work-sharing construct limits the execution to only one processor. Other processors are going to wait for it to terminate. OpenMP offers this possibility of expressing sequential execution through the SINGLE and MASTER directives. SINGLE means that the following code must be executed by any (and just one) of the participating processors. MASTER specifies that the working processor entering the sequential section must be the master.

Directive	Description
C$OMP [END] PARALLEL DO [clauses]	Start [end] of a parallel DO loop
C$OMP [END] PARALLEL SECTIONS [clauses]	Start [end] of a set of parallel sections
C$OMP SECTION	Start of a new section inside a SECTIONS
C$OMP [END] PARALLEL [clauses]	Start [end] of a set of work-sharing constructs
C$OMP [END] DO [clauses]	Start [end] of an iterative work-sharing construct
C$OMP [END] SECTIONS [clauses]	Start [end] of a non-iterative work-sharing construct
C$OMP [END] SINGLE	Start [end] of a code section executed only by one of the participating processors
C$OMP [END] MASTER	Start [end] of a code section executed only by the master processor
C$OMP BARRIER	Force a barrier synchronization including all processors

Table 4: Standard OpenMP directives

A set of clauses defined by OpenMP can be supplied to the previous directives and work sharing constructs to specify SHARED and PRIVATE variables (see Table 5). SHARED variables are accessed by all the processors. PRIVATE variables are defined local to each processor. Either preserving or not the value that they have before entering - FIRSTPRIVATE - and/or after leaving - LASTPRIVATE - a parallel region.

The REDUCTION clause can be used to specify the variables that are involved in a reduction operation.

Clause	Description
SCHEDULE(STATIC[,CHUNK])	Static/interleave application-level scheduling in a parallel loop
SCHEDULE(DYNAMIC)	Dynamic application-level scheduling in a parallel loop
SCHEDULE(GUIDED)	Guided application-level scheduling in a parallel loop
PRIVATE (var1 [,var2...])	Specified variables are local to the work-sharing construct, they are not initialized by default
FIRSTPRIVATE (var1 [,var2...])	Specified variables are local to the work-sharing construct, each copy is initialized with the current value
LASTPRIVATE (var1 [,var2...])	Specified variables are local to the work-sharing construct, the last value is returned to the original variable after closing the parallelism
SHARED (var1 [,var2...])	Variables are shared among the processors working in parallel
REDUCTION (op: var1[,var2...])	Variables are involved in a reduction operation. The specific operation is indicated by "op"

Table 5: Standard OpenMP directive clauses

4.2.1.2. NANOS extensions to the OpenMP directives

The purpose of the following extensions to the OpenMP directives is to provide support for unstructured parallelism, processor grouping and control the functionality desired in spawning the parallelism [8].

The NAME directive (see Table 6) has been designed to extend the parallelism supported by the PARALLEL SECTIONS directive and the SECTIONS work-sharing construct. It allows the definition of precedence relations among different sections. This directive assigns a name to a section and allows the specification of its predecessors, using the clause PRED (name1 [,name2...]), and of its successors (clause SUCC).

The ONTO (expr1 [,expr2]) clause, given to a SECTION / NAME directive, specifies the processor that should be used as the master of the parallel section and optionally the number of processors to use inside it. This number of processors will be available to spawn further parallelism. The RELATIVE / ABSOLUTE clause is used to indicate whether the virtual processor identifier obtained from *expr1* should be considered relative to the current executing processor or as an absolute virtual processor identifier. Identifier range is assumed to be from 0 to P-1, where P is the number of processors currently allocated to the application. For the PARALLEL SECTIONS / PARALLEL DO directives, the CPUS clause specifies the number of processors that the application wants to be allocated to the execution of the parallel construct.

The clauses WDSTATIC and GWDSTATIC are related to specific work generation for loops. They indicate that work descriptors, instead of nano-threads, should be used to spawn the parallelism of the following parallel loop. It is assumed that, at this point, the application is spawning the inner-most level of parallelism, usually small enough to be considered fine-grain parallelism, so the programmer wants to use the most efficient mechanism for spawning.

The WDSTATIC clause indicates that local work descriptors have to be used for spawning the loop. Using local work descriptors, the work is supplied from the master processor to the slaves in a one-to-one basis. This allows to establish processor groups and distribute different loops to different processors.

The GWDSTATIC clause indicates that global work descriptors have to be used. Using global work descriptors, the work is supplied to all processors allocated to the application. This should be used for loops in the exploitation of a single-level of parallelism.

