Dynamic Speedup Calculation through Self-Analysis

Julita Corbalán and Jesús Labarta
Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)
{juli,jesus}@ac.upc.es

Abstract

In an multiprocessor environment, with applications running concurrently, the scheduler is responsible for optimizing the system utilization. It distributes processors among applications according to a scheduling policy. Some policies allocate processors taking into account information such as the expected speedup. This information is usually provided by the users to the scheduler as an a priori input, and it is obtained by running the applications several times with different input sets. However, the large number of executions needed to obtain an accurate information constitute the major drawback of this approach, since they may consume a lot of time. A recent work has suggested that the efficiency of the applications can be dynamically estimated. This information can be used by the scheduler, avoiding the necessity of a priori information. The goal of our work is to present a different approach to dynamically compute the speedup achieved by parallel applications in order to provide this information to the scheduler. This approach is based on the traditional speedup equation. We will show that the dynamically calculated speedup approaches the speedup calculated as the relationship between the parallel execution with one processor and the parallel execution with P processors.

1 Introduction

In a multiprocessor environment, with parallel applications sharing the available resources, the distribution of processors is a key issue to obtain a good system utilization. The scheduler is responsible for distributing processors among applications. This distribution can be performed taking into account several parameters such as the current distribution of processors, the time that the applications are waiting to run, the number of processors requested by each application, the amount of time consumed by the applications, or the amount of work of the system. These parameters are either already known or can be directly measured by the scheduler.

However, many researchers have shown that using application characteristics such as the speedup or the average parallelism improve the performance of the scheduler [2][3][12]. In particular, Parsons and Sevcik [13] showed that, if applications executing in a system have different speedup curves, the knowledge of this behavior is useful to the scheduler, since it can assign more processors to those applications that will take advantage of them.

The speedup is the relationship between the sequential and the parallel execution time. Traditional approaches statically computed the speedup by executing the sequential and the parallel version several times with different input sets, in order to provide this information to the scheduler. This was because they assumed that the speedup could not be dynamically calculated since, in order to obtain the parallel and sequential execution time, applications must run until completion.

However, the speedup obtained by a parallel application depends on several factors such as the degree of parallelization achieved by the application, its input data, the architecture, and the placement of the processors. The degree of parallelization of the application is a constant factor. On the other hand, parameters such as the input set or the placement of the processors (which is very critical in a NUMA machine like the O2000) may change from one execution to another. Consequently, the speedup can change depending on these variable parameters and cannot be calculated without executing the application.

Since the knowledge of the speedup is useful to the scheduler, but different executions of one application may have different speedup curves, in this work we propose a new approach to dynamically measure the speedup. We call our implementation *self-analyzer*, and it calculates at run-time the speedup obtained in the parallel regions of the applications. The speedup computation involves the relationship between two measures: the reference (or baseline) and the parallel execution time. We propose a mechanism to obtain the reference time without having to run the sequential version, but it is obtained during the execution of the parallel application. In this way, it will not be necessary to perform several time-consuming executions.

We will show that the speedup calculated[1] by our approach corresponds with the speedup calculated with the traditional approach (the relationship between the parallel execution with one processor and the parallel execution with P processors). We have also analyzed the overhead introduced by our mechanism and we have found it negligible.

The remainder of this paper is organized as follows. Section 2 presents the motivations of this work.Section 3 describes our approach to dynamically compute the speedup in parallel applications. Section 4 presents the evaluation of our proposal, including both a validation of the dynamically calculated speedup and the analysis of the introduced overhead. Finally, in Section 5 we summarize the main conclusions of this work.

1. We only focus our attention on the parallel regions of applications.

2 Motivation and Related Work

In this work, we present a new approach to dynamically compute the speedup. Nguyen *et al* [10] propose that the efficiency obtained by the parallel applications can be dynamically calculated. Then, the speedup can also be dynamically computed as a function of the efficiency. Their approach is called *self-tuning*. This enables the scheduler to use such information to make the decisions. In order to calculate the efficiency, they measure the different sources of overheads that cause loss of efficiency and subtract them from 1.

Figure 1 shows the formulation proposed by Nguyen in [10] to calculate the efficiency and the relationship between efficiency and speedup[4]. The sources of overhead, showed in Figure 1, are the *system overhead*, the *idleness* and *communication (processor stall time)*[2].These components were obtained by using the hardware counters provided by the architecture, and by instrumenting the parallel library.

