Performance-Driven Processor Allocation

Julita Corbalán, Xavier Martorell, Jesús Labarta

Departament d’Arquitectura de Computadors (DAC)

Universitat Politècnica de Catalunya (UPC)

{juli,xavim,jesus}@ac.upc.es

Abstract

This work is focused on processor allocation in shared-memory multiprocessor systems, where no knowledge of the application is available when applications are submitted. We perform the processor allocation taking into account the characteristics of the application measured at run-time. We want to demonstrate the importance of an accurate performance analysis and the criteria used to distribute the processors. With this aim, we present the *SelfAnalyzer,* an approach to dynamically analyzing the performance of applications (speedup, efficiency and execution time), and the Performance-Driven Processor Allocation (PDPA), a new scheduling policy that distributes processors considering both the global conditions of the system and the particular characteristics of running applications. This work also defends the importance of the interaction between the medium-term and the long-term scheduler to control the multiprogramming level in the case of the clairvoyant scheduling policies[1]. We have implemented our proposal in an SGI Origin2000 with 64 processors and we have compared its performance with that of some scheduling policies proposed so far and with the native IRIX scheduling policy. Results show that the combination of the *SelfAnalyzer+PDPA* with the medium/long-term scheduling interaction outperforms the rest of the scheduling policies evaluated. The evaluation shows that in workloads where a simple equipartition performs well, the PDPA also performs well, and in extreme workloads where all the applications have a bad performance, our proposal can achieve a speedup of 3.9 with respect to an equipartition and 11.8 with respect to the native IRIX scheduling policy.

1 Introduction

The performance of current shared-memory multiprocessors systems heavily depends on the allocation of processors to parallel applications. This is especially important in NUMA systems, such as the SGI Origin2000 [SGI98]. This work attacks the problem of the processor allocation in an execution environment where no knowledge of the application is available when applications are submitted.

Many researchers have considered the use of application characteristics in processor scheduling [Brecht96] [Chiang94][Marsh91][Nguyen96][NguyenZV96][Parsons96]. In these works, parallel applications are characterized by different parameters such as the maximum speedup, the average parallelism, or the size of the working set. Performing the processor allocation without taking into account these characteristics can result in a bad utilization of the machine. For instance, allocating a high number of processors to a parallel application with small speedup will result in a loss of processor performance.

Traditionally, characteristics of parallel applications were calculated in two different ways. The first approach is that the user or system administrator performs several executions under different scenarios, such as the input data or the number of processors, and collects several measurements. A second approach, used in research environments [Brecht96] [Chiang94] [Helmbold90] [Leutenegger90] [Madhukar95] [Parsons96] [Sevcik94], defines a job model, characterizing the applications by a set of parameters, such as the average of parallelism or the speedup. This information is provided to the OS as an *a priori* input, to be taken into account in subsequent executions.

This approach has several drawbacks. First of all, these tests can be very time-consuming, even, they can be prohibitive due to the number of combinations. Furthermore, many times the performance of the application depends on the particular input data (data size, number of iterations). Second, the behavior of the applications is influenced by issues such as the characteristics of the processors assigned to them, or the run-time mapping of processes to processors, or the memory placement.

[1] Those scheduling policies that consider the application characteristics

These issues determine the performance of the applications and are only available at run-time. Finally, the different analytic models proposed so far are not able to represent the behavior of the application at run-time. Moreover, analytic models try to characterize the application when it is individually executed, not in a shared environment. Most of the previous approaches are based on analytic models.

On the other hand, the typical way to execute a parallel application in production systems is through a long-term scheduler, i.e. a queueing system [Feitelson95]. The queueing system manages the number of applications that are executed simultaneously, usually known as the multiprogramming level[1]. In this execution environment, jobs are queued until the queueing system decides to execute it. This work is based on execution environments where the applications arrival is controlled by a long-term scheduler.

This work relies on the utilization of the characteristics of the applications calculated at run-time and on using this information for processor scheduling. In particular, we propose to use the speedup and the execution time with *P* processors. This work is focused on demonstrating the importance of: the accuracy of the measurements of the application characteristics, the criteria used to perform the processor scheduling, and the coordination with the queueing system, in the performance that may be achieved by parallel applications. With this aim, we present: (1) a new approach to measure the speedup and the execution time of the parallel applications, the *SelfAnalyzer*, (2) a new scheduling policy that uses the speedup and the execution time to distribute processors, the *Performance-Driven Processor Allocation (PDPA)* policy, (3) and a new approach to coordinating the (medium-term) scheduler with the queueing system (long-term scheduler).

Our approach has been implemented in an Origin2000 with 64 processors. Applications from the SPECFp95 benchmark suite and from the NAS benchmarks have been used to evaluate the performance of our proposal. All the benchmarks used in the evaluation are parallelized with OpenMP [OpenMP2000] directives. Finally, in the current implementation we assume that applications are malleable [Feitelson97], applications that can adjust to changing allocations at runtime.

Results show that the combination of the *SelfAnalyzer+PDPA* with the medium/long-term scheduling interaction outperforms the rest of the scheduling policies evaluated. The evaluation shows that, in workloads where a simple equipartition performs well, the *PDPA* also performs well, and in extreme workloads where all the applications have a bad performance, our proposal can achieve a speedup of 3.9 with respect to an equipartition and 11.8 with respect to the native IRIX scheduling.

The remainder of this paper is organized as follows: Section 2 presents the related work. Section 3 presents the execution environment in which we have developed this work. Section 4 presents the *PDPA* scheduling policy. Section 5 presents the evaluation of the *PDPA* compared to some scheduling policies proposed so far and the IRIX scheduling policy. Finally, section 6 presents the conclusions of this work.

2 Related Work

Many researchers have studied the use of characteristics of the applications calculated at run time to perform processor scheduling. Majumdar *et al* [Majumdar91], Parsons *et al* [Parsons96], Sevcik [Sevcik94][Sevcik89], Chiang *et al* [Chiang94] and Leutenegger *et al* [Leutenegger90] have studied the usefulness of using application characteristics in processor scheduling. They have demonstrated that parallel applications have very different characteristics such as the speedup or the average of parallelism that must be taken into account by the scheduler. All these works have been carried out using simulations, not through the execution of real applications, and assuming *a priori* information.

