CHAPTER 2

General Overview of Shared-Memory Multiprocessor Systems

Abstract

The performance of a multiprocessor system is determined by all of its components: architecture, operating system, programming model, run-time system, etc. In this Chapter, we focus on the topics related to multiprocessor architectures, scheduling policies and programming models.

We analyze the different proposals in multiprocessor architectures, focusing on the CC-NUMA SGI Origin2000, the platform for our work.

We also describe and classify the scheduling policies previously proposed in the literature. Some of them will be presented in more detail since they are more related with the work done in this Thesis. Others are presented only for completion.

Finally, we analyze the standard programming models adopted by users to parallelize their applications.

2.1 Introduction

In this Chapter, we give a general overview about three of the main elements in a multiprocessor system: architecture, operating system, and programming model.

We will give an insight about some of the multiprocessor architectures, focusing on the SGI Origin 2000 [93][93], which has been the target architecture in this Thesis. This machine is a shared-memory multiprocessor architecture and it is representative of multiprocessor machines with a medium-high number of processors used in commercial environments and supercomputing centers.

The second component, and the main point of this Thesis, is the operating system. In particular, this Thesis focuses on the part of the operating system that is in charge of distributing processors among applications: the scheduler. In this Chapter, we briefly present some of the proposals made in the last years on processor scheduling. We want to give a general overview about processor scheduling. More specific proposals related with the topic of this Thesis are distributed among the different chapters.

Finally, we also describe standard programming models used to parallelize applications: MPI, OpenMP, and the hybrid model MPI+OpenMP. In this Thesis, we have adopted the OpenMP programming model.

This Chapter is organized as follows. Section 2.2 describes the SGI Origin 2000 architecture. Section 2.3 presents the related work concerning to proposals to share processors among applications. Section 2.4 presents related work to coordination of scheduling levels. Section 2.5 presents related work to job scheduling policies and Section 2.6 related work to processor scheduling policies. Section 2.7 presents the standard programming models used to parallelize applications: MPI and OpenMP. Finally, Section 2.8 presents the summary of this Chapter.

2.2 Multiprocessor architectures

In this Section, we present a brief description about some multiprocessor architectures. A multiprocessor is a system composed by more than one processor. They appeared with the goal of improving the throughput of the system, executing more applications per unit of time, and to improve the execution time of individual applications, executing them with multiple processors.

There are two main trends in multiprocessor architectures: architectures that implement a global address space, shared-memory multiprocessors, and architectures that implement several separated address spaces, distributed-memory multiprocessors.

Shared-memory multiprocessors are the direct extension from uniprocessors to multiprocessors. It has been shown a difficult task to have shared-memory multiprocessors with a high number of processors, mainly because of the cost of maintain the memory coherence. The alternative, systems with separated address spaces, scale better than shared-memory multiprocessors because they do not have to maintain the memory coherence. However, they require more effort by the programmers. In this kind of architectures, the programmer must explicitly re-design the algorithms to execute in different address spaces.

2.2.1 Shared-memory architectures

Shared-memory multiprocessor architectures are systems with the common characteristic that any processor can access any memory address. In general, advantages of these systems are that they are easy to use and to program because they are a direct evolution from uniprocessor architectures. That means that user applications and system policies can be used in these systems without modification. However, to exploit as much as possible the potential of multiprocessors, it is convenient to modify applications and policies to consider particular characteristics of each architecture.

The main problems related with shared-memory architectures are the memory bandwidth, the cache coherence mechanism, and the semantics of memory consistency. These problems imply that shared-memory multiprocessors are difficult to scale (large shared-memory multiprocessors are composed by 64, 128, 512 or 1024 processors). The memory bandwidth limits the number of simultaneous accesses to memory that applications can do.

The memory coherence generates the false sharing problem. It appears when two different data are accessed by two processors and these data are in the same cache line. In this case, cache coherence protocols generate that the cache line repeatedly travels across the multiprocessor from one processor to the other.

There are two main types of shared-memory architectures: symmetric multiprocessors and CC-NUMA architectures.

Symmetric Multiprocessors, SMP´s.

Figure 2.1 shows a typical configuration of a SMP machine. SMP´s are architectures with a limited number of processors, typically from 2 to 16 processors. Processors are usually inter-connected through a crossbar or a bus. The main characteristic of SMP´s is that the cost to access memory is the same for all the processors. This kind of architecture receives the name of CC-UMA, **C**ache-**C**oherent **U**niform **M**emory **A**ccess.

SMP´s have the advantage of their accessible cost, then their use have been extended as web servers and database servers. Their main problem is their limited scalability due to the interconnection mechanism used (between memory and processors). SMP´s that use crossbars are fast, but crossbars supporting many processors are very expensive. Buses are cheaper than crossbars, but they have a limited bandwidth and do not scale at all for more than around 12 processors.

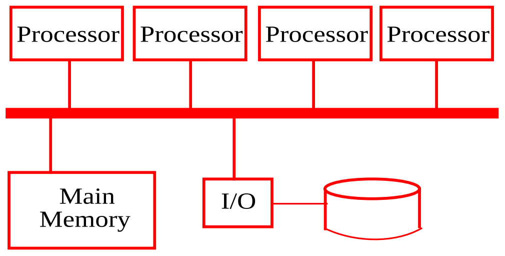


Figure 2.1: SMP architecture

Cache-Coherent Non-Uniform Memory Access, CC-NUMA.

The other group of architectures are CC-NUMA architectures. CC-NUMA machines consist of many processors, and they can scale up to 1024 processors. These machines are the biggest ones that implement shared-memory with cache coherence implemented by hardware. In these architectures, processors are usually grouped and interconnected through a more complex network. The goal of these architectures is to hide memory latency to scale. The main characteristic of CC-NUMA machines is that the cost to access memory is not always the same. It depends on which processor is doing the access and where the data is placed. In CC-NUMA machines, the memory is virtually shared but physically distributed.

The main advantage of CC-NUMA respect to SMP´s is that they scale better, and they can have many more processors. However, CC-NUMA machines have a much more sophisticated design. That implies that they are much more expensive than SMP´s, and then not so accessible. The other problem related to CC-NUMA machines is the memory access. As we have commented yet, the cost to access memory depends on the processor and the memory node. This will imply that the scheduling should be designed and implemented “with care”, taking into account this characteristic.

2.2.2 Distributed-memory architectures

Distributed-memory architectures have the common characteristic that they offer separated address spaces. Compared with shared-memory machines, distributed-memory machines have the advantage that they scale better, mainly because they do not have to maintain information related to each cache line distribution and status. For this reason, typically distributed-memory machines are systems with a high number of processors.

We can differentiate two main types of distributed-memory machines: Massively Parallel Processors and Networks of workstations.

Massively Parallel Processors systems, MPP´s.

MPP systems consist of a high number of processors connected through a fast (generally proprietary) interconnection network. Processors can explicitly send / receive data from / to remote memory nodes by special instructions or through the network API. In any case, the hardware does not provide any memory consistency mechanism. In these architectures, the cost to access to remote data also depends on the distance between processors and memory, and on the path to access it. Due to this hardware simplification, MPP´s can scale to a very high number of processors.

The main advantage of this architecture is the scalability. Nevertheless, MPP’s have architectural design problems such as the interconnection network designs or the data injection mechanism design. One of the disadvantages of these systems are that they are expensive, mainly because of the interconnection network.

In MPP´s, each processor executes its own copy of the operating system, that implies that the resource management is more complicated that in shared-memory machines.

Networks of Workstations, NOW´s.

Figure 2.2 shows a NOW. They usually consist of several uniprocessor servers connected over a network. The main difference with MPP systems is the interconnection network. In NOW’s, different nodes are usually interconnected with a commodity network and each node is a commodity processor. In fact, each workstation could be a different system with different characteristics.

These kind of systems have a clear advantage compared with MPP´s, they are more economic. In an environment where there are several users, each one with its own workstation, if we connect all of them with a general purpose network, and we install a resource manager, we achieve a NOW. These systems can grow in an easy way. Problems related with these architectures are that they do not offer good performance to individual parallel applications because the processor interconnection is not designed with this goal, and the resource management is not easy because we can have heterogeneous workstations.