However, one of the major limitations of this method to calculate the efficiency is that it is closely dependent on the architecture. In some current multiprocessors systems, such as the Origin 2000[15][6], the *stall time* cannot be measured, since the corresponding hardware counter is not provided by the architecture. On the other hand, there are some features of the system that do have influence on the speedup, and that are not totally taken into account in the previous equation. One of the most important, which has a significant influence on the performance in the O2000, is the relationship between the number of cache misses in the parallel and the sequential execution. The *stall time* only computes the execution time lost due to the access to remote data. However, it does not take into account the execution time consumed in the access to the local data, which also has a big impact in the total execution time. This execution time does not remain constant when the number of processors changes. Nonetheless, it is assumed constant in the equation of Figure 1.

An interesting effect that may appear when running an application in parallel is the *superlineal speedup* (i.e. when the speedup achieved with p processors is greater than p)[5]. It may occur when the accumulated misses of all processors that are running a parallel application is lower than the number of misses of the sequential application. This effect cannot be detected using the previous equation since, in the one hand, it does not consider the cache misses neither in the sequential execution nor in the parallel one, and, on the other hand, the efficiency, as it is measured, ranges from 0 to 1, and thus, the speedup will never be greater than p.

2. In their architecture, communication appears when processors have to access to remote cache, causing a processor stall.

$$Efficiency(p) = 1 - \left(\frac{WT(p) - UT(p)}{WT(p)}\right) - \frac{IT(p)}{WT(p)} - \frac{PST(p)}{WT(p)} \longrightarrow Speedup(p) = Efficiency(p) \times p$$

WT(p)= elapsed execution time with p processors
IT(p)= accumulated idle time
UT(p)= accumulated user-mode execution time
PST(p)= accumulated processor stall time

Figure 1: Efficiency and speedup equations. The efficiency is calculated as a function of the sources of overhead: *system overhead*, the *idleness* and *communication*. The speedup is calculated as a function of the efficiency.

To conclude, the *Self-Tunning* approach has two major drawbacks. First, it cannot be implemented in many current arquitectures such as the O2000. And second, it does not take into account that there are arquitectures components, such as the cache, that may improve the performance of the parallel applications. Therefore, a better approach to compute the speedup must consider the execution time rather than some particular components of it. And this is what we present in this work.

3 Self-Analyzer

Our *self-analyzer* can correctly work in applications with a particular internal structure. This Section describes first the requirements that have to be accomplished by the applications. Then, we present the methodology that allows a dynamic computation of the speedup, and finally, we discuss some implementation issues.

3.1 Applications

A great number of scientific applications are characterized by their predictable behavior. These applications are known as *iterative parallel applications* [12]. Figure 2 shows the structure of these applications. We can observe that they are composed of a set of parallel loops inside a sequential loop. We refer to this structure as *iterative parallel region*, and, to the set of parallel loops inside the sequential loop as *parallel region*.

Since a parallel region is inside a sequential loop, it will be executed *n* times, where *n* is the number of iterations of the sequential loop. If this parallel region executes always the same code, with different data, its execution in the iteration *i+1* will be similar to its execution in the iteration *i*. This characteristic has also been exploited in the self-tuning proposal [10][12], assuming that the speedup achieved by any subset of iterations will be similar. We will consider a parallel region as the minimum section of code needed to calculate the speedup.

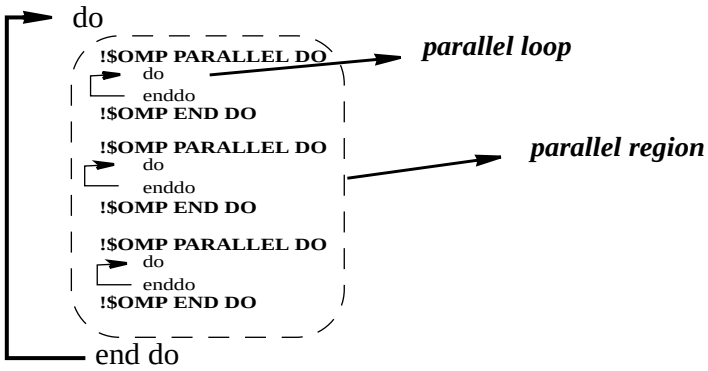

Figure 2: *Iterative parallel region*. Some functions from the *self-analyzer* are inserted in the source code to calculate the speedup.