Some researchers propose that applications should monitor themselves and tune their parallelism, based on their performance. Voss *et al* [Voss99] propose to dynamically detect parallel loops dominated by overheads and to serialize them. Nguyen *et al* [Nguyen96] [NguyenZV96] propose *SelfTuning*, to dynamically measure the efficiency achieved in iterative parallel regions and select the best number of processors to execute them considering the efficiency. These works have demonstrated the effectiveness of using run-time information.

Other authors propose to communicate these application characteristics to the scheduler and let it to perform the processor allocation using this information. Hamidzadeh [Hamidzadeh94] proposes to dynamically optimize the processor allocation by dedicating a processor to search the optimal allocation. This proposal does not

[1] In our environment, the multiprogramming level is normally set to allow the simultaneous execution of a small number of applications (two, three or four).

consider characteristics of the applications, only the system performance. Nguyen *et al* [Nguyen96][NguyenZV96] also use the efficiency of the applications, calculated at run-time, to achieve an *equal-efficiency* in all the processors. Brecht *et al* [Brecht96] use parallel program characteristics in dynamic processor allocations policies, (assuming *a priori* information). McCann *et al* [McCann93] propose a scheduler that dynamically adjust the number of processors allocated to the parallel applications to improve the processor utilization. Their approach considers the application-provided idleness to allocate the processors, resulting in a large number of re-allocations.

To obtain application characteristics, previous systems have taken approaches such as the use of the hardware counters provided by the architecture, or monitoring the execution time of the different phases of the applications. Weissman [Weissman98] uses the performance counters provided by modern architectures to improve the thread locality. McCann *et al* [McCann93] monitor the idle time consumed by the processors. Nguyen *et al* [Nguyen96][NguyenZV96] combined both, the use of hardware counters and the measurement of the idle periods of the applications.

The most studied characteristic of parallel applications has been the speedup. Several theoretical studies have analyzed the relation between the speedup and other characteristics such as the efficiency. Eager, Zahorjan and Lazowska define in [Eager89] the speedup and the efficiency. Speedup is defined for each number of processors *P* as the ratio between the execution time with one processor and with *P* processors. Efficiency is defined as the average utilization of the *P* allocated processors. The relationship between efficiency and speedup is shown in Figure 1

$$S(P) = \frac{T(1)}{T(P)} \quad \longrightarrow \quad E(P) = \frac{S(P)}{P}$$

Figure 1: Speedup and efficiency definitions

Helmbold *et al* analyze in [Helmbold90] the causes of loss of speedup and demonstrate that the super-linear speedup exists basically due to memory cache effects.

Our work has several characteristics that differ from the previously mentioned proposals. First of all, with respect the parameters used by the scheduling policy, our proposal considers two characteristics of the applications: the speedup and the execution time. We also propose to consider the variation in these characteristics proportionally to the variation in the number of allocated processors. Second, we differ in the way the application characteristics are acquired. We believe that parameters such as the speedup can only be accurately calculated as the relation between two measurements, as opposed to [Nguyen96]. Furthermore, since the execution time of the applications is used by the scheduler, we propose a new approach to estimate the execution time of the whole application. Our measurements are based on the time, not on the hardware performance counters. In this way our method is independent from the architecture. Third, we have implemented and evaluated our proposal using real applications and a real architecture, the Origin2000. Simulations do not consider important issues of the architecture such as the data locality. And finally, we consider the benefit provided by the interaction of the (medium-term) scheduler with the long-term scheduler (queueing system).

3 Performance-Driven Processor Allocation

This section presents the three components of this work. Figure 2 shows the general overview of our execution environment. (1) Parallel applications calculate their performance through the *SelfAnalyzer* which informs the scheduler about the achieved speedup with the current number of processors, the estimation of the execution time of the whole application, and the requested number of processors. (2) Periodically (at each *quantum*[1] expiration) the scheduler wakes up and applies the scheduling policy, the *PDPA*. The *PDPA* distributes the processors among the parallel applications considering their characteristics, global system status, such as the number of processors allocated in the previous *quantum*, and the requested number of processors of each application. Once the processor allocation has been decided, the scheduler enforces it by suspending or resuming the application's processes. The scheduler informs the applications about the number of processors assigned to each one and applications are in charge of adapting their parallelism to their current allocation. In our work, the scheduler is a user-level application, and it must enforce the processor allocation through the native operating system calls such as suspend, or resume. Finally, the scheduler interacts with the queueing system to dynamically modify the multiprogramming level (3). The result is a multiprogramming level adapted to the particular characteristics of the running applications.

[1] A typical quantum value is 100 ms

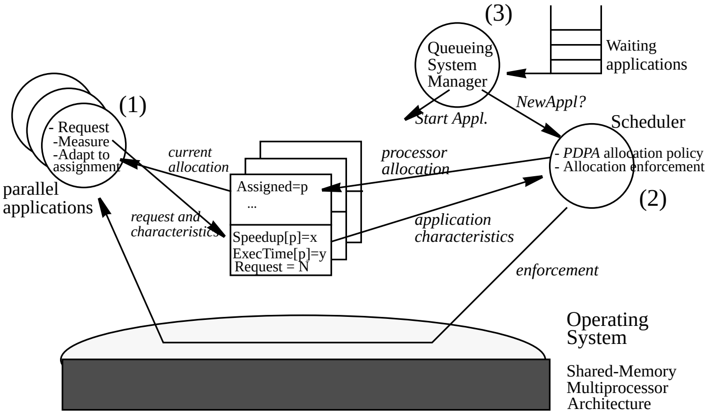

Figure 2: General overview

3.1 Dynamic Performance Analysis: *SelfAnalyzer*

The *SelfAnalyzer* [Corbalan99] is a run-time library that dynamically calculates the speedup achieved by the parallel regions, and estimates the execution time of the whole application. The *SelfAnalyzer* exploits the iterative structure of a significant number of scientific applications. The main time-consuming code of these applications is composed of a set of parallel loops inside a sequential loop. Iterations of the sequential loop have a similar behavior among them. Then, measurements for a particular iteration can be considered to predict the behavior of the next iterations, also exploited in [Nguyen96].