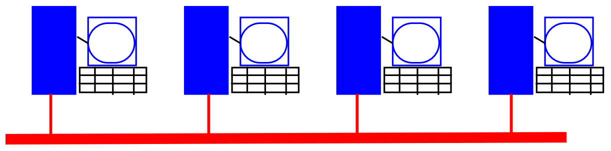


Figure 2.2: Networks of workstations

2.2.3 Mixed approaches

The two previously presented groups of architectures have been combined with the aim of including the best of each option. One of these approaches are clusters of SMP´s.

Clusters of SMP´s consist of several SMP nodes interconnected. Figure 2.3 shows the typical configuration of a cluster of SMP´s, several SMP´s machines connected through an interconnection network. Inside each SMP node we have shared-memory, and between nodes we have distributed-memory. This configuration has become very popular last years because small SMP´s have proliferated due to their prices, and connecting several of them we can achieve a system with a high number of processors. Every day is more usual that users have SMP´s at their work rather than workstations, then clusters of SMP is a natural evolution.

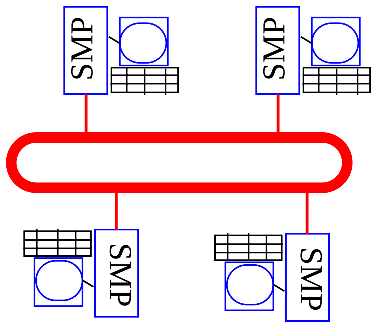


Figure 2.3: Clusters of SMP

Another different approach is the presented by some architectures such as the T3E [19]. The T3E is essentially a distributed-memory machine, however, it provides special instructions to specify shared-memory regions. These memory regions can be shared between more than one processor, but in some cases with a limited access (read only / write only). The main difference with shared-memory machines is that, in this case, the hardware does not provides any support to ensure the memory consistency.

2.2.4 CC-NUMA architecture: SGI Origin 2000

Since we have adopted a practical approach, it is very important to understand the architecture of the system. The SGI Origin2000 [93] is a shared-memory multiprocessor system, that means that it has a global shared address space. We have selected the SGI Origin 2000 by several reasons. The first one is the availability, we have received the support to evaluate our contributions in this system[1]. Moreover, we consider that the SGI Origin 2000 is a modern architecture, quite extended, and the most used shared-memory multiprocessor system in supercomputing centers. Finally, the number of processors available, 64, is quite enough to evaluate our contributions in a real multiprocessor system.

The SGI Origin2000 is a scalable architecture based on the SN0 (Scalable Node 0), which is, in his turn, composed by two superscalar processors, the MIPS R10000.

The MIPS R10000 CPU

The MIPS R10000 CPU[110][68] is the CPU used in the SN0 systems. The MIPS R10000 is a "four-way"[2] superscalar RISC processor. Superscalar means that it has enough independent, pipelined execution units that it can complete more than one instruction per clock cycle. The main features of the MIPS R10000 are a nonblocking load-store unit to manage memory accesses, two 64-bit ALU's for address computation, arithmetic, and logical operations, see Figure 2.4.

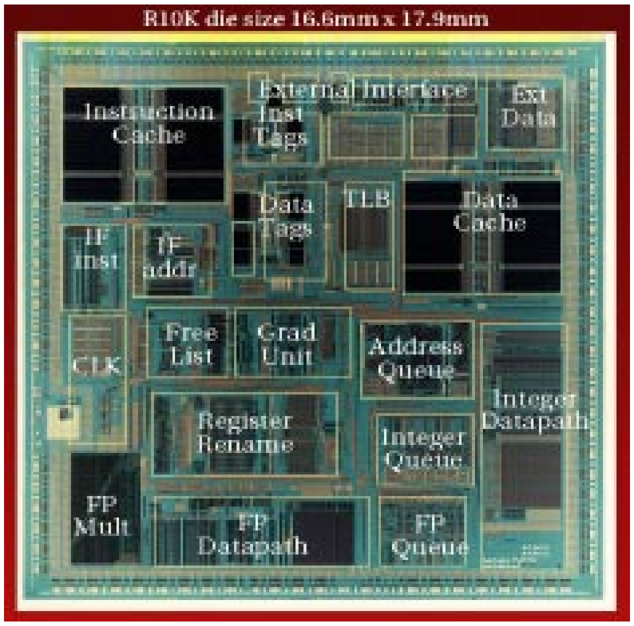


Figure 2.4: MIPS R10000 [68]

1. We have evaluated our contributions using all the processors in the system.
2. It can fetch and decode four instructions per clock cycle.

The MIPS R10000 uses a two-level cache hierarchy. A Level-1 (L1) internal to the CPU chip (separated data and instructions), and a second level cache (L2) external to it. Both of them are non-blocking caches, that means that the CPU does not stall on a cache miss. Both the L1 and L2 uses a Least Recently Used (LRU) replacement policy. The R1000 is representative of the most powerful processors available at the moment this Thesis was started.

SN0 Node Board

Each SN0 (Scalable Node 0)[93] contains two MIPS R10000 CPU's. Figure 2.5 shows a block diagram of a node board.

Each memory DIMM contains the *cache directory*. The use of the cache directory implies that the main memory DIMMs must store additional information. This additional information is proportional to the number of nodes in the system. The SN0 uses a scheme based on a *cache directory*, which permits cache coherence overhead to scale slowly.

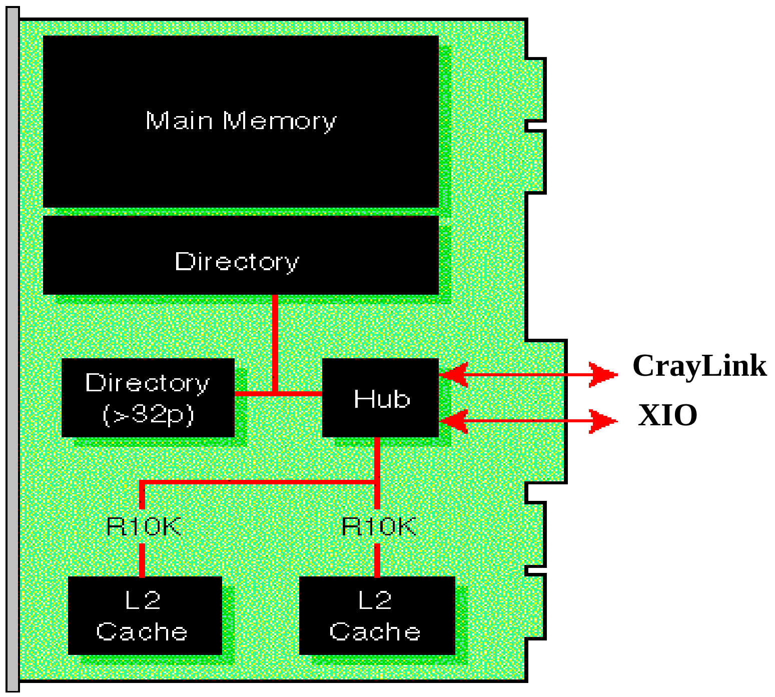


Figure 2.5: SN0 Node Board [93]

The final component is the hub. The hub controls data traffic between CPUs, memory, and I/O. The hub has a direct connection to the main memory on its node board. This connection provides a raw memory bandwidth of 780 MBps to be shared by the two CPUs on the node board. Access to memory on other nodes is through a separated connection called the CrayLink interconnect, which attaches to either a router or another hub.

When the CPU must refer data that is not present in the cache (L2 cache), there is a delay while a copy of the data is fetched from memory into the cache. In the SN0, the cache coherence is the responsibility of the hub chip. The cache coherence protocol allows the existence of multiple readers or only one writer (which will "*own*" the cache line). This protocol implies that, depending on the algorithm, multiple cache lines can travel trough the different nodes to maintain the data coherence. It is important to know the memory behavior to understand the application performance, very influenced by architectural configurations.

SN0 Organization

SN0's are interconnected to compound the complete system [93]. Figure 2.6 shows the high-level block diagrams of SN0 systems. Each "N" box represents a node board. Each "R" circle represents a router, a board that routes data between nodes. Through the hub, memory in a node can be used concurrently by one or both CPU's, by I/O attached to the node, and - via router - by the hubs in other nodes.