This characteristic is shared by a large number of scientific applications. In [11] it is shown that five out of ten SPLASH [14] applications and the seven PerfectClub [1] applications are iterative. Applications that may also be iterative are those that perform computational fluid dynamics, as for instance the *tomcatv* from the SPECFP95, or those that perform weather prediction like the *swim*, from the SPECFP95. Other applications that also follow this scheme are some of the NAS Benchmarks, such as the *BT* or the *SP* and real applications such as *crash simulations* for cars.

3.2 Dynamic speedup computation

In this Section we present the methodology followed by the *self-analyzer* to calculate the speedup achieved by the parallel region of applications. Note that we propose to dynamically compute the speedup in order to provide this information to the scheduler, so that it can choose an optimal processor allocation to reduce the execution time of this part. We assume that the sequential part is intrinsic to the application and our technique cannot help to reduce its effect.

When the *self-analyzer* enters an *iterative parallel region*, it runs a few iterations of this loop with only one processor in order to calculate the reference. This reference is the average of the execution time of these iterations executed with one processor. The rest of iterations are executed with the number of processors available.

$$\text{Speedup } (p) = \frac{\text{Execution time of the } \textit{parallel region} \text{ with 1 processors}}{\text{Execution time of the } \textit{parallel region} \text{ with } p \text{ processor}}$$

Figure 3: Traditional speedup equation.

During the rest of the loop, the *self-analyzer* collects the execution time of each iteration of the parallel region. Every few iterations, the execution times of the parallel region is averaged in order to achieve a more accurate measure. This average is used to compute the speedup, which is

dynamically calculated following the equation presented in Figure 3. Due to the behavior of these applications, we assume that the dynamically computed speedup for a particular number of processors will be constant for any set of iterations. Moreover, it will be the same as the speedup obtained in the complete execution of the iterative parallel region.

The speedup is continuously recalculated in order to detect both, variations in the behavior of the application and on the number of assigned processors. Each time the *self-analyzer* detects a variation in the number of processors it discards the execution time of the current iteration. If the *self-analyzer* did not discard this iteration, the speedup would fall down due to the overhead introduced by the data movements. These data movements are generated by the re-distribution of iterations.

Another issue concerning the implementation is that the *self-analyzer* recalculates the speedup even when the number of processors does not change. Each time it has a new speedup value it does not replace the old one. Instead, it maintains a certain history by means of calculating the speedup as a function of the two values. Figure 4 presents the weight that we assign to the previously and the current calculated speedup. We assign a weight of 0.6 to the old speedup and a weight of 0.4 to the new speedup[3].

$$\mathrm{Speedup}(P) = (\mathrm{SpeedupOld}(p) \times 0.6) + (\mathrm{SpeedupNew}(p) \times 0.4)$$

Figure 4: We calculate the speedup as a function of the old and the new speedup.

3.3 Implementation issues

In order to calculate the speedup, the *self-analyzer* extends the parallel library with four function calls that perform the speedup calculation. These function calls are the following:

- *init_parallelism*, it is called before the execution of the sequential loop. It initializes data structures such as timers, or speedup data.
- *open_parallel_region*, is called at the beginning of each *parallel region*. It obtains the timestamp and manages the number of processors to use (1 or *P*).
- *close_parallel_region*, is called at the end of each *parallel region*. It obtains the timestamp, calculates the speedup and informs the scheduler.
- *end_parallelism*, is called after the execution of the sequential loop.

The code of the parallel applications has to be modified in order to introduce the calls to the *self-analyzer*. In this work, these calls have been introduced by hand, but they can be easily generated by a compiler, following the OpenMP directives.

3. We have tested other combinations and we have found that this provides the best accuracy

The *self-analyzer* needs the support of a parallel library which allows the user to select the amount of parallelism to be created on each parallel loop. In this work we use the NthLib [7] as parallel library with the support of the NANOS compiler[9]. The NANOS compiler processes the OpenMP directives and generates code to access to the NthLib. The NthLib generates work taking into account the available number of processors, and it can support dynamic variations in the number of processors. Moreover, the NthLib interacts with the scheduler, allowing the *self-analyzer* to communicate to the scheduler the speedup achieved by the parallel application.

In the next Section, we evaluate whether the speedup calculated by our proposal corresponds with the real speedup and the overhead introduced in the execution time of the parallel applications.

