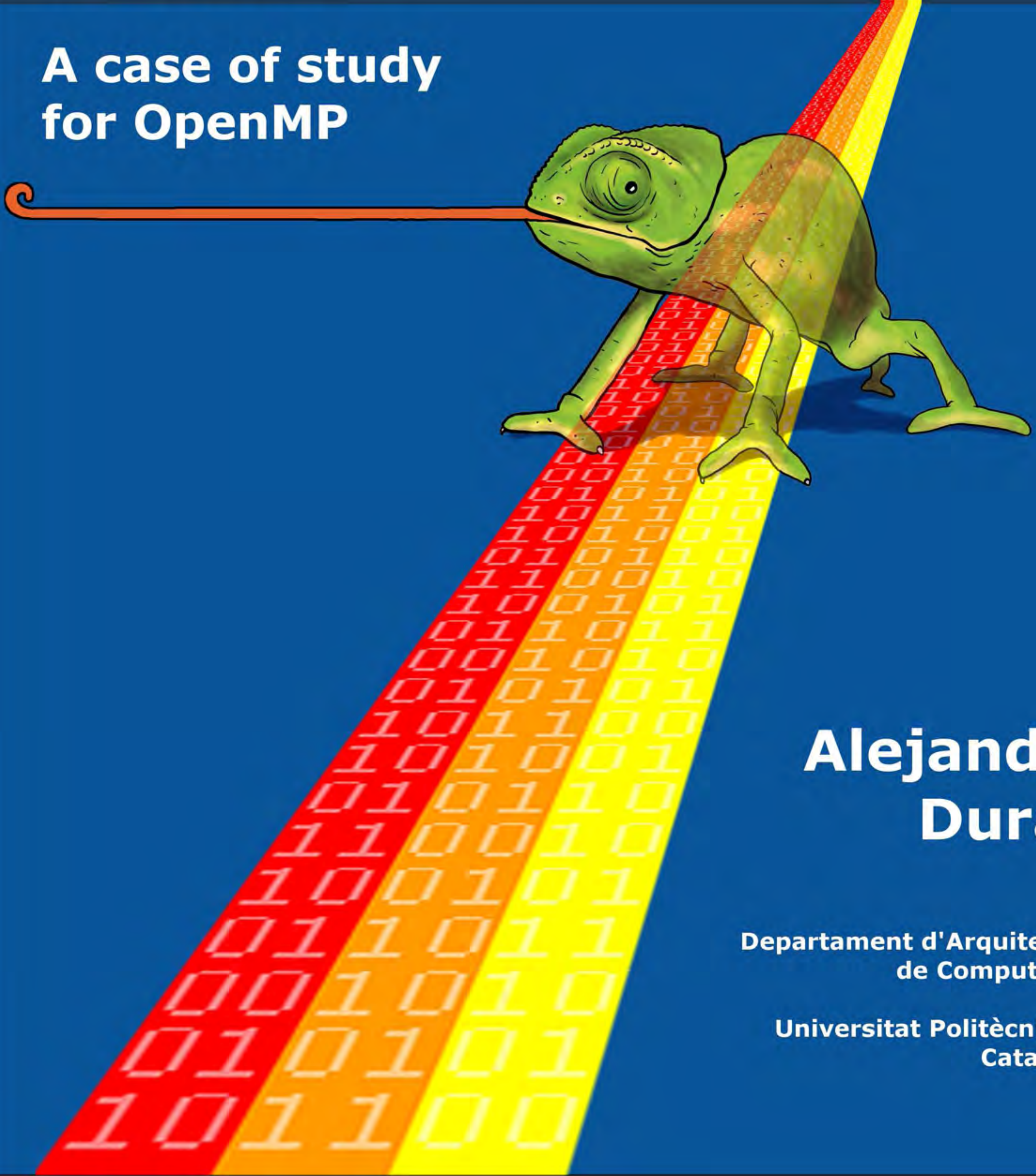
ADVERTIMENT. La consulta d’aquesta tesi queda condicionada a l’acceptació de les següents condicions d'ús: La difusió d’aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d’investigació i docència. No s’autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d’un lloc aliè al servei TDX. No s’autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you’re accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it’s obliged to indicate the name of the author

Self-tuned parallel runtimes

A case of study for OpenMP



Alejandro Duran

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Self-tuned parallel runtimes

A case of study for OpenMP

Alejandro Duran González

Advisors: Julita Corbalán and Eduard Ayguadé

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Thesis submitted in fulfillment of the requirements of the degree:

Doctor per la Universitat Politècnica de Catalunya

September 2008

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: ....
Autor de la tesi: ....

Acorda atorgar la qualificació de:

No apte
Aprovat
Notable
Excel·lent
Excel·lent Cum Laude

Barcelona, …………… de/d'…..................…………….. de ....….

El President	El Secretari
....	....
(nom i cognoms)	(nom i cognoms)

El vocal	El vocal	El vocal
....	....	....
(nom i cognoms)	(nom i cognoms)	(nom i cognoms)

Abstract

Although parallel computing is increasingly common, particularly due to the spread of multicore processors, the programming of these parallel systems remains difficult. A whole new of set of problems that do not exist in sequential programming is present in parallel programming: identification of parallelism, work and data distribution, load balancing, synchronization and communication. Novel users, in general, do not have the necessary expertise to deal with those problems and the help the receive from the languages, compilers and tool is still not enough. As a result, the productivity of most programmers in parallel systems is severely hampered.

One possible way to ease the problem is have adaptive runtime components as a part of a parallel environment. This components would obtain information about the application and the execution environment and based on their embedded expertise can take decision that maximize the performance of the applications reducing the burden put into the programmer. The goal is not to develop methods that outperform a hand-tuned applications for specific architectures but come as close as possible with a minimum effort from the programmer.

In this thesis, we develop self-tuned algorithms for three different aspects of parallel exploitation of the OpenMP parallel language: parallel loop schedul-

[1]The work of this thesis has been supported by Ministry of Education of Spain (under contracts TIC2001-0995-C02-01, TIN2004-07739-C02-01 and TIN2007-60625), the European Commission (in the context of POP project (IST-2001-33071) and the SARC integrated project #27648 (FP6)), the HiPEAC Network of Excellence (IST-004408), the MareIncognito BSC-IBM agreement and the IBM CAS Program. We also would like to thank the Barcelona SuperComputing Center (BSC) for the use of their machines.

ing, thread allocation in multiple levels of parallelism and task granularity control. The algorithms are fed with information obtained from on-line profilers that gathers information about the characteristics of the application. With this information,

Our evaluation results shows that the performance obtained using the self-tuned algorithms is in most cases as good as the one obtained by applications optimized by hand. We think this suggests that the use of self-tuned algorithm can help to maximize the ratio performance over effort so the entry level to the parallel computation is lower.

Acknowledgements

The process of this thesis has been very long and as such I have to give to thanks many people: some for their technical advice, some for their moral support and many for just being there. The list is long and, as usual, i am in a rush so if I forget to mention someone, please , do not be upset with me.

It all began with Xavier Martorell and Jesús Labarta who guided me at start of my research career and set me on my PhD path. I think both can be considered some sort of "shadow" directors of this thesis as often had to heard my doubts and problems and always provide me with good insights. Thanks, to Julita, who started helping me at the very beginning and, although we have not always agreed, she has been very helpful (and sort of my personal psychologist). I want to thank Eduard, as well, for his advice and guidance, particularly, in the later stages of the thesis.

The list of people who I have been collaborating with in the different stages of is as long as it has been the road to this thesis. I'd like to thank particularly the people involved in the Mercurium compiler (Jairo, Roger) and also of the NANOS group at UPC (Marc, Juanjo (my super officemate :D), Xavi, ...) and to others (Montse, David, Juanjo (again!),...).

Most of the work of this thesis has been done in two cities: the beautiful Barcelona where I live and Toronto in Canada. I ended spending 10 months in the IBM Lab in Toronto which were both fruitful in work and experiences. I want to thank the IBM Center of Advanced Studies and Jesús for such an opportunity. Also, I would like to thank to all the people that helped me while I was there starting with Kelly who helped me countless times (and also gave me countless rides to the IBM Lab :). I would like to mention all the other people with whom I shared so much time in the Lab (Kit, Faryaaz, Brad, Cyrus, ...). Also, thanks to Raúl Silvera for his technical help and to Susan for providing our little barcelonian community with a shelter where to live :)

One big thanks to my family and friends for always providing a such a good support. Knowing that in periods of stress you can have some fun in good company is a great safety net to perform hard endeavors :). I know I have been always the most communicative guy about my work but I have always appreciated the interest.

But, most and foremost, I want to thank to those who had to live with me because I know it has not always been easy (sleepless nights and long periods of stress tend to make coexistence a little more *interesting*). Thanks for your unconditional support and help in the difficult moments.

Contents

List of Tables

LIST OF TABLES

List of Figures

Chapter 1

Introduction

The real voyage of discovery consists not in seeking new landscapes but in having new eyes.

Marcel Proust

French writer (1871 – 1922)

Abstract

This chapter overviews the context of the parallel world in which the work has been developed. It describes how the process of writing a parallel program works and how this thesis tries to improve this process by using runtime self-tuned algorithms. We briefly describe the contributions of this thesis. This chapter also outlines the structure of the remaining of the document.

1.1 A parallel world

In recent years parallel computation has finally, after years of prophecy, become ubiquitous. Lead by the spread of commodity multicore processors, parallel programming is not anymore an obscure discipline only mastered by a few rocket scientists but a skill required to many programmers from a variety of fields (e.g. databases, games, systems, . . .).

Unfortunately, the amount of able parallel programmers has not increased at the same speed as the number of parallel systems. One of the reasons for this increasing gap is the lack of an easy way to write parallel code.

Parallel programming is inherently different from sequential programming. Programmers must deal with a whole new set of problems: identification of parallelism, work and data distribution, load balancing, synchronization and communication.

At first, it seemed like compilers could help to make a seamless transition from sequential to parallel by means of automatically parallelizing the user code. But, this approach has only produced limited results due to the complexity of understanding the semantics of the user programs and, thus, parallelize them properly. Meanwhile, parallel programmers have embraced several languages designed to allow the creation of parallel applications. In these languages, the programmer is not only responsible of identifying the parallelism but also of specifying such low-level details as, for example, the exact mapping of data or the way the work should be distributed across the available computing units (i.e. scheduling).

This low-level details do not express the parallelism in the application but how the parallelism of the application needs to be exploited (i.e. how it needs to be mapped to the execution environment). How the parallelism of an application will be exploited depends on several parameters (e.g. scheduling, threads allocated, . . .) that in most parallel languages the user needs to set. Users usually need to spend time analyzing the different options for the different parameters to find which values obtain the best performance for their execution environment.

The fact that users need to take care of such details is not only a waste of time for them but it also hurts the portability of the application as the decisions need to be reevaluated from one execution environment to another. Even a change in the input data of the application or the number of processors used may result in previous right decisions being erroneous.

Moreover, the trend in today's systems is that the execution environment keeps changing in order to comply with business considerations [SWC+07, USR02, WLW+07, CSW+08]. This means that users will have a hard time trying to tailor the parallel exploitation to a changing environment. Clearly, a different method needs to be used.

For these reasons, developing mechanisms that allow the applications to automatically decide which is the best value of the different parameters that configure the parallel exploitation of an application with minimum user intervention is an important line of research.

1.2 Programming models

The process of transforming a sequential execution of a program into a parallel execution one consist, roughly, of two steps: first, the identification of all the parallel units of a program and its relationships (i.e. synchronization and communication). Second, the exploitation of these units using the resources available in the execution environment of the application.

Ideally, the compiler using flow analysis and data dependence analysis would be able to detect which parts of a program can run concurrently. Unfortunately, several problems like pointer aliasing, lack of interprocedural information or memory indirections have severely hampered the ability of compilers to understand what are the relationship of different parts of program. This lack of knowledge forces compilers to be very conservative in their effort to parallelize user codes which results, with a few exceptions, in parallel versions which express far less parallelism than the amount that really exists in the application and, as a result, the obtained performance is not very good [ZUR04].

Instead of relying on compiler auto-parallelization, programmers who want to take the maximum advantage from the parallelism of their application have used an assortment of explicit parallel languages to express the available parallelism in their applications.

Most notably, the *Message Passing Interface* (*MPI*) [For95] has been extensively used for distributed memory environments and, also, in shared memory environments.*MPI* runs multiple replicas of the same program in parallel and the user is responsible of distributing the work across the different replicas. The information exchange between the different replicas is achieved by means of calls to explicit communication messages.

Also, the *OpenMP* [Org08] language has become in recent years the *de facto* standard for programming in shared memory systems, particularly because it has been endorsed by all important vendors. *OpenMP* extends the Fortran (by means of special comments), C and C++ (by means of #pragma directives) languages with constructions that allow to specify regions of a program that are executed in parallel by multiple threads. Other constructions allow the distribution of work across multiple threads as well as their synchronization. The communication between threads is implicit as *OpenMP* uses a shared memory model where all the threads can see all the memory.

The *Unified Parallel C*(*UPC*) [con05] is a global-address space language that extends the C language. As *MPI*, multiple copies of a *UPC* program run at the same time. *UPC* only provides support for work distribution for parallel loops.*UPC* provides language extensions that allow the programmer to explicitly distribute the data across the different copies (which may be in different nodes).

The *Cilk* [FLR98] language extends, as well, the C language with constructions that allow to easily write task-parallel programs. It provides language constructions to create parallel tasks and synchronize them but it has no constructions for data parallelism (e.g. parallel loops). *Cilk* forces the application to have a recursive structure to be efficient which means that in many cases the programmer needs to rewrite its program.

The problem with these languages is that the specification of the parallelism is tied with how the parallelism needs to be exploited. For example, if we had a simple code, like the one in Figure 1.1, where there are three loops *i,j,k*. Of these loops the *i* and *j* loops are parallel as there are no dependences between the iterations.

```
for ( i = 0; i < N ; i++ ) {
  for ( j = 0; j < M; j++ ) {
    r = 0;
    for ( k = 0; k < K ; k++ ) {
      r = f(r) + g(k);
    }
    a[i,j] += r;
  }
}
```

Figure 1.1: Simple code (i and j loops are parallel)

If we were, for example, to use *OpenMP* (see Chapter 2 for details on *OpenMP*) to parallelize that program we could write several different parallelizations. So, for example, we could write a very simple parallelization like the one in Figure 1.2 where only the outer i loop is executed in parallel. In this example, the number of threads used and which iterations will be executed by each thread are unspecified so the implementation *defaults* will be used (which are not necessarily the best).

```
#pragma omp parallel for
for ( i = 0; i < N ; i++ ) {
  for ( j = 0; j < M; j++ ) {
    r = 0;
    for ( k = 0; k < K ; k++ ) {
      r = f(r) + g(k);
    }
    a[i,j] += r;
  }
}
```

Figure 1.2: Simple code (*OpenMP* version)

```
#pragma omp parallel for num_threads(2) schedule(static)
for ( i = 0; i < N ; i++ ) {
#pragma omp parallel for schedule(dynamic,8)
  for ( j = 0; j < M; j++ ) {
    r = 0;
    for ( k = 0; k < K ; k++ ) {
      r = f(r) + g(k);
    }
    a[i,j] += r;
  }
}
```

Figure 1.3: Simple code (*OpenMP* version with nested parallelism)

Another user may, for example, choose to write a different parallelization (see Figure 1.3) where, now, both the i and j loop will be executed in parallel by means of using *nested parallelism*. The user also specifies that the number of threads allocated to the i parallel loop should be two and that an `static` assignment of iteration should be used while that for the j loop a `dynamic,8` assignment of iterations to threads should be used.

Which of the two parallelizations is the best? The answer to this question depends on several factors. Some, like the values of N and M or the granularity of the functions f and g, depend on the application. Others, like the cacheline size, depend on the final architecture where the program runs. Others, like the number of available threads, depend on execution environment of the application and may change from one execution to another even in the same architecture. So, which is the best parallelization of the two will usually depend on so many different parameters that is difficult to know before hand.

This forces the users to test the two versions (and possibly many others) in their target environment. This analysis is time consuming and error prone so users could end taking the wrong decision. Of course, experienced users have intuitions of what things will work well and which ones will not that allows them to simplify this analysis. But, intuition is not a rational tool and even these users will make mistakes from time to time. Even worse, when any of the factor mentioned before (i.e. input data, architecture or execution environment) changes this analysis will be need to be done again. What happens in practice many times is that users have different parallel versions of the same sequential program: one for each specific scenario.

At least, in the *OpenMP* memory model the communication happens implicitly while in models based on messages like *MPI* the user needs to care, as well, of how the information flows between the threads. So in *MPI* it is even more explicit how the parallelism needs to be executed.

Clearly, while these languages allow an experienced user to develop a high-performance version of his serial application, the burden put into the casual programmer which is just starting or needs to parallelize a small portion of a program is too much. A better approach is needed to help these users start in the parallel world.

1.3 Goal of this thesis

Our philosophy is that the programmer is responsible of identifying the parallel parts of the program and their relations (i.e. communication and synchronization) and the runtime library is responsible of automatically finding the best way to exploit the available parallelism. So, for our simple code, the user would mark all the parallelism by marking both the *i* and *j* loop (see Figure 1.4). The actual decision of how to execute those loops (i.e. number of threads, serialize them or not, assignment of iterations to threads, ...) should be decided by the implementation (compiler plus the runtime library).

```
#parallel
for ( i = 0; i < N ; i++ ) {
#parallel
  for ( j = 0; j < M; j++ ) {
    r = 0;
    for ( k = 0; k < K ; k++ ) {
      r = f(r) + g(k);
    }
    a[i,j] += r;
  }
}
```

Figure 1.4: Ideal parallelization of the simple code

In this thesis, we want to demonstrate that is possible for the runtime library of a parallel environment to adapt itself to the application and the environment and thus reducing the burden put into the programmer when he develops a parallel program.