We believe that the speedup should be calculated as the relationship between two measurements: the sequential or reference execution time and the parallel execution time. In [Corbalan99] we demonstrated that the speedup calculated as a function of only one measurement can not detect significant issues such as the super-linear speedups. Figure 3 shows the formulation used by the *SelfAnalyzer* to calculate the speedup and to estimate the execution time.

To calculate the speedup, the *SelfAnalyzer* measures the execution time of each outer sequential iteration and also monitors the sequential and parallel regions inside the outer loop. It executes some initial iterations of the sequential loop with a predefined number of processors, (*baseline*), to be used as reference for the speedup computation. Once T(baseline) is computed, (1) in Figure 3, the application goes on measuring the execution time but with the number of processors allocated by the scheduler. If *baseline* is one processor, the calculated speedup will correspond with the traditional speedup measurement. Since the execution of some initial iterations with one processor could consume a lot of time, we propose to set the *baseline* greater than one processor. In [Corbalan99] we demonstrate that setting *baseline* to four processors is a good trade-off between the information provided by the measurement and the amount of overhead introduced because of executing the first iterations with a small number of processors. However, this approach has the drawback that it does not allow us to directly compare speedups among applications. Setting *baseline* to four processors, the speedup with four processors of an application that scales well will be one and the speedup with four processors of an application that scales poorly will be also one.

We use Amdahl's law [Amdahl67] to normalize the speedups inside an application. Amdahl's law bounds the speedup that an application can achieve with P processors based on the fraction of sequential code.

We call this function the Amdahl's Factor (AF), see (2) in Figure 3. In this way, we calculate the AF of the *baseline* and use this value to normalize the speedups calculated by the *SelfAnalyzer.*

Considering the characteristics of these parallel applications, and taking into account their iterative structure,

$$(1)\ S(p) = \frac{T(\text{baseline})}{T(p)} \times AF(Baseline)\text{, where } AF(Baseline) = \frac{1}{\left(f + \frac{(1-f)}{Baseline}\right)}\ (2)$$

$$(3)\ ExTime(p) = ConsumedTime + \left(\frac{AF(Baseline) \times T(\text{baseline})}{S(p)} \times ItersRemaining\right)$$

Figure 3: Calculation of the speedup and execution time estimation

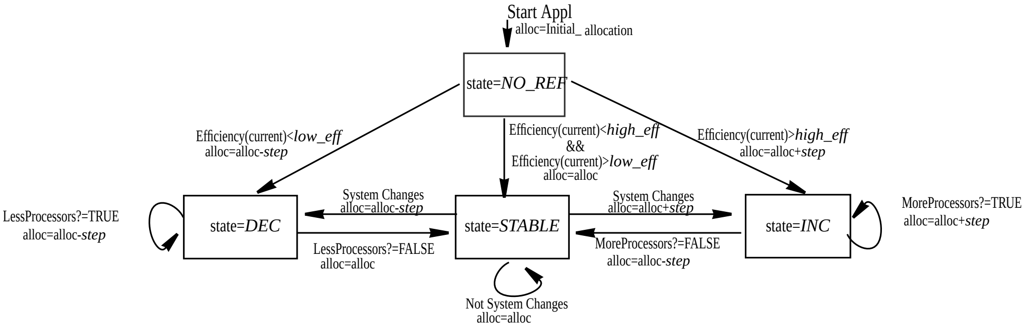

Figure 4: *PDPA*: Application state diagram

we are able to estimate the complete execution time of the application by using the calculated speedup and the number of iterations that the application executes, (3) in Figure 3. This estimation is calculated by adding the consumed execution time until the moment with the estimation of the remaining execution time. The remaining execution time is calculated as a function of the number of iterations not yet executed and the speedup that the application is achieving on each iteration.

To calculate the speedup and the execution time, the *SelfAnalyzer* needs to detect the following instrumentation points in the code: the starting of the application, the iterative structure, and the start and end of each parallel loop. In the current implementation, the invocation of the *SelfAnalyzer* at these points can be done in two different ways: (1) if the source code is available, the application can be re-compiled and the *SelfAnalyzer* calls can be inserted by the compiler. (2) If the source code is not available, both the iterative structure and the parallel loops are dynamically detected.

When the source code is not available, we detect the instrumentation points using dynamic interposition [Serra2000]. Calls to parallel loops are identified by the address of the function that encapsulates the loop. This sequence of values (addresses) is passed to another mechanism that dynamically detects periodic patterns. It receives as input a dynamic sequence of values and it is able to determine whether they follow a periodic pattern. Once we detect the iterative parallel region, the performance analysis is started.

In this case the number of times that the iterative structure executes is not available. In that case, the *SelfAnalyzer* is not able to estimate the execution time of the application and it assumes that the most useful characteristic to the scheduler is the execution time of one outer iteration.

As far as the status of the performance calculation is concerned, applications can be internally in two different states: either *Performance Not yet Calculated* (*PNC*), or *Performance Calculated* (*PC*). The application is in the *PNC* state when the speedup with the current number of assigned processors has not been yet calculated, and in the *PC* when the speedup has been calculated. At the start of the application and each time the processor allocation is changed, the application is in the *PNC* state. If the processor allocation is modified when the application is in the *PNC* state, the current calculations (speedup and execution time) are discarded, and a new calculation with the current number of processors is started.

3.2 The Performance-Driven Processor Allocation: *PDPA*

The *PDPA* allocates processors among the applications considering issues such as the number of processors used in the system, the speedup achieved by each application, and the estimation of the execution time of the whole application. The goal of the *PDPA* is to minimize the response time, while guaranteeing that the allocated processors are achieving a good efficiency.

The *PDPA* considers each application to be in one of the states shown in Figure 4. These states correspond with trends of the performance of the application. These states and the transitions among them are determined both by the performance achieved by the application and by some policy parameters. The *PDPA* parameters are the target efficiency (*high_eff*), the minimum efficiency considered acceptable (*low_eff*), and the number of processors that will increment/decrement the application allocation (*step*). In Section 3.2.2 we will present the solution adopted in the current approach to define these parameters.