4 Evaluation

The goal of the *self-analyzer* is to dynamically calculate the speedup achieved by a parallel region. In order to calculate the speedup, the *self-analyzer* executes in sequential some iterations of the *iterative parallel region* with the aim of obtaining the baseline measure. In this Section we evaluate whether the dynamically computed speedup corresponds to the traditional speedup (i.e. that achieved by executing independently the parallel and the sequential version) and whether the *self-analyzer* introduces overhead in the execution time.

4.1 Execution environment

In order to evaluate the *self-analyzer,* we have executed four benchmarks from the SPECFP 95: *tomcatv, swim, apsi and turb3d*. These benchmarks have been parallelized by hand using the OpenMP directives. The calls to the *self-analyzer* have been also introduced by hand. They have been compiled with the NANOS-Compiler[9] and linked with the NthLib parallel library [7][8].

All the measures have been obtained using a Origin 2000 with 64 processors and the executions have been carried out with the machine dedicated. The kernel threads have been bounded to the processors in order to avoid great variations in the speedup due to uncontrolled thread migrations.

We also evaluate the *hydro2d* from the SPECFP95, which is an iterative parallel application, with the particular feature that all the iterations do not execute the same amount of work. These conditions mean that the *hydro2d* does not comply with the requirements of the *self-analyzer*. However, we will also analyze the behavior of our mechanism for this particular case.

4.1 Dynamic speedup calculation

In this Section we evaluate the accuracy of our dynamically computed speedup by comparing it with the real speedup.

$$\text{Efficiency(p)} = 1 - \left(\frac{\text{idletime}}{\text{usertime}}\right) \longrightarrow \text{Speedup(p)} = \text{Efficiency(p)} \times \text{P}$$

Figure 5: The only source of overhead considered is the idleness.

In Figure 6 we show the speedup calculated for the following scenarios:

- *TS*. Traditional Speedup. The parallel application has been executed with each number of processors. The speedup has been calculated as the relationship between the execution time of one and *P* processors. In addition, it represents the maximum speedup that the application can achive for each particular number of processors, since its processor allocation is constant during the complete execution.
- *SA*(N exec). *Self-Analyzer*, N executions. The speedup has been dynamically calculated by the *self-analyzer*. We have performed one execution for each number of processors.
- *SA*(1 exec). *Self-Analyzer*, 1 execution. The speedup has been dynamically computed by the *self-analyzer*. We have performed only one execution by application, and the number of available processors has been dynamically changed every certain number of iterations. The aim of this curve is to analyze whether is possible to provide the scheduler with the speedup of the application by running the first iterations with diferent number of processor. In this way, after some iterations, the scheduler will know the behavior of the rest of the application depending on the number of assigned processors.
- *ST'*. *Self-Tuning*'. a variation of the *self-tuning*, explained below.

The *ST'* is a variation of the *self-tuning*, proposed in [10]. We have tried to implement the approach to calculate the efficiency proposed in the *self-tuning* and we have observed than in our environment and with our applications, the *system overhead* is negligible. Moreover, as pointed out before, the *processor stall time* cannot be measured in our system. We have analyzed the benchmarks and we have observed that the main source of overhead is the *idle time*. We have instrumented the parallel library in order to obtain the *idle time* and the speedup is calculated as follows.

We observe that in the case of the *tomcatv*, *swim*, *apsi* and *turb3d* the speedup calculated by the *SA(N exec)* is very similar to the *TS* and, most important, the shape of the curves is the same. In the case on the *ST'* the calculated speedup does not correspond with the *TS*, since in this set of applications, the speedup is basically determined by the relationship between the number of cache misses in the sequential version and in the parallel one.

