

Improving Processor Allocation through Run-Time Measured Efficiency

Julita Corbalán, Jesús Labarta

Departament d'Arquitectura de Computadors (DAC), Universitat Politècnica de Catalunya(UPC)

{juli,jesus}@ac.upc.es

Abstract

In a multiprocessor architecture it is very important to allocate processors to applications in a proportional way to the performance that applications are achieving. Not considering this performance can result in an under-utilization of the multiprocessor, and also it can slowdown the execution time of parallel applications. However, the performance of parallel applications is not known before their execution. In this work, we propose to use dynamically measured application efficiency of OpenMP applications to improve the performance of two scheduling policies proposed so far, the equipartition and the equal_efficiency. The modified scheduling policies will request parallel applications to achieve a target_efficiency to receive more processors. We refer to the modified equipartition and equal_efficiency as equip++ and equal_eff++. We also propose to use a dynamic multiprogramming level to avoid the under-utilization of the machine introduced by these new scheduling policies when using a static multiprogramming level. We have evaluated this work by executing several workloads in a SGI Origin2000 with 64 processors. Results show that the combination of (target_efficiency+dynamic multiprogramming level) achieves, in the worst case, the same performance as the equipartition and the equal_efficiency, and in the best case it achieves a speedup of up to 1.3 in individual applications, and in specific workloads a speedup of up to 2.5, respect to the original algorithms.

1 Introduction

In a multiprocessor architecture it is very important to allocate processors to applications in a proportional way to the performance that applications are achieving. Not taking the performance into account can result in an under-utilization of the multiprocessor, allocating pro-

cessors to applications that do not scale well, or even it can slowdown the execution time of parallel applications.

This leads to the problem of the performance of parallel applications is not known before their execution. The traditional approach calculates the characteristics of parallel applications either running them many times with several number of processors or modeling the application based on some known parameters such as their average parallelism. This information is then passed to the scheduler as an *a priori* input.

This approach has the drawback that it can be very time consuming. Moreover, it can result in an inaccurate information since characteristics such as the speedup depend on run-time parameters i.e. the particular memory mapping, etc. In [Corbalan99] we propose a new approach to dynamically calculate the speedup of iterative parallel applications. The main idea is to calculate the speedup based on the relationship between the execution time of one loop iteration of the application, with a *baseline* (4) number of processors, and the execution time of one loop iteration with the number of allocated processors. These measurements are done at run-time.

Several scheduling policies consider applications characteristics and propose to execute, for instance, the “smallest job first”, SJF [Majumdar88], or the “largest job first”, LJF [Chen88]. Other works that also analyze different application characteristics and their use in processor scheduling are [Majumdar91], [Sevcik89] and [Chiang94]. However, they are mainly focused on deciding which application must be executed as a function of their execution time. These scheduling policies assume that information about the workload is available *a priori*, and that the number of processors requested by applications is fixed during the complete execution, and it is known at submission time. Other works determine

the number of processors that must be allocated to applications as a function of their speedups [Eager89] [Ghosal91], but having the complete speedup curve as input.

Our research is based on execution environments where no knowledge of the applications is provided *a priori*, and it is focused on determining the number of processors that must be allocated to each application. In this work, we assume that applications are submitted to a queueing system and are executed in a simple *First Come First Served* policy (FCFS) and that they are malleable [Feitelson97]. We propose to use dynamically measured application efficiency to improve the performance of two scheduling policies proposed so far, the equipartition [McCan93] and the equal_efficiency [NguyenZV96]. The main idea is to allocate more processors to those parallel applications that achieve a *target_efficiency*, allocating processors proportionally to the application's performance. The goal of this work is to show that by introducing small modifications in a scheduling policy we can consider the application efficiency, and that improves both the global system performance and the individual application performance. We refer to the modified equipartition and equal_efficiency as equip++ and equal_eff++ respectively. We also propose using a dynamic multiprogramming level to avoid the under-utilization of the machine introduced by these new scheduling policies when using a static multiprogramming level. Our approach has been implemented in a SGI Origin2000 with 64 processors. Applications from the SPECfp95 Benchmark Suite and from the NASPB have been used to evaluate the performance of our proposal. All the benchmarks used in the evaluation are parallelized with OpenMP directives [OpenMP00].

The remainder of this paper is organized as follows: Section 2 describes the execution environment. Section 3 presents the equipartition and the equal_efficiency algorithms. Section 4 explains the modifications introduced in the equipartition and the equal_efficiency to consider the efficiency. Finally, Section 5 and Section 6 presents the evaluation and the conclusions of our work.