The implementations of both local work descriptors (LWD) and global work descriptors (GWD) are explained later in Chapter 7 (Subsection 7.1.10).

Directive/Clause	Description
C$OMP NAME (name) PRED(name1[,name2...]) SUCC (name1 [,name2...])	Named section directive including clauses to indicate predecessor and successor relationships for this section
ONTO (expr1 [, expr2])	Clause indicating where to execute the work and the number of processors to be used
RELATIVE | ABSOLUTE	Clause indicating that processor numbering is relative to the master processor or is an absolute processor identifier
CPUS (expr)	Clause indicating number of processors in a work-sharing construct
SCHEDULE(WDSTATIC [, chunksize])	Optimized (inner-level) static/interleave scheduling supporting processor groups
SCHEDULE(GWDSTATIC [, chunksize])	Optimized (inner-level) static/interleave scheduling to all available processors

Table 6: Extensions to the OpenMP directives and clauses

Sample program.

```
      PROGRAM nanos
      PARAMETER (N=1024)
      DOUBLE PRECISION A(N,N)
      ! initial code (sequential portion)
C$OMP PARALLEL SECTIONS
C$OMP NAME(left) succ(backward,below)
      ! left section code
C$OMP NAME(right) succ(forward,above)
      ! right section code
C$OMP NAME(backward) succ(above)
      ! backward section code
C$OMP NAME(forward) succ(below)
      ! forward section code
C$OMP NAME(above)
      ! above section code
C$OMP NAME(below)
      ! below section code
C$OMP END PARALLEL SECTIONS
      !final code (sequential portion)
      END
```

PROGRAM nanos

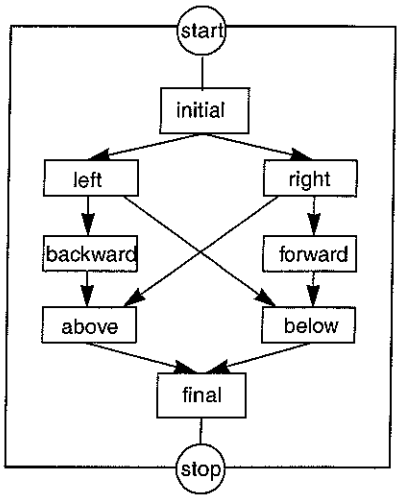

Figure 20: Sample code annotated with OpenMP directives and associated HTG

As an example of use of the NAME directive extension, Figure 20 shows a sample Fortran code annotated with OpenMP directives and the associated HTG structure obtained by the NANOS compiler from the analysis of the directives. The benefits of expressing parallelism through NAME directives instead of SECTION is that the execution is more dynamic, allowing, in some cases, to hide the unbalance among the execution of several parallel sections.

4.2.2. Code generation from the OpenMP directives

From the directives described in the previous subsection, the compiler builds / refines the HTG representing the application being compiled and a parallel program is obtained from the HTG representation of the parallel application. In our environment, the parallel code generated for the application has the following characteristics:

- A Fortran function is generated for each node in the HTG.
- Input and output control dependences are embedded into the code as calls to the run-time library.
- Output data dependences are represented through successor relations.
- Input data dependences are represented through by a per-node counter [13][12].

A Fortran function is generated both for simple and compound nodes. Code generated for simple nodes performs the set of operations contained in the node. Code generated for a compound node consists of a control function (associated to the start node) that sequences the execution of the internal nodes. To do this, the control function evaluates if it is useful to execute in parallel the functions associated with the enclosed nodes based on the available system resources. In order to improve code generation, it is possible to join two or more functions in one. In particular, functions associated to the start and stop nodes in a compound node are usually joined.

All services needed by the generated code such as creation of parallelism, control of dependencies, thread management, etc. are currently provided by a run-time user-level threads package (nthLib), although this also could be done through direct code injected by the compiler.

Following the previous scheme, the emitted code is an executable representation of the HTG.

Chapter 5. Kernel-level Interface & Functionality

Abstract

In this chapter we present the extensions we propose for the kernel-level interface and functionality designed in this thesis, for supporting the Nano-Threads Programming Model.

The kernel scheduling framework includes new data structures inside the kernel and shared with the user level to allow the implementation of kernel-level scheduling policies which take the application as the scheduling target.

The kernel interface promotes the cooperation between the user and kernel levels and enforces efficiency by selecting the most efficient way of communicating kernel level events to the application.

" ... hay tantos nanosegundos en un segundo como segundos en treinta y dos años."