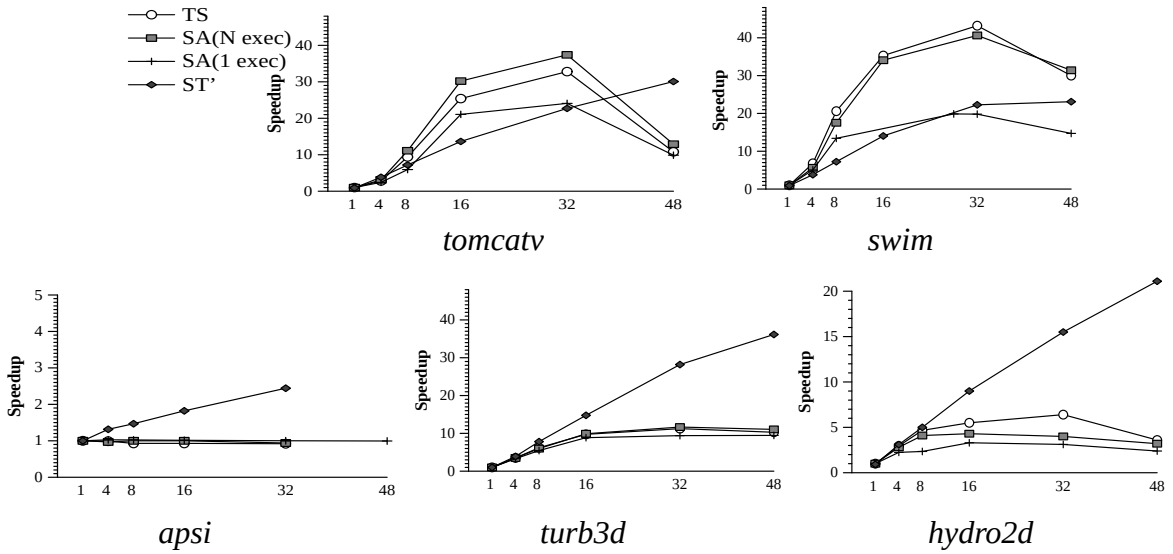

Figure 6: Speedup comparison. The y axis is the calculated speedup and the x axis is the number of processors available.

The *SA(1 exec)* is able to detect the speedup achieved by the *apsi* and *turb3d*, which achieved low and medium speedup values. In the case of the *tomcatv* and *swim* the curve of the speedup follows the same shape but the absolute values are lower than the real speedup. The reason is because a dynamic change in the number of processors causes a dynamic change in the data distribution and that implies the access to remote pages.

The performance of the *tomcatv* and *swim* applications is closely related to the data placement. They can obtain a super-lineal speedup due to the drastical reduction of the number of cache misses. Then, if processors are assigned at the beginning of the execution, data is correctly placed. However, a change in the number of processors allocated in the middle of the execution causes data reallocation and then, continuos accesses to remote pages. As these applications are strongly affected by memory allocation, this fact causes a decrement on the speedup achieved by the *SA(1 exec)*. Demonstrating the influence on the execution time of the data placement in a NUMA machine is out of the scope of this work. But, in order to give an insight about such influence, we have carried out a simple experiment. We have executed 4 out of 900 iterations of the *swim* with 4 processors and the rest with 32 processors, (4*4, 896*32). We have repeated the experiment but changing the order (896*32, 4*4). The number of iterations is executed with 4 and 32 processors is the same in both cases. However, we found a difference in the execution time of about the 20%.

Therefore, if we want to provide the scheduler with the complete speedup curve after some iterations of the sequential loop, the results are influenced by the very frequent processor re-allocation, which implies data re-allocation. This causes an increase of accesses to remote data and then a decrease in the absolute speedup. Nevertheless, we can observe that the speedup curve of

SA(1 exec) is parallel to the TS curve, which at least indicates that the *SA (1 exec)* curve follows the same trend that the real curve. One may consider this information enough to give an insigth to the scheduler about the behavior of the application. On the other hand, if we look at the *ST*' curves, neither the absolute numbers, nor the shape of the curves look like the correct curve.

In the case of the *hydro2d* the speedup calculated in both the *SA(N exec)* and *SA(1 exec)* does not corresponds with the *TS*, but with some number of processors the speedup is very close it.

To conclude, results from this Section have shown that our approach to dynamically compute the speedup approximates very precisely the traditional speedup curves, in terms of the shape of the curves for the *SA(1 exec)* and absolute numbers for the more logical comparison, *SA(N exec)*. Note that, the scheduler, in order to assign an optimal number of processors, does not need a 100% precise expected speedup but a precise information about the trends, and this is achieved by our approach.

4.2 Overhead introduced by the *self-analyzer*

The *self-analyzer* executes some iterations of the *iterative parallel region* in sequential in order to obtain the baseline measure. This sequential execution of a parallel code may introduce overhead. In this Section we evaluate the overhead introduced by the *self-analyzer* in the total execution time. All the measures have been obtained in the same conditions as in previous Section.

Figure 7 presents the complete execution time of the five benchmarks executed ranging the number of processors from 1 to 48. The curve with circle marks corresponds to the execution of the application without the *self-analyzer*. The curve with box marks corresponds to the execution time with the *self-analyzer*, which implies the execution of some of the parallel code in sequential, and collecting some measures during the complete execution of the application.