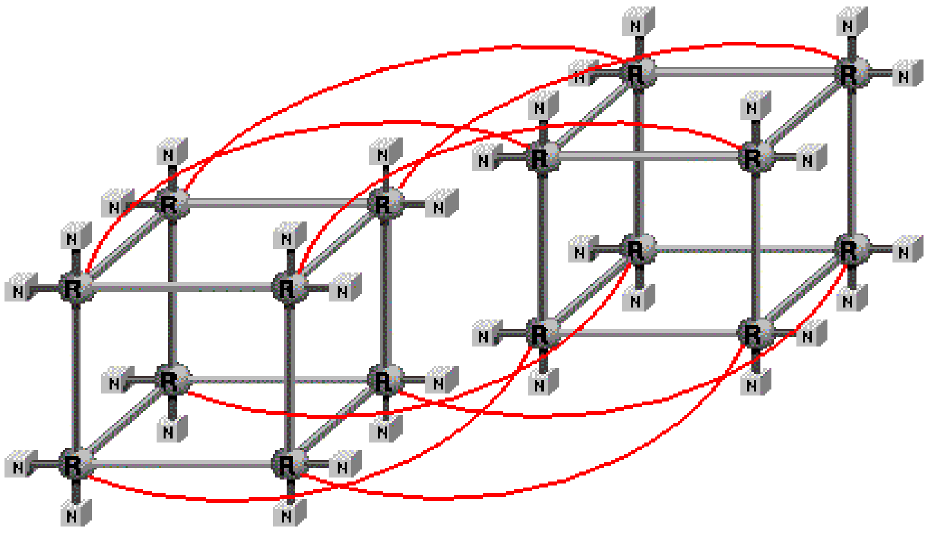


Figure 2.6: SGI Origin 2000 organization [93]

The hub determines whether a memory request is local or remote based on the physical address of the data. Access to memory on a different node takes more time than access to memory on the local node, because the request must be processed through two hubs and possibly several routers. Table 2.1 shows the average latencies to access the different memory levels [53].

Table 2.1: SGI Origin2000 average memory latencies

Memory level	Latency (ns)
L1 cache	5.1
L2 cache	56.4
Local memory	310
4 cpus remote memory	540
8 cpus remote memory	707
16 cpus remote memory	726
32 cpus remote memory	773
64 cpus remote memory	867
128 cpus remote memory	945

The hardware has been designed so that the incremental cost to remote memory is not large. The hypercube router configuration implies that the number of routers information must pass through is at most n+1, where n is the dimension of the hypercube.

This configuration, and the different cost to access memory, implies that it is important to consider the memory placement. Data are not replicated, it is in one memory cache, or memory node. If the processor, or processors, that are accessing to these data are in the same node, accesses to these data will be very fast. However, if data have to travel through the interconnection network, the cost to get the data could be three times slower.

The particular characteristics of our SGI Origin2000 are the following:

- 64 250 MHZ IP27 Processors
- CPU: MIPS R10000 Processor Chip Revision 3.4
- FPU: MIPS R10010 Floating Point Chip Revision 0.0
- Main memory size: 12.288 Mbytes
- L1: Instruction cache size: 32 Kbytes
- L1: Data cache size: 32 Kbytes
- L2: instruction / data cache size: 4 Mbytes

2.3 Scheduling policies

In this Section, we will present the related work in processor scheduling. We will follow the scheduling classification presented in the introduction of this Thesis, see Figure 2.7.

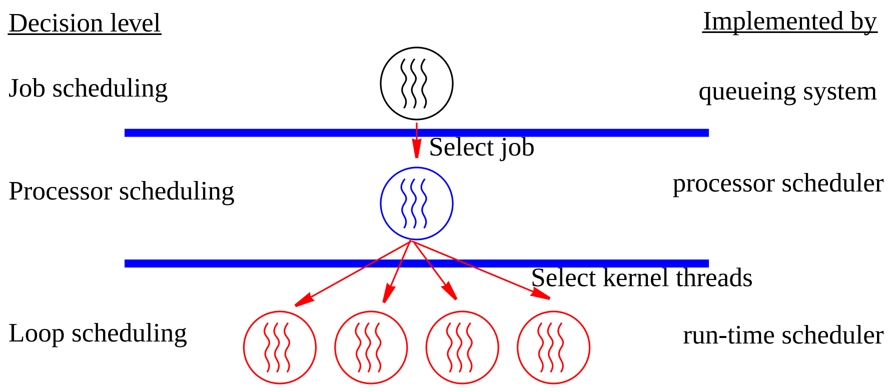


Figure 2.7: Decision levels

The first level, job scheduling, decides which job must be started at any moment. It selects one job for execution, from a list of queued jobs. In a distributed system, the job scheduling policy could include a decision related to where, in which node, to start the application. The job scheduling policy is implemented by the queueing system, and it is activated at a low frequency in the system.

The second level, processor scheduling, decides the scheduling between processors and kernel threads of running applications. The processor scheduling policy is implemented by the processor scheduler in the O.S. kernel. It is activated at a medium frequency in the system, a typical value is between 10 and 100 ms.

Finally, the loop scheduling policy decides which user-level threads to execute from each application. The loop scheduling policy is implemented by the run-time library and it is always implemented at the user-level.

In this Chapter, we focus on the processor scheduling and job scheduling level because they are implemented by the system. Loop scheduling policies are more related to the particular algorithm, the user, and the compiler, and out of the focus of this Thesis.

There exist several possibilities to execute a workload of parallel jobs in a multiprocessor system: from executing one job at any time to executing all the jobs together. Multiple combinations of different policies at each level have been proposed. Some of then, by default, eliminate some of the other decision levels. For instance, most of the job scheduling policies assume that there is not processor scheduling policy in the system (a simple dispatch).

In this Section, we present some related work with our proposal of coordinating the different scheduling levels, and related work with job scheduling and processor scheduling. Related to the scheduling policies, we will try follow a similar classification to the proposed by Feitelson in [32]. However, it does not classify scheduling policies in these three levels, for this reason we will re-organize its classification to match with our classification.

Feitelson classifies in [32] the scheduling policies as single-level and two-level policies, see Figure 2.8. In single-level scheduling, the O.S directly allocates processors to kernel threads. In two-level scheduling, the O.S separates the scheduling decision in two steps: (1) it allocates processors to jobs, and (2) inside each job several kernel threads are selected to run. Single-level policies are also called time-sharing policies, and two-level policies are also called space-sharing policies.

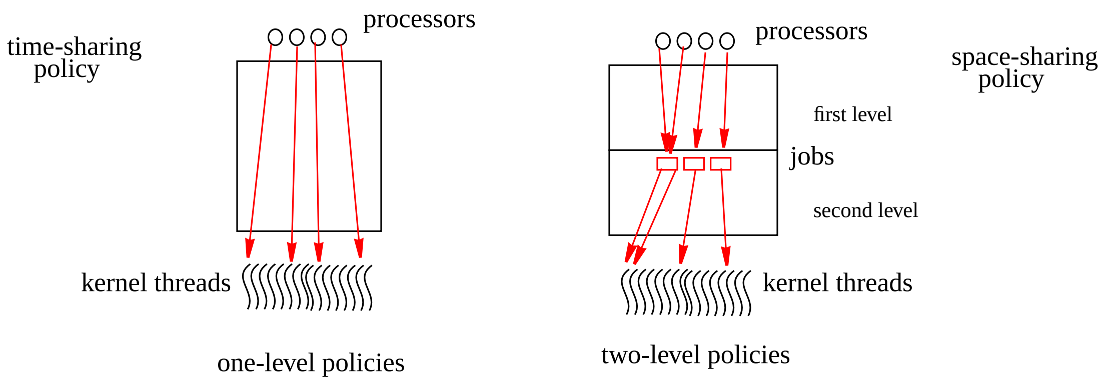


Figure 2.8: One-level policies vs. two-level policies

Feitelson classifies space-sharing policies into four classes: Fixed, Variable, Adaptive, and Dynamic, see Table 2.2. The decisions taken by Fixed and Variable policies are only related to job scheduling. On the other hand, decisions taken by Adaptive and Dynamic policies are related to processor scheduling.