For this purpose we study three different parameters that are involved in the parallel exploitation of the *OpenMP* parallel language, which we have chosen as our target language:

Parallel loops scheduling Parallel loops are one of the main sources of data parallelism but the performance that user can obtain from them depends heavily on which iterations are executed by each of the threads. This decision is known as loop scheduling.

Thread allocation When there is just one level of parallelism all the threads of the application are assigned to the lonely parallel region. But, now a days, a very common technique to improve the load balance of applications is using multiple levels of parallelism. In these cases, a decision needs to be made about how the threads of the application are allocated across the different parallel regions.

Parallel tasks cut-off Parallel tasks are another important source of parallelism. One the main problems is making sure that the created tasks have enough granularity to overcome the overheads of their creation. Because at the same point in the program the granularity of the tasks created will vary (e.g. recursive functions) users will, typically, embed a cut–off function in their program that will be decide if a task is created or not.

In all the cases, we want to understand the different alternatives of execution and understand under which circumstances is better to use each one to propose a self-tuned algorithm that will first perform an on-line profiling of the application and based on the information gathered it will adapt the value of the parameter to the one that maximizes the performance of the application.

Our goal is not to develop methods that outperform a hand-tuned application for a specific scenario, as this is probably just as difficult as compiler code outperforming hand-tuned assembly code, but methods that get close to that performance with a minimum effort from the programmer. In other words, what we want to achieve with our self–tuned algorithms is to maximize the ratio performance over effort so the entry level to the parallelism is lower.

1.4 Contributions of this thesis

To demonstrate our ideas, we have developed several mechanisms in a real system that based on information gathered at execution time adapt different parameters related to the exploitation of the parallelism in the *OpenMP* language.

The following list summarizes the contributions of this thesis:

- An self–tuned loop scheduler for parallel loops like the *OpenMP* DO workshare.
- An self–tuned algorithm to decide the allocation of threads for an hybrid MPI+*OpenMP* application.
- An self–tuned algorithm to decide the allocation of threads in innermost level of an *OpenMP* application with multiple levels of parallelism.
- An extensive analysis of scheduling strategies in the new *OpenMP* task model.
- An self–tuned cut–off for *OpenMP* tasks that serializes them based on granularity prediction to increase the granularity of parallel tasks.

1.5 Organization of this thesis

The remaining of this thesis is organized as follows:

Chapter 2 describes the main elements (programming models, hardware model and the runtime library) of the environment where we have developed our thesis.

Chapter 3 describes a feedback technique to adapt the parallel loop scheduler at execution time to improve the load balance of the application. The mechanism adapts to the dynamic characteristics of the application (i.e. input data) and of the execution environment (e.g. number of processors).

Chapter 4 describes a feedback technique to adapt the distribution of the threads available to the application across different parallel regions in order to improve the load balance of the application.The mechanism adapts to the dynamic characteristics of the application (i.e. input data) and of the execution environment (e.g. number of processors).

Chapter 5 presents our initial evaluation of the new *OpenMP* task model. We describe our parallelization of different benchmarks with this model and our evaluation of different scheduling strategies for it. This allows us to identify areas that users will need to optimize manually so we can provide self-tuning algorithms.

Chapter 6 describes an analysis of the performance of different scheduling policies for *OpenMP*tasks and, based in that analysis, a feedback-guided policy that dynamically increases the granularity of parallel tasks by means of serializing small tasks to generate coarser tasks.

Chapter 7 presents the conclusions of this thesis and discusses some possible lines of future work.

Chapter 2

Background and Context

If you don't crack the shell, you can't eat the nut.

Persian Proverb

Abstract

This chapter presents some important concepts that will be used in the following chapters of this thesis. It also describes our execution environment. We present a brief introduction to the *OpenMP* language and to the *MPI+OpenMP*hybrid model that we use in part of our work. We present the concept of *OpenMP* runtime library and why is important for us. Finally, we discuss how we can retrieve information from the applications to feed our self-tuned algorithms.

2.1 Introduction

In this chapter, we introduce several key concepts to the work presented in the following chapters of this thesis.

The target execution environment of our techniques are multiprocessors with a global shared memory address space. We, briefly, describe the most important characteristics of these architectures.

There are several parallel programming languages available for these architectures but we decided to focus in the *OpenMP* language as it widely viewed as a *de facto* standard for shared memory programing. We describe its main features: parallel regions, worksharings and synchronization constructs. We also explain two other *OpenMP* advanced features that are important for our work: nested parallelism and task parallelism. We then explore the *MPI+OpenMP* hybrid model that we have used for part of our work.

We describe what the *OpenMP* runtime library is and why is a key component in all of our proposed algorithms. We, also, give a brief description of the two runtime libraries we have used to implement our proposals: IBM's XL runtime library and the NANOS NthLib runtime library .

For our self-tuned algorithms we need to obtain information from the applications at runtime to use it as input. We discuss how we can obtain such information and which are the limitations of the different information retrieval options. These limitations, in turn, impose the structure that we expect from the application in order to apply the self-tuned methods that will be presented in the following chapters.

2.2 Execution environment

Through the work of this thesis we have used different execution environments (POWER3 [PDM$^+$98], POWER4 [Cor01], POWER5 [KST04] and Itanium-2 [Cor04]) but all them have a common characteristic: they are multiprocessor architectures with hardware shared memory. This means, that all the processors in the system have a global view of the memory of the

Architecture (Vendor model)	**Frequency**	**Number of Processors (Cores)**	**L2 Cache Size**	**Memory**	**Operating System**
POWER3 (IBM Nighthawk-2)	375 MHz	16 (16)	8 Mb	64 Gb	AIX 5.1
POWER4 (IBM Regatta)	1.1 GHz	16 (32)	1.44 Mb	128 Gb	AIX 5.1/5.2
POWER5 (IBM eServer p5 595)	1.6 GHz	32 (64)	1.92 Mb	514 GB	AIX 5.3
Itanium-2 (SGI Altix 4700)	1.6 GHz	64 (128)	8 Mb	986 Gb	SUSE Linux Enterprise 10

Table 2.1: Summary of the execution environments of this thesis.

system and they can access any memory location with a regular load/store operation. The cost of accessing different memory locations does not necessarily need to be the same. In fact, many shared memory systems are NUMA (Non-uniform Memory Access) systems where the access to time to a memory location varies from processor to processor.

In Table 2.1, we summarize the main characteristics of the execution environments we have used for our experiments. Because all our execution environments are shared memory architectures we chose as our target programming model one that was designed with this kind of architectures in mind: *OpenMP*.

2.3 OpenMP

OpenMP [Org08] has emerged in recent years as the *de facto* standard for parallel programming in shared memory systems. *OpenMP* is based on the idea of an execution environment with a global shared address space. Programmers define which loops and sections of code can be executed in parallel.

OpenMP works by inserting directives in the sequential source code. In fact, the same code can generate a sequential or parallel version just activating or deactivating the *OpenMP* directives using a compiler flag. *OpenMP* also defines a set of intrinsics and environment variables that allow to configure the behavior of the parallel execution. The *OpenMP* elements provide the application programmer with the tools to run multiple threads in parallel, distribute work among them and synchronize them.

Another advantage of this programming model is that parallelization can be done incrementally. Users can perform a profile of their sequential program and just parallelize the most time consuming parts of their code. The program can then be tested and evaluated. If the program achieves enough speedup, the parallelization can be finished at this point. Otherwise, users can parallelize additional code.

2.3.1 Parallel regions

OpenMP uses a fork-join execution model. This means that an *OpenMP* program starts the execution as a single threaded program and at some point it generates parallelism by creating multiple threads which are afterwards terminated and the single threaded execution continues until the next fork point. The region between the fork and the join point is called a parallel region. Figure 2.1 illustrates the *OpenMP* fork-join model.

To create a parallel region in an *OpenMP* program we need to use the `parallel` construct. Figure 2.2 shows the syntax of this construct for the C and C++ languages while Figure 2.3 shows the syntax for the Fortran language.

When the initial thread (also known as serial thread) encounters a `parallel` construct it creates new group of *OpenMP* threads (known as a `team of threads`) and it becomes the master of it. All the threads of the team start executing the code of the parallel region (i.e. all of them are doing the same). The number of threads that will be created can be determined by the user using the `OMP_NUM_THREADS` environment variable, the `omp_set_num_threads` intrinsic, or the `num_threads` clause in the `parallel` construct. If the user does not specify a number of threads the implementation will choose the number. It is important to note that once a parallel region starts the number of threads remains fixed until the end of the region. But, between different

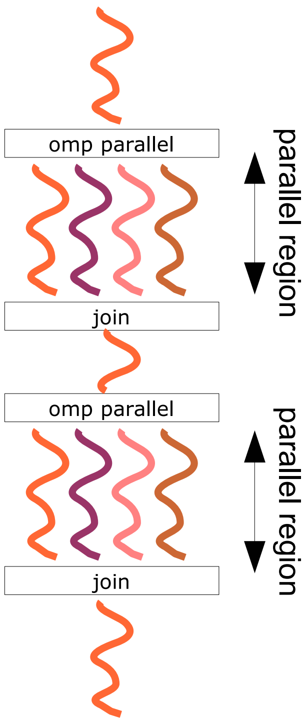


Figure 2.1: *OpenMP* fork-join model

```
#pragma omp parallel [clauses]
   structured-block
```

Figure 2.2: C/C++ `parallel` construct syntax

```
!$OMP PARALLEL [clauses]
   statements
!$OMP END PARALLEL
```

Figure 2.3: Fortran `parallel` construct syntax

parallel regions the number of threads can change. This fact gives *OpenMP* applications, unlike other programming models, the characteristic that they can adapt the parallelism that they spawn to the number of processors available (malleability) each time they cross a parallel boundary.

At the end of the parallel region there is an implicit barrier that acts as a synchronization point for all threads of the team. When all the threads of the team reach the barrier all but the master thread will be logically[1] destroyed.

Another useful clause is the `if` clause. When the user specifies an `if` clause the expression contained in the clause will be evaluated before the team is created. If it evaluates to false the parallel region is executed only by the serial thread and no other threads are created. Otherwise the team is created as usual.

The remaining clauses are used for data scoping and they will be discussed in Section 2.3.3.

2.3.2 The worksharing constructs

All threads in a parallel region are initially executing the same code and thus doing exactly the same work. But, in general, our interest in having multiple threads is, that instead of all of them computing the same, they can split the work between them so the overall computation goes faster.

In *OpenMP*, this is achieved by means of the `worksharing` constructs. The most important `worksharing` constructs are the `sections` workshare, the loop worksharing constructs and the `single` workshare.

The SECTIONS construct

The `sections` construct allows the programmer to specify that several parts of his program can be executed in parallel and that they should be distributed among the threads of the team of the parallel region where the `sections` construct appears.

[1]Most implementations will keep them alive to be reused afterwards for efficiency purposes but from the *OpenMP* point of view they are terminated.

```
#pragma omp sections [clauses]
{
  [#pragma omp section]
     structured block
  [#pragma omp section
     structured block
  ]
  ...
}
```

Figure 2.4: C syntax of the `sections` construct

```
!$OMP SECTIONS [clauses]
[!$OMP SECTION]
     statements
[!$OMP SECTION
     statements
]
  ...
!$OMP END SECTIONS [nowait]
```

Figure 2.5: Fortran syntax of the `sections` construct

Figure 2.4 shows the syntax of the `sections` construct for the C/C++ languages. Figure 2.5 shows the syntax for Fortran language.

Each of the `section` directives defines a region of code that can be executed independently of the others by a thread. When the threads reach the `sections` construct they start executing all the defined `section` in a dynamic order. There is an implicit barrier at the end of the `sections` construct unless the `nowait` clause is specified.

The remaining clauses of the `sections` construct are used for data scoping and they will be discussed in Section 2.3.3.

Figure 2.6 shows a simple example with the `sections` construct. In this

```
#pragma omp parallel
#pragma omp sections nowait
{
#pragma omp section
  foo();
#pragma omp section
  bar();
}
```

Figure 2.6: Example of a SECTIONS construct

```
#pragma omp for [clauses]
  for-loop
```

Figure 2.7: Syntax of the `for` construct

```
!$OMP DO [clauses]
  do loop
!$OMP END DO [nowait]
```

Figure 2.8: Syntax of the `do` construct

example, we defined two different sections: the *foo* section and the *bar* section that can be executed in parallel by any thread of the team. When threads finish the execution of the section they skip the implicit barrier at the end of the construct because the `nowait` clause is specified.

The loop worksharing constructs

Most commonly, the different parts of the program that a programmer would like to execute concurrently are the different iterations of a loop. In this case, two worksharing constructs exist: the `for` construct for C/C++ and the `do` construct for Fortran. They are basically the same and only differ in some minor issues related to the base language. We can see the syntax for the `for` construct in Figure 2.7 and for the `do` construct in Figure 2.8.

When threads reach one of the loop worksharing constructs they start executing all the iteration of the loop in parallel. The `schedule` clause determines which iterations will be executed by each thread of the team. *OpenMP* defines the following loop schedules:

static The `static` schedule specifies that the iterations are to be divided proportionally between of the threads of the team. Also, all iterations assigned to a thread must be contiguous.

static,n Also known as *interleave* schedule, the `static,n` schedule specifies that the iterations are grouped into `chunks` of size n. Then, these chunks are distributed giving one to each thread. If there are still unassigned chunks we start giving another to the first thread and so on.

dynamic With the `dynamic` schedule, iterations form chunks of size n as in the previous schedule but instead of a fixed assignment of chunks they are distributed on a first-come first-serve approach. This means that in two instances of the same loop the actual assignment can be different.

guided The `guided` schedule, assigns iterations in chunks dynamically as the threads request them but the size of each chunk is proportional to the number of iterations remaining to assign. So, initially the chunks will be big but the last ones will be small.

auto The `auto` schedule gives implementations the ability to decide any possible assignment of iterations to threads. This can range from simple static schedules to complex feedback guided schedules. In part, this schedule has been inspired by our work described in Chapter 3.

runtime When a programmer specifies a `runtime` schedule it delays the decision of the actual schedule until the application is executed. This allows the programmer to change the schedule of the loops (to one of the previous ones) using an API call or an environment variable.

As in the `sections` construct there is an implicit barrier at the end of the loop worksharing region unless the programmer makes use of the `nowait` clause.

Other clauses of the loop worksharing constructs are used for data scoping and they will be discussed in Section 2.3.3.

The SINGLE construct

The `single` construct allows the programmer to specify a section of his program that while is the middle of the parallel region it will be only executed by on the threads of team. This allows to serialize some parts of the code (e.g. input/output) without having destroy the team and create it again afterwards.

```
#pragma omp single [clauses]
{
  structured block
}
```

Figure 2.9: C syntax of the `single` construct

```
!$OMP SINGLE [clauses]
      statements
!$OMP END SINGLE [nowait]
```

Figure 2.10: Fortran syntax of the `single` construct

Figure 2.9 shows the syntax of the `single` construct for the C/C++ languages. Figure 2.10 shows the syntax for Fortran language.

When the threads reach a `single` construct only one of them will execute the code associated to the single region. The other threads wait at an implicit barrier at the end of the region unless the `nowait` clause is specified. Note, that if the `nowait` clause is specified multiple threads can be in different `single` regions at the same time.

The remaining clauses of the `single` construct are used for data scoping and they will be discussed in Section 2.3.3.

2.3.3 Data scoping

OpenMP defines several attributes that are associated to the program variables that specify the visibility and initial value of those variables inside the different parallel and worksharing regions of the program. We discuss here the most important clauses that modify these attributes which are common to the `parallel` and worksharing constructs: `shared`, `private`, `firstprivate` and `reduction`.

Shared and private variables

Variables in any region (except some task regions) are *shared* by default. This means that any access from any thread will be seen by the others as they are sharing the same variable. The `private` clause allows to specify that certain variables should be *private* to each thread.

When threads enter a region they allocate space for all their *private* variables and any modification to these variables will only be visible by that thread and not by any other thread of the team.

The `shared` clause allows the user to specify that some variables will be *shared*.

Firstprivate variables

The `firstprivate` clause allows the programmer to specify a list of variables which will be private (i.e. there will be a copy for each thread) but the initial value of each of the private copies will be the value of the original variable when the region is reached.

In task regions that are not directly nested in a `parallel` or worksharing region variables will be by default `firstprivate`.

Reduction variables

The `reduction` clause allows the programmer to specify a list of variables where the program performs a reduction operation.

For each variable marked as `reduction` a new private variable will be created in the region for each thread. The initial value of this private variable will be the neuter of the reduction operation (e.g. 0 for the addition operation). At the end of the region, all the private reduction variables of the threads will be combined into a single value that will be then stored in the original variable.

2.3.4 Communication and synchronization in *OpenMP*

Besides being able to cooperatively execute work, threads will eventually need to synchronize and communicate information between them. Because *OpenMP* uses a shared memory model all the communication happens implicitly when a thread writes to a shared variable and another reads it. The user is still responsible to ensure that all threads have the same view of the

```
#pragma omp barrier
```

Figure 2.11: C `barrier` syntax

```
!$OMP BARRIER
```

Figure 2.12: Fortran `barrier` syntax

memory because the *OpenMP* memory model is a relaxed one that does not ensure a coherent memory view from different threads. This is accomplished using the `flush` directive. To help the user, *OpenMP* defines a number of implicit `flush` operations that cover most common use cases.

For thread synchronization, *OpenMP* defines three main constructs that we describe next: the `barrier`, `critical` and `atomic` constructs.

Barrier construct

As we have seen many *OpenMP* constructs have an implicit barrier at the end of their regions that acts a synchronization point for all the threads of a team.

The `barrier` construct allows to insert explicit barrier synchronizations that will act in exactly the same way: no thread is allowed to go beyond the `barrier` construct until all the threads of the team have reached it.