2 Execution Environment

2.1 General overview

In our execution environment, parallel applications request a number of processors and calculate their performance through a dynamic performance analysis library, *SelfAnalyzer* [Corbalan99]. The *SelfAnalyzer* informs the scheduler about the current speedup. The

scheduler is a user-level application that periodically (at each *quantum*¹ expiration) wakes up and applies the scheduling policy distributing processors among parallel applications. Once the processor allocation has been decided, the scheduler enforces it by suspending or resuming the applications processes. Our scheduler implements the process control approach proposed in [Tucker89]. The scheduler informs applications about the number of processors assigned to each one and applications are in charge of adapting their parallelism to their current allocation. The arrival of the applications is controlled by a long-term scheduler, the queueing system. This queueing system allows the concurrent execution of as many applications as the multiprogramming level defines.

3 Scheduling policies

3.1 Equipartition

Equipartition [McCann93] is a space sharing policy that, to the extent possible, maintains an equal allocation of processors to all jobs. The initial allocation for each job 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 requested² processors drops out. This algorithm continues until either there are no remaining jobs or until all P processors have been allocated. The only information provided by the application is the maximum number of processors that it can use. Reallocations are done at job arrival and completion.

3.2 Equal_efficiency

The goal of the equal_efficiency [NguyenZV96] is to maximize the system efficiency. The idea is to allocate more processors to those applications that have better efficiency and fewer processors to applications with worse efficiency.

The equal_efficiency initially assumes that all the applications have the same efficiency, then it allocates the same number of processors to all of them during a quantum. In this quantum applications measure their efficiency and inform the scheduler. Once informed about application's efficiency, the scheduler moves processors from applications with low efficiency to applica-

-
1. One quantum is 100 ms
 2. Specified as a command line parameter of the application or setting an environment variable

tions with high efficiency, and the algorithm continues allocating processors.

In order to avoid a great number of re-allocations, which will imply a great overhead, the `equal_efficiency` extrapolates the efficiency curve [Dowdy88], from the most recently measured efficiency (calculated by the *SelfAnalyzer*). Once extrapolated, the `equal_efficiency` 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.

4 Improving the scheduling policies by requesting a *target_efficiency*

4.1 Equip++

Equipartition has been modified to ensure that running applications are achieving a *target_efficiency*. It applies the same equipartition algorithm with the modification that before allocating a new processor to an application, the `equip++` not only checks if the number of requested processors is greater than the number of allocated processors but also checks if the efficiency of `cpus_allocated+1` has been already calculated.

If such value is available, the algorithm checks if it is greater than *target_efficiency*. In that case the processor is allocated to the application, and the algorithm goes on with the next processor and application. Otherwise, the application will not receive more processors in the next quantum. If the efficiency of `cpus_allocated+1` has not been calculated, the processor is allocated to the application, and the algorithm continues allocating processors.

4.2 Equal_eff++

The `equal_efficiency` allocates processors to those applications that achieve the *best efficiency*. However, *best efficiency* is not a synonym of *good efficiency*. The `equal_efficiency` policy has been modified in the same way as the equipartition policy.

The algorithm is based on the `equal_efficiency` algorithm but before allocating a processor to an application, the `equal_eff++` checks whether the efficiency of `cpus_allocated+1` has been previously calculated. In that case, the efficiency must be greater or equal than *target_efficiency*. Otherwise, the application will not receive more processors in the next quantum. If the effi-

ciency of `cpus_allocated+1` has not been calculated, the processor is allocated and the algorithm continues.

4.3 Dynamic Multiprogramming Level

Obviously, with these algorithms some processors can remain unallocated, resulting in an under-utilization of the machine. To avoid this problem, we propose the same approach as in [Corbalan00], using a dynamic multiprogramming level. More information can be found in [CorbalanL00]. The main idea is to dynamically adjust the number of running applications to the system load by communicating the medium and long term schedulers.

5 Evaluation

5.1 Architecture, applications and workloads

Workloads have been executed in a SGI Origin2000 with 64 processors. To evaluate our proposal we have selected four different applications: swim, hydro2d, apsi, and bt. Swim, hydro2d and apsi are applications from the SPECfp95, bt is from the NASPB. Each one of them has different behavior considering the speedup. Table 1 presents the characteristics of these applications, from higher to lower speedup. The complete performance analysis of these applications and their speedup curves can be found in [Corbalan99].