"Interacciones", Sheldon L. Glashow, Tusquets Editores, S.A., mayo 1994.

5.1. Kernel scheduling framework

In this section, we present the design decisions taken in the design of the kernel-level scheduling framework and the resulting functionality, in detail.

5.1.1. Design decisions

During the design of the kernel-level interface, several design decisions guided this work to provide a new, highly dynamic, space/time-sharing environment using two levels of kernel scheduling. The kernel offers a set of scheduling policies and the system manager can decide which one is active, considering the workload. The active kernel-level policy implements the first scheduling level. Low-level management involving individual processors becomes the second level. The motivation for these two levels of scheduling is that we consider that the application should be the target for the first level and the individual processes the target for the second level.

The first design decision consists of ensuring that the kernel-level scheduling policy is able to work with the information provided dynamically by the applications with respect to resource requests (mainly processors). Applications use the kernel interface to set the number of requested processors at any time. The scheduling policy uses this value to redistribute processors to applications.

The second design decision is to consider the application as the scheduling target. The application is seen as a whole by the kernel scheduling policy, taking into account its global needs and not the needs of every independent process/thread. Processors are assigned to applications by the kernel scheduling policy. Each processor is labeled with the application to which it is assigned. The application becomes the *preferred* application of the processor. The practical consequence of this decision is that the processor is going to execute threads in that application unless the scheduling policy decides to reallocate it to another application. This is going to enforce affinity between a set of processors and their preferred application.

The third design decision is that any change in the number of processors allocated to an application (both in the number and in their status) is automatically communicated to the user-level through the kernel interface. As a consequence, the interface should be powerful enough to provide all this information.

In traditional operating systems, processors becoming idle usually take work (processes) from the operating-system ready queue. They do not ensure that the process taken from the ready list belongs to the same application where the processor was previously running. This goes against our decision of considering the application as the scheduling target. The correct way to go to execute work in another application is to find which is the new application that should receive processors and be assigned to it. In this way, the kernel avoids to get a process ready to run which could belong to an application which has not been selected to run by the scheduling policy. This situation could lead some processors to stay idle when all the work currently distributed is consumed. In this direction, the fourth design decision is that every time the kernel scheduling policy redistributes processors, it will also prepare a list, consisting of applications that will receive processors in the future. Applications in this list will receive more processors in case the currently running applications terminate or release some processors.

The fifth design decision has to do with the events that will trigger a new redistribution of processors. We are considering to redistribute processors every time the current quantum expires. We consider a quantum to be the maximum time between two global redistributions of processors. We are not considering to redistribute processors every time an application starts or terminates (or even every time an application changes its request for processors) because we think that this would introduce a lot of movements and the application would suffer because of this. We consider that the interface for requesting and releasing processors is asynchronous with respect the scheduling policy. New applications will be inserted at the end of the list of work, just in case a processor becomes idle. These applications are going to be considered by the scheduling policy during the next global redistribution. A presumably good alternative would be to redistribute processors when the list of work is found empty. By now, we consider that the list of work is large enough for all the current quantum.

For completion, the sixth design decision is that applications not explicitly coded to cooperate with the kernel (not making use of the NANOS kernel interface) are also considered by the operating system when deciding the allocation of processors. Any useful information available, such as the number of ready processes/threads limited by the number of physical processors in the system, can be used by the kernel to estimate the requirements of these applications. Therefore, these applications do not escape from kernel scheduler control. Space-sharing is also applied to them, ensuring that their performance will be very close to the one achieved using the traditional time-sharing scheduling policies. As these applications could not control their own internal scheduling and this can lead to starvation or considerable degradation of the performance, the kernel scheduling has to take into consideration to perform a round-robin assignment of processors to their processes.

The seventh design decision is related to the behavior of the second scheduling level. A processor that should get a new process/thread to run proceeds in the following way: First, it tries to select any process with pending work in kernel mode. This favors kernel work and allows kernel tasks to execute as usual. Processor affinity may be taken into account for running such kernel tasks. Next, it tries to get work from its current preferred application. If it succeeds, it proceeds to execute to user-mode and works inside its application. This is expected to be the normal behavior, all along the current quantum. Otherwise, it extracts the next application from the list of work, and assigns itself to the application. In case the list of work is empty, the processor could perform a redistribution of processors.