Moreover, Feitelson classifies applications into three types: rigid, moldable, and malleable. Rigid are those applications that can only be executed with the number of processors requested. Moldable are those applications that can be executed with any number of processors, and different from the number of processors requested, but it must be fixed during the complete execution of the application. And malleable are those applications that can be executed with a varying number of processors.

Table 2.2: Space-sharing classification proposed by Feitelson

Type	Hardware	Decision level	Application
Fixed	Determines the partition size	job scheduling	Rigid parallelism
Variable	Does not determine the partition size	job scheduling	Rigid parallelism
Adaptive	Does not determine the partition size	processor scheduling	Moldable
Dynamic	Does not determine the partition size	processor scheduling	Malleable

We classify policies based on the capacity of decision that they have. Then, we classify them in job scheduling policies (Fixed and Variable), and processor scheduling policies (Adaptive and Dynamic). Job scheduling policies only decide which application to execute, not the number of processors to allocate to each one. Processor scheduling policies can apply a time-sharing (one-level), space-sharing (two-levels), or gang scheduling policies (time-sharing and space-sharing).

2.4 Coordinating scheduling levels

2.4.1 Coordinating the processor scheduler and the run-time

The coordination between processor scheduler and run-time has been directly or indirectly used in other proposals. In the NANOS project [63][64], Martorell et al. use an interface between the processor scheduler and the run-time. This interface includes information from the processor scheduler to the run-time such as the number of processors allocated, and information from the run-time to the scheduler such as the number of processors requested. In fact, the interface between processor scheduler and run-time used in this Thesis is an extension of the NANOS interface. In *Process Control* [105], the run-time is informed when it has too many kernel threads activated, and it reduces the number of kernel threads. In *Scheduler Activations* [2], Anderson et al. propose a mechanism to manipulate threads and to communicate the processor scheduler with the run-time library. McCann et al. also propose in [65] an interface between the processor scheduler and the run-time. They also propose some processor scheduling policies that will be commented in next Section.

Rather than to have an interface or a mechanism to communicate different levels, a different approach consists of deducing this information by observing the other scheduling level decisions. For instance, in ASAT [91] the run-time observes the number of processes in the system to deduce if the load is very high. In that case, the run-time decides to reduce its parallelism. This approach is also followed by the native IRIX run-time library.

As we have commented before, we have mainly completed and extended the NANOS interface. In Chapter 3 we describe the main modifications introduced in the NANOS interface.

2.4.2 Coordinating the queueing system and the processor scheduler

The explicit coordination between the queueing system and the processor scheduler has not been explicitly proposed. At this level, the typical interaction between levels consist of the observation to deduce information. A typical practice is to periodically evaluate the load of the system and to decide whether any parameter, such as the multiprogramming level, must be modified. This evaluation can be automatically done by the queueing system, or by the system administrator.

There are three main traditional execution environments. In the first one, the system applies a job scheduling policy, but not a processor scheduling policy. In that case, the multiprogramming level is determined by the number of applications that fill in the system. In that case, there is no coordination because there is no processor scheduling level. However, the multiprogramming level and the load of the system are indirectly controlled by the queueing system. This execution environment has the problem of the fragmentation. In this environments, jobs are rigid and they only can run with the number of processors requested. In these systems, fragmentation becomes an important issue because they are typically non-preemptive.

The second one is an execution environment without job scheduling policy, and with a processor scheduling policy. In these systems, the main problem is that the system can become overloaded, because there is no control about the number of applications executed. If at any moment, a high number of jobs arrive to the system, the system performance will degrade.

The third one is an execution environment that applies a job scheduling policy and a processor scheduling policy. In this case, depending on the granularity that the queueing system modifies its parameters, such as the multiprogramming level, the system frequently end up being low loaded (with unused processors but with queued applications), or overloaded (with much more processes than processors).

To summarize, there are two main problems related to the no coordination between processor scheduler and queueing system: If the system implements some mechanism to control the multiprogramming level, the main problem will be fragmentation. In the second case, if the system is uncoordinated, the problem will be that the system can saturate due to an excessive use of resources.

2.5 Job scheduling policies

Job scheduling policies decide which job should be executed, and in which order. Depending on the system. The job scheduling policy can also include the decision about in which node to execute it.

Job scheduling policies include Fixed and Variable policies. In fact, we consider that a job scheduling policy can be classified in the two classes, it depends more on the hardware than in the job scheduling policy itself. Feitelson considers as Fixed those policies that are executed in a system where the hardware determines partition sizes and only one application can be executed per partition. And, as Variable, those policies executed in a system where partition sizes can be specified by the queuing system.

Figure 2.9 shows some of the job scheduling policies presented in this Section. We differentiate between those policies that do not use application information, and those policies that use *a priori* information (execution time estimation).

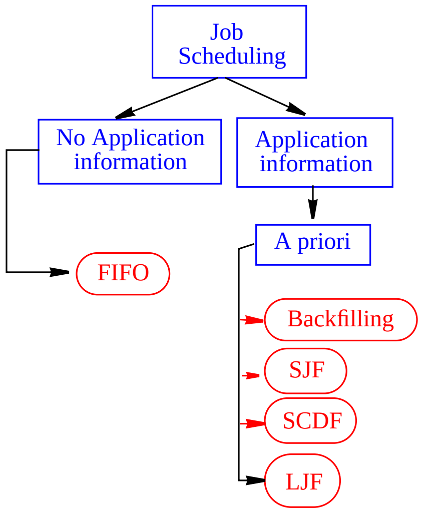


Figure 2.9: Job scheduling policies

2.5.1 Fixed policies

In fixed policies, the partition size is limited by the hardware, and set by the system administrator, typically for reasons related to access control or hardware limitations. This allows certain parts of the machine to be dedicated to certain groups of users. The main problem related to fixed partitioning is internal fragmentation. If a job requires a small number of processors, the rest are left unused. A partial solution is to create several partitions with different sizes, then users can choose the most suitable.

2.5.2 Variable policies

Variable partitioning is similar to fixed partitioning except that partition sizes are not predefined, partitions are defined according to application request. Partitions can be previously created with different sizes or arbitrary created. However, internal fragmentation may occur in cases where the allocation is rounded up to one of the predefined sizes, but the job does not use the extra processors. If the architecture does not impose any restriction, internal fragmentation can be avoided by creating arbitrary partitions. However, external fragmentation remains an issue because a set of processors can remain idle because they are not enough to execute any queued job.

The simplest approach is a *First Come First Serve* (FCFS or FIFO) approach, jobs are executed in the same order of arrival to the system. The favorite heuristic in uniprocessor systems is the *Shortest Job First* (SJF) [49]. However, this is not a good approach in multiprocessors because it needs to know the execution time in advance and long jobs might be starved. In parallel systems, jobs may be measured by the number of processors they request, which is suposed to be known in advance. Using this information, the *Smallest Job First* [60], and the *Largest Job First* [114] have been proposed. The first one is motivated by the same reason that the SJF. However, it turns out to perform poorly because jobs that require few processors do not necessarily terminate quickly [55][50]. To solve this problem a new approach appear, the *Smallest Cumulative Demand First* [55][60][89]. In this policy, jobs are ordered considering the product of number of processors and expected execution time. However, this policy did not show better results that the original *Smallest Job First* policy [50], and also suffers from the same problem than the *Shortest Job First*: it needs to know the application execution time in advance.

Finally, a job scheduling policy that has shown to perform well is called *Backfilling* [56]. Backfilling approach also needs a job execution time estimation. In *Backfilling*, jobs are executed in a first come first serve order but it allows to advance small jobs in the queue if they do not delay the execution of previously queued jobs [56][97]. Feitelson at al. show in [67] that the precision of user run-time estimations do not significantly affect the performance of Backfilling.