We can observe that in the *tomcatv, swim, apsi* and *turb3d* the *self-analyzer* does not introduces a significant overhead. The reason stems from the behavior of the sequential loops, which have a large number of iterations (400). In addition, these iterations are quite short. Consequently, the impact of the sequential execution of few iterations is minimum. The worst case is the *turb3d* where the overhead reaches the 10% in the execution with 48 processors.
The overhead introduced by the *self-tuning*' is not significant since we measure the idle time when threads are not doing useful work for the applications. But, even though this version of the *self-tuning* does not introduce overhead, we have observed that the access to information such as the number of cache misses may introduce a lot of overhead. This type of information [4] is collected through system calls and thus it is very costly to obtain.

4. The secondary cache misses is the more related information to the processor stall time

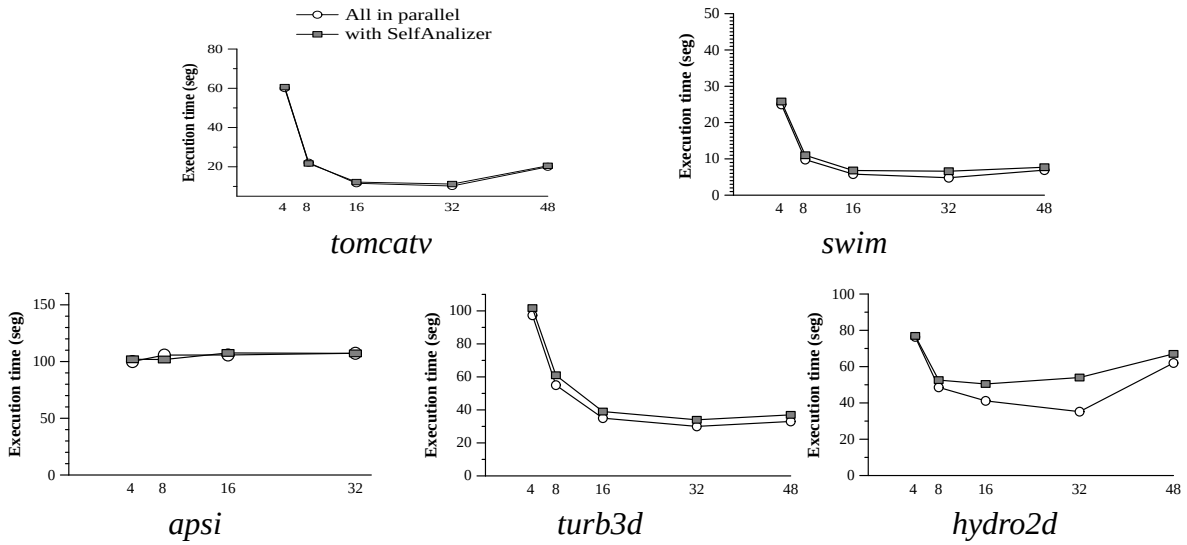

Figure 7: Overhead introduced when executing with *self-analyzer*.

The case of the *hydro2d* is more critical since it has very few iterations and the first iterations process more data than the rest. Then, the sequential execution of these iterations have a great impact on the total execution time.These kind of applications are critical for us: first, we execute the first iterations in sequential, and second the execution time of these iterations is greater than the rest.

In the next Section we present some modifications to the initial approach in order to reduce the overhead in the execution time, for the applications which experiences this particular behavior.

4.3 Enhancements to reduce the overhead

The overhead introduced by the *self-analyzer* depends on the applications. If either the application executes few iterations or the iterations are expensive in terms of execution time, the overhead introduced becomes significant.

It seems that the problem lies in the time that the application spends executing with one processor. The solution that we propose is increasing the number of processors that we use as reference from one to either two or four. This is a partial solution, since if we take as reference the execution time with four processors, we cannot obtain an absolute value of the speedup, as it happens in the case of one processor. Nevertheless, we expect to obtain the same shape in the speedup curve as in the case of having the reference taken with one processor.