The `barrier` construct syntax for the C/C++ languages is shown in Figure 2.11 and for the Fortran language in Figure 2.12.

Critical construct

The `critical` constructs allows to specify that a certain sections of code can only be executed by one thread at a time. This allows to protect sensitive data structures from multiple simultaneous manipulations (i.e. data races). The syntax for the C/C++ languages is shown in Figure 2.13 and for the Fortran language in Figure 2.14.

When a thread reaches the beginning of a `critical` region it waits until no other thread is inside any other (or the same) critical region and then it

```
#pragma omp critical [(name)]
{
  structured block
}
```

Figure 2.13: C/C++ `critical` syntax

```
!$OMP CRITICAL [(name)]
  block of statements
!$OMP END CRITICAL [(name)]
```

Figure 2.14: Fortran `critical` syntax

```
#pragma omp atomic
  statement
```

Figure 2.15: C/C++ `atomic` syntax

```
!$OMP ATOMIC
  statement
```

Figure 2.16: Fortran `atomic` syntax

tries, to atomically, gain access to it. When it exits it signals that it has exited so other threads can try to gain access to the `critical` regions.

Optionally, a name can be specified in a `critical` region. This allows to have multiple threads inside multiple `critical` regions as long as all of them have different names (one of them can be *unnamed*).

Atomic construct

The `atomic` construct avoids that multiple threads update the same memory address at the same time. The syntax for the C/C++ languages is shown in Figure 2.15 and for the Fortran language in Figure 2.16.

Only a limited number of statements are allowed to be protected with an `atomic` construct: arithmetic operations and assignment (e.g. $x = x + n$), logic operations and assignment (e.g. $x <<= 2$), increment, decrement, post-increment, post-decrement and, in Fortran, some intrinsics like *max* and *min*.

The `atomic` construct is restricted to simple statements so the compiler

```
#pragma omp parallel num_threads(2)
#pragma omp sections
{
  #pragma omp section
  #pragma omp parallel num_threads(2)
    foo();
  #pragma omp section
  #pragma omp parallel num_threads(3)
    bar();
}
```

Figure 2.17: Example of nested parallel regions

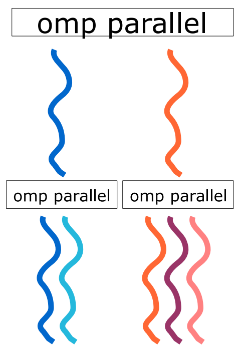


Figure 2.18: Teams of threads in nested parallel regions

can use special atomic operations from the architecture to improve the performance of the application.

2.3.5 Nested parallelism

OpenMP parallel regions can be nested inside other parallel regions. This means that new teams of threads are created. The master of the new parallel region is the thread that encountered the nested parallel construct but all other threads will be new *OpenMP* threads. By doing so, we obtain a tree structure of teams of threads. For example, the code in Figure 2.17 would create an outer parallel region with two threads. Then each of those threads would create an inner parallel region one with two threads that will execute *foo* and another with three threads that will execute *bar*. A tree representing this structure of teams is shown in Figure 2.18 where each thread has a different color. Note how the master threads are reused in the inner teams.

```
#pragma omp task [clauses]
  structured-block
```

Figure 2.19: C/C++ `task` construct syntax

```
!$OMP TASK [clauses]
  statements
!$OMP END TASK
```

Figure 2.20: Fortran `task` construct syntax

Nested parallelism is particularly useful in situations where there is not enough work for all threads in an outer level but exploiting only the inner level would not yield a good performance because of the overheads of creating the inner parallel regions. Nested parallelism offers a compromise in those situations [AML+99,AGMJ04]. Nested parallelism , as we will see in Chapter 4, can also be used to improve the performance when there is unbalance in the outer levels of parallelism.

2.3.6 *OpenMP* tasks

The latest specification of *OpenMP* (3.0) has shifted from a thread–centric to a task–centric execution model, based on the fork-join paradigm where threads are execution vehicles that have access to the shared memory [ACD+07].

The task construct

The `task` construct allows the programmer to explicitly specify a unit of parallel work called a *task*. Tasks are useful for expressing unstructured parallelism and for defining dynamically generated units of work.

Figure 2.19 shows the syntax of the `task` construct for the C/C++ languages and Figure 2.20 for the Fortran language.

A `task` construct may be lexically or dynamically nested inside an outer task construct. Tasks created by this construct may be executed by any thread in the team immediately or they may be deferred until a later time. The major difference from the *OpenMP* task model compared to other existing ones is that tasks are created `tied` by default. `Tied tasks` can be

```
void traverse (struct tree *tree)
{
  if ( tree->left )
    #pragma omp task
      traverse(tree->left);
  if ( tree->right )
    #pragma omp task
      traverse(tree->right);
  #pragma omp taskwait
  process(tree->data);
}
```

Figure 2.21: Simple tree traversal with *OpenMP* tasks

executed by any thread but if, after the execution start, the task is suspended, execution can only be resumed by that same thread that suspended it. So, it is said that the task is `tied` to the thread that started the execution.

The `untied` clause allows the programmer to change this behavior if that does not cause any problem (e.g. use of `threadprivate` in `untied tasks` is discouraged). `Untied tasks` have no scheduling restrictions so they can be executed at any time by any thread.

The remaining clauses of the `task` construct are related to the creation of a data environment for the new task (i.e. data scoping) and another that allows dynamic serialization of tasks based on a condition (i.e. `if` clause) analogous to the `if` clause of the `parallel` construct.

The taskwait construct

The `taskwait` construct completes the task support. `Taskwait` suspends execution of the current task until completion of all of its child tasks.

Dynamic task aggregation

OpenMP also allows an implementation to serialize tasks and execute them immediately as part of the parent task (although they need to have their own data environment). This allows the runtime to implement cut-off techniques in order to reduce overheads by dynamically aggregating several tasks into a single one.

If we had a recursive traversal in a tree like the one in Figure 2.21, where a

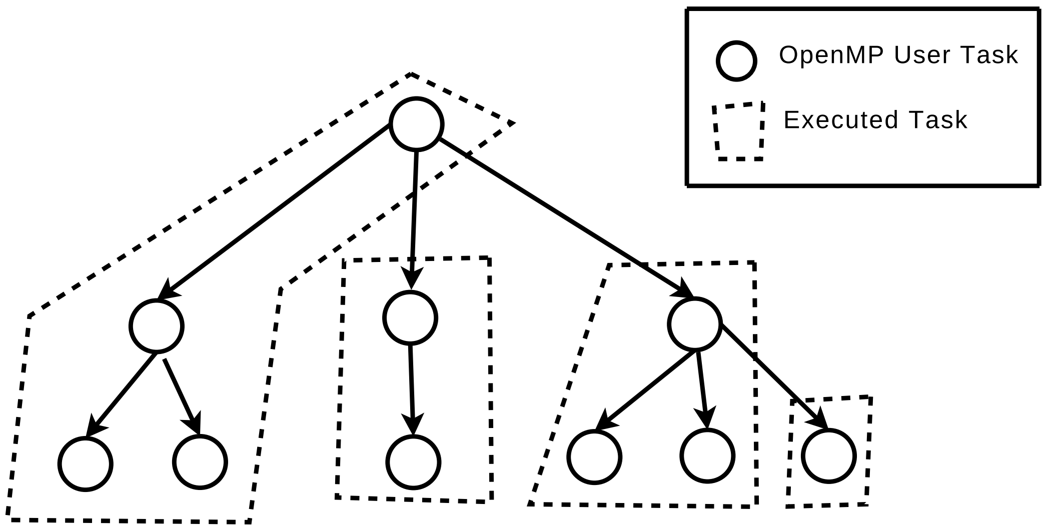


Figure 2.22: A possible execution for a recursive tree with *OpenMP* tasks

tree is visited in post-order, and we had a tree like the one show in Figure 2.22. All nodes of the tree are potential *OpenMP* tasks because the programmer used a `task` construct for each node of the tree. But, the runtime may decide to aggregate them (the dashed areas in Figure 2.22) by not creating some of the user specified tasks and execute them serially.

2.4 The Message Passing Interface

The Message Passing Interface (*MPI*) [For95] is a library interface that provides a programming paradigm to parallelize applications. *MPI* follows an SPMD model (*Single Program Multiple Data*). When the application starts *MPI* allocates as many different processes as the user wants and all of them execute the same program but on different data. The programmer controls the distribution of work by means of primitives that allow to determine how many total process there are in a job (*mpi_comm_size*) and the process rank in that group (*mpi_comm_rank*). Once the application has started, unlike in *OpenMP*, the number of processes remains fixed until the end.So, it is said that *MPI* applications are not malleable (i.e. they cannot adapt to changes in the resource allocation of the application).

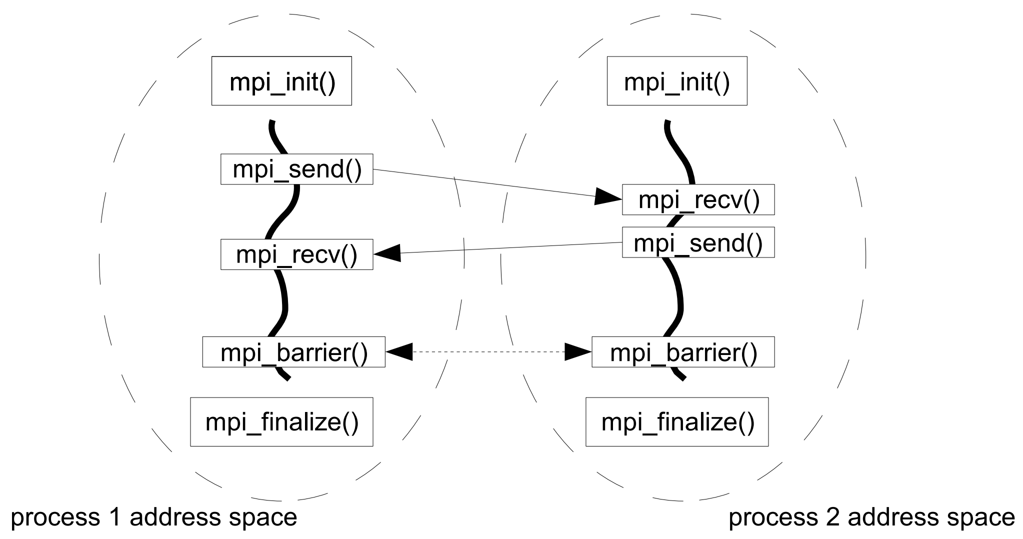


Figure 2.23: *MPI* execution model

Because these process can be in different nodes connected through a network the address spaces of each process is independent and it cannot be accessed from any of the other processes of the application. Consequently, *MPI* provides a set of primitives that allows the user to communicate data explicitly between the different processes. Some of these communication primitives are point-to-point between a pair of processes (both synchronous and asynchronous). Some others are collective operations in which multiple processes cooperate in the communication (e.g. to make a reduction operation). *MPI* also provides some synchronization primitives like barriers.

Figure 2.23 shows how *MPI* works. The *mpi_init* and *mpi_finalize* functions mark the beginning and the end of the *MPI* application and must be executed in each of the processes. The *mpi_send* primitive allows a process to send some data to another process. Each *mpi_send* must be paired with an *mpi_recv* in the receiving process. The *mpi_barrier* allows to establish a *rendezvous* point for all the processes of the application. In *MPI* applications, many times the synchronization time is spent in communication primitives rather than in more pure synchronization primitives like *mpi_barrier*.

Although *MPI* was conceived to be used in distributed computations across nodes without shared memory, it has been used in shared memory

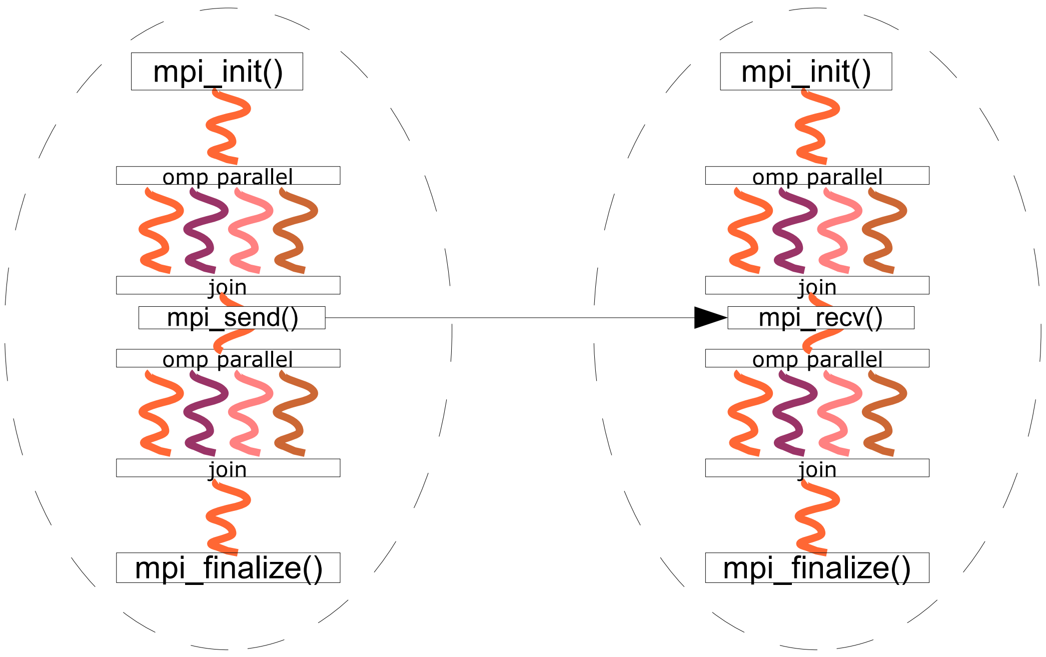


Figure 2.24: The *MPI+OpenMP* model

architectures with great success. This is because in these architectures the library is optimized to use shared memory for the communication instead of actually sending messages.

2.5 *MPI+OpenMP*

Another interesting way to obtain multiple levels of parallelism is using more than just one parallel programming paradigm to make a hybrid model. That is the case of the *MPI+OpenMP* model.

The main idea in this model is to use *MPI* to parallelize the outer level of parallelism and then use *OpenMP* to parallelize the inner level instead of using *OpenMP* for both as in the case of *OpenMP* nested parallelism.

Figure 2.24 shows the *MPI+OpenMP* execution model. When the application begins *MPI* processes are started as usual. Then, they encounter an *OpenMP* `parallel` region so each of the *MPI* processes will have *inside* a team of *OpenMP* threads that will work in parallel on the data of that *MPI* process. Note that *MPI* communication is still possible. Usually because of performance and thread-safety reasons the *MPI* communication is restricted to the serial thread. In *MPI+OpenMP*, while the number of *MPI* processes remains fixed during all the execution, the number of *OpenMP* threads that are used in the inner `parallel` regions can change from one to another. This gives *MPI+OpenMP* application a degree of malleability that we will be exploiting in Chapter 4.

This model is specially suited for clusters of SMPs, which have become increasely common over the last years. By using *MPI* in the outer level the application can be distributed across different nodes of a cluster. By using *OpenMP* in the inner level all the CPUs of each node can be used very easily.

This model can also be useful, as in the case of *OpenMP* when there is not enough work at *MPI* level but exploiting only the *OpenMP* level can have a very high overhead and, also, where load imbalance exists.

2.6 The *OpenMP* Runtime Library

To correctly understand what the runtime library is we need first some insight into the *OpenMP* compilation process. Figure 2.25 shows an scheme of this process.

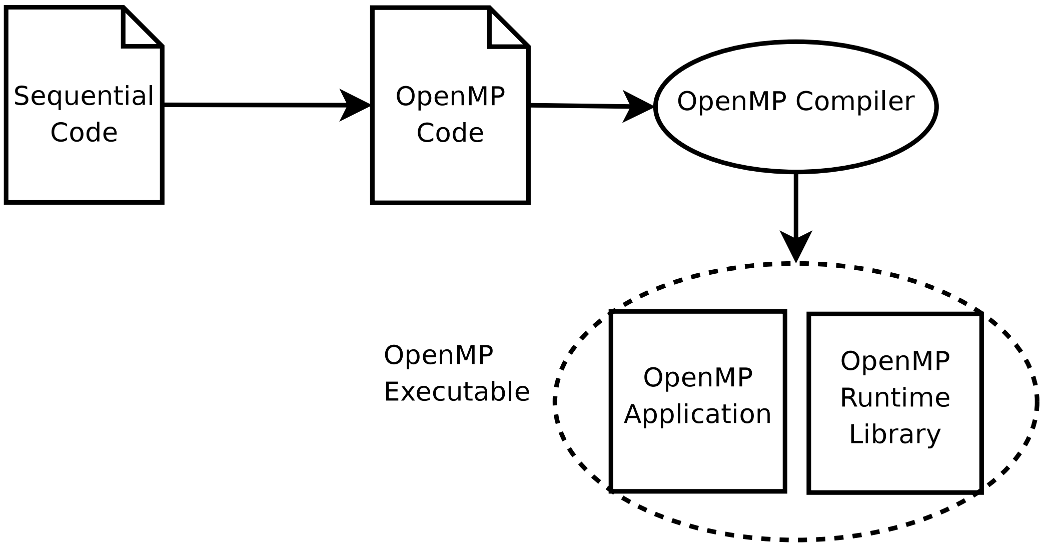


Figure 2.25: The *OpenMP* compilation and execution process

Once a user has parallelized his application, by means of annotating it with *OpenMP* directives, he has to use an *OpenMP* enabled compiler. This compiler will transform the sequential code into parallel code following the indications of these directives. These transformations often require the use of several different services ranging from very low-level ones (e.g locks, threads, barriers, ...) to high-level ones (e.g. worksharings). In Figure 2.27, we can see a very simple transformation that the compiler could generate for our *Hello World* example from Figure 2.26.

```
void foo () {
#pragma omp parallel
  printf(``Hello world\n'');
}
```

Figure 2.26: *OpenMP hello world* example

The exact transformation will vary from one *OpenMP* compiler to another, and it will usually be more complex than our simple example, but typically the code affected by the *OpenMP* directive will be outlined to a new function (*__foo1* in the example) and then the compiler will insert a call to a parallel primitive (*nth_parallel_region_* in the example) that implements the appropriate semantics of the *OpenMP* directive. So, there will be several of these primitives each related to some *OpenMP* directive.