3.2.1 Application state diagram

The *PDPA* can assign four different states to applications: *NO_REF(*initial state*)*, *DEC*, *INC*, and *STABLE* (see Figure 4). Each quantum the *PDPA* processes the performance information provided by the applications, compared with the performance achieved in the previous quantum, and with the policy parameters, and decides the application state for next quantum. The state transitions determine the processor allocation for this application in the next quantum, even if the next state is the same.

All the applications start in the *NO_REF* state. This state means that the *PDPA* has no performance knowledge about this application (at the starting point). The processor allocation associated with the starting of a new application is the same as an equipartition (approximately total_processors_machine/total_applications), if there are enough free processors, otherwise it assigns the available free processors. Once the *PDPA* is informed about the achieved speedup with the previous allocation, it compares the efficiency[1] with *high_eff* and *low_eff*. If the efficiency is greater than *high_eff*, the *PDPA* considers that the application performs well and sets the next state as *INC*. If it is lower than *low_eff*, the *PDPA* considers that the application performs poorly and sets the next state as *DEC*. Finally, the *PDPA* may consider that the application has an acceptable performance that does not justify a change and the PDPA sets the next state as *STABLE*.

If the next state is *INC*, the application will receive in the next quantum the current number of allocated processor plus *step*. If the next state is *DEC* the application will receive in the next quantum the current number of allocated processor minus *step*. If the next state is *STABLE* the processor allocation will be maintained.

The *INC* state means that the application has performed well until the current *quantum*. In this state the *PDPA* uses both the speedup and the estimation of the execution time to decide the next state. The MoreProcessors() algorithm presented in Figure 5 is executed to determine the next state. MoreProcessors() returning TRUE means that the additional processors associated to the transition to this state has provided a “real benefit” to this application. In that case the next state is set to *INC*. MoreProcessors() returning FALSE means that the additional processors were not useful to the applications. In that case the next state is set to *STABLE*. If the next state is *INC*, the application will receive *step* additional processors in the next quantum. If the next state is *STABLE*, the application will loose the *step* additional processors received in the last transition.

The *DEC* state means that the application has performed badly until the current *quantum*. The LessProcessors() algorithm presented in Figure 5 is executed to determine the next state. LessProcessors() returning TRUE means that the application has not yet achieved an acceptable performance. In that case the next state will be *DEC*. LessProcessors() returning FALSE means that the performance is currently acceptable and the next state must be *STABLE*. If the next state is *DEC*, the application will loose *step* more processors in the next *quantum*. If the next state is *STABLE* the application will retain the current allocation.

The *STABLE* state means that the application has the maximum number of processors that the *PDPA* considers acceptable. Typically, once an application becomes *STABLE* it remains *STABLE* until it finishes. The allocation in this state is maintained. Only if the policy parameters are defined dynamically might the *PDPA* change the state of an application from *STABLE* to either *INC* or *DEC*. If *low_eff* has been increased and the efficiency achieved with the current allocation is not acceptable, the next state will be *DEC* and the application will loose *step* processors. In a symmetric way, if *high_eff* has

```
MoreProcessors()
{
    RelativeSpeedup=ExTime(LastAllocation)/ExTime(current)
    IncrementProcessors=current/LastAllocation
    if (    Efficiency(current)>=high_eff) &&
            Speedup(current)>Speedup(LastAllocation) &&
            RelativeSpeedup>=(IncrementProcessors*high_eff)) return TRUE
    else    return FALSE
}
LessProcessors()
{
    if (Efficiency(current)<low_eff) return TRUE
    else    return FALSE
}
```

Figure 5: Algorithms to determine if the application achieves a good or bad performance

[1] Calculated as the ratio between the speedup with *P* processors and *P*.

been decreased the next state will be *INC* and the application will receive *step* additional processors.

3.2.2 *PDPA* parameters

As we have commented before, there are three parameters which determine the “aggressiveness” of the *PDPA*. These parameters can be either statically or dynamically defined. Statically defined, for instance by the system administrator, or dynamically defined, for instance as a function of the number of running applications.

In the current *PDPA* implementation *high_eff* and *low_eff* are dynamically defined and *step* is statically defined. The *PDPA* calculates the values of *high_eff* and *low_eff* at the start of each quantum, before processing the applications. The value of *high_eff* is calculated as a function of the ratio between the total number of processors allocated in the last *quantum* and the number of processors in the system. We have adopted this solution because this ratio is a good hint about the level of scalability that the *PDPA* must require of parallel applications to allocate them more processors. The higher this ratio is, the higher the *high_eff* value will be. Experimentally, the *high_eff* values ranges from 1.0 (ratio>0.9) to 0.7 (ratio<0.75). The value of *low_eff* is defined as a function of *high_eff*. In the current implementation it has been set to the value of *high_eff* minus 0.2.

Step is a parameter that sets the increments or decrements in the allocation of an application. This parameter is used to limit the number of re-allocations that are suffered by the applications. Setting *step* to a small value we achieve more accuracy in the number of allocated processors but the overhead introduced by the re-allocations can be significant. In the current implementation, this parameter has been tuned empirically and set to four processors.

3.2.3 Implementation issues

The *PDPA* checks the internal status of the applications and maintains the processor allocation to those applications that are in the *PNC* state. Transitions in the state diagram are only allowed either when all the applications are in the *PC* state or if there are unallocated processors. The aim of this decision is to maintain the allocation of those applications that are calculating their speedup. If we modify the speedup of an application in *PNC* state as a consequence of the processing of another application, it could result in inaccurate allocations.

To those applications that are in *PC* state, the *PDPA* allocates a minimum of one processor. This decision has been taken considering that the efficiency of an application with one processor is 1.0. This assumption is also done in scheduling policies such as the equipartition and the equal_eff. Moreover, it simplifies the *SelfAnalyzer* and the *PDPA* implementation.