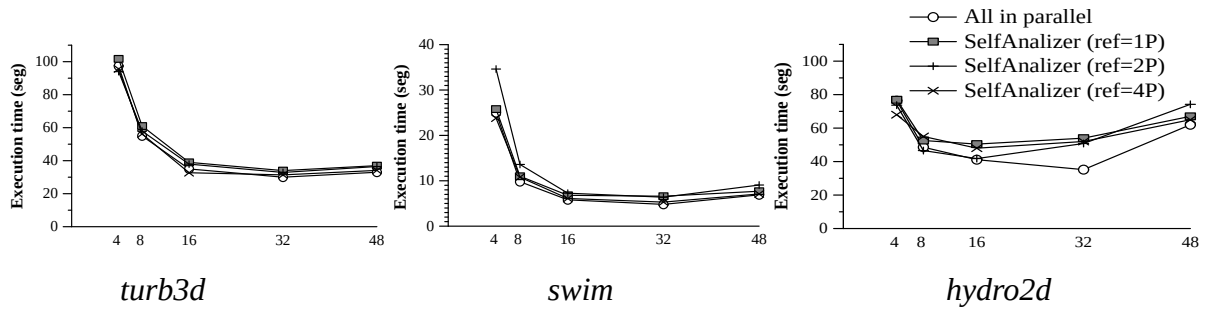

Figure 8: Execution time of the parallel applications when the baseline reference is the execution time of one, two and four processors. The x axis is the number of processors and the y axis is the execution time.

Figure 8 presents the overhead introduced by the *self-analyzer* when using as reference the execution time of one [5], two and four processors compared to the execution time of the application when using the total number of processors available.

We see that, when the number of processors that we use as reference is increased, the overhead introduced is reduced. In the case of the *turb3d* the overhead has been completely eliminated and in the case of the *hydro2d* it has been eliminated in the execution with four processors and reduced from 50% to 27% for 48 processors. We also present the execution time of the *swim* in order to present the behavior of an application that initially did not present any overhead.

It seems clear that, if we take the reference measure with four processors, the overhead should be reduced. But the point now is whether the calculated speedup is correct. Figure 9 shows the speedup calculated with the *self-analyzer* when using as reference the execution time of one, two and four processors. We compare these speedups with the traditional speedup, calculated as in Section 4.1 For the *turb3d*, we can observe that the elimination of the overhead (as shown in Figure 8) does not influence the accuracy of our approach, and the speedup curves for 2 and 4 processors as reference measure is almost the same as the *TS*. For the *hydro2d* the reduction of the overhead slightly increases the accuracy (the curves corresponding to taking the reference at 2 or 4 processors are closer to the *TS* than the curve corresponding to 1 processor).

For the *swim* application, we observe a similar behavior in the curves corresponding to having the reference measure at 2 or 4 processors than the *SA (1 exec)* approach, although in these experiments we have obtained the speedup for each particular number of processors through individual executions. The *swim* has the particular feature that it has a few number of parallel loops before the *iterative parallel region*. Since these loops are out of the control of the *self-analyzer*, they are executed with the total number of processors available, thus they establish the initial data distribution. The *self-analyzer* starts the *iterative parallel region* with a different number of processors

5. First approach of the *self-analyzer*

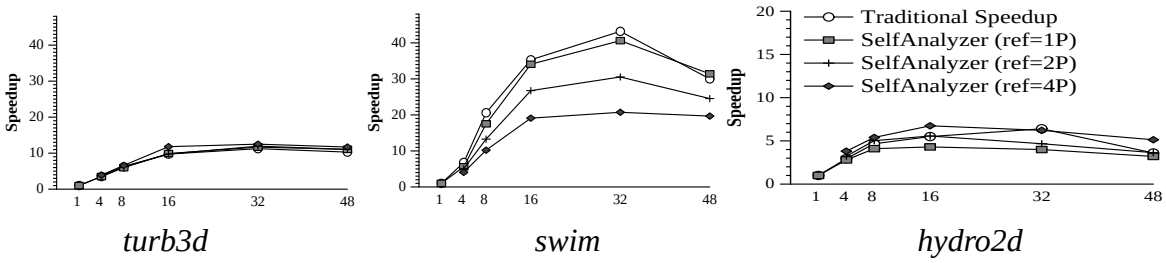

Figure 9: Speedup calculated when taking as reference the execution time of one, two and four processors. The x axis is the number of processors and the y axis is the calculated speedup.

than in the previous parallel loops, which implies accessing remote data. Since the execution time of the *swim* is very related to the data distribution, the different number of processor used as reference affects significantly the achieved speedup.