2.6 Processor scheduling policies

Processor scheduling policies decide how many processors to allocate to each job. At this level of decision, the processor scheduler can implement three types of policies: time-sharing, space-sharing, and gang scheduling. In time-sharing, the processor scheduler considers the kernel thread as the unit of scheduling. Space-sharing policies consider the job as the unit of scheduling. They divide the processor scheduling policy into two-levels: processors to jobs, and processors allocated to jobs to kernel threads. Gang scheduling is a combination of time-sharing and space-sharing: it implements a time-sharing among jobs, where each job implements a one-to-one mapping between processors and kernel threads. Figure 2.10 shows the three processor scheduling options.

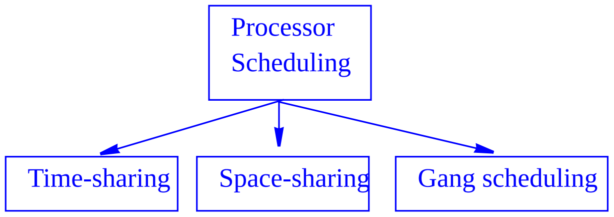


Figure 2.10: Processor scheduling policies

2.6.1 Time-sharing policies

Time-sharing policies are the automatic extension from uniprocessor policies to multiprocessor policies. Time-sharing policies do not consider grouping of threads. These policies are usually designed to deal with the problem of many-to-few mapping of threads to processors. There are two main approaches to deal with this problem: local queues (typically one per processor) or a global queue (shared by all the processors). Figure 2.11 shows the main approaches proposed in time-sharing policies.

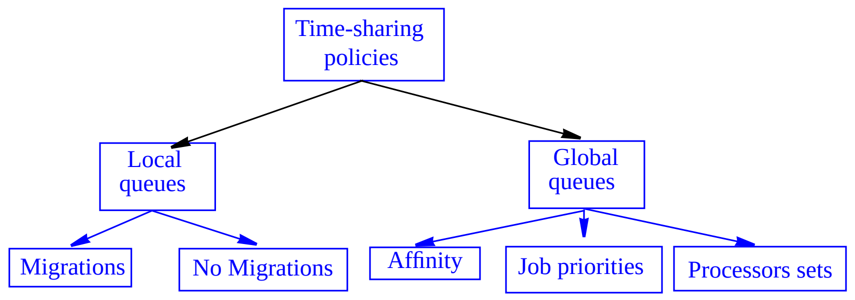


Figure 2.11: Time sharing policies

Local queues

To use local queues is the natural approach for distributed memory machines, but it is also suitable in shared-memory machines. Since normally each processor gives work for its own queue, there is no need for locks (no contention). In shared-memory machines, local queues also helps to maintain the memory locality. However, this approach has the problem that it needs a good load balancing mechanism to map threads to processors.

The use of local queues implies that threads are directly mapped to processors. This provides threads a sense of locality and context. Threads can keep data in local memory, supporting programming models that exploit locality. For example, a very useful programming paradigm is to partition the problem data set, and perform the same processing on the different parts, using some communication for coordination. In SPMD programming, this data partitioning is explicitly part of the program.

The main problem related to local queues is the load balancing. The load balancing was shown crucial to achieve an adequate performance [28], mainly in distributed systems. There is some controversy in the literature in whether the correct load balancing of new threads is enough [29], or maybe thread migration (“preemptive load balancing”) is also required [22][51].

There are several proposals to deal with the problem of thread placement. In the case where no migrations are considered, one simple approach is to use a random mapping[3][48]. Obviously, this scheme has a bad worst-case, because it is possible to choose the most highly loaded processor. A design that has been suggested a number of times is to partition the system using a buddy-system structure. A buddy-system creates a hierarchy of control points: the root controls the whole system, its two children, each controls half of the system, and so on. This control is used to maintain data about load on different parts of the system.

The second issue is load balancing where thread migrations are allowed. Important characteristics of load balancing algorithms are that they distribute the load quickly and that they be stable. Quick load distribution is achieved by non-local communication to disseminate information about load conditions[57]. Stability means that the algorithm should not over-react to every small change. It is achieved by balancing only if the difference in loads achieves a certain threshold [70], or by limiting the number of times a thread can migrate.

Global queues

A global queue is easy to implement in shared-memory machines but not a valid choice for distributed memory machines. Threads in a global queue can run to completion or can be preempted. In preemptive systems, threads run for a time and then return to the queue. In this Section, we will comment preemptive systems. The main advantage of using a global queue is that it provides automatic *load sharing*[3][32]. Drawbacks are possible contention in the queue and lack of memory locality.

A global queue is a shared data structure that should be accessed *with care*. The main methods used are through locks and using wait-free primitives. There is also some work on fault tolerance [111]. The direct implementation of a global queue uses locks to protect global data and allow multiple processors to add and delete threads. A lot of work has been done regarding the efficient implementation of locks. A global queue is mainly the same as in uniprocessor systems. This typically implies that entries are sorted by priorities, and multi-level feedback is used rather than maintaining strict FIFO semantics. The problem of using locks is the contention [10][44]. Indeed, measurements on a 4 processors machine show that already 13.7% of the attempts to access the global queue found it locked, and the trend indicated that considerable contention should be expected for large number of processors [103]. The alternative is use a bottleneck-free implementation of queues, that does not require locking such as the fetch-and-add primitive [73].

The approach of partitioning and then scheduling using a global queue shared by all the processors in the partition was implemented in the Mach operating system [11][12]. The partitioning is based on the concept of a processor set, in this case a global priority queue is associated with each processor set.

The order in which threads are scheduled from the global queue is determined by their relative priorities. Most implementations use the same algorithms that were used in uniprocessor systems.

Another important consideration in thread scheduling is matching threads with the most appropriate processor for them to run on. This is normally equivalent to the question of data that may reside in the processor cache. The scheduling policy that tries to schedule threads on the same processor on which they ran most recently, under the assumption that this particular processor may still have some relevant data in its cache is called *affinity scheduling* [4][8][23][98][106].

3. The term “load sharing” originated in distributed systems where it describes systems that do not allow a processor to be idle if there is work waiting at another processor.

All the mentioned scheduling schemes operate on individual threads, and do not consider the fact that these threads belong to competing jobs. As a result, jobs tend to receive service proportionally to the number of threads they spawn. An alternative approach is to give jobs equal degrees of services [55], threads belonging to a job with lots of threads should have a lower priority than threads belonging to jobs with few threads. It is also possible to prioritize jobs. This can be done by keeping the threads in separate per-job queue. Processors serve these queues in a round robin way, and set the time quantum for each scheduled thread according to the priority of the job to which this thread belongs [83].

2.6.2 Space-sharing policies

Space-sharing is done by partitioning the machine among applications that run side by side. Applications run in these partitions as in a dedicated machine. This approach is motivated by the desire to reduce the operating system overhead on context switching [105]. With space-sharing, the system is more involved in allocating processors to jobs. The application run-time system may then schedule user threads on these processors[104][113].

In *adaptive partitioning*, the partition size is defined when application is launched, and fixed during the complete execution time. In *dynamic partitioning*, the processor allocation can change at run-time. Adaptive and dynamic partitioning are both considered as processor scheduling policies because they are involved in processor scheduling decisions, not job scheduling.

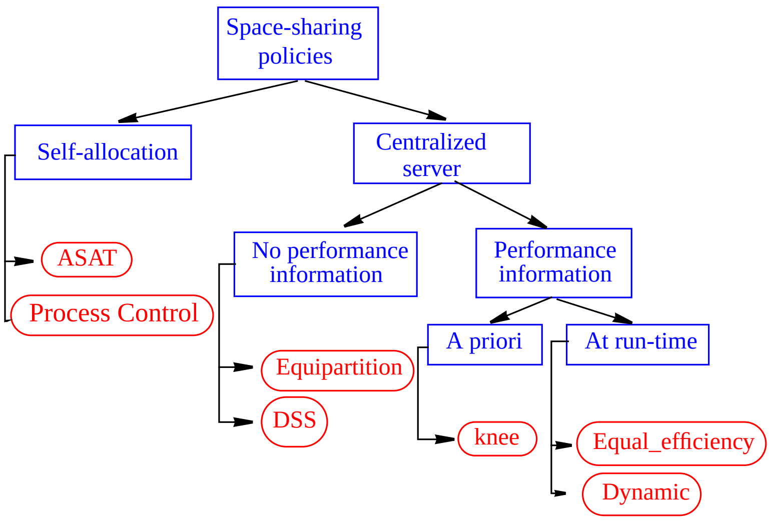


Figure 2.12: Space-sharing policies classification

In adaptive and dynamic partitioning, the system decides how many processors to allocate to each application. In adaptive partitioning, processor allocation must be maintained because the applications are moldable, not malleable. Moldable applications are those than can be executed with any number of processors, but are not able to adapt their parallelism to changes on the processor allocation. Then, the processor allocation must be maintained during the complete execution of the application. In dynamic partitioning, the processor allocation can change at run-time. However, most of the adaptive policies can be considered also as dynamic processor allocation policies since they depend more on the running application than on the policies themselves.