All these primitives are commonly grouped into a library that it is distributed with the compiler and it is referred as the *OpenMP* Runtime Library (or just runtime library or even runtime). While the *OpenMP* compiler is in charge of the static semantics of the *OpenMP* application the runtime library is in charge of its dynamic semantics (i.e. the actual parallel execution of the

```
void __foo1 ( ) {
    printf(``Hello world\n'');
}

void foo () {
   nth_parallel_region_(__foo1 ,...);
}
```

Figure 2.27: Simplified transformation of an *OpenMP* program

program). Because of this, some of its core services are related to parallelism management (e.g. creation of threads, work distribution, ...). These runtime library services are invoked at runtime so they can potentially adjust their behavior to real time factors that were unknown at compile time (e.g. the input data, the system configuration, the system load, ...).

In our work we have built upon this property to design our self-tuned algorithms. We modify the appropriate parts of the runtime library so it adapts to information that is obtained at execution time. We used the IBM XL runtime library to implement all our proposals but the last one (i.e. the self-tuned task granularity cut-off). Because we were experimenting with *OpenMP* tasks, we used the only research *OpenMP* compiler that supported them at that time: the Nanos Mercurium compiler [BDG+04] and the Nanos runtime library that comes with it.

2.6.1 XL runtime library

The XL runtime library is the commercial *OpenMP* runtime library shipped by IBM with its compilers. *OpenMP* threads are implemented on top of *POSIX threads* (or pthreads).

These pthreads will poll a queue to find work to execute. In this queue the runtime library places information about the different parallel regions and worksharing that they have to execute [ZSA04].

The library offers different services (fork/join, barriers, worksharing creation, ...) that can provide the worksharing and structured parallelism expressed by the *OpenMP* standard.

2.6.2 Nanos runtime library

The Nanos runtime library is a research *OpenMP* runtime library that uses user-level threads based on the nano-threads programming model [Pol93, GAL+99]. These nano-threads are then scheduled on top of *POSIX threads* that are allocated when the program starts. This schema allows the runtime library to be very flexible in terms of associating nano-threads to pthreads.

The library offers different services (fork/join, synchronize, dependence control, environment queries, ...) that can provide the worksharing and structured parallelism expressed by the *OpenMP* standard.

We used it in on our work with *OpenMP* tasks as there was at the time no other runtime library that supported the new *OpenMP* task model.

We implemented the *OpenMP* tasks as nano-threads. Because the nano-threads can move from pthread to pthread this allows us to support task switching and experiment with both `tied` and `untied` tasks.

2.7 Application structure

To self-tune parameters of the applications we need information about those applications so the algorithms we design use this information to take the best decisions. We can obtain this information from three different sources:

1. First, we can provide an interface so the programmer can directly provide the information to our algorithms. Obviously, because one of our goals was to reduce the burden of the user we have avoided this option.

2. Then, the compiler could analyze the application and try to estimate different parameters (e.g. execution time, size of structures, ...). But as some information (e.g. size of input data, number of processors) is unknown at compile time this approach can only achieve so much.

3. Last, we can attach some profiling mechanism to the application. When the application runs the profiler will gather the information that will later be used to feed the decision algorithms.

We have opted to use the third possibility as it has the potential to adapt to changes and it avoids the need of user intervention. To profile and use this information there are two different options:

1. First, we can profile one (or more) executions for the purpose of obtaining information as a training. This information will be then used in later executions of the application. The main advantage of this

approach is that once we have done the training the profiling can be disabled and the applications get the benefit from it the *whole* execution lifetime. The drawback is that all the decisions are bound to the details of the training (input data, available resources, . . .) which can hinder the flexibility of the decisions.

2. Our other option is to profile all the executions and use the information in that same execution. With this option we can adapt to any scenario because the “training” occurs every time the application is executed. Of course, this is also a drawback because the profiling has to be carefully designed so the overhead does not become significant. The other important drawback of this approach is that we need that the information obtained from the profiler can be actually used. That means, that we need to find some repeatability in the application so the information we obtain from a small part of the execution can be used for the remaining part of the execution.

In this work we are going to focus in applications that have some part of the code executed multiple times either because it is inside a loop region or because it has some recursive pattern.

Figure 2.28 shows the typical pattern of an application with a repetitive behavior inside a loop. Usually there is an initialization phase, followed by a computation phase that is repeated multiple times and an optional finalization phase at the end. This kind of structure is very common in applications in the scientific domain [BBB+91, JdW06] that apply iterative methods to find the solution to a problem.

```
call initialization()
do step=1,nsteps
   call compute_step_i(step)
end do
call finalization()
```

Figure 2.28: Skeleton of an iterative application

Due to this iterative behavior, it is possible to gather information during the execution of the first iteration (or a few more if needed) of the loop and from that information we can change different parameters of the parallel execution (e.g. scheduling, number of threads, ...) so they are optimized for the execution of the following iterations of the loop.

Figure 2.29 shows a pattern that is common in many recursive algorithms. As in the previous case, there is usually an initialization phase at the beginning of the computation and finalization phase at the end. Then, the computation itself is called recursively until some condition is true. At each computation call several new instances of the computation function can be invoked.

```
void computation (void)
{
  if (final_condition) return;
  while (condition)
   computation();
}

void main ()
{
  initialization();
  computation();
  finalization();
}
```

Figure 2.29: Skeleton of a recursive application

This kind of application pattern generates an execution tree where there is a computation root and this one has some computation children and so on. What we can do in this case is gather information about the application while executing a branch of this tree so the runtime library can adapt is behavior to optimize the execution of the remaining branches of the tree.

Chapter 3

Self-tuned parallel loop scheduling

A learning experience is one of those things that says, 'You know that thing you just did? Don't do that.'

Douglas Adams

British writer (1952 – 2001)

Abstract

Parallel loops are one of the main sources of data parallelism but their performance depends on which iterations are executed by each of the threads (i.e. the loop scheduling). This chapter presents a mechanism for self-tuned parallel loop scheduling that we call `adjust`. The `adjust` scheduler works by first gathering information about the application that allows to characterize its loops to make a scheduling decision. Our evaluation of the `adjust` scheduler shows that in most cases it can achieve a decision as good as the one that would choose an advanced user.

3.1 Introduction

Parallel loops are the most important source of parallelism in numerical applications. *OpenMP*allows the exploitation of loop-level parallelism thorough the `do` work-sharing and `parallel do` constructs. Iterations are the work units that are distributed among threads as indicated in the `schedule` clause: `static`, `dynamic` and `guided` (all of them with or without the specification of a `chunk` size).

While in a `static` schedule the assignment of iterations to threads is defined before the computation of the loop starts, both `dynamic` and `guided` do the assignment dynamically as the work is being executed. With a `dynamic` schedule, threads get uniform chunks while with a `guided` schedule, chunks are progressively reduced in size in order to reduce scheduling overheads at the beginning of the loop and favor load balancing at the end.

Deciding the appropriate scheduling of iterations to threads may not be an easy task for the programmer, specially when it depends on dynamic issues, such as input data, or when memory behavior is highly dependent on the schedule applied. Load imbalance or high cache miss ratios, respectively, are usually symptoms of inappropriate iteration assignments. In *OpenMP*, the programmer can play with the predefined schedules mentioned above or embed its own scheduling strategy in the application code if none of them are appropriate. The `chunk` size (or number of contiguous iterations assigned to a thread) is a parameter that needs to be appropriately set in order to avoid non-friendly memory assignments of iterations and/or excessive runtime overheads in the process of getting work. Even worse, the decisions may depend on parameters of the target architecture (going against performance portability, one of the key issues in *OpenMP*).

In order to decide a schedule strategy, some simple rules of thumb are usually applied: `static` for those loops with good balance among iterations; unbalanced loops should use an interleaved schedule (`static` with `chunk`) or some sort of dynamic schedule (`dynamic` or `guided` depending on the loop shape). However, the use of dynamic schedules usually incurs in high scheduling overheads and its non-predictive behavior tends to degrade data locality (non-reuse of data across loops or multiples instances of the same loop). Although these rules work for a large number of simple cases, they are far from complete and can lead to poor decisions. Other schedules need to be built by the user, embedding code and data structures into the application to implement them.

We present a proposal to remove such burden from the programmer by letting the runtime library decide which is the most appropriate schedule for a given loop based on information gathered while the application is executing.

3.2 Motivation and related work

In order to motivate our proposal, we will consider a synthetic loop in which the cost of each iteration is $cost(i) = k/i$, k being a parameter that depends on the number of iterations of the loop. This function distributes almost all the weight of the loop to the first iterations (the first 1% of the iteration space accounts for 50% of the cost of the loop). During the execution, each thread accesses a matrix indexed with its thread identifier. Therefore, the loop only has temporal locality. The loop has been executed 500 times on a 4-way IBM Power4 system. Figure 3.1 shows the results obtained with different schedules (with $k = 100000$).

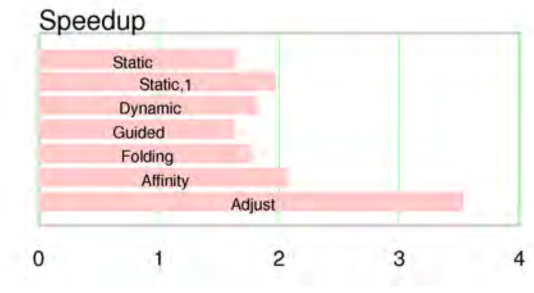


Figure 3.1: Speed-up with different schedules for a synthetic case (4 CPUs)

In this loop, using a `static` schedule leads to a highly unbalanced execution, with a speedup of 1.64 with respect to the execution with one thread.

Although the use of a `static` schedule with a chunk size of one increases the speedup to 1.97, it still does not achieve good balance. With $k = 10000$, the work of the first thread is 1.6 times the work of the fourth. Therefore, this is an example in which it seems appropriate to use either a `dynamic` or `guided` schedule. Using a `dynamic` schedule we get a speedup of 1.82 due to the high scheduling overheads and degradation of temporal locality. A `guided` schedule, which usually tends to reduce these scheduling overheads, is decreasing the speedup to 1.63; this is due to the fact that some threads get excessively large chunks at the beginning that are not well balanced with the remaining ones.

Other schedules, similar in nature to `guided`, have been proposed in the literature, such as trapezoid scheduling [TN93], factoring [SHF92] and tapering [Luc92]. These schedules are variations of the previous ones and are tailored for certain load unbalance patterns. Some other schedules try to take in account the geometric form of the iteration space. For example folding is a variation of `static` in which iterations i and $N - i$ are assigned to the same thread, N being the total number of iterations. For the previous synthetic example, this schedule is unable to improve the behavior achieving a speedup of 1.77. A study of the most suited schedule for different loops, grouped by their iteration execution time variance is presented in [YL94].

Other proposals try to achieve load balancing by applying work stealing. They usually assign iterations to threads in a `static`-like manner; in Affinity Scheduling (AS) [ML94] threads steal chunks of work from other processors as soon as they finish with the initially assigned work. This work stealing adds a dynamic part to the work assignment that does not favor memory behavior. Affinity Scheduling is available in the IBM *OpenMP* runtime library as a non-standard feature that can be specified in the `OMP_RUNTIME` environment variable. In our synthetic example, affinity scheduling achieves the highest speedup of all the available schedules (2.07). Dynamically partitioned affinity scheduling (DPAS) [SE94] learns from work stealing in order to derive a new `static`-like schedule, to be used in subsequent instances of the loop, in which each thread has a different chunk size. Other proposals such as Feedback Guided Dynamic Schedule (FGDS) [Bul98] and Feedback

Guided Load Balancing [DR00] avoid the dynamic part by simply measuring the amount of unbalance (without applying work stealing) and they derive a similar `static` schedule. Hamidzadeh and Lilja [HL94] used a different approach where a processor is reserved to compute partial schedules based on the load of each processor that are placed on the processors work queues. The obvious disadvantage of this approach is that a processor needs to be sacrificed which in small systems can be a problem.

Probably, the best schedule would be an ad-hoc schedule trying to reduce scheduling overheads, optimize load balancing, avoid false sharing or a combination of these issues that compensate them. However, in more complex cases than a synthetic case, it may be impossible for the the programmer to compute this ”ideal schedule” because it depends on variables only available at run-time (architecture, input data, interaction between loops or processes, . . .).

Our proposal advances one step further in the use of dynamically derived schedules and show how they can optimize the behavior of applications. The runtime library is able to characterize the execution of a loop and learn from past executions in the same run in order to gradually enhance the assignment of iterations to threads. The objective is to relieve the user from the task of deciding the best schedule for each loop and ideally lead to a performance close to hand-tuned application. For instance, in the same synthetic example described above, our proposed adaptive scheduler (called `adjust`) achieves an speedup of 3.53. The scheduling is achieved in a completely transparent way with no additional specification from the programmer in the source code.

New similar algorithms have been presented since our proposal [ABD$^+$03]. Zhang et al [ZBC$^+$04, ZV05] designed an adaptive scheduler specifically designed for hyper-threaded processors [hyp] where they could disable some of the threads in a core to obtain better performance. Tabirca et al. [TTYF04] presented some variants of feedback-guided schedules which are unclear to represent improvements over our work. Kejariwal et al. [KNP06] proposed an algorithm that adjust the chunk size of a schedule as the schedule progresses based on the variance of the iterations executed up to that point.

```
do step=1, Nsteps
!$omp parallel do
  do i=1, N
     f(i)
  end do
!$omp parallel do
  do i=1, N
     g(i)
  end do
end do
```

Figure 3.2: Common iterative pattern in scientific applications

3.3 Objective

Our goal in this work is to show that an efficient scheduler for parallel loops can be found by the runtime library with minimal (or no) information from the user and/or the compiler by means of gathering information, with run-time profiling, about the application that allows to characterize the application and that this process can be done with a reasonable (or even negligible) overhead.

Although the compiler could provide static information derived from the analysis of the source code or even could be provided by the user as hints, such as the initial schedule for the loop or the identifiers of other loops with similar memory access and/or workload patterns, in the worst case the information can only be gathered at run-time because the information about the loop may not be known at compilation time. So, we have decided to take the approach of only using run-time information to explore its feasibility.

We want to take advantage of the fact that, as we saw in Chapter 2, in most numerical applications the same loops are executed multiple times inside a *sequential time step* loop. This pattern (shown in 3.2) allows the runtime library to gather information in the first execution of the *time step* loop to characterize the application. Based on this characterization the runtime library is able to decide a good assignment of iterations to threads for all the parallel loops.

When deciding this assignment, the runtime library should try to maximize the following principles in order to improve the performance of the application loops:

Balance loops so that all threads get the same amount of work (i.e. computational work); this does not imply the same amount of iterations.

Preserve spatial locality by assigning contiguous chunks of iterations to the same thread whenever possible. This will optimize the access to memory containers (cache lines or pages) and reduce the likelihood of false sharing.

Preserve temporal locality by reusing the same schedule in subsequent execution instances of the same loop (or an affine one). This will favor data reuse.

It is important that even if schedule is already decided, the characterization process continues in order to detect further opportunities for refinement or changes in the behavior of the application (i.e. a change of phase in the execution). Since this information gathering could have a significant impact on performance and we want the profiling to be active all the time the runtime library should support different levels of profiling detail (and consequently of overhead). The most appropriate level of detail will be used depending on the status of the characterization process.

We present in the following sections an adaptive run-time scheduler, which we call `adjust` and the profiling mechanism that achieve these objectives.

3.4 Run-time profiling support

To implement an adaptive scheduler the runtime library needs to offer some support. In particular, two things are needed: the capability to dynamically profile parallel loops execution to obtain information and, being able to propagate that information to future similar instances of the same loop.

3.4.1 Profiling

In order to decide the most suitable schedule, the runtime library needs to collect information that characterizes the behavior of the loop that will be later passed to our `adjust` scheduler.

At run time, the profiler collects information about the iteration space (loop bounds and number of iterations) and about the amount of work of the iterations (i.e. execution time of the iterations).

The process of profiling the application is not free. The more detailed is the information we obtain, the higher it will be the impact of the profiling process in the performance of the application. Fortunately, the detail of information that the runtime library needs to take scheduling decisions is not always the same: the first time a loop is executed, or after detecting a high perturbation in its current characterization, is usually the only moment when we need very detailed information. But, once the runtime library detects a stable characterization (see Section 3.5), it could switch to a not very detailed profiling in order to minimize unnecessary overheads.

Our runtime library supports two different profile granularities modes: *thread granularity* and *subchunk granularity*. When *subchunk granularity* is used to avoid excessive overhead of measuring every single iteration[1], groups of iterations, which we call *subchunks*, are measured together. The size of these subchunks is variable for each thread and depends of the number of iterations that they have been assigned. That means that, if a thread has few iterations assigned each of its subchunks will represent very few iterations (it could even be one iteration per subchunk) but if the thread has a large number of iterations assigned the number of iterations each subchunk will represent will be large as well. In our current implementation we empirically chose to measure a maximum of 25 subchunks per thread.

When *thread granularity* is used, the measures are done for the overall iterations assigned to each thread (i.e. it is the same as if there was only one subchunk per thread) so the overhead is very low.

The profiling granularity mode is controlled by the adaptive scheduler (see Section 3.5) based on the state of the loop characterization before its execution but the objective is to use the *thread granularity* as often as possible to reduce the overheads to a minimum.