Applications in *PC* state are sorted by speedup. This arrangement is done to give a certain priority to those applications that perform better, and assuring that these applications will receive processors. Finally, the *PDPA* maintains the history of the applications states, and does not allow that applications change from *STABLE* to either *DEC* or *INC* more than three times. The number of transitions is limited to avoid an excessive number of reallocations that will generate a loss of performance. It has been tuned empirically considering the particular characteristics of the workloads used. Further research with different workloads and applications will allow us to tune this parameter.

3.2.4 Interface

Table 1 shows the main primitives of the interface between the parallel library and the scheduler, and between the *SelfAnalyzer* and the scheduler. The first four rows are used by the parallel library to interact with the scheduler: requesting for cpus, checking the number

Table 1: Interface

Function	Description
int cpus_request(int P)	Request for P cpus to the scheduler
int cpus_current()	Returns the number of cpus allocated to the application
int cpus_preempted_work()	Returns the number of preempted threads
work_t get_preempted_work()	Returns the pointer to the first preempted thread
int cpus_speedup(int P, double speedup)	Sets the speedup achieved when P cpus are allocated to the application
int cpus_predicted_time(int P,double time)	Sets the execution time estimated when P cpus are allocated to the application

of allocated cpus, checking whether there are preempted threads and recovering them. These are the main functions to implement the dynamic processor allocation mechanism. The last two primitives are used by the *self-Analyzer* to inform the scheduler about the calculated speedup and the estimation of the execution time of the application.

3.3 Queueing system coordination: Dynamic multiprogramming level

As we have commented before, the multiprogramming level defines the number of applications running concurrently in the system. Non-clairvoyant scheduling policies typically allocate as many processors as possible to the running applications, since they are not able to determine how they will perform. They assign the minimum between the total requested number of processors and the number of processors of the machine.

But, even when the total requested number of processors is greater than the total number of processors in the machine, the *PDPA* may decide to leave some processors unallocated. In that case, the logical approach is to allow the queueing system to start a queued application. We propose to check after each re-allocation the scenario conditions and to decide whether a new application can be started. The conditions that must be met are the following:

- Are there free processors?
- Are all the running applications in the states *STABLE*, or *DEC*?
- Even if there are some application in the *INC* phase, does the number of unused processors reach a certain percentage? (currently defined by the administrator in a 20%)

These conditions are checked in the *NewAppl()* function call implemented by the scheduler and consulted by the queueing system.

4 Execution environment and implementation

The work done in this paper has been developed using the NANOS execution environment: The NanosCompiler, NthLib, and the CpuManager (the medium-term scheduler).

Applications are parallelized through OpenMP directives. They are compiled with the NanosCompiler [NANOS99], which generates code to NthLib [Martorell95][Martorell96]. NthLib constructs the structure of parallelism specified in the OpenMP directives and it is able to adapt the structure of the application to the number of available processors. Moreover, it interacts with the CpuManager through a kernel interface in the following way: NthLib informs the scheduler about the number of requested processors and the scheduler informs NthLib about the number of processors available to this application.

The CpuManager [CorbalanML99] is a user-level processor scheduler. It implements the *PDPA* scheduling policy. It follows the approach proposed in [Tucker89], that assumes that applications perform better when the number of running threads is the same as the number of processors.

For the following experiments, the CpuManager implements the queueing system. Then, in this particular implementation it communicates with the *PDPA* by calling it directly. The queueing system launches a new application each time a running application finishes, and every quantum it asks to the *PDPA* whether a new application can be started.

5 Evaluation

In order to evaluate the practicality and the benefits of the *PDPA* we have executed several parallel workloads under different scenarios:

Equip: Applications are compiled with the NanosCompiler and linked with NthLib. The CpuManager is executed and it applies the equipartition policy proposed in [McCann93]. Equipartition is a space sharing policy that, to the extent possible, maintains an equal allocation of processors to all jobs. The initial allocation is set to zero. Then, the allocation number of each job is increased by one in turn, and any job whose allocation has reached the number of requested[1] processors drops out. This process continues until either there are no remaining jobs or until all *P* processors have been allocated. The only information provided by the application is its current processor requirement**s**.

PDPA: Applications are compiled with the NanosCompiler and linked with NthLib. The CpuManager applies the *PDPA* scheduling policy. Three different variations have been executed to demonstrate the usefulness of the different components of our approach. (1) *PDPA*, as proposed in Section 3. (2) *PDPA*(S), the *PDPA* only considers the speedup. The benefit in the execution time provided by the extra processor allocation is not considered. (3) *PDPA*(idleness), the speedup is calculated as a

[1] Specified as a command line parameter of the application or setting an environment variable

function of efficiency. In this case, we have tried to implement the approach proposed in [NguyenZV96], which calculates the efficiency measuring the sources of overhead: idleness, processor stall time, and system overhead. In our applications, we found the system overhead to be negligible, and in current architectures, like the Origin2000, the hardware does not provide the performance counters to calculate the processor stall time. Due to the difficulties of implementing their complete approach, we have implemented a similar approach only considering the idleness as source of overhead.

Equal_eff: Applications are compiled with the NanosCompiler and linked with the NthLib. The CpuManager applies the equal_eff proposed in [NguyenZV96]. The goal of the equal_eff is to maximize the system efficiency. It uses the dynamically calculated efficiency of the applications, obtained through the *SelfAnalyzer*, to extrapolate [Dowdy88] the complete efficiency curve. Once extrapolated, the equal_eff works in the following way: it initially assigns a single processor to each application, and then it assigns the remaining processors one by one to the application with the currently highest (extrapolated) efficiency.

SGI-MP: Applications are compiled with the MIPSpro F77 compiler and linked with the MP-library. The commercial IRIX scheduling policy has been used. In this case, the NANOS execution environment is not involved at all. The queueing system has been used to control the multiprogramming level. In this scenario, the environment variables that define the application adaptability have been set to the following values[1]: MP_BLOCKTIME=200000 and OMP_DYNAMIC=TRUE.

5.1 Architecture, applications and workloads

All the workloads have been executed in an Origin2000 [Laudon97][SGI98] with 64 processors. Each processor is a MIPS R10000 [Yeager96] at 250 MHZ, with two separated instruction and data L1 cache (32 Kbytes), and a secondary unified instruction/data cache (4 Mbytes).