It is important to comment that, the ability to apply a dynamic processor scheduling policy does not necessarily imply frequent processor re-distribution. The processor scheduling policy could decide the same distribution each time that it is re-applied. Such a stable allocation is, in fact, a desirable situation to avoid processor migrations. It is important to be able to react to changes in both the applications or the system, but also to maintain the allocation if conditions are stable.

In this Section, we focuses in the first level of decision of the processor scheduler, processors to jobs. This decision is also called the *processor allocation policy*. The second level can be implemented by the processor scheduler or by the run-time itself. In our case, the processor scheduler takes this decision. We have followed the proposal made by Tucker and Gupta in *process control* [105]. They propose to do a one-to-one mapping to achieve the optimal execution point. This mechanism is presented in Chapter 3.

Adaptive policies

In adaptive partitioning, the partition size can be decided by the system (on its own), or in cooperation with the application. If the system decides the partition, a simple approach is giving each job as many processors as it requests. However, if the sum of processors requested is greater than the total number of processors, jobs will receive less processors than their request [113][89]. In the extreme case, processor allocation could be reduced to one processor. This approach increases the execution time with respect to the case that each job uses as many processor as it requests, but guarantees that short jobs do not have to wait for large jobs.

One approach is to perform an Equipartition [54][55] among jobs. Equipartition tries to allocate an equal-size partition for all current jobs (limited by the job request). Jobs can run simultaneously and do not have to wait in the queue. Equipartition achieves good performance in both minimizing response time and maximizing throughput.

A few other algorithms have been proposed that do not derive directly from the equipartition. For instance limiting the maximum partition size that a single job can obtain [86]. Another approach is based on a state machine. At each instant the system is in a certain state, which identifies the ideal partition size [86]. With a system with P processors, the states are {P, P/2, P3,..., 1}. Jobs are allocated to partitions whose size is determined by the state. A more sophisticated approach is to base the decisions on predictions of the queuing time, and of how long current jobs will run [26]. These policies do not consider the application characteristics, just the job request.

Another alternative is to decide the partition size in collaboration with the application. One option is to consider the *execution signature*, i.e. how much time it would require on different partition sizes [59][79]. The system combines this information with the load information to derive the optimal partition size. In general, information about the total work and the efficiency of the job is beneficial [13]. The problem of this approach is that it assumes that the information is precise and that the application cooperates with the system. One alternative is that the system collects the information itself, based on previous executions of the job [41]. Eager *et al.* propose in [30] to allocate the number of processors that achieve the *Knee* of the curve execution time vs. efficiency.

Dynamic policies

Dynamic partitioning modifies the processor allocation of applications at run-time to react to different events such as the system load, the application request, performance information, etc. For example, when an application enters in a sequential phase its processors can be reallocated to another application [113][106][112].

The main reason to change the partition sizes at runtime is the desire to improve fairness and resource utilization. Thus, when a new job arrives, a fair share of processors should be preempted from running jobs, and given to the new job. When a job terminates, its processors should be divided among the other jobs. In essence, this is the *Equipartition* policy [105][55][65]. *Dynamic Space Sharing* (DSS) is a dynamic processor allocation proposed by Polychronopoulos et al. in [84] that decides an allocation proportional to the number of requested processors.

Efficiency can also become an issue. Consider a job that it is written so that the work is divided into 8 equal pieces. Running such a job on a partition of 7 processors would not provide any additional benefit respect to a partition of 4 processors, leading to waste of resources [113]. Nguyen et al. propose *Equal_efficiency* in [74][75][76]. The goal of the Equal_efficiency is to achieve an equal efficiency in all the processors. In that case, when a job is submitted the system executes it for short periods on different partition sizes and measures the job efficiency. The policy allocates processors (one by turn) to those applications that achieve the higher efficiency. The main problem of the Equal_efficiency is that it allocates processors tho those applications that achieve the best efficiency, however, the "best" is not a synonym of good efficiency. Another related problem to this policy is the way the estimate the efficiency achieved by running applications.

McCann propose in [65] *Dynamic*, a dynamic space-sharing policy that reallocates processors from one parallel job to another based on changes in the parallelism. Jobs inform the scheduler about the number of processors they could use. They also informs the scheduler when there are no ready application threads to run. Processors are then moved from jobs that do not use them to processors that could use them. Dynamic calculates the "use of processors" considering the idle periods of the application, resulting in a great number of reallocations, as it was shown in [76].

Some versions of dynamic partitioning operates in the following way: instead of explicitly allocating processors to jobs, jobs create threads that execute on available processors. If there are too many threads in the system, jobs voluntarily reduce the number of threads. If there are free processors, jobs create new threads to use them. Examples of this approach includes *Process Control* [105] and ASAT [91].

The processor scheduling policy implemented by the native operating system IRIX 6.5 is a combination of a priority based time-sharing policy with an affinity scheduling.

2.6.3 Gang Scheduling

Gang Scheduling is a technique proposed by Ousterhout in [78] that combines space and time sharing and it was presented as the solution to the problems of static space-sharing policies. We define gang scheduling as a scheme that combines three features [32]:

- Application threads are grouped into gangs (typically all the threads of the application in the same gang)
- Threads in each gang are executed simultaneously, using a one-to-one mapping
- Time slicing is used among gangs (all the threads in the gang are preempted at the same time)

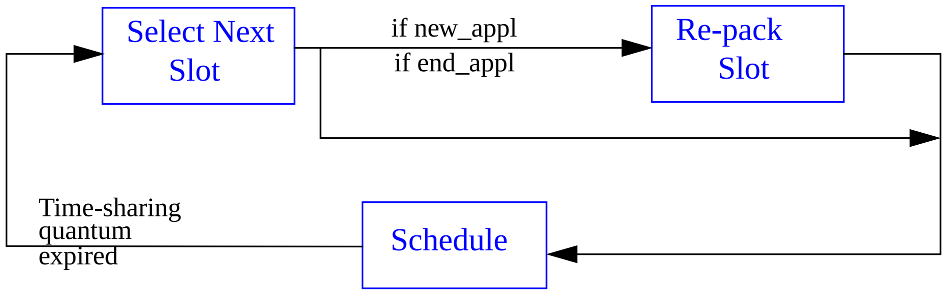


Figure 2.13: Gang scheduling policies diagram

Figure 2.13 shows the diagram that represents the gang scheduling policies behavior. Periodically, at each time-sharing quantum expiration, the system selects a new slot to execute. A slot is a set of applications that will be concurrently executed. If during the last quantum, any new job has arrived to the system, or any job has finished, the slot will be re-organized. The algorithm to re-organize one slot (or the complete list of slots) is called the re-packing algorithm. The re-packing algorithm is applied to migrate some job from any other slot to the currently selected. Traditionally, the scheduling phase is traduced by a single dispatch, to allocate as many processors to jobs as they request. Gang scheduling policies mainly differs in the re-packing algorithm applied.

Figure 2.14 shows a taxonomy as a function of the three elements that define a gang scheduling policy: the re-packing algorithm, the job mapping, and the scheduling algorithm applied.