[1]Our initial test showed that measuring every iteration was prohibitively expensive

3.4.2 Loop information

The runtime library needs to provide some way to associate data for each loop in the application. To distinguish which loop is going to be executed we use the address of the outline routine generated by the compiler to encapsulate the loop. But, even for the same loop we want to distinguish different instances if they have different iteration spaces (i.e. lower and upper index of the loop[2]) because the scheduling decisions will be different for each iteration space, and therefore, we need to save the information separately. Furthermore, we want to have different schedules for the same loop in environments where the number of resources allocated to the application varies from time to time (e.g. multiprogrammed environments with resource management [CML05]). For this reason, we also use the number of CPUs allocated to the application when the loop is executed to find the appropriate scheduling information. This way we can have different schedules if the number of processors allocated to the application changes.

When the loop is going to be executed an identifier $\{F,IS,C\}$ is generated, where F is the address of the loop outlined function, IS is the iteration space and C is the number of CPUs assigned to the application. This identifier is used as index in a hash table to find the data associated to the loop in this particular execution.

The information that is kept for each loop $L_{\{\mathrm{F,IS,C}\}}$ is the following:

- The identifier $\{F,IS,C\}$.
- The execution time profiled for each of the subchunks (if *subchunk granularity* was used) or threads (if *thread granularity* was used) in the previous execution (if there was one).
- The balancing information about the loop (see Section 3.5).
- The relation of weights between iterations (see Section 3.5).
- The last schedule used.

[2]Loops are normalized so we do not need to take into account the loop step

- The best schedule so far (see Section 3.5).

This information is passed to the `adjust` scheduler.

3.5 The ADJUST loop scheduler

In this section, we describe the prototype implementation of a self-tuned scheduler for parallel loops that we call `adjust`.

The `adjust` scheduler uses the information gathered by the profiler to decide which iterations will execute each of the threads. The scheduler is invoked by the runtime library each time a parallel loop $L_{\{\mathrm{F,IS,C}\}}$ is about to be executed so it decides the schedule to use for that loop. It is also called when the parallel loop finalizes to process the profile information gathered by the runtime library during the execution of the loop.

3.5.1 The processing step

When a parallel loop finalizes its execution `adjust` is invoked to process the profiling information. To reduce the overhead of this step when `adjust` is invoked instead of doing the computation at that moment it enqueues the processing request in a global processing queue. When threads are idle (e.g. in a barrier) they dequeue the requests and process them. This allows to partially overlap the processing computation with regular thread synchronization. If when the parallel loop is going to be executed again the processing is still pending it is then executed at that point.

The processing step characterizes the loop based on the profile information and the state of the loop up to that moment. Two characteristics of the loop are discovered: the balance of the loop and the relation of weights between iterations.

The characterization of the balance of the loop is composed by three different parameters:

- First, a state that indicates if `adjust` found a schedule that resulted in a balanced execution of the loop $L_{\{\mathrm{F,IS,C}\}}$ and how confident `adjust` is

State	**Meaning**
Unknown	The runtime library has no balance information.
Unbalanced	The runtime library found that it is unable to balance the loop.
Balanced	The runtime library found a schedule that balanced the loop.
Highly balanced	The runtime library is confident that the schedule applied is balanced.

Table 3.1: Possible balance states

that it will be a balanced execution again in the future. We summarize the possible states and their meaning in Table 3.1.

- The number of consecutive executions that this balance state has been maintained for the loop. The higher this number is the higher is the confidence that `adjust` has that reusing past scheduling decision will lead to the same result.

- The actual definition of balance for this loop (i.e the percent of unbalance allowed). Instead of having a global threshold for all the loops one for each loop $L_{\{\mathrm{F,IS,C}\}}$ is kept. This allows to increase the threshold if we are confident with the decisions of a loop to avoid variations in one execution of the loop that are due small variations in the load of the system. Initially, the threshold it is set to a 10% of imbalance, but as `adjust` gains confidence it increases the limit to 20% and later to 25%.

If the execution time of each thread does not deviate from the average more than the actual definition of balance the execution is considered balanced, otherwise the execution is unbalanced. Using this information, a transition in the balance state automaton, shown in Figure 3.3, is performed.

Usually the initial balance state for a loop is the *Unkown* state (see next section for more details). After a successful balanced execution the state changes to the *Balanced* state. After N unsuccessful executions the state moves to the *Unbalanced* state. While in this state a single balanced execution, normally due to a change in the behavior of the loop, changes the state

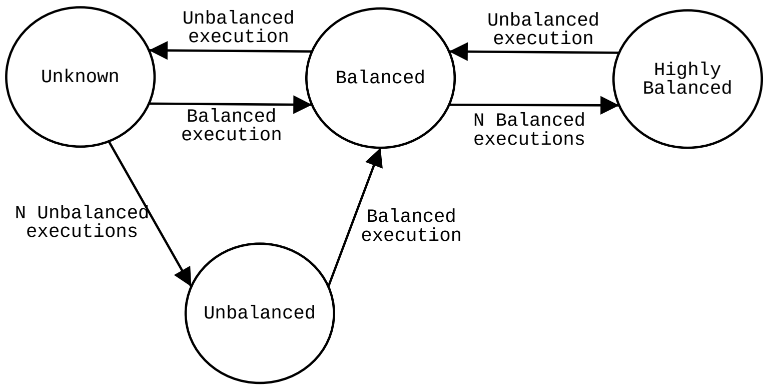


Figure 3.3: Balance information transitions

to the *Balanced* state. While in the *Balanced* state any imbalanced execution reverts the state to the *Unkown* state. After N successful balanced executions in the *Balance* state, the confidence on the decision increases and the state changes to the *Highly Balanced* state. An unbalanced execution in this last state reverts to the *Balanced* state. Note that when there is confidence (i.e. *Highly Balanced* state) in the balance of the loop one bad execution will not destroy it immediately but just move it to a state with less confidence (i.e. *Balanced* state), but a second bad execution in the next N will revert the state to *Unknown* meaning that `adjust` is not sure anymore and it needs to evaluate its decisions again. This gives some tolerance to perturbations while being able to adapt to changes in the behavior. We used 10 as the value for N in order to obtain a fast transition to the *Highly Balanced* state based on our observations of the behavior of the applications.

To find the relation of weights between iterations the average weights of the iterations executed by each thread is computed. If it does not deviate from a certain threshold the iterations are considered to be of *constant weight* otherwise the relation between the weights is considered to be *unknown*. In the current implementation only these two patterns are handled but others patterns could be recognized if their properties are useful for scheduling.

Finally, in the processing step, the schedule decision used is saved as the last schedule applied. If this schedule also happens to be the best schedule so far it is saved as such.

It is worth to mention that the processing step discards all the profiling information (and consequently it does no processing at all) the first time a scheduler is tried (either the initial one or after a change). The reason behind this is that those first samples can be misleading because the first time a schedule is used the number of cache misses, due to cold misses, is larger than in a regular execution where the cache is already settled. This cache effect will be reflected in the execution time of the iterations. Consequently, any scheduling decision taken based on those times will be misleading.

3.5.2 The scheduling decision

The first time that `adjust` is invoked by the runtime library there is no information about the loop so a new structure is allocated for it. First, it tries to find the most similar structure that corresponds to the same loop but with a different iteration space. If it finds one, then the initialization is inherited from it: it copies the balance information, the iteration information and a modified version of the schedule applied to the other iteration space (adding or subtracting iterations). If it cannot find one structure that is similar then it is initialized to the *Unknown* balance state and the hypothesis that all iterations have all the same weight is tried.

`Adjust` decides which schedule (and its parameters) to use for a given execution of a loop $L_{\{\mathrm{F,IS,C}\}}$ based on the current information for the loop (i.e. the number of iterations, available resources and past behavior). Currently, one of two the following schedules will be chosen:

- The standard *OpenMP* `static` schedule kind. This schedule kind is a good choice when the loop does not have load balance problems as it has good temporal and spatial locality.

- Non-uniform `static`. This schedule is similar to the previous one. Each thread is also assigned a chunk of contiguous iterations that are determined prior to the loop execution. However, chunks assigned to threads may be of different size. When the size of each chunk is properly chosen the load balance that is achieved can be as good as in any

Balance state	Iteration cost	Schedule
Unknown	Constant	Static
Unknown	Non-constant	Non-uniform Static
Unbalanced	*	Best schedule found
Other	*	Reuse previous

Table 3.2: Schedule decision function

dynamic schedule. But, the temporal and spatial localities that will be obtained will be as good as in the `static` case because of the assignment of contiguous iterations and schedule reuse. Therefore, this is a good choice for loops with load balance problems.

The scheduling decision is summarized in table 3.2. If the loop is considered to be balanced (i.e. in the *Balanced* or *Highly Balanced* states) the last schedule applied is reused to improve temporal locality. This schedule will be the one that previously obtained the balanced execution. If the loop is considered unbalanceable (i.e. in the *Unbalanced* state) the schedule will be the best schedule found which it will be used from there on. When nothing is known about the balance of the loop, either because a proper schedule that balances the loop has not yet been found or because it has not reached the threshold to give up, a `static` schedule will be used if the iterations weights are constant, otherwise a `non-uniform static` schedule will be used.

The assignment of iterations for each thread when the non-uniform static schedule is used works as follows:

1. We compute the amount of work that each thread should be assigned by dividing the total amount of work (i.e. the sum of all iterations work) by the number of threads.

2. Subchunks of iterations are assigned sequentially to the first thread until assigning a new subchunk to that thread will result in the amount of work for the thread being higher than the amount of work per thread we previously computed. In that case, the subchunk is split assuming all iterations in the subchunk have the same weight, and the number of

Balance state	Granularity used
Unknown	*subchunk granularity*
Other	*thread granularity*

Table 3.3: Granularity mode decision function

iterations assigned to the first thread are adjusted consequently. The remaining iterations of that subchunk will be assigned to other threads.

3. Next, we start assigning iterations to the second thread in the same way, and we continue doing so until we get to the last one.

4. All the remaining iterations are assigned to the last thread.

Also when the schedule is decided, `adjust` also decides how much information it will need in the future and it sets the granularity of the measurements that the runtime library will do while the loop executes. As, we explained in section 3.4 the runtime library supports two profiling modes: subchunk (fine granularity) and thread (coarse granularity).

The decision is based on the balance state of the loop as shown in table 3.3. The rationale is the following, if we are in the *Unknown* state `adjust` will need very detailed information in order to decide a new scheduler, but if we are in any other state that means that `adjust` will be reusing a previous scheduler and then it will only need enough information to know if the execution was well balanced or not. To know that is enough to have information about the execution time of each thread, so coarser measurements can be used.

3.6 Evaluation

3.6.1 Environment

We run all benchmarks in a p690 32-way Power4 [BBF$^+$01] machine at 1.1 GHz with 128 GB of RAM. We used the IBM's XLF compiler with the *-O3 -qipa=noobject -qsmp=omp* flags with a modified version of the XL runtime library that has the `adjust` scheduler implemented. The operating system was AIX version 5.1.

3.6.2 Methodology

In order to evaluate the proposed schedule, we have used some programs from the SPEComp suite [ADE+01] (*swim*, *ammp*, *gafort*, *apsi*, *wupwise* and *art*), the NAS *OpenMP* benchmarks [BBB+91] (*bt*, *ft*, *cg*, *sp* and *mg*) with the class A data input, and a computational kernel that calculates the *legendre* polynomial.

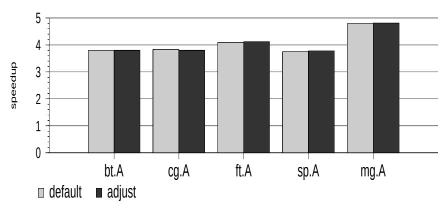
We compare, for each application, the regular version that comes with the *OpenMP* `schedule` clause from the developer with a version that uses our `adjust` schedule described in Section 3.5. Because, we could not modify the compiler we made use of the `runtime` schedule and then we set the environment variable `OMP_RUNTIME` to use the `adjust` schedule.

The *bt*, *ft*, *cg*, *sp*, *mg*, *swim*, *apsi* and *wupwise* programs use an `static` schedule in their main loops. The *ammp*, *gafort* and *art* applications use a `guided` schedule in their main loops. The *legendre* kernel has two triangular loops that are programmed with a “folding” schedule embedded in the application code but we also tried some *OpenMP* schedules to compare them. It is also interesting to note that in the *mg* application the amount of iterations of the loops changes from one time step to the next.

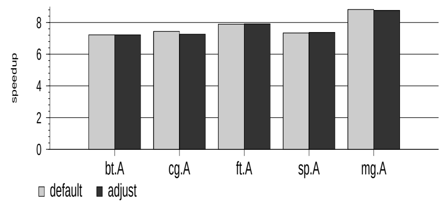
In all the cases, we used the average of several executions to compute the speed-up using the execution time of the serial version as the baseline compiled with the same optimizations as the parallel version.

3.6.3 Results

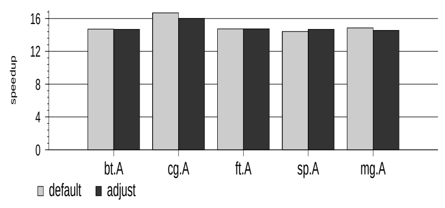
Figure 3.4 shows the results for the NAS benchmarks, on the x-axis are the different benchmarks and on the y-axis is the speedup achieved for the default schedule and for the `adjust` schedule. We can observe that the speed-up obtained with both schedules is almost equivalent (differences are always below 5%). This is explained because the NAS benchmarks parallel loops are well balanced and they have very good locality so there is not room for improvement with our `adjust` schedule. But, this also means that the `adjust` schedule is able to decide to use a `static` schedule for this type of loops, which are quite common, with a negligible overhead.



(a) 4 processors



(b) 8 processors



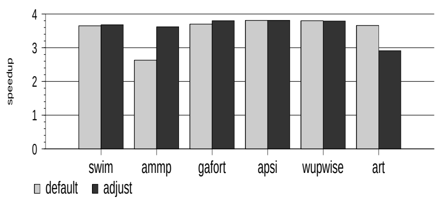
(c) 16 processors

Figure 3.4: Speedups for the NAS benchmarks

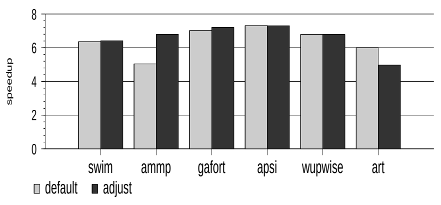
Figure 3.5 shows the results for the SPEComp benchmarks. We can observe, again, that in those applications (*swim*, *apsi* and *wupwise*) that use a `static` schedule the `adjust` schedule has no significant improvements but it decides to use the proper schedule without a significant overhead. In the case of *ammp*, the `adjust` schedule obtains an significant increase of the speed-up of the application (27% with 4 processors, 22% with 16 processors). The reason for this improvement is that the *non-uniform static* schedule computed by `adjust` has much better spatial and temporal locality than the `guided` schedule, because iterations are executed contiguously and the schedule is reused across executions of the same loop.

Figure 3.6 shows a paraver trace[3] [par] of an execution of the *ammp* application with the `adjust` scheduler. We can observe how the first execution of the main loop is heavily unbalanced, but afterwards the executions of that

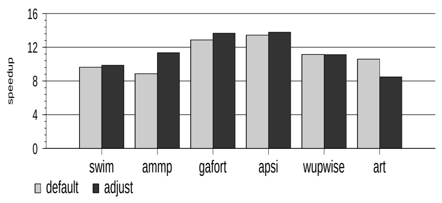
[3]In paraver traces, time is in the x-axis and threads are allocated in the y-axis. The black color represents time the thread is idle. All other colors represent execution of the different parallel loops of the application by a given thread.



(a) 4 processors



(b) 8 processors



(c) 16 processors

Figure 3.5: Speedups for the SPEComp benchmarks

loop are well balanced because `adjust` found a good `non-uniform static` schedule.