To evaluate our proposal we have selected four different applications: swim, hydro2d, apsi, and BT (class=A). The swim, hydro2d and apsi are applications from the SPECFp95, the BT is from the NASPB [Jin99]. Each one of them has different behavior considering the speedup. Table 2 presents the characteristics of these applications, from higher to lower speedup. Swim achieves a super-linear speedup, BT has a moderate-high speedup, hydro2d has low speedup and apsi has very bad speedup. In all the applications, except in apsi, the maximum speedup is achieved with 32 processors. The complete performance analysis of these applications and their speedup curves can be found in [Corbalan99].

Compilation of benchmarks from the SPECFp has been done using the following command line options for the native MIPSpro f77 compiler: -64 -mips4 -r10000 -Ofast=ip27 -LNO:prefetch_ahead=1. Compilation of the BT has been done using the Makefile provided with the NASPB distribution.

Table 3 describes the four different workloads designed to evaluate the performance of the *PDPA*. The column instances is the number of times that the application is executed and the request column is the number of requested processors.

Workload 1 is designed to evaluate the performance of the *PDPA* when applications perform well, and the allocation of the equipartition policy directly achieves a good performance. Workload 2 has been designed to evaluate the *PDPA* performance when some of the applications perform well and some perform badly. Workload 3 evaluates the performance when applications have a medium and bad speedup and, finally, workload 4 evaluates the *PDPA* when all the applications have very bad performance. Since we are not assuming *a priori* knowledge of the applications, we have set the requested number of processors to 32 in all the applications.

[1] These values have been tuned empirically to perform well under all the applications used in this work

Table 2: Parallel applications

Characteristic/Application(input)	swim(ref)	BT.A	hydro2d(train)	apsi(ref)
Exec.Time. in Sequential	212.2 sec.	1066.21 sec.	223.7 sec.	99 sec.
Speedup with 8/16/32 processors.	21.6/36.5/**44.2**	6.1/12.4/**20.85**	4.6/5.4/**6.3**	0.93/0.93/0.92

Table 3: Workload description

	swim		BT		hydro2d		apsi	
	instances	request	instances	request	instances	request	instances	request
w1	6	32	6	32				
w2			6	32			6	32
w3					6	32	6	32
w4							12	32

The multiprogramming level has been set to four in all the executions. The queueing system applies a *First Come First Served* policy, and we assume that all the applications have been queued at the same time[1].

The dynamic page migration mechanism of IRIX has been activated and we have checked that results are slightly better than without this mechanism.

5.2 Results

Figure 6 presents the average execution time per application in the different scenarios for the four workloads. We also show the total execution time of the workloads under the different scheduling policies. Results from workload 1 show that the *PDPA*-based scheduling policies (*PDPA* and *PDPA*(S)) perform well, compared with equipartition. The *PDPA*(idleness) does not perform well, demonstrating the importance of an accurate estimation of the performance. In this workload, the equal_eff performs well since the applications can efficiently use a large number of processors. We can also appreciate the importance of considering the benefit provided by the additional processors to the applications. If we observe the average execution time of swim, we see how the *PDPA* outperforms the *PDPA*(S). The reason is that the *PDPA*(S) allocates more processors to some instances of swim, allocating less processors to the rest of running applications. With the *PDPA*(S) the standard deviation in the execution time of the different instances is greater than in *PDPA*. The execution time range is (6.5,14.6) in *PDPA*(S) and (6.5,8.5) in *PDPA*. The importance of considering the benefit provided by the additional processors is more significant when the load of the system is high. In that case, without considering this parameter the processor allocation can become unfair. In the rest of workloads the difference between *PDPA* and *PDPA*(S) is less significant, since the load of the system is low.

In the workload 2, the *PDPA*-based scheduling policies outperform the rest of scheduling policies. In this case, the workload execution time has been significantly

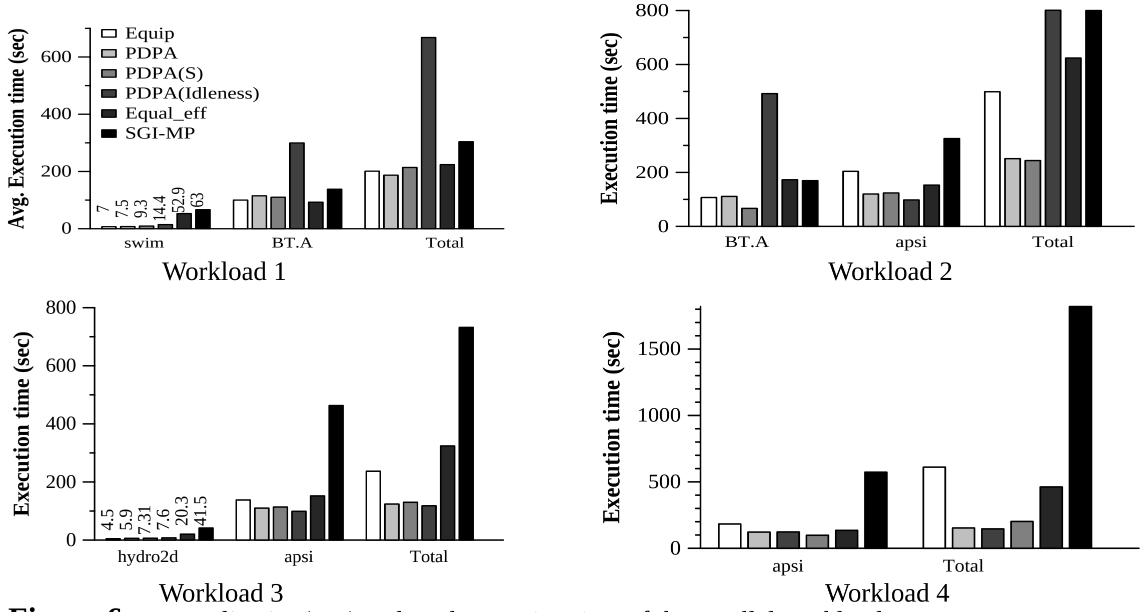

Figure 6: Per application(avg) and total execution time of the parallel workloads.