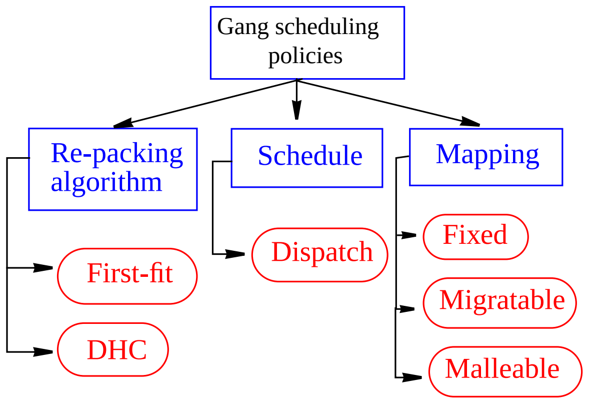


Figure 2.14: Gang scheduling taxonomy

The simplest version of gang, where threads are always rescheduled in the same set of processors is the most popular one. However there have been proposed more flexible versions [81]. One of this is migratable preemptions, where threads are preempted in one set of processors and resumed in another to reduce the fragmentation.

Other similar scheme to gang scheduling is Coscheduling[78]. Coscheduling was originally defined by Ousterhout to describe systems where the operating system attempts to schedule a set of threads simultaneously, as in gang scheduling. However, if it cannot, it will resort to scheduling only a subset of the threads simultaneously.

Feitelson and Jette argue in [34] that Gang Scheduling solves the problem of taking incorrect decisions while performing job scheduling. Feitelson and Rudolph comments in [35] that academically speaking Gang Scheduling is inferior to dynamic partitioning, but that dynamic partitioning is difficult to implement and that drawbacks of gang are not so critical. They conclude that "*the advantages of Gang Scheduling generally outweigh its drawbacks*".

Several works have analyzed the problem of fragmentation in Gang Scheduling and have proposed several re-packing algorithms as solution to this problem. Feitelson in [33] analyzes several algorithms of job re-packing for Gang Scheduling and concludes that the best option is a buddy system or to use migration to re-map the jobs (based on a first-fit algorithm). Feitelson and Rudolph propose and evaluate Distributed Hierarchical Control (DHC) in [35][37]. DHC is a design using a hierarchy of controllers that dynamically re-partitions the system according to changing job requirements. They show, through simulations, that DHC achieves performance comparable to off-line algorithms.

Zhou *et al.* [115] also attack the fragmentation problem and present ideas such as job re-packing, running jobs on multiple slots, and minimizing the number of time slots in the system to improve the buddy scheme for Gang Scheduling. Setia [88] shows through simulations that Gang Scheduling policies that support job migration offer significant performance gains over policies that do not use remapping. Other authors have analyzed other aspects of the performance of Gang Scheduling under different kind of applications. Silva and Scherson propose Concurrent Gang [94] to improve the performance of I/O intensive applications. Using simulations they show that Concurrent Gang combines the advantages of Gang Scheduling for communication and synchronizations intensive applications with the flexibility of a Unix scheduler for I/O intensive applications. They also classify applications through run-time measurements in [95] to provide better service to I/O bound and interactive jobs under gang. They propose to improve the utilization of idle times (idle slots and blocked tasks) and to control the spinning time of tasks. Gare and Leutenegger [40] analyze the relation between job size and quantum allocation. They allocate a number of quanta inversely proportional to the number of processes per job to reduce the slowdown.

Traditionally, systems that apply a gang scheduling policy do not include a job scheduling policy because one of the goals of gang scheduling is to reduce the impact of incorrect job scheduling decissions.However, Zhang et al. [116] propose Backfilling gang scheduling to improve the system performance. They show that, even theoretically gang scheduling policies allows the execution of any new job, due to resource limitations some jobs must remain queued until some running job finishes. Combining Backfilling and Gang scheduling the queued time can be reduced.

Polychronopoulos et al. attack gang scheduling problems by proposing Sliding Window-DSS (SW-DSS) and Variable Time Quanta-DSS (VTQ-DSS) [85], two variations of their implementation of the Dynamic Space-Sharing (DSS) policy. SW-DSS and VTQ-DSS introduce a time-sharing among jobs, allocating a number of processors proportional to the number of requested processors. These approaches do not consider the performance achieved by running applications, and the number of processors received is proportional to the number of processors requested.

2.7 Programming models

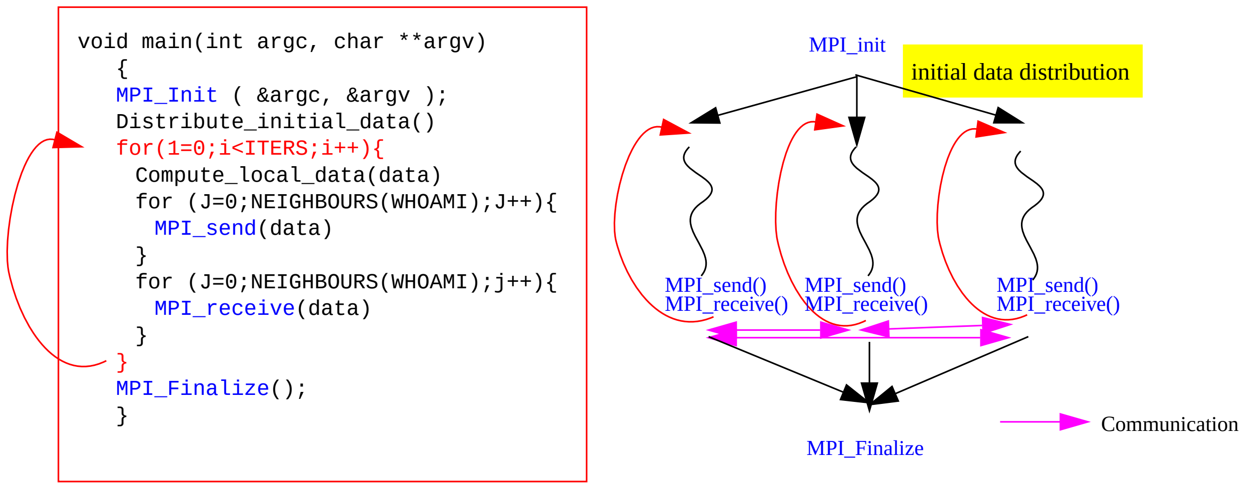
There are two main programming models to parallelize sequential programs: MPI and OpenMP.

2.7.1 Message Passing Interface: MPI

MPI [66] is based on the idea of an execution environment with different address spaces. The MPI programming model divides the algorithm in a fixed number of tasks, and each task is executed in a process. MPI forces that the programmer divides the complete algorithm, it does not allow a progressive parallelization. Moreover, since the different tasks will be executed in different address spaces, data communication must be done by explicit message passing.

Most of the parallel applications are iterative, then the typical SPMD behavior of an MPI application is the following (see Figure 2.15):

- The master task reads the data and distributes the data among the rest of tasks.
- Each task calculates one iteration of the loop.
- Each task sends the data to its neighbours
- Each task receives the data from its neighbours
- Each task executes the next iteration and so on.



```
void main(int argc, char **argv)
   {
   MPI_Init ( &argc, &argv );
   Distribute_initial_data()
   for(1=0;i<ITERS;i++){
     Compute_local_data(data)
     for (J=0;NEIGHBOURS(WHOAMI);J++){
       MPI_send(data)
     }
     for (J=0;NEIGHBOURS(WHOAMI);j++){
       MPI_receive(data)
     }
   }
   MPI_Finalize();
   }
```

Figure 2.15: MPI example

This behavior implies few points of data communication per task and no task re-scheduling. Since the number of tasks is fixed during the complete execution of the application, no application malleability is allowed in MPI applications. In addition, since the data communication is done explicitly and at a few points, load balancing is not easily implementable.