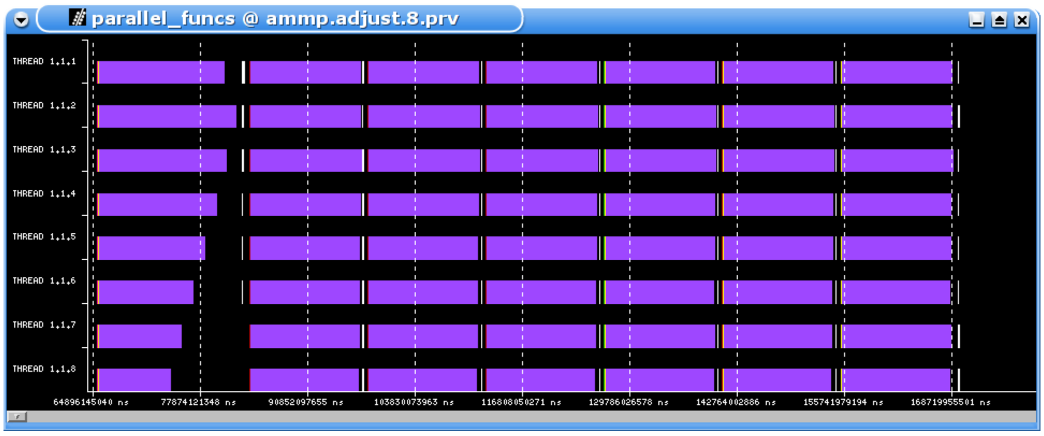
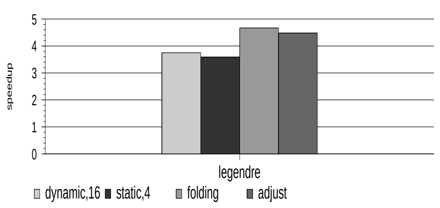
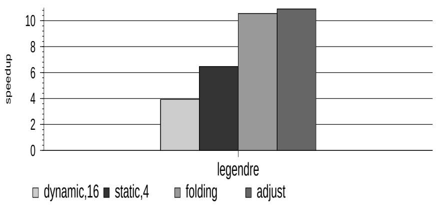


Figure 3.6: Trace of a execution of *ammp* with the `adjust` scheduler

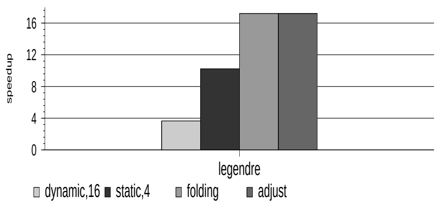
The improvement in the *gafort* is negligible (below 4%). Because *gafort* makes random vector accesses thus we do not obtain the locality gains seen in *ammp*. Even so, the `adjust` schedule has computed a schedule that has as good load balance as the `guided` schedule. The *art* benchmark is the worse case our method can find as there is just one loop executed once. Thus, we



(a) 4 processors



(b) 8 processors



(c) 16 processors

Figure 3.7: Speedups for the *legendre* kernel

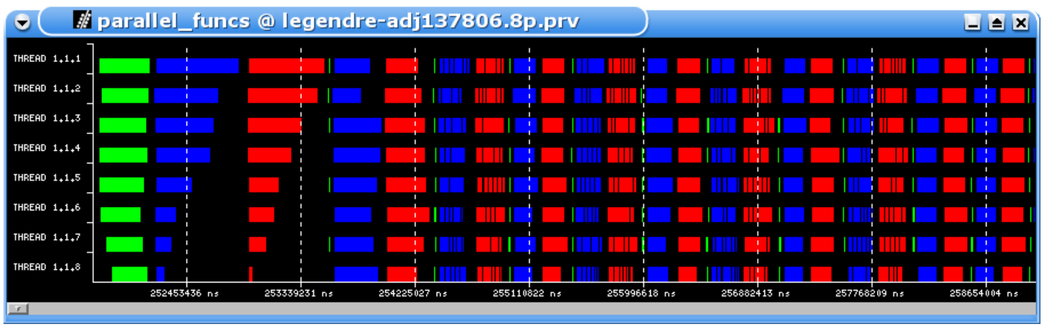


Figure 3.8: Trace of a *legendre* execution with the `adjust` scheduler

cannot make use of the knowledge obtained from that first execution because as we said, our method needs loops that are executed multiple times. To solve these cases a schedule that is more flexible with imbalanced codes than `static` should be used the first time (such as the Affinity schedule [ML94]), but currently the `guided` schedule performs a 20% better than our proposal.

In figure 3.7, we can observe the results for the *legendre* kernel. In addition to the default folding schedule encoded in the application and our `adjust` schedule, we tried several *OpenMP* schedules and we present the best ones that we found: a `dynamic,16` schedule for 4 processors and an `static,4` schedule for 8 and 16 processors. Note that one user that decided to use `dynamic,16` as schedule after some basic analysis in a small system would be fooled if later he would run it in a bigger production system. Figure 3.8 shows a paraver trace of an execution of the *legendre* kernel with the `adjust` scheduler. We can observe that the are two triangular loops whose first execution is heavily unbalanced but after the first execution of each one `adjust` is able to find a good `non-uniform static` schedule that balances both loops. This leads to a performance close to the *hardcoded* folding schedule (differences are below 5%) and much better than the best *OpenMP* schedule (improvements range from 16% with 4 processors to 41% with 16 processors).

3.7 Influence in the *OpenMP* standard

This proposal was submitted to the *OpenMP* Language committee as feature request and in the upcoming *OpenMP* 3.0 specification [Org08] there will be a new schedule kind called `auto`. When user selects this schedule kind for a loop he is specifying that the implementation should choose the schedule kind for the loop. This selection could be done by means of any combination of compiler and/or runtime library techniques. So, the `auto` schedule kind allows to expose to the user the kind of self-tuned scheduler that we have presented in this chapter.

A self-tuned scheduler can also be chosen by an implementation as the default scheduler when a user does not specify a `schedule` clause for a parallel loop. In this way, an *OpenMP* implementation can choose to use a self-tuned scheduler as a default to help novel users to obtain better performance with less effort. And, if a user spends time analyzing the application he can still specify through the `schedule` clause an specific schedule disabling the self-tuning mechanism.

3.8 Conclusions

In this chapter, we have presented the `adjust` scheduler. The `adjust` scheduler is a feedback guided scheduler that adaptively decides a scheduling policy for each of the loops of an application.

The `adjust` scheduler from the programmer (or user) removes the burden of deciding which scheduler is the best one for the parallel loops in his application.

Our evaluation showed that is able to do so in all the benchmarks we used but one (*art*). With `adjust`, loops that would require the use of `static`, `dynamic`, `guided` or even other schedule kinds not available in *OpenMP*, such as folding, can be efficiently executed.

Our algorithm main drawback is that in the first execution of a loop, because we decided to take a pure run-time approach, `adjust` has no information an can lead to a very bad decision. In the worst case, if the loop is executed just once there is no way to mend this decision because our technique requires that loop is executed multiple times in order to take advantage of the profiling.

Several approaches can be used to minimize the problem. First, `adjust` could use as initial data, instead of a predetermined `static` schedule, the one that the programmer specified through the *OpenMP* `schedule` clause. Another possibility is that the compiler generates information for `adjust` based on code analysis. These information would be used only as a hint in the first execution. Afterwards, `adjust` would decide based on the profile information gathered during the first execution of the loop. Another possibility would be to use information from the first iterations of the loop to dynamically adapt the scheduler on the first execution of the loop.

Another possible way to improve our algorithm could be to adopt the characterization for a another loop (for which a characterization has been done) and make fine-grain measurements. This characterization reuse may be important in order to reduce the time required to reach a stable characterization state. Reuse hints could be provided by the programmer or the compiler (e.g. providing information about affine loops). It could even be possible that the runtime library discovers affinity relationships between loops (i.e. loops whose characterization is the same or change in the same way) that are executed inside an iterative sequential loop.

Chapter 4

Self-tuned balanced thread allocation

Fere libenter homines id quod volunt credunt

Gaius Julius Caesar
Roman statesman and general
(100 BC - 44 BC)

Abstract

One of the limiting factors to obtain good performance when using multilevel parallelization schemes, like *MPI+OpenMP* or *OpenMP* nested parallelism, is avoiding load balance issues. This can be accomplished defining a good thread distribution that balances the total computational power across the *outer* level of parallelism. In this chapter we present a self-tuned technique that obtains information about the computational load of the application and based on that information computes a thread distribution that balances the application correctly.

4.1 Introduction

When parallelizing an application, the use of some sort of nested parallelism paradigm is becoming increasingly common. Nested parallelism is used mainly to increase the amount of available parallelism so applications can scale-up beyond a small number of processors (otherwise some processors would be idle) [JJY+03].

Usually the programmer accomplishes this by either using a mixed model like *MPI+OpenMP* (where the user launches several *MPI* instances and each of them will create *OpenMP* threads) or a more pure model like nested parallelism in *OpenMP* (where each of the threads of the initial parallel region creates an inner parallel region).

A common problem with nested applications is deciding how to distribute all the available threads across the different inner regions (as it is important to avoid threads being reused in multiple regions). Although some extensions have been proposed to the *OpenMP* standard [GOM+01] currently the user needs to do this grouping manually by means of the `num_threads` clause as shown in Figure 4.1 for the case of nested *OpenMP* parallelism.

```
#pragma omp parallel num_threads(ngroups)
{
int g = omp_get_thread_num();
#pragma omp parallel num_threads(group_size[g])
  work()
}
```

Figure 4.1: Nested *OpenMP* example with manual allocation of threads

A bad decision can lead to severe load imbalance in the application execution. To make a good decision the user needs to know how the data of the application is going to be distributed across the different groups of processors. This requires the user to do an analysis before choosing an allocation of threads to the different parallel regions. This analysis will need to be done each time the data input of the application or the number of available processors changes because the distribution usually needs to be different.

To make the decision of the allocation of threads automatically we present two algorithms: `DPB` (Dynamic Processor Balancing) and `DWB` (Dynamic

Weight Balancing). Both algorithms gather information at runtime of the computational load assigned to each outer parallel region and based on that information they compute a new allocation of threads in the inner regions that maximizes the balance of the application.

4.2 Motivation and Related Work

We can analyze which are the limiting factors of single level parallelizations with the help of a simplified version of the structure of *BT-MZ*, one of the codes included in the NAS multizone benchmarks [JdW06], parallelized with *OpenMP*.

Figure 4.2 shows the *BT-MZ* code that performs a computation over a *blocked* data structure. For each block of data (or zone), some work is performed in a subroutine called *adi*. A first level of parallelism appears since all zones can be computed in parallel. This corresponds to a *block* level parallelism and is coded by the parallelizing directives `parallel` and `do`. A `static` work distribution will be performed among the threads that execute this first level of parallelism. The definition of another level of parallelism is possible: the computation performed in each zone of data is organized in the form of a parallel loop. The code in subroutine *adi* contains a parallelizing directive for this loop. For this second level of parallelism, a `static` scheduling is also set. A *time step* loop encloses the overall computation and at the end of each iteration, there is some data movement to update zone boundaries. This iterative structure is common in most numerical applications and it is necessary in any technique that dynamically improves the behavior of an application based on its past behavior.

The programmer can choose between two parallelization strategies that exploit a single level of parallelism. The first one exploits the inter–zone parallelism. We will call it the *outer* version. The second one exploits the intra–zone parallelism. This is the *inner* version. The performance for the *outer* version is clearly limited by the number of zones. Using more threads than the number of zones does not contribute to improve its performance.

In addition, zones may have different sizes and lead to an unbalanced

```
  ...
  do step = 1, niters
     ...
C Inter-zone parallelism
!$OMP PARALLEL NUM_THREADS(num_groups)
!$OMP DO SCHEDULE(static)
     do zone = 1, num_zones
       CALL adi ( zone, ...)
     end do
!$OMP END DO
!$OMP END PARALLEL
     ...
C Update zone boundaries
     ...
  end do
  ...
  end

  subroutine adi ( zone_id, ... )

C Intra-zone parallelism
!$OMP PARALLEL NUM_THREADS(zone_threads(zone_id))
!$OMP DO SCHEDULE(static)
  do j = 1, k_size(zone_id)
     ...
  end do
!$OMP END DO
!$OMP END PARALLEL
  end
```

Figure 4.2: Main structure of *BT-MZ* with *OpenMP* nested parallelism.

execution in which some of the threads waste execution cycles waiting for the others to finish their work. In case of important size differences, the unbalance degree might cause a noticeable performance degradation. So, two main factors are limiting the performance of the *outer* version:

- The relation between the number of available threads and the number of zones.
- The unbalance degree expressed through the size differences between the zones.

In both cases, the *outer* version will not increase its performance when an important number of threads (32 or more) are available. Note that these factors are independent of whether the *outer* level is coded with with *OpenMP*, as in the example, or another language as *MPI*.

Other issues limit the *inner* version performance. The most important issue is granularity. Creating the work in this level of parallelism among a large number of threads might cause that the work assigned to each thread is too small to take profit from the parallel execution. The finest granularity that can be exploited is conditioned by the runtime overheads related to parallelism creation and termination, work distribution and thread synchronizations. These overheads are going to be noticeable when executing with a large number of threads (again, 32 or more). These problems where already noted by Jin et al. [JJJT03]. They showed the different overheads between parallelizing the outer or the inner loop of a Cloud Modeling Code.

Several works have tried to overcome these problems by employing a multilevel parallelization instead of just parallelizing the *outer* or the *inner* level.

Several research works [Smi00, Hen00, SSOB02, CE00, DMSC99, BYS+01, KMB00, aMS00, MKB00, Smi99, LR99] present the possibility of mixing more than one programming model for exploiting nested parallelism. Typically, applications, which execute under such parallel strategy, define a first level of parallelism using a distributed memory paradigm, like *MPI*, plus a second level of parallelism implemented with shared memory model, usually programmed with *OpenMP*.

Other researchers have worked with nested parallelism in a pure *OpenMP* model [TTSY00, GAM+02, AGMJ04, AML+99, BS03, Bli02]. In this case, both levels of parallelism are exploited with *OpenMP* parallel regions.

The results from these experiments show that, although not in all the cases [Hen00, CE00, SSOB02], in most of them multilevel parallelism strategies perform better than conventional single level strategies. This is particularly true in applications with load balancing problems [DMSC99, AML+99].

But the use of nested parallelism is not with its own problems. By exploiting both levels of parallelism (inner and outer levels), nothing is gained, unless threads are arranged in a way that avoids the grain size and work unbalance problems. Regarding the grain size problem, the only solution is to forbid that all available threads execute the inner level of parallelism. So, thread clustering strategies are the solution. The main idea is to create, for each block or zone, an different set of threads that will execute the work defined at the inner level of parallelism. The NUM_THREADS clause in the source code is used for that purpose.

This strategy solves the grain size problem, but it does nothing to solve the possible work unbalance created in the outer level of parallelism. A common approach has been establishing some method to partition the data across the different threads so all have the same load [SKK00, BFL+01]. The problem of this solution is that is often tailored to the specific data of the application.

Another option, presented by Huang and Tafti [HT99, Taf], is defining the thread sets that balance the computational power (i.e. the total number of threads) according to the amount of work assigned to each parallel branch in the outer level [BS99, GOM+01, AGMJ04]. In the example, this is done by having different values for the argument of the `NUM_THREADS` clause, depending on the *zone* that a set of threads is going to work on.

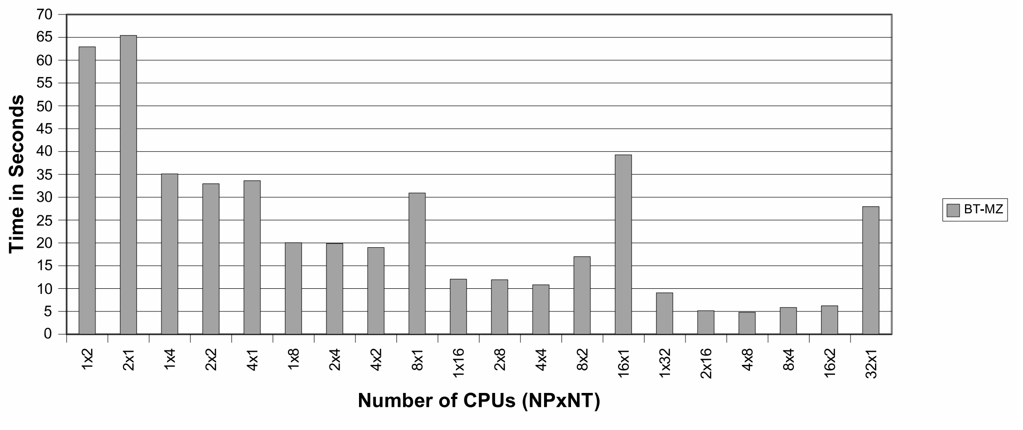


Figure 4.3: *BT-MZ* class A execution times on a IBM Regatta

To illustrate the impact of this thread clustering strategy, Figure 4.3 shows how the execution time of the *BT-MZ* application changes with different allocations of threads in the outer level (NP) and inner level (NT). NT represents the average value as its exact value is different for each zone since their have different sizes in this application. For each number of processors (2, 4, 8, 16 or 32) the best parallelization strategy is different. This difficulty in determining the most appropriate thread distribution between the levels of parallelism is the main motivation for the work in this Chapter.

The thread grouping mechanism was studied and organized as a proposal to extend the *OpenMP* language by Gonzàlez et al. [GOM+01,JJY+03]: some constructs were defined to allow the programmer specify the most appropriate thread distribution between the levels of parallelism. Also, this proposal presented an optimal algorithm to compute the thread distribution. But this work relies on the programmer providing the required information: the number of branches in the outermost parallel level and the computational weight associated to each branch.

Our proposal is to rely on runtime mechanisms to automatically derive the optimal thread distribution. Measurements are obtained at runtime as an approximation of the computational weight of each nest of parallelism. From those weights we show how it is possible to obtain a suitable thread distribution for nesting parallelism exploitation that avoids the problem of imbalance.

4.3 Objective

The parallelism that exists in applications with multiple levels of parallelism, particularly when is only two levels, can be seen as groups of threads. The number of groups is the number of threads (or processes) in the outer level of parallelism. This is specified by the user either with a `num_threads` clause in the case of nested *OpenMP* or through the *MPI* launcher in the *MPI+OpenMP* case. The number of threads in each group is determined by the number of threads in each of the inner levels of parallelism. The threads in each of the groups are different so the sum of the threads of the groups cannot be greater than the number of total threads.

So, a thread distribution is defined by: the number of groups in the outer level of parallelism and the number of threads in each group in the inner level of parallelism. These factors plus the distribution of work among threads determine the appropriateness of a thread distribution. Depending on the unbalance of the work distribution a thread distribution will or will not succeed in improving the performance of the application. If the thread

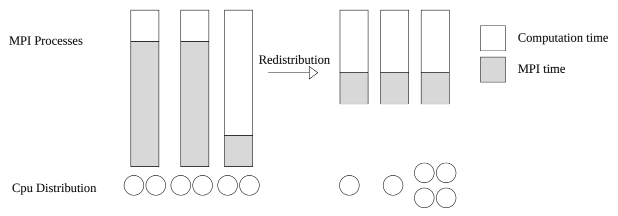


Figure 4.4: CPU redistribution example (*MPI+OpenMP* application)

allocation for the inner levels of parallelism is not in concordance with the work distribution work unbalance will persist. Thus, three elements limit the benefits that can be obtained from a thread distribution: the number of outer groups, the number of threads assigned to each group, and the work distribution.