5 Conclusions

In this work we have presented our approach (*self-analyzer)* to dynamically calculate the speedup of parallel applications. This information may be useful for the processor scheduler in order to achieve a better processor distribution.

First, we have analyzed a previous proposal which estimates the speedup based on the efficiency, and the efficiency is computed taking into account a set of sources of loss of performance. We have found some drawbacks to this approach:

- It considers some parameters of the architecture that cannot be measured in current architectures
- It does not consider some architecture features that influence on the execution time, as for instance the relationship between the number of cache misses in the sequential and parallel version.

Our proposal exploits the characteristics experienced by *iterative parallel applications*, which have a set of parallel loops inside a sequential loop. Our methodology to dynamically compute the speedup works as follows: we first obtain the reference measure by executing with one processor few iterations of the outer loop. Then, the rest of iterations are executed with the number of available processors. The speedup is computed by dividing the execution time of the parallel version by the reference execution time.

The accuracy of our methodology to compute the speedup has been evaluated by comparing the dynamically computed speedup with that obtained in the traditional way. We also present a comparison with a particular implementation of the *self-tuning* proposal.

Results show that the dynamically obtained speedup is almost equal to the traditional speedup, despite of the fact that it is computed at run-time. In addition the overhead introduced is negligible for almost all applications. In order to reduce the overhead for those applications where it does have an impact, we have optimized the methodology by means of collecting the reference measure with four processor instead of one. With this optimization the overhead is almost eliminated.

Finally, one interesting conclusion we extract from this work concerns the fact that the speedup achieved by a parallel application in a NUMA machine does not only depends on the particular structure of the application and the number of processors assigned but also it strongly depends on the memory and processor distribution. Since this allocation is performed at run-time, it seems reasonable that dynamically computing the speedup is more accurate than calculate it through several previous executions with different input data, and provide this information to the scheduler as an *a priori* input.

Future work includes the extension the *self-analyzer* to applications that are not *iterative parallel* and the design and implementation of new scheduling policies that can take advantage of the *self-analyzer.*

6 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 Center of 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, and Xavier Martorell for kindly providing the NthLib.

7 References

[1] M. Berry, D. Chen, P.Koss, D.Kuck, S.Lo, Y.Pang,L.Pointer, R. Roloff, A. Sameh, E. Clementi, S.Chin, D. Schneider, G. Fox. P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum and J, Martin. “The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers”. *The International Journal of Supercomputer Applications*, 3(3):5-40,1989

[2] T.B. Bretch and K. Guha, “Using parallel program characteristics in dynamic multiprocessor allocation policies”. *IEEE Performance Evaluation* 27&28 (1996) 519-539. ftp://www.cs.yorku.ca/pub/brecht/Brecht_Guha.ps. York University.

[3] D. L. Eager, R. B. Bunt, “Characterization of programs for scheduling in multiprogrammed parallel systems”. *Performance evaluation* 13 (1991) pp. 109-130.

[4] D. L. Eager, J. Zahorjan and E. Lazowska, “Speedup Versus Efficiency in Parallel Systems”. *IEEE TRans. Computer.* 38 (3) 1989 pp 408-423

[5] D. P. Helmbold, Ch. E. McDowell, “Modeling Speedup (n) greater than n”. *IEEE Transactions Parallel & Distributed Systems* 1(2) pp. 250-256, Apr. 1990. University of California, Santa Cruz.

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

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

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

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

[10] T.D. Nguyen,J. Zahorjan, R. Vaswani, “Maximizing Speedup through Self-Tuning of Processor Allocation”. *IPPS 96,* Technical report UW-CSE-95-09-02. University of Washington

[11] T.D. Nguyen,J. Zahorjan, R.Vaswani, “Parallel Application Characterization for multiprocessor Scheduling Policy Design”. *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lectures Notes in Computer Science. Springer-Verlag* 96. University of Washington

[12] T. D. Nguyen,J. Zahorjan, R. Vaswani, “Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling”. *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lectures Notes in Computer Science. Springer-Verlag* 96. University of Washington

[13] E.W. Parsons, K.C. Sevcik, “Benefits of speedup knowledge in memory-constrained multiprocessors scheduling”, University of Toronto.*IEEE Performance Evaluation* 27&28 (1996) 253-272

[14] J.P.Singh, W.D.Weber, and A. Gupta. “SPLASH: Standford Parallel Applications for Shared Memory”. *Computer Architecture News*, 20(1):5-44, 1992

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