Table 1: Parallel applications

Characteristics/ Application(input)	Swim	Bt	Hydro2d	Apsi
Exec.Time. in Sequential	212 sec.	1066 sec.	223 sec.	99 sec.
Speedup with 8/16/32/48 processors	21.6/36.5/ 44.2 /30.0	6.1/12.4/ 20.85 / 20.59	4.6/5.4/ 6.3 /3.6	0.93/ 0.93/ 0.92

Table 2 describes the five different workloads used in this work. The `inst.` column (instances) is the number of times that the application is executed and the `req.` column (request) is the number of processors requested by each instance. The multiprogramming level has been set to four applications in all the executions. The dynamic page migration mechanism of IRIX has been activated.

W1 is designed to compare the performance of the scheduling policies when applications perform well, and the allocation of the equipartition policy achieves directly a good performance. W2 has been designed to compare the performance when some of the applications perform well and some perform bad. W3 evaluates the performance when applications have a medium and bad

speedup. W4 compares the performance when all the applications have a very bad performance. Finally, W5 is designed to compare the performance when all the applications perform very good, (swim achieves a super-linear speedup with 16 processors), and the instrumentation analysis could be more critical in that case. We have set to 32 the requested number of processors in all the applications. Since applications of the same workload are submitted at the same time, and we apply a *FCFS* policy, applications have been submitted intercalated, for instance swim-bt-swim-...-bt.

Table 2: Workload description

	swim		bt		hydro2d		apsi	
	inst.	req	inst.	req	inst.	req	inst.	req
w1	6	32	6	32				
w2			6	32			6	32
w3					6	32	6	32
w4							12	32
w5	12	32						

5.2 Execution Environment

The work done in this paper has been developed using the NANOS execution environment [Martorell95][CorbalanML99]. Applications are parallelized through OpenMP directives. The CpuManager implements the *equip*, *equal_eff*, *equip++* and *equal_eff++*. The scenarios that we have evaluated are

Table 3: Scenarios evaluated

Scenario	Policy	SelfAnalyzer		Multi. Level	
		Static	Dyn.	sml	dml
Equip	equip			X	
Equip++(static)+dml	equip++	X			X
Equip++(static)+sml	equip++	X		X	
Equip++(dyn)+dml	equip++		X		X
Equip++(dyn)+sml	equip++		X	X	
Equal_efficiency	equal_eff	X		X	
Equal_eff++(static)+dml	equal_eff++	X			X
Equal_eff++(static)+sml	equal_eff++	X		X	

shown in Table 3. Static column of the *SelfAnalyzer* means that the source code has been modified by the compiler and re-compiled with the *SelfAnalyzer*. Dyn.. column (dynamic) means that the source code is not modified and the instrumentation done by the *SelfAnalyzer* is dynamically loaded by a dynamic interposition tool (DITools [Serra00]). We differentiate the static and dynamic scenarios to evaluate the possible overhead that could be introduced when the *SelfAnalyzer* is dynamically loaded. In the case of the *equal_eff++* the evalua-

tion is not done since we consider that results from the *equip++* are valid for the *equal_eff++*.

Moreover, we have evaluated the influence of the value of *target_efficiency* in the performance of the scheduling policies proposed in this work. To evaluate this influence, we have executed the workloads with three different values of *target_efficiency*: 50%, 70% and 90%. This evaluation has been performed only with the *equip++(dyn)*, assuming that results are valid to the rest of scenarios.

Results of this evaluation have been eliminated from this document due to the lack of space. They can be found in [CorbalanL00]. They show that a *target_efficiency* of 70% is the most appropriate, since it avoids the excess of processors generated by a *target_efficiency* of 50%, without increasing the execution time in excess, and the small number of processors generated by a *target_efficiency* of 90%. The rest of the experiments will be performed with a *target_efficiency* of 70%.

5.3 Equipartition vs. Equip++

Figure 1 shows the results of executing from W1 to W5 with *equip* and *equip++*. Numbers above columns are the average of the number of processors allocated to each application. Y axis shows the execution time, in seconds, of each application in the workload and the total execution time of the workload. We have executed