Our proposal focuses only on tuning the thread distribution for the inner levels. Our runtime implementation is not going to adjust the number of outer groups or the work distribution. These two factors will be specified in the application by the programmer using the appropriate *OpenMP* directives. Given a work distribution and a number of groups the runtime will be able to derive the best thread assignment for the inner level. The only requisite for our implementation is that the program behaves in a iterative manner to be able to adapt the number of threads in each group after gathering some information about the application.

Figure 4.4 illustrates our objective. In this example, there is one group with more computation time than the others. This causes the global execution time to increase as the other two groups (the ones in the left) spend their time waiting at synchronization points (send/receive, barriers, . . .). In this case, our goal is to take the decision of redistributing the processors so we take one processor away from both of the groups on the left and give them to the one on the right. This decision makes the two victims go slower but the application execution time is reduced due to a better utilization of the resources.

4.4 Adaptive Thread Allocation

4.4.1 General design

Initially the runtime library assumes that all groups have the same amount of work and distributes all the available threads uniformly among them. Then, the runtime library gathers information through a dynamic profiler that allows to infer how the computation is distributed among the groups. This distribution of work will in turn be translated to changes to the thread distribution: increasing the number of threads to those groups with more work and decreasing the number of threads to those with less work.

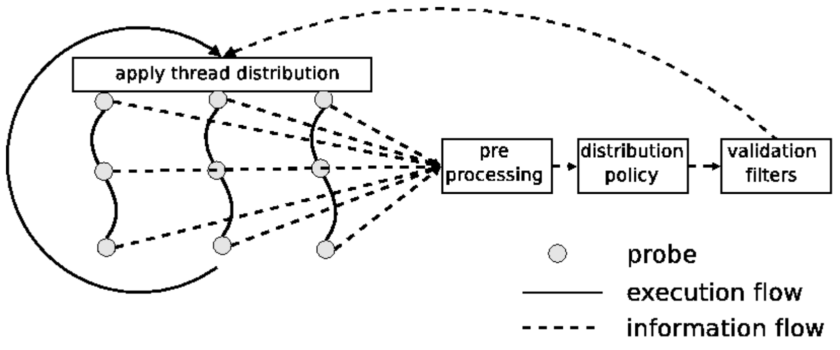


Figure 4.5: Framework design overview.

Figure 4.5 shows the design of our general framework for computing the optimal size of each group. It is a typical feedback guided scenario in which several phases can be distinguished:

- First, several profile probes are placed in the runtime library to obtain information about the application. This information is processed to represent the work load of each of the groups of threads of the application.
- When enough information is gathered the *distribution policy* is invoked to compute a new distribution of threads based on the data that was obtained by the profiler.
- Last, the distribution is passed through a series of filters that validate that the new distribution will yield some benefit. A function predicting

the benefit of the new distribution is used to avoid unnecessary changes in the distribution. Notice that the lack of such mechanisms would leave the system open to undesirable effects (e.g.constantly moving threads across the groups or even thread ping-pong).

We assume, that for each nest of parallel regions where the runtime will automatically compute a thread distribution the programmer previously defined a number of groups and a work distribution schema.

4.4.2 Run-time profiling support

To implement our adaptive method for thread distribution we need to obtain information about the computational weight of the different groups of threads in the outer level of parallelism. Thus, we need to dynamically profile the application to obtain this information.

Because *OpenMP* and *MPI* have some differences the profiling strategy needs to be slightly different depending if the outer level is one or the other.

Profiling OpenMP

We want to profile each of the outer regions of the *OpenMP* nest to obtain the computation time of each group of threads.

Our runtime system places time probes at the beginning and at the end of the execution of each thread in an outer parallel region. This allows for a good approximation of the work each group of threads does, unless the runtime system introduces unreasonable overheads that distort the measurements. Notice that the way the probes are placed make the accumulated time include all the overheads related to the inner parallel regions (e.g. thread creation/termination, barrier synchronizations, load imbalance, . . .). We expect these overheads to be low enough so they do not interfere in the sampling.

A better, but more complex, approach would be placing the probes at the beginning of the worksharing constructs and before the implicit barrier at the end of each worksharing construct in the inner regions. Accumulating the time in each worksharing we would obtain a good sample of the amount of work of each group of threads.

The information gathered by the profiler is associated to a region so each the outer regions can potentially have a different distribution of threads.

Profiling *MPI*

In contrast to *OpenMP* in *MPI* there is no concept of region as all the threads are always executing code. But, we need to define a region so we can sample the work load of each of the outer groups (i.e. each *MPI* process).

As we have seen previously in Chapter 2, many scientific applications have the characteristic that they are iterative, that is, they apply the same algorithm several times to the same data. We exploit this characteristic to accumulate meaningful times for computation and communication usage.

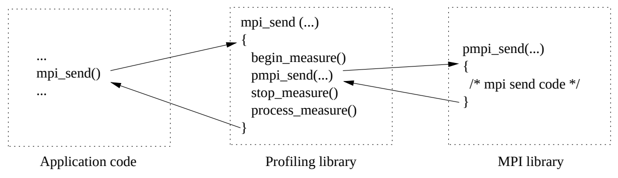


Figure 4.6: *MPI* profiling mechanism.

MPI defines a standard mechanism to instrument *MPI* applications that consists of providing a new interface that it is called before the real *MPI* interface [For95]. Figure 4.6 shows how the standard *MPI* profiling mechanism works. The application is instrumented using this profiling mechanism. When a *MPI* call is invoked from the application the library measures the time spent in the call and add its to a total count of time spent in *MPI* calls.

The iterative structure of the application is detected using a Dynamic Periodicity Detector library (DPD) [FCL01]. DPD is called from the instrumented *MPI* call and it is fed with a value that is a composition of the *MPI* primitive type (send, receive, . . .), the destination process and the buffer address. With this value DPD will try detect the patterns of periodic behavior in the application. Each period that is detected is a different region similar to the *OpenMP* ones. Our profiler will keep track of the time each of the *MPI* processes spent executing in the region and how much time each process in *MPI* synchronization (either explicit or implicit like in blocking communication calls).

Information preprocessing

A few measures from the profiler for a single region are averaged together before further processing to avoid an unexpected interference of the system to have a big impact in the characterization process.

Then, the execution time measurements are normalized to the minimum sample. That is, the runtime finds the thread with the minimum execution time and it divides the rest of values by this one with an integer division. This normalization erases the small variations of the sampling process giving a more meaningful collection of computational weights for each group of threads.

All the information is then passed to the distribution policy so it can compute, if needed, a new distribution.

In this step, the runtime could potentially compute additional metrics from the sampled data, like the degree of unbalance, that could be useful to future distribution policies.

4.4.3 Thread Distribution Policies

When enough information is gathered by the profiler a *thread distribution policy* is invoked. With this information the goal of the policy is to generate a processor distribution where all the groups of threads spend the same amount of time in computation, reducing the synchronization time as much as possible.

We have developed two different policies to achieve this goal:

Dynamic Processor Balancing

Each time the Dynamic Processor Balancing (DPB) policy is invoked it tries to improve one of the groups of threads of the application: the one with highest computation time. It does so by increasing its thread allocation with threads that are stolen from another group (which we call the victim group).

The victim group is the group with the minimum computation time of those that have more than one thread (every group has always at least one thread).

Once the victim is selected, we compute an ideal execution time for the group we want to improve, t_{i+1}, with the formula:

$$t_{i+1}(highest) = t_i(highest) - t_i(sync)/2 \tag{4.1}$$

Where $t_i(highest)$ represents the execution time of the group we want to improve in the last execution of the region and $t_i(sync)$ the time that was lost due to synchronization. This time t_{i+1} is what we want to achieve with the new distribution of threads. In each step we expect to balance both groups which means reducing the time distance between them (i.e the synchronization time) by half.

The value $t_i(sync)$ is computed differently if the outer group uses *OpenMP* or MPI:

In *MPI* this is computed by subtracting the time spent in *MPI* calls by group that spent most time, which should be the same we are trying to improve, to the victim group:

$$t_i(sync) = t_{mpi}(victim) - t_{mpi}(highest) \tag{4.2}$$

This heuristic assumes that the *MPI* time of the process that has more computation time is the minimum *MPI* time that any of the process can have.

In *OpenMP* we compute the synchronization time as the time spent in the barriers, which can be computed subtracting the execution time of the group that spent less time in the region to the group that spent the most time:

$$t_i(sync) = t_i(highest) - t_i(lowest) \tag{4.3}$$

With this *future* time t_{i+1} the number of threads that should be moved between the groups to obtain that time based on the last execution time, is computed with the following formula:

$$threads = \frac{threads_i(highest) * t_i(highest)}{t_{i+1}(highest)} - threads_i(highest) \qquad (4.4)$$

Where $threads_i(highest)$ is the current number of threads allocated to the group we want to improve. If the *threads* computed value is equal to all the threads of the victim group then all but one are stolen because every group needs at least one thread.

One problem with this policy is that because it only interacts with two groups at a time it make take several invocations before an stable thread distribution is found. Because of that, we developed the *Dynamic Weight Balancing* policy.

Dynamic Weight Balancing

The Dynamic Weight Balancing strategy is based on the work from Gonzàlez et al. [GOM+01] where the user feeds the algorithm by means of annotations in the code that specify a factor representing how much work each group of threads has. We use largely the same algorithm but using as input the the measures gathered by the profiler of the runtime library .

```
min=mininum(samples)
for i = 1 to ngroups
    weight(i) = samples(i)/min
    howmany(i) = 1
end for
while ( sum(howmany) < num_threads )
    find i such as weight(i)/howmany(i) is maximum
    howmany(i) = howmany(i) + 1
end while
```

Figure 4.7: DWB thread distribution algorithm.

Figure 4.7 shows the pseudocode of the algorithm. The variable **weight** contains the proportion of the work load that each group has computed from the profiler samples (**samples** variable). The variable **ngroups** refers to the number of groups defined in the outer level of parallelism. The variable **howmany** specifies the number of threads to be used for the execution of the parallelism in the inner levels (i.e the size of each group). The variable **num_threads** refers to the total number of threads available.

First, the algorithm assigns one thread per *group*. This ensures that at least one thread is assigned for the execution of the inner levels of parallelism that each *group* encounters. After that, the rest of threads are distributed according to the proportions in vector **weight** so the groups with a higher work load get more threads than those with a lower work load.

DWB in contrast to *DPB*, can compute a good distribution of threads in a single invocation of the policy. Also, note that because in each invocation threads from multiple groups can be re-allocated the impact from moving threads between groups can be higher with this policy.

4.4.4 Validation of a thread distribution

After the policy computes a thread distribution a number of filters are used to validate that we will be obtain a benefit after applying the new distribution.

Critical path validation filter Moving threads is not free. There is some penalty mainly because of data movement across caches. This filter discards those cases where moving threads between groups will not result in a performance increment that overcomes the penalty of the movement.

```
threshold= number between 0 and 1
find maxgroup such as samples(maxgroup)/howmany(maxgroup) is maximum

new_critical_path=samples(maxgroup)/howmany(maxgroup)
if ( new_critical_path < prev_critical_path and
     (new_critical_path - prev_critical_path) >
     (threshold * prev_critical_path) ) then
  return true
else
  return false
fi
```

Figure 4.8: Critical path validation algorithm.

Using the time samples and the new thread distribution, this filter computes an estimation of the critical path that would result if the new distribution was applied. The time samples are divided by the number of assigned threads in the new distribution. If the maximum value of these divisions (i.e. the critical path) is greater than the one obtained with the current thread distribution the distribution discarded. It is also discarded if the time difference is below a certain threshold. Figure 4.8 shows the described algorithm pseudocode. We suggest 5% as the threshold value.

Ping-pong effect This filter is in charge on detecting ping-pong situations, where a number of distributions are applied cyclically without any real gain. This filter uses the history of computed thread distributions to detect this situation. When the filter detects the ping-pong effect it chooses the distribution that worked best and it filters out any other.

More threads than chunks of work anomaly The distribution algorithm is not aware of the number of chunks of work that will be defined in the inner levels of parallelism. The parallel regions in the inner levels might offer different degrees of parallelism translated to how chunks of work are distributed among the threads. It is possible that one parallel region offers enough chunks so all the threads can work, while others do not. Our implementation is sensitive to this effect. But if this situation occurred it will be reflected in the measurements and the thread distribution algorithm will solve it.

4.4.5 Applying the new thread distribution

When the policy decides a new allocation the runtime library informs each group of threads of their new processor availability by leaving the new information in a shared memory zone of the process associated to the region. It also resets the profile information for that region so new data is gathered when it is executed again.

After that, the *OpenMP* runtime library should adjust the parallelism level (i.e. number of running threads) of the inner parallel regions to comply with the policy decision.

From the application point of view this can be done in two ways:

Non-preemptively Two synchronization points are defined at the entrance and exit of the inner parallel regions. When an application arrives at a synchronization point, it checks for changes in its allocation and adjusts its resources adequately. So, this means that while the group is inside a parallel region could potentially run with more (or less) resources than those actually allocated to it.

Preemptively In this version, the runtime library does not wait for the application to make the changes but it preempts immediately the processors stopping the running threads on them. As this can happen inside a parallel region, the runtime library needs the capability to recover the work that was doing or it has assigned to that thread in order to exit the region. This is not an easy task as available resources may change several times inside a parallel region leading to deadlocks if not carefully planed. Further information of this approach can be found in the work of Martorell et al. [MCN$^+$00].

Our implementation, in IBM's XL library, uses the first approach. As the parallel regions our work focuses are small enough, the time a process does not comply the allocation is small enough that there are no meaningful differences between the results obtained with both approaches.

4.5 Evaluation

We used two synthetic applications and two benchmarks from the NAS Multizone benchmark suite to evaluate the Dynamic Processor Balancing and Dynamic Weight Balancing policies. This section describes first the evaluation of *DPB*, then the evaluation of *DWB* and finally some experiments comparing both policies.

Due to external factors related to the availability and access to the execution environments the experiments were performed in different machines and runtime libraries . This is the reason why they are presented separately in this evaluation.

4.5.1 Applications

Synthetic applications

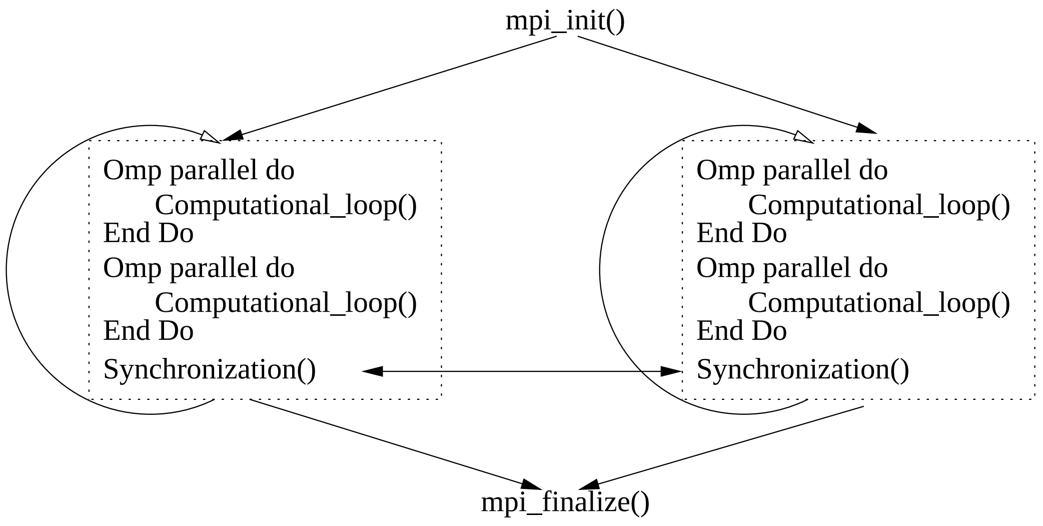


Figure 4.9: *MPIO* structure

MPIO *MPIO* is a synthetic *MPI+OpenMP* application that we used to explore the potential of the DPB policy. Figure 4.9 shows the structure of this synthetic application: it includes a simple external loop with two internal parallel loops. Two *MPI* processes execute the external loop and the internal loops are parallelized with *OpenMP*. At the beginning and at the end of each external iteration there is a message interchange to synchronize the *MPI* processes.

MPIO receives two parameters that allow to specify the workload of each of the *MPI* processes. So, different imbalance scenarios can be defined and tested.

Multi-block (Mblock) The *mblock* benchmark is a multi-block algorithm that performs a simulation of the propagation of a constant source of heat in an object. The output of the benchmark is the temperature at each point of the object. The heat propagation is computed using the Laplace equation. The object is modeled as a multi-block structure composed of a number of

Groups	% of imbalance
2	0%
4	0%
8	34%
16	60%

Table 4.1: Imbalance summary for the *large* input

rectangular blocks. Blocks are connected through a set of links at specific positions. After an initialization phase, an iterative solver computes the temperature of each point in the structure. Each block computation can be done in parallel, also parallelism exist inside each block. Propagation of the temperature between blocks can also be done in parallel. Both levels are parallelized with *OpenMP*.

The size of the blocks can be defined as an input of the application. This allows the definition of different imbalance scenarios. We used two different inputs: one with sixteen blocks, called *large*, and another with eight blocks, called *large8*.

The *large* input has twelve blocks of size 20x30x42 and four of size 40x40x60. This input generates different scenarios of imbalance (computed as suggested by De Rose et al. [RHJ07]). Table 4.1 summarizes the imbalance that exists when this input is distributed across a different number of groups. As we see, it allow us to test for three imbalance scenarios: 0% (i.e. no imbalance), 34% and 60%.

The *large* input has six blocks of size 64x64x64 and two blocks of size 128x128x128. Table 4.2 summarizes the imbalance that exists when this input is distributed across a different number of groups. As we see, it allow us to test for three imbalance scenarios: 0% (i.e. no imbalance), 52%, 75% and 88%. Note that in the case with 16 groups because there are only eight blocks it does not make sense to define 16 groups (as half of the groups would have no work at all) but an unconscious user could still make this mistake.