The last design decision refers to the mechanisms used to inform the user-level about the kernel-level processor allocations. We use an upcall mechanism and shared memory to inform applications about kernel-level scheduling decisions. We have selected the following mechanisms for informing each event taking performance into account: Processor allocation and blocking on I/O are communicated to user-level through upcalls because there is already an available processor to do this work and the overhead is small [5]. The events for returning processors to the kernel, processor preemption and unblocking are communicated through shared memory [32][84]. In this case, the application polls the shared memory at every user-level scheduling point to see whether there is any of such events. Fine-grain parallelization helps in that such polling is made often enough to detect most of the operating-system requests [86].

5.1.2. Operating system scheduling framework

As it has been stated both in Chapter 3 and in the previous design decisions, the NANOS operating-system level scheduling is application-oriented and takes applications as the scheduling target. It searches for an environment where the execution of the applications can proceed as smooth as possible, without large or abrupt processor movements. In this subsection, we present the main ideas that are used by the operating-system scheduling mechanisms to manage and distribute physical processors among the running applications.

The first idea we take for achieving a smooth kernel-level scheduling comes from the First-Class User-Level Threads approach [84]. It consists of providing the applications with a *grace* time when any processor has to be preempted. With such a grace time, the application is given an opportunity to reach a user-level *safe* point and release voluntarily the processor. In this way, each time the operating system applies the active scheduling policy, the processors need not to be moved immediately, but, instead, they can be moved when they are released by the applications or forcefully after the grace time has expired. This means that each processor has to be informed about which application will be its destination. In some sense, it is like pre-allocating processors to the applications where they are going to execute in a small amount of time.

The second idea extends the previous one, and consists of collecting all the information about the pre-allocation of processors into a kernel structure, called the *work list*. The *work list* will contain, at any given instant, the applications that are going to receive processors in the near future. Each entry in the *work list* is a reference to an application, indicating a number of processors that will choose this application as their preferred one. In order to better balance the processor allocation among the different applications, one application can be represented by more than one entry in the *work list*, each entry possibly indicating a different number of processors. Each time a processor visits an entry in the *work list*, it decrements the counter and, from now on, it will select processes from this application to run on it. When the counter reaches zero, the entry is removed from the *work list*.

The scheduling algorithm distributes processors taking into consideration all the requests and the current allocation. This results on a set of applications that will run during the next quantum and the corresponding distribution of processors. From this set, applications that the number of processors does not change for the next quantum, are not disturbed and continue running. Applications in the set that have to release processors are signaled through shared memory. The applications that will receive more processors or will start running are put in the *work list* with the corresponding number of processors they will use in the next quantum.

The *work list* is then filled with enough more entries that will feed work to the processors that could become idle, after consuming the work prepared for the next quantum.

The *work list* decouples the work done by the kernel-level scheduling policy from the actual individual processor movements between applications. By means of the *work list*, the events of requesting more processors and starting a new application are equivalent because an starting application is exactly the same as its request for the first processor. These events add a new entry at the end of the *work list*.

Also, the events of terminating an application or releasing processors, both voluntarily and under a grace time, and processor preemptions will proceed in the same way. The free processors have to get an entry from the *work list* and assign to the application.

Taking into account locality issues, a processor can get an entry different from the first, to select an application into which it had run before.

Every time the operating system applies the current scheduling policy, it fills a new *work list*, which will be used till the end of the current quantum.

The previous approach favors malleable applications. They will be able to reconfigure their parallelism to use more or less processors. And the approach guarantees that, in case of processor preemption, the application will be accurately informed and can easily recover the preempted work. The implementation of this framework is presented in Chapter 7 (Subsection 7.2.3).

5.2. Kernel interface

In this section, we describe the kernel interface established between parallel applications and the operating system, following our design of the NPM. As explained in the previous section, this interface enables dynamic application adaptability and runtime management of parallelism within individual applications. At the same time, it assists the operating system in allocating processors to parallel applications in order to minimize application turnaround times and maximize processor utilization. Table 7 shows the proposed operating system interface to support the processor scheduling. It is described in the following subsections and several examples are presented.

Operating system interface	Description
int cpus_request (int ncpus);	Processor request/release
int cpus_requested (void); int cpus_current (void); int cpus_askedfor (void); int cpus_preempted_work (void);	Informational services
work_t cpus_get_preempted_work (void);	Returns a descriptor for a preempted thread
int cpus_processor_handoff (work_t work);	Transfers the physical processor to the supplied thread
int cpus_release_self (void);	Voluntarily releases the current processor

Table 7: Operating system interface

5.2.1. Processors request and supply of virtual processors

At any time, applications use the `cpus_request(ncpus)` primitive to set the number of requested processors that they want to run on. The operating system will try to assign as many processors as possible without exceeding `ncpus`. The primitive can be used dynamically at runtime to request more or less processors than those currently assigned. It is illegal to request a number of processors above the number of processors installed in the system. The kernel maintains two different numbers of processors: `current` and `requested`.