[1] Instances from different applications have been merged in the queue

reduced because of the communication with the long-medium term scheduler. The speedup with respect to both the *PDPA*(idleness) and the SGI-MP is 3.2.

Workload 3 does not show large differences in the individual performance, although the number of processors allocated to applications by the *PDPA*-based scheduling policies is very small, allowing the long-term scheduler to start a new application, resulting in a better system utilization. This better utilization can be observed in the execution time of the workload. The *PDPA*-based scheduling policies achieve speedups from 2 (with respect to the equip.) to 6.2 (with respect to the SGI-MP).

Finally, in workload 4, the *PDPA*-based scheduling policies outperform the rest, mainly in the execution time of the workload, and also in the individual performance. Allocating a small, but sufficient, number of processors to the apsi avoids undesirable memory interferences. Considering the workload execution time, the *PDPA*-based scheduling policies achieve speedups from 2.0 with respect to the equip. to 6.76 with respect to the SGI-MP.

We want to comment on the performance achieved in the case of the SGI-MP environment. The problem is the large number of unnecessary context-switches. These context-switches generate a loss of performance because they imply the reload of the data cache, remote memory accesses, and increase the system time consumed by the application. For instance, consider one apsi execution in the workload 4 in the *PDPA* and in the SGI-MP environments. In the *PDPA* the apsi has consumed a 0.1% of the execution time in system mode (0.23sec. in system-mode and 204sec. in user-mode). In the SGI-MP case, the apsi has spent a 27% in system mode (152sec. in system mode and 562.7sec. in user-mode).

Figure 7 shows the processor allocation made by the different scheduling policies when executing the parallel workloads. Each column shows the average of processors allocated to each different application. In these graphs, we can observe how the scheduling policies that take into account the application characteristics distribute the processors accordingly with the application performance. Since there are a minimum of two instances of each application running concurrently the highest of the columns should normally not exceed thirty-two processors (in the case of workload 4 sixteen).

We can observe how *PDPA* and *PDPA*(S) distribute the processors proportionally to the application performance. *PDPA*(S) is less restrictive and it assigns more processors. On the other hand, equal_eff does not have a rule to stop the processor allocation to the applications. This is the reason why the equal_eff allocates a higher number of processors to applications that perform badly, like apsi. *PDPA*(idleness) is not able to detect the good or bad behavior of the applications. The idleness is shown as a bad hint of the real efficiency achieved by the parallel applications. We can also observe in the case of the SGI-MP, how applications have adapted their par-

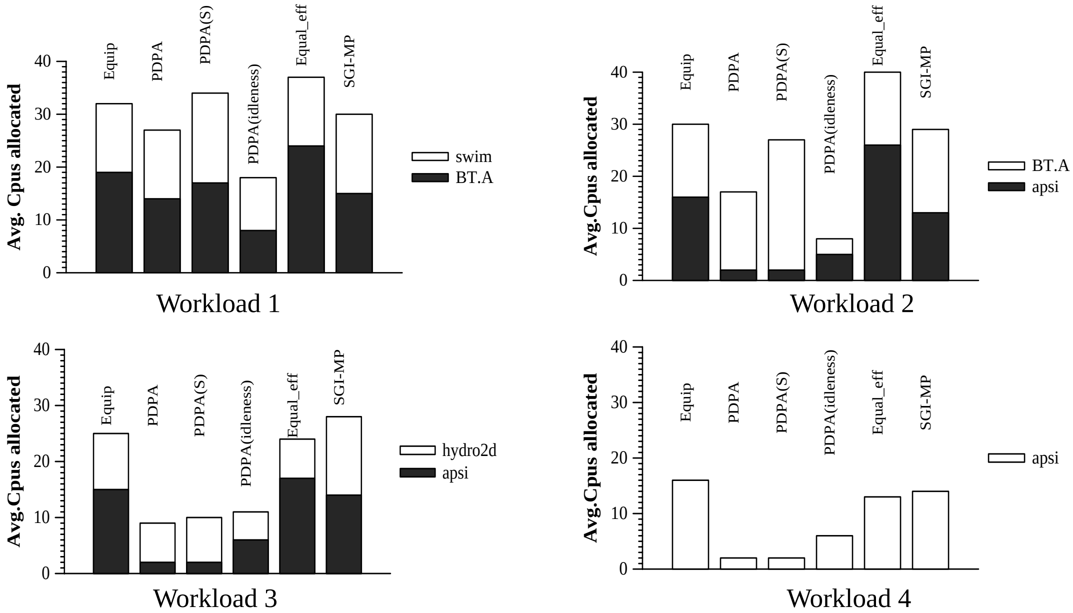

Figure 7: Processor allocation (avg) of each application under the different scheduling policies

allelism to the available processors, in a similar way to the equip.

6 Conclusions

In this work, we have presented *Performance-Driven Processor Allocation*, a new scheduling policy that uses both global system information and the application characteristics provided by the *SelfAnalyzer*, a dynamic performance analyzer. *PDPA* allocates processors to applications that will take advantage of them, avoiding unfair allocations, allocating processors to applications that do not benefit from them, or even prejudicial allocations, resulting in an increase in the execution time.

This work has been implemented and evaluated on an SGI Origin2000. We have demonstrated that it is important for the scheduler to receive accurate information about the application characteristics. Our evaluation shows that *PDPA* outperforms the considered scheduling policies.

Finally, in this work we have considered the usefulness of the interaction between the medium and the long-term scheduler. Our experience has shown that it is convenient to allow this kind of communication to improve the performance of the global system. This conclusion is valid for *PDPA* and also to any scheduling policy that allocates processors to applications based upon their performance.

7 Acknowledgments

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511, the ESPRIT Project NANOS (21907) and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1999FI 00554 UPC APTIND. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

The authors would like to thank José González and Toni Cortés for their valuable comments on a draft version of this paper.

8 References

[Amdahl67] G. M. Amdahl, “Validity of the single processor approach to achieving large-scale computing capabilities”, in Proc. AFIPS, vol. 30, pp. 483-485, 1967.