Groups	% of imbalance
2	0%
4	52%
8	75%
16	88%

Table 4.2: Imbalance summary for the *large8* input

NAS-MZ benchmarks

We evaluated two applications from the NAS Multizone benchmark suite [JdW06]: *BT-MZ* and *SP-MZ*. Versions of both benchmarks exist with *MPI+OpenMP* and with nested parallelism for *OpenMP*. These benchmarks solve discretized versions of the unsteady, compressible Navier Stokes equations in three spatial dimensions. The two applications compute over a data structure composed by blocks. The computation processes one block after another. Then, some data is propagated between the blocks. Parallelism appears at two levels. At the outermost level, all the blocks can be processed in parallel. At the innermost level, the computation in each block can be coded through parallel `do` loops. This structure allows for the definition of a two-level parallel strategy. The main difference between the two benchmarks is the composition of the blocks, which is going to be the main issue in the evaluation. In the case of the *BT-MZ*, the input data is composed by blocks of different sizes, while in *SP-MZ* all blocks are of the same size.

Both benchmarks come with a two load balancing algorithms. These algorithms represent slightly more than a 5% of the total code. The first algorithm is a data balancing algorithm that distributes blocks in the outer level of parallelism trying that all groups have a similar amount of computational work. The second one assigns a number a threads in the inner level of parallelism to each of the outer groups based on the computational load of the zones assigned to each outer group so it works in a similar to our *DWB* algorithm. Both methods are computed before the start of the computation based on knowledge of the data shape and computational weight of the application. Because they use the knowledge of the application we call *Application Data Balancing* to the first one and *Application Processor*

Balancing to the second one. They represent the best a programmer can do with knowledge of the application so we consider them us the upper bound for evaluating our automatic approach.

BT-MZ For each block, *BT-MZ* computes different phases. All of them implement a nest of three `do` loops: one per dimension. Usually, the outermost loop corresponds to the K-dimension and it is parallelized. Two phases must be parallelized on the J-dimension because of data dependences.

Applying a single level strategy forces the programmer to choose between two possibilities. Exploiting the parallelism between blocks or exploiting the parallelism inside the block. But both approaches have their own problems.

For example, using input class A, *BT-MZ* works with an input data composed by 16 three-dimensional blocks. Table 4.3 shows the size of each block, according to the dimension sizes.

Block	**I-dimension**	**J-dimension**	**K-dimension**	**Size**	**Proportions**
1	13	13	16	2704	1
2	21	13	16	4368	1.61
3	36	13	16	7488	2.76
4	58	13	16	12064	4.46
5	13	21	16	4368	1.61
6	21	21	16	7056	2.61
7	36	21	16	12096	4.47
8	58	21	16	19488	7.20
9	13	36	16	7488	2.76
10	21	36	16	12096	4.47
11	36	36	16	20736	7.66
12	58	36	16	33408	12.35
13	13	58	16	12064	4.46
14	21	58	16	19488	7.20
15	36	58	16	33408	12.35
16	58	58	16	53824	19.9

Table 4.3: Block sizes for *BT-MZ* class A.

Applying a single level strategy forces the programmer to choose between two possibilities. Exploiting the parallelism between blocks, which is limited by two factors: only 16 threads can obtain work as there are only 16 blocks, and what is worst, the parallelism is highly unbalanced. Last column on table 4.3 shows the proportions between the blocks. Between the the smallest block (block 1) and the largest one (block 16) there is a factor of 19.9. Exploiting the parallelism inside the block, the loops over the K-dimension and J-dimension (this last one only in two phases) are parallelized. According to the information in table 4.3, the K-dimension is 16 for all blocks and the J-dimension varies within 13, 21, 36 and 58. When the loop on the K-dimension is executed in parallel, only 16 threads will obtain work. Again, this limits the performance. Therefore, best option is to use a two-level strategy, combining the *inter* and *intra* block parallelism. This strategy generates 16 per 16 chunks of work. Therefore, even with a large number of threads all of them will get work. But, the problem of unbalance persists.

The *BT-MZ* class B input is very unbalanced as well but there are 64 blocks instead of just 16 as in the class A and the performance problems are very similar.

SP-MZ The *SP-MZ* benchmark is very similar to the *BT-MZ* benchmark. The main difference between both benchmarks is related to the sizes of the blocks in the input data structure. In the *SP-MZ* benchmark all the blocks have the same size so the outer level of parallelism is well balanced. Table 4.4 summarizes the number of blocks and the size of each one for the *SP-MZ* input classes A and B.

Class	Number of blocks	Size of each block
A	16	32x32x16
B	64	38x26x17

Table 4.4: Description of *SP-MZ* inputs

The computation evolves over different phases, where each phase is implemented by three nested loops, one per dimension. The outermost loop in each phase is always parallelized. As in the case of *BT-MZ*, a single level strategy can not be efficiently applied. Just exploiting the *inter* block parallelism is not unbalanced at all, but suffers from the same limitation regarding the number of threads to be used. Thus, again, a two level strategy is the best option: combining *inter* and *intra* block parallelism.

4.5.2 *MPI+OpenMP* experiments

Execution environment

The evaluation was performed in a single node of an IBM RS-6000 SP with 8 nodes of 16 Nighthawk Power3 @375MHz (192 Gflops/s) with 64 GB RAM of total memory. A total of 336Gflops and 1.8TB of Hard Disk were available. The operating system was AIX 5.1. The *MPI* library was configured to use shared memory for message passing inside the node. Only the *DPB* policy was implemented in this version of the runtime library . All experiments were run in exclusive mode.

MPIO benchmark

We used the *MPIO* benchmark to test the potential of the DPB policy. With *MPIO* we can define imbalance scenarios by defining the amount of the total load that is assigned to first *MPI* process (the lower this number is the bigger the imbalance is). Four different scenarios of imbalance were defined: 43,33%, 36,67%, 16,67% and 6,67% of the total load assigned to the first *MPI*.

We executed *MPIO* both with *DPB* and without it. Figure 4.10 shows the percentage of improvement with *DPB* over the version without it. We can see that as the amount of imbalance increases the improvements obtained by using *DPB* also raise. This is due to the fact that when there is a high amount of imbalance even a moderately well balanced version is a huge improvement.

In figure 4.11 the processor distribution that *DPB* assigned to each of the *MPI* processes is shown. We can see that the percentage of processors allocated to each *MPI* process is very close to the percentage of the work that we assigned to that *MPI* process.

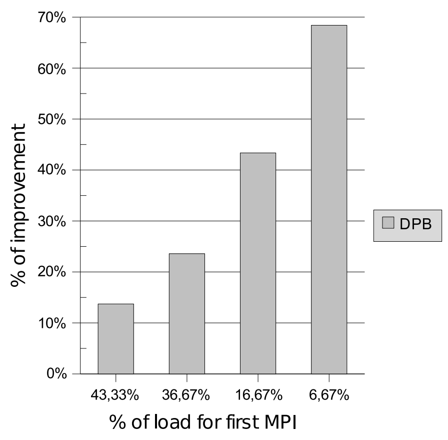


Figure 4.10: *MPIO* Speed-ups versus imbalanced version (16 CPUs).

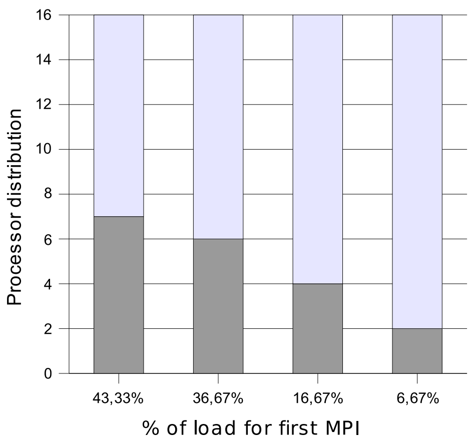


Figure 4.11: *MPIO* processor distribution (16 CPUs).

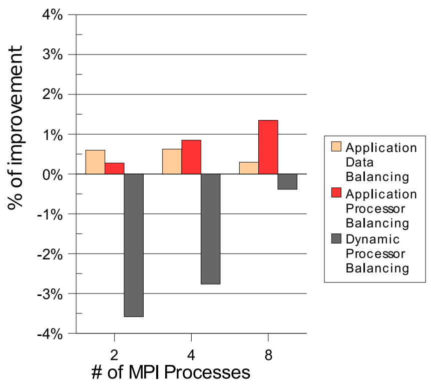


Figure 4.12: *SP-MZ* Class A improvements (16 CPUs).

Regular applications (*SP-MZ*)

We want programmers to use our proposal without even having to spend time to find if their application is balanced or unbalanced. But, this will only be feasible if in the case that the application is not unbalanced we do not introduce significative overheads because of the dynamic profiling.

To find out how significant are the overheads in the case of balanced applications we have evaluated the *SP-MZ* benchmark.

Figure 4.12 shows the results obtained for the execution of the *SP-MZ* benchmark with the class A input. We can observe that the maximum overhead introduced by *DPB* is 4% which we think is quite acceptable.

We also evaluated the *SP-MZ* with the bigger class B input. The results are shown in Figure 4.13. We can observe two interesting things:

- With two *MPI* processes, *DPB* has just a 1% of overhead but both of the load balancing algorithms embedded in the application have overheads close to the 25%. This is not because a poor processor distribution but it was a problem of the implementation yielding the processor too much under certain circumstances.

- In the eight *MPI* processes case, we can see that all the load balancing algorithms obtain an improvement close to 10% in an application that was supposed to be balanced.

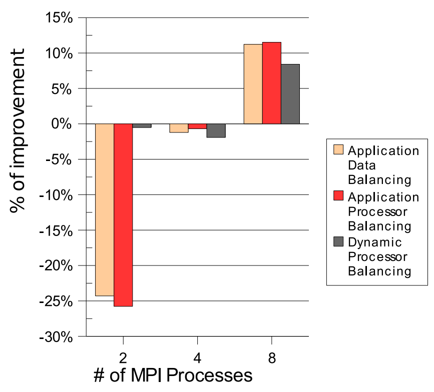


Figure 4.13: *SP-MZ* Class B improvements (16 CPUs).

From these experiments we can conclude that the overheads introduced by *DPB* are small enough to be feasible to use it as a default strategy if we do not know, or do not care to know, if the application is balanced. And, if there happens to be an odd case where the application exhibits an small unbalance *DPB* will detect it and correct the behavior of the application.

Irregular applications (*BT-MZ*)

The situation where the data domain is irregular in its geometry, and in consequence in its computational load, is very frequent on scientific codes mainly because of the nature of the entities being modelled (weather forecasting, ocean flow modelling, ...). The evaluation of the *BT-MZ* benchmark will allows to us quantify if how useful can *DPB* be to improve those codes. Our goal is to be as close as possible to the *Application Processor Balancing*(*APB*) that is coded in the *BT-MZ* application.

Figure 4.14 shows the improvement of the different load balancing strategies over the regular version for *BT-MZ* with the class A data input. We can see that *DPB* is closely tied with the *APB* algorithm. In an execution with two *MPI* processes execution, *DPB* obtains a better improvement (26% vs 18% of improvement). In an execution with four *MPI* processes, both algorithms are closely tied. But, in an execution with eight *MPI* processes, the

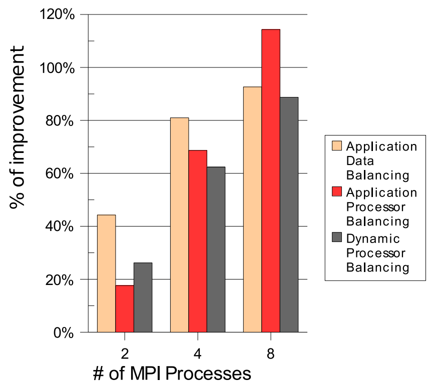


Figure 4.14: *BT-MZ* Class A improvements (16 CPUs).

DPB improvement is significantly lower (a 14% less than with *APB*).

As we mentioned in Section 4.4.3, because *DPB* works with pairs of processes as the number of *MPI* processes increases so it does the time that it takes *DPB* to find a stable distribution. So, for longer executions *DPB* should be an option as good as the *Application Processor Balancing* algorithm.

To confirm this hypothesis we obtained the improvements of the different algorithms over the last 20 iterations of the algorithm where the *DPB* algorithm should have found a stable distribution. Figure 4.15 shows these improvements. In this case, the difference in the improvements obtained with *DPB* and the *APB* decreases until a very acceptable 2%, that is due to the inherent overheads of continuous profiling of the application to detect changes in the behavior.

Figure 4.16 shows the processor distribution that *DPB* chooses to overcome that unbalance (each *MPI* process has a different color). We can see that the distributions found by the algorithm are quite complex.

We also evaluated the *BT-MZ* benchmark with the class B input. We can see the improvements obtained with the different load balancing algorithms in Figure 4.17. The class B has a larger number of iterations than the class A, and we can see that the behavior of *DPB* and *Application Processor Balancing* is very similar to the last iterations of the class A. This means

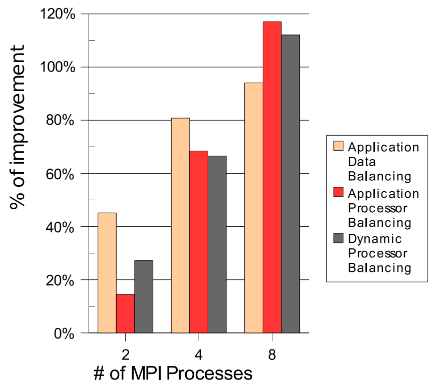


Figure 4.15: *BT-MZ* Class A improvements (last 20 time steps, 16 CPUs).

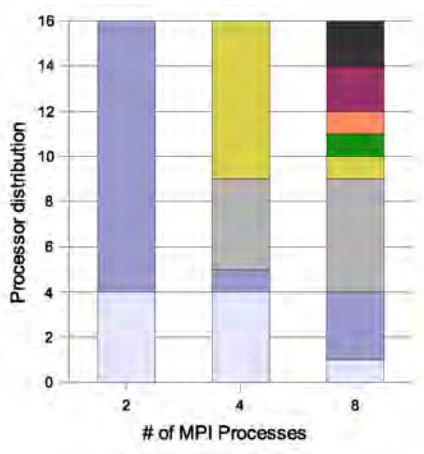


Figure 4.16: *BT-MZ* Class A processor distribution (16 CPUs).

that the warm-up cost of *DPB* is hidden here because it is only a small fraction of the total amount of iterations.

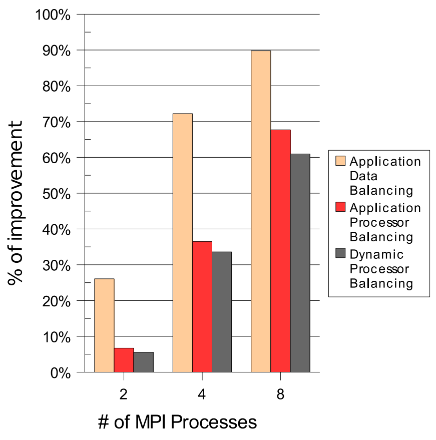


Figure 4.17: *BT-MZ* Class B improvements (16 CPUs).

Now, if we compare the improvements of *DPB* with the *Application Data Balancing*(*ADB*), in many cases this algorithm obtains better improvements than *DPB*. This is because data redistributions allow finer movements than processor distribution (because each group needs at least one processor that it can not be split). Even so, in some situations the improvements *ADB* obtains are lower than *DPB*. Taking into account that *ADB* is very difficult to apply in a transparent way we think that *DPB* shows that it can improve the performance of unbalanced applications, that do not suffer from the warm-up costs, without programmer intervention at all.

MPI vs *MPI+OpenMP*

Figure 4.18 shows a comparative of the execution times for different combinations of *MPI* and *OpenMP* for the *BT-MZ* benchmark: ranging from pure *OpenMP* (only one *MPI* process) to pure *MPI* (only one *OpenMP* thread in each *MPI* process) through hybrid combinations. We can see that the pure *OpenMP* approach obtains better results than the *MPI* approaches or the hybrid approaches. But, when we apply our *DPB* policy we get even better results because the load imbalance is further reduced.

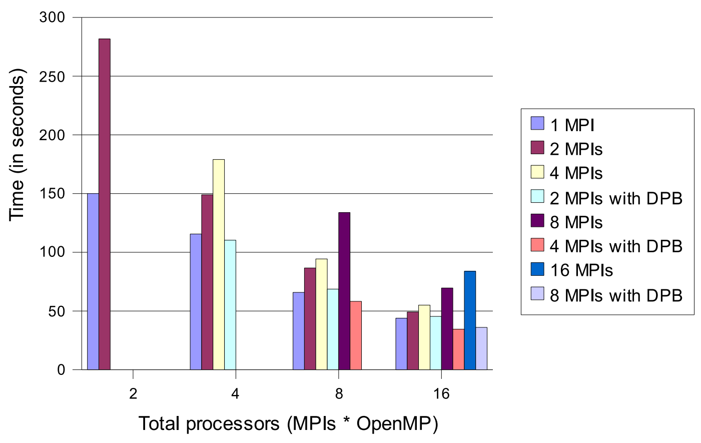


Figure 4.18: *BT-MZ* Class A. *MPI* vs *MPI+OpenMP*.

These results show that when an application is unbalanced is better to use an hybrid approach that allows to overcome the unbalance either with a hardcoded algorithm, as those described in Section 4.5.1, or, even better, automatically as with our proposals. When the application is well balanced, on the other hand, it is probably better to use a single level approach, if enough parallelism can be generated from a single level, as noted by Cappello et al. [CE00] (whether *MPI* or *OpenMP* is an ongoing debate).

4.5.3 Nested *OpenMP* experiments