A request for less processors using `cpus_request()` implies that the application voluntarily wants to release some processors, possibly because the current degree of parallelism inside the application cannot exploit them. When the kernel scheduler detects this situation, it automatically steals `current` minus `requested` number of processors. The

application may also release the desired processors, instead of letting the scheduler to steal them (see the following subsection).

5.2.2. Specific processor release

The application can release a specific running physical processor, as an answer to an operating system request using the `cpus_release_self()` primitive. This primitive attempts to release the physical processor on which the calling virtual processor runs. When this primitive returns, it means that either the processor was not needed by the scheduler or that a new processor has been allocated to the application. The data shared with the operating system reflects both situations, informing about the currently assigned processors.

5.2.3. Preempted work recovery

The kernel also offers the possibility of transferring a physical processor from the current virtual processor to another virtual processor of the same application. This feature is provided to allow an application to recover the execution of a preempted virtual processor. To allow an application to recover the execution of preempted work, the kernel offers the `cpus_processor_handoff(work)` primitive [15]. The virtual processor identified by `work` receives the current physical processor. `Work` can be the identifier of a preempted virtual processor or a reference to the new context to be resumed at user-level. It is assumed that the identifiers of the preempted virtual processors could be obtained through the `cpus_get_preempted_work()` primitive, described in the following subsection.

5.2.4. Current status of processor allocation

The following primitives provide information about the current status of processor allocation for the calling application. The primitive `cpus_requested()` informs about the number of processors that the application has requested from the operating system. The `cpus_current()` primitive returns the number of physical processors actually owned by the application.

The application is also informed about the need of releasing some of the available processors, like in [84][152]. The primitive `cpus_askedfor()` returns the number of processors that the operating system is reclaiming from the application.

The `cpus_requested()`, `cpus_current()` and `cpus_askedfor()` primitives help the application to know at any time which is the status of the number of processors allocated to the application by the operating system.

At any time, the operating system can decide to reassign processors from one application to another. Applications loosing processors are informed about this fact and the contexts of the preempted works are put at their entire disposition to be resumed by one of the remaining assigned processors, if necessary. The following functions are used to obtain this information:

Preempted work recovery is supported through the following primitives: `cpus_preempted (void)`, and `work_t cpus_get_preempted_work (void)`. `Cpus_preempted` returns the number of processors which have been preempted from the application by the operating system. And `cpus_get_preempted_work()` returns a reference to a preempted context that can be resumed at user-level. The functionality of these primitives has been extended, when they are implemented through shared memory, to allow the application to select which one of the preempted virtual processors to resume.

5.2.5. Coding examples

The primitives presented in the previous subsection can be used from the user-level to request processors and obtain information about the resources allocated and their status. As an example of using the kernel interface, we present the way an application requests processors and the basic implementation of the idle function inside NthLib.

5.2.5.1. Requesting processors and spawning parallelism

Example 5.1 shows the code used for requesting and releasing processors inside the application. In this example, the kernel-level interface is called from the Fortran language. For this reason, the names of the primitives have been prefixed by the prefix `nthf_`. In line 2, a new parallel region starts and the application requests N processors, using the primitive `nthf_cpus_request()`. After that, the parallelism is spawned. We present a simple code to spawn a parallel loop (lines 3 to 9). The number of processors currently allocated is got from the operating system (line 4) using the primitive `nthf_cpus_current()`. After that, a nano-thread is created on each available processor. After the parallelism has been joined in line 9, the primitive `nthf_cpus_request()` is used again to release the processors and continue the execution with one for the next sequential portion of code.

Example 5.1. Requesting processors and spawning parallelism

```
1    ...
2    nthf_cpus_request (N)
3    self = nthf_self()
4    ncpus = nthf_cpus_current()
5    CALL nthf_depadd (self, ncpus)
6    DO nth_p = 0, ncpus-1
7       CALL nthf_create (func_loop_01, 0, nth_p, 1, nth_mask, 4, self, ...)
8    ENDDO
9    CALL nthf_block ()
10   nthf_cpus_request (1)
11   // Sequential code
12   ...
```

5.2.5.2. The virtual processors scheduling loop

Example 5.1 presents a simple idle function based on the proposed kernel-level interface. The code can be used inside the nano-threads library to force each virtual processor, after execution of an application task, to check for the operating system conditions and search for new application tasks at user-level.