[Brecht96] T. B. Brecht, K. Guha. ”Using Parallel Program characteristics in dynamic processor allocation”, Performance Evaluation, 27&28, pp. 519-539, 1996.

[Corbalan99] J. Corbalán, J. Labarta, “Dynamic Speedup Calculation through Self-Analysis”, Technical Report number UPC-DAC-1999-43, Dep. d’Arquitectura de Computadors, UPC, 1999.

[CorbalanML99] J. Corbalán, X. Martorell, J. Labarta, “A Processor Scheduler: The CpuManager “, Technical Report UPC-DAC-1999-69 Dep. d’Arquitectura de Computadors, UPC, 1999.

[Chiang94] S.-H. Chiang, R. K. Mansharamani, M. K. Vernon. “Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies”, In Proc. of the ACM SIGMETRICS Conference, pp. 33-44, May 1994.

[Dowdy88] L. Dowdy. “On the Partitioning of Multiprocessor Systems”. Technical Report, Vanderbilt University, June 1988.

[Eager89] D. L. Eager, John Zahorjan, E. D. Lawoska. “Speedup Versus Efficiency in Parallel Systems”, IEEE Trans. on Computers, Vol. 38,(3), pp. 408-423, March 1989.

[Feitelson95] D. G. Feitelson, B. Nitzberg. “Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860”, in JSSPP Springer-Verlag, Lectures Notes in Computer Science, vol. 949, pp. 337-360, 1995.

[Feitelson97] D. G. Feitelson. “Job Scheduling in Multiprogrammed Parallel Systems”. IBM Research Report RC 19790 (87657), October 1994, rev. 2 1997.

[Hamidzadeh94] B. Hamidzadeh, D. J. Lilja, ”Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing”, Int. Conf. on Parallel Processing, vol II, pp. 39-46, 1994.

[Helmbold90] D. P. Helmbold, Ch. E. McDowell, “Modeling Speedup (n) greater than n”, IEEE Transactions Parallel and Distributed Systems 1(2) pp. 250-256, April 1990.

[Jin99] H. Jin, M. Frumkin, J. Yan. “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance”. Technical Report: NAS-99-011, 1999.

[Laudon97] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA Highly Scalable Server”. Proc. 24th Int. Symp. on Computer Architecture, pp. 241-251, 1997.

[Leutenegger90] S. T. Leutenegger and M. K. Vernon. “The Performance of Multiprogrammed Multiprocessor Scheduling Policies”, In Proc. of the ACM SIGMETRICS Conference, pp. 226-236, May 1990.

[Madhukar95] M. Madhukar, J. D. Padhye, L. W. Dowdy, ”Dynamically Partitioned Multiprocessor Systems”, Computer Science Department, Vanderbilt University, TN 37235, 1995.

[Majumdar91] S. Majumdar, D. L. Eager, R. B. Bunt, ”Characterisation of programs for scheduling in multiprogrammed parallel systems”, Performance Evaluation 13, pp. 109-130, 1991.

[Marsh91] B. D. Marsh, T. J. LeBlanc, M. L. Scott, E. P. Markatos, “First-Class User-Level Threads”. In 13th Symp. Operating Systems Principles, pp. 110-121, Oct. 1991.

[Martorell95] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, “Nano-Threads Library Design, Implementation and Evaluation”. Dept. d’Arquitectura de Computadors - UPC, Technical Report: UPC-DAC-1995-33, September 1995.

[Martorell96] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, “A Library Implementation of the Nano-Threads Programming Model”. Proc. of the Second Int. Euro-Par Conf., vol. 2, pp. 644-649, Lyon, France, August 1996.

[McCann93] C. McCann, R. Vaswani, J. Zahorjan. “A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors”. ACM Trans. on Computer Systems, 11(2), pp. 146-178, May 1993.

[NANOS99] NANOS Consortium, “Nano-Threads Compiler”, ESPRIT Project No 21907 (NANOS), Deliverable M3D1. Also available at http://www.ac.upc.es/NANOS, July 1999.

[Nguyen96] T.D. Nguyen, J. Zahorjan, R.Vaswani, “Parallel Application Characterization for multiprocessor Scheduling Policy Design”. JSSPP, vol.1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.

[NguyenZV96] T. D. Nguyen, J. Zahorjan, R. Vaswani, “Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling”, in JSSPP volume 1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.

[OpenMP2000] OpenMP Organization. “OpenMP Fortran Application Interface”, v. 2.0 http://www.openmp.org, June 2000.

[Parsons96] E. W. Parsons, K. C. Sevcik. “Benefits of speedup knowledge in memory-constrained multiprocessor scheduling”, Performance Evaluation 27&28, pp.253-272, 1996.

[Serra2000] A. Serra, N. Navarro, T. Cortes, “DITools: Application-level Support for Dynamic Extension and Flexible Composition”, in Proceedings of the USENIX Annual Technical Conference, pp. 225-238, June 2000.

[Sevcik94] K. C. Sevcik, ”Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems”. Performance Evaluation 19 (1/3), pp. 107-140, Mar 1994.

[Sevcik89] K. C. Sevcik. “Characterization of Parallelism in Applications and their Use in Scheduling”. In Proc. of the ACM SIGMETRICS Conference, pp. 171-180, May 1989.

[SGI98] Silicon Graphics Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. http://techpubs.sgi.com, Document Number 007-3430-002, 1998.

[Tucker89] A. Tucker, A. Gupta, “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors”. In 12th Symposium Operating Systems Principles. pp. 159-166, December 1989.

[Voss99] M. J. Voss, R. Eigenmann, ”Reducing Parallel Overheads Through Dynamic Serialization”, Proc. of the 13th Int. Parallel Processing Symposium, pp. 88-92, 1999.

[Weissman98] B. Weissman, “Performance Counters and State Sharing Annotations: A Unified Approach to Thread Locality”, Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 127 - 138, 1998.

[Yeager96] K. C. Yeager, “The MIPS R10000 Superscalar Microprocessor”. IEEE Micro vol. 16, 2 pp. 28-40, 1996.