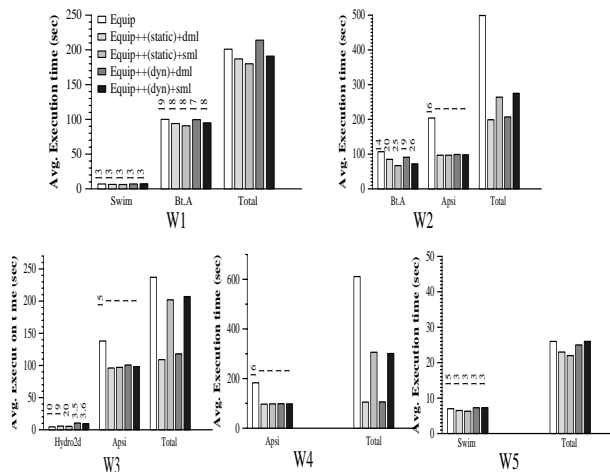


Figure 1: Equipartition vs. Equip++

with dynamic and static multiprogramming level (dml/sml) to evaluate the influence of the dml in the individual execution time of parallel applications, and to differentiate the benefit provided by either allocating the

processors proportional to the performance or the dynamic multiprogramming level.

Based on these results we can conclude that `equip++` outperforms `equip`. That means that it is very important to consider the performance that parallel applications achieve to perform the processor allocation. We can also observe that the dynamic loading of the *SelfAnalyzer* does not introduce a significant overhead. The third and fifth columns of each graph show the benefit provided by requesting a *target_efficiency* to applications. A second important conclusion is that in processor allocation policies that can leave unused processors it is very important to use a dynamic multiprogramming level to avoid the under utilization of the machine. The second and fourth columns show the additional benefit provided by the dynamic multiprogramming level. We can also comment that in workloads where applications co-exist with good and bad speedup, such as the W2, processors are re-distributed from applications with bad speedup (`apsi`) to applications with good speedup (`bt`). The processor re-distribution has two effects, first `bt` changes from 107 sec. with the `equip` ($\text{speedup}=9.96$, 14 proc.) to 67 sec. with the `equip++(static)+dml` ($\text{speedup}=15.9$, 25 proc.). And second, `apsi` is also benefit of this re-distribution, it changes from 204 sec. ($\text{speedup}=0.48$, 16 proc.) to 97 (speedup=1.02, 1 proc.). Finally, we want to comment the allocation of processors to `hydro2d` in W3. We can observe the great difference between the allocation in the (static) and (dyn) cases. This is because the execution of `hydro2d` is quite affected by the processor re-allocation introduced by the *SelfAnalyzer* (data re-distribution). `Hydro2d` has several nested parallel regions and the mechanism introduces overhead, resulting in a reduction in the achieved speedup.

5.4 Equal_efficiency vs. Equal_eff++

Figure 2 shows the results of executing the workloads with the `equal_efficiency` and `equal_eff++`. We can observe that in workloads including applications with bad speedup, the `equal_eff++` outperforms the `equal_efficiency`. This is because the `equal_eff++` limits the number of processors that are allocated to this kind of applications, and these processors can be used more efficiently by another application. In workloads where only applications with good speedup are involved the performance of the scheduling policies is more influenced by the particular characteristics of the `equal_efficiency` than by the limit in the processor allocation. The `equal_efficiency` does not work with real speedups, it works with extrapolated values. The function that extrapolates the speedup curve is highly influ-

enced by the speedup values calculated. Since the `equal_efficiency` allocates processors to those applications that have the highest efficiency, a small difference in the speedup calculated can result in a great difference in the processor allocation. This fact results in a great

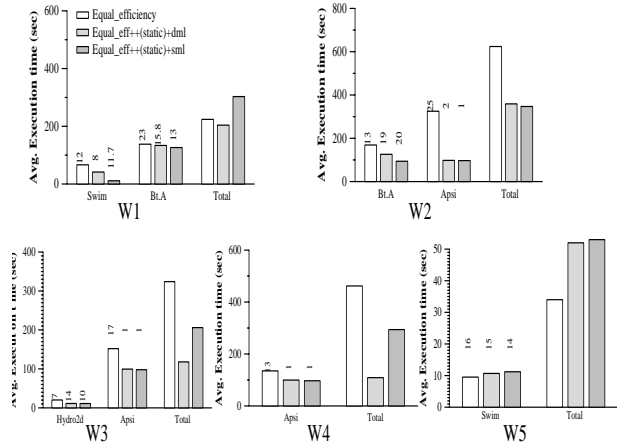


Figure 2: `Equal_efficiency` vs. `Equal_eff++`

variance in the number of processors allocated to the same application in different executions.