2.7.2 OpenMP

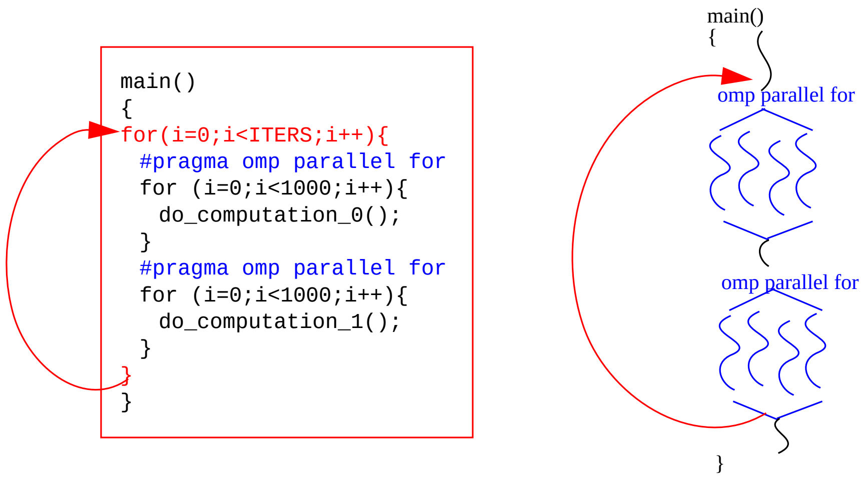
OpenMP [77] is based on the idea of an execution environment with a global (shared) address space. The OpenMP programming model does not explicitly divide the algorithm in tasks. Programmers define which loops, and Sections of code, can be executed in parallel. The compiler and the run-time, as a function of the number of processors, and the loop scheduling policy selected, will generate the tasks that will execute the algorithm in parallel.

Since the OpenMP programming model assumes a shared address space, communication is implicitly done using variables. Programmers do have not to include any explicit directive or function to access the data used by multiple tasks.

In OpenMP, scheduling decisions are taken once per parallel loop or parallel Section. This fact gives OpenMP applications the characteristic that they can adapt the parallelism that they spawn to the number of processors available (malleability). Of course, to do that they need support from the run-time library, but in MPI this characteristic is not possible because it only takes scheduling decisions at the initial of the application.

OpenMP works by inserting directives in the sequential source code. In fact, the same code can generate a sequential or parallel version just activating or disactivating the OpenMP directives. Another advantage of this programming model is that parallelization can be done incrementally. Users can perform a profile of their sequential program and just parallelize the most time consuming loops of their code. The program can then be tested and evaluated. If the program achieves enough speedup, the parallelization can be finished at this point. Otherwise, users can parallelize additional loops.

Figure 2.16 shows an OpenMP example. The code in the left side shows a simple program with two loops. The *omp parallel for* directive specifies that the following loop is parallel.



```
main()
{
for(i=0;i<ITERS;i++){
  #pragma omp parallel for
  for (i=0;i<1000;i++){
    do_computation_0();
  }
  #pragma omp parallel for
  for (i=0;i<1000;i++){
    do_computation_1();
  }
}
}
```

Figure 2.16: OpenMP example

The figure in the right side corresponds to the parallelism that will be generated in this example. The program is executed in sequential until the *omp parallel for* directive is found. The *omp parallel for* directive creates a *team*[4] of threads that will collaborate to execute the parallel loop.

2.7.3 OpenMP directives

The OpenMP programming model provides directives that specify parallel regions, work-sharing constructs, synchronization, and data environment constructs.

Parallel regions constructs

A parallel region is a block of code that is going to be executed by multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts a parallel execution. When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team.

```
!$OMP PARALLEL [clause[[,] clause]...]
Block
!$OMP END PARALLEL
```

Figure 2.17: Parallel region directive

Figure 2.17 shows the OpenMP code to specify a parallel region. To give more information about the valid clauses see [77]. The number of physical processors actually hosting the threads at any given time is implementation-dependent. Once created, the number of threads in the team remains constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the runtime system from one parallel region to another. The OMP_SET_DYNAMIC primitive and the

4. Threads may be physically created at this point and waiting for work in an idle loop

OMP_DYNAMIC environment variables can be used to enable and disable the automatic adjustment of the number of threads. Within the dynamic extent of a parallel region, thread numbers uniquely identify each thread. Threads numbers are consecutive and range from zero, for the master thread, up to one less than the number of threads within the team. *Block* denotes a structured *block* of Fortran statement. It is noncompliant to branch into or out of the block. The code contained in *block* is executed by each thread. The END PARALLEL directive is an implicit barrier at this point. Only the master thread of the team continues the execution past the end of a parallel region.

Work-Sharing constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed dynamically within a parallel region, if not, it is treated as whether the thread that encounters it were a team of size one. The main work-sharing construct directives are DO and SECTION.

Figure 2.18 shows the DO directive format. It specifies that the iterations of the immediately following DO loop must be executed in parallel. Iterations of the DO loop are distributed across threads that already exists. One of the clauses that users can specify defines the loop scheduling to use. It can be STATIC, DYNAMIC, GUIDED or RUNTIME.

```
!$OMP DO [clause [[,]...]
do_loop
[!$OMP END DO [NOWAIT]]
```

Figure 2.18: DO directive

Figure 2.19 presents the SECTIONS directive which is a non-iterative work-sharing construct. It specifies that the enclosed Sections of code are to be divided among threads in the team. Each Section is executed once by a thread in the team.

```
!$OMP SECTIONS [CLAUSE [[,]CLAUSE]...]
[!$OMP SECTION]
block
[!$OMP SECTION
block]
...
!$OMP END SECTIONS [NOWAIT]
```

Figure 2.19: SECTION directive

Synchronization constructs

The OpenMP programming model provides directives to synchronize threads. The MASTER directive specifies that the code enclosed within the MASTER and END MASTER directives is executed by the master of the team. The CRITICAL and END CRITICAL directives restrict access to the enclosed code to only one thread at a time. The BARRIER directive synchronizes all the threads in a team. Other synchronization directives are the ATOMIC, FLUSH and ORDERED.

Data environment constructs

The data environment constructs controls the data environment during the execution of parallel constructs. The THREADPRIVATE directive makes named common blocks and named variables private to a thread but global within the thread. There are several data attribute clauses that specify, for instance, if a particular variable is PRIVATE or SHARED among threads in a team.

2.7.4 MPI+OpenMP

Last years, clusters of SMP's have appeared as a possible architectural structure that combines the benefits of the two approaches: the scalability of distributed-memory machines and the easiness of use of shared-memory architectures. The programming model proposed for these new architectures is an hybrid programming model: MPI+OpenMP. This programming model is also useful in DSM's such as the SGI Origin2000. Figure 2.20 shows the behavior of a MPI+OpenMP application. It tries to exploit the advantage provided by both programming models.

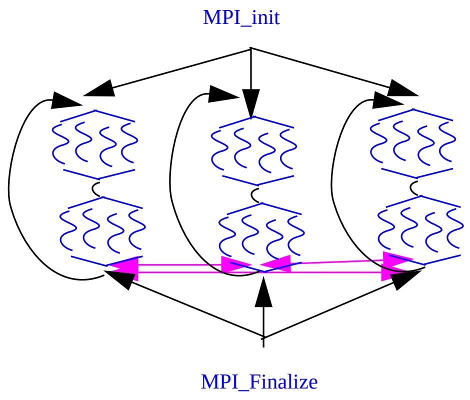


Figure 2.20: MPI+OpenMP

2.8 Summary

In this Chapter, we have briefly described the main multiprocessor architectures focusing in the SGI Origin 2000. The SGI Origin 2000 has been the target architecture of this Thesis because the availability and because it is representative of modern shared-memory multiprocessors architectures.

We have also presented an overview of the main approaches in scheduling policies. We classify scheduling policies in job scheduling, processor scheduling, and loop scheduling. This Thesis focuses on job scheduling and mainly in processor scheduling policies, then loop scheduling policies are not presented.

Finally, we have presented the two main programming models used to parallelize sequential applications: MPI and OpenMP. MPI is based on the idea that the application will be executed in a system with separated address spaces. Then, it partitions the problem in different task that will be executed in separated address spaces. On the other hand, OpenMP is based on the idea that the application will be executed in a system with a global shared address space. OpenMP parallelizes applications by specifying which parts of the code (mainly loops) can be executed in parallel. It delegates the decision of the parallelization to the run-time library, who takes this decision at the run-time based on the loop scheduling policy, application parameters such as the data size, and the number of processors available. We have also briefly presented a new programming model resulting from the combination of MPI and OpenMP. It is based on the idea of exploiting characteristics of systems that combines separated address spaces and global address spaces. This programming model incorporates the benefits and problems of the two previously presented programming models.