The sample idle function consists of a main loop, to be executed forever, starting at line 3. The first thing to do is to check whether the operating system is reclaiming any processor to be released (line 4). This is possible, for instance, immediately after applying the kernel-level scheduling policy and having redistributed the physical processors. In case a number of processors are reclaimed, the current virtual processor stops, releasing the physical processor, using the primitive `cpus_release_self()` (line 5). Another operating system condition that should be checked is whether there are any preempted processes in the application. This is done in line 7. In case there are preempted processors, lines 8 and 9 get one of the preempted processes and transfer the current physical processor to it. The current virtual processor will also be stopped, in this case. Finally, when reaching lines 11 and 12, the virtual processor has not detected any warning condition from the operating system, gets a new nano-thread to execute and dispatches it.

In this example, we assume that when the virtual processor is stopped while executing either the primitive `cpus_release_self()` or `cpus_processor_handoff()`, its context can be later reused for another physical processor to be assigned to the application, continuing the execution from that point. Another solution would be to provide a specific entry point for new processors entering the application.

Example 5.2. Sample idle function

```
1    // Executed for each virtual processor
2
3    for (;;) {
4       if (cpus_asked_for()) {
5          cpus_release_self();
6       }
7       if (cpus_preempted_work()) {
8          work_t w = cpus_get_preempted_work();
9          cpus_processor_handoff (w);
10      }
11      nth = nth_getwork ();
12      // dispatch new nano-thread
13   }
```

5.3. Kernel-level scheduling policies

The goal of the scheduling policies supporting the NPM should be to help in achieving the main goal of this thesis, that is, to provide high global-system performance. In this work we are using the proposed environment to test several policies and compare among them and with existing operating-system scheduling policies. Several scheduling policies have been implemented at kernel level. The main purpose of this implementation is to validate the design and implementation of the kernel scheduling mechanisms.

This work does not search for the best policy, but instead tries to demonstrate that the ideas and mechanisms proposed are valid. A proposal for a specific policy to be used inside NPM is out of the scope of this thesis. The following subsections describe the behavior of the four policies during the execution of application workloads. Three of the policies presented have been obtained from the literature [95][47][89][75]: Equipartition, Batch and Round-robin. The fourth comes from the experimentation done along this work. Our experience indicates that applications usually request an even number of processors and most of times the number of processors is a multiple of 4. From this experience, the Cluster policy is provided.

The following policies are evaluated and results are shown in Chapter 9, along with some examples of the behavior of the applications running under them.

5.3.1. Equipartition (Equip)

This policy divides the number of physical processors by the number of running applications, and it assigns the resulting number of processors to each application. When an application requests less processors than the result of the division, the processors exceeding that number are not assigned to any application. In case there are more applications than processors, only the first P applications are executed, assuming there are P processors.

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

5.3.2. Batch

This policy allocates processors to applications in strict ordering of arrival. Each application receives all requested processors, till all the processors in the computer have been allocated. The last application receiving processors will receive less processors than those requested if there is not so many processors available in the machine. Also, if the requests for processors change during execution, the number of applications receiving processors may change. The *work list* is used to ensure that those applications not receiving all the requested processors will receive some of them when processors are voluntarily released by other applications.

When using the Batch policy, applications arriving first are benefitted from that and applications arriving later are delayed till the first ones terminate.

5.3.3. Round-robin (RR)

The operating system allocates processors to applications in a round robin manner. Each application receives as many processors as requested. The policy remembers the last application receiving processors at each quantum. At the next quantum, it continues allocating processors from the next application. At each reallocation, the last application receiving processors can receive less processors than those requested, if there is not so many processors available in the system. The *work list* is used to ensure that those applications not receiving all the requested processors will receive some of them when processors are voluntarily released by other applications.

The Round-robin policy will motivate a large number of context switches between the applications, due to time-sharing.

5.3.4. Processor Clustering (Cluster)

The Processor Clustering policy allocates processors in clusters of 4 processors. In a first step, this policy allocates a cluster of 4 processors to all running applications. If some applications are not receiving processors because there is a large number of applications in the system, they are added to the *work list* for the current quantum. In case a number of processors remain unallocated, this policy starts a second step, allocating again in clusters of 4. And so on, till all the processors have been allocated or all the requests have been satisfied. When less than four processors remain to be allocated, some applications can receive two or even one processor to maintain all processors working.