6 Conclusions

In this work we have adapted two scheduling policies proposed so far, the `equip` and the `equal_efficiency`, to impose a *target_efficiency* that must be accomplished by the parallel applications to receive processors. Moreover, a dynamic multiprogramming level has been used to avoid the under-utilization of the machine that introduces the *target_efficiency*. These scheduling policies have been implemented in a SGI O2000 with 64 processors and evaluated under several workloads with different speedup characteristics.

Observing the evaluation of the `equip` vs. `equip++` and `equal_eff` vs. `equal_eff++` we can extract three main conclusions: First, considering the application efficiency to perform the processor allocation is very important to improve both the system performance and the application performance. Second, the communication between the medium and the long term scheduler improves the global system performance. And finally, it is also important to base the processor scheduling decisions on real values, not on extrapolated ones, since it can result in an incorrect or unfair allocation.

7 Acknowledgments

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511, 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 Xavier Martorell and Jose Gonzalez for their valuable comments on a draft version of this paper.

8 References

- [Corbalan99] J. Corbalán, J. Labarta, “Dynamic Speedup Calculation through Self-Analysis”, Tech. Report UPC-DAC-1999-43, Dep. d’Arquitectura de Computadors, UPC, 1999.
- [CorbalanML99] J. Corbalán, X. Martorell, J. Labarta, “A Processor Scheduler: The CpuManager”, Tech. Report UPC-DAC-1999-69 Dep. d’Arquitectura de Computadors, 1999.
- [Corbalan00] J. Corbalán, X. Martorell, J. Labarta, “Performance-Driven Processor Allocation”, *Pro. of the 4th Symposium on Operating System Design & Implementation (OSDI2000)*, 2000.
- [CorbalanL00] J. Corbalán, J. Labarta, “Improving Processor Allocation through Run-Time Measured Efficiency”, Tech. Report UPC-DAC-2000-59.
- [Chen88] G-I. Chen and T-H Lai, “Scheduling independent jobs on hypercubes”, In 5th symp. Theoretical Aspects of Computer Science, pp. 273-280, Springer-Verlag, Lectures Notes in Computer Science Vol. 294, Feb 1988.
- [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”, Tech. Report, Vanderbilt University, June 1988.
- [Eager89] D. L. Eager, J. Zahorjan, E.D. Lawoska. “Speedup Versus Efficiency in Parallel Systems”, *IEEE Trans. on Comp.*, vol. 38,(3), pp. 408-423, March 1989.
- [Feitelson97] D. G. Feitelson. “Job Scheduling in Multiprogrammed Parallel Systems”. IBM Research Report RC 19790 (87657), October 1994, rev. 2 1997.
- [Ghosal91] D. Ghosal, G. Serazzi, S. K. Tripathi, “The processor working set and its use in scheduling multiprocessor systems”, *IEEE Trans. Soft. Eng.*17(5), pp. 443-453, May 1991.
- [Majumdar88] S. Majumdar, D. L. Eager, R. B. Bunt, “Scheduling in multiprogrammed parallel systems”, in *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 104-113, May 1988.
- [Majumdar91] S. Majumdar, D. L. Eager, R. B. Bunt, “Characterisation of programs for scheduling in multiprogrammed parallel systems”, *Perf. Ev.* vol. 13 pp. 109-130, 1991.
- [Martorell95] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, “Nano-Threads Library Design, Implementation and Evaluation”. Dept. d’Arquitectura de Computadors : UPC-DAC-1995-33, Sep. 1995.
- [McCann93] C. McCann, R. Vaswani, J. Zahorjan, “A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors”, *ACM Trans. on Comp. Sys.*, 11(2), pp. 146-178, May 1993.
- [NguyenZV96] T. D. Nguyen, J. Zahorjan, R. Vaswani, “Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling”. *JSPP*, vol. 1162 of *Lectures Notes in Computer Science. Springer-Verlag*, Univ. of Washington, 1996.
- [OpenMP00] OpenMP Organization. “OpenMP Fortran Application Interface”, v. 2.0 <http://www.openmp.org>, June 2000.
- [Serra00] A. Serra, N. Navarro, T. Cortes, “DITools: Application-level Support for Dynamic Extension and Flexible Composition”, in *Proc. of the USENIX Annual Tech. Conference*, pp. 225-238, June 2000.
- [Sevcik89] K. C. Sevcik, “Characterization of Parallelism in Applications and their Use in Scheduling”, in *Proc. of the ACM SIGMETRICS Conf.*, pp. 171-180, May 1989.
- [Tucker89] A. Tucker, A. Gupta, “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors”, in *12th Symp. Operating Systems Principles*. pp. 159-166, dec. 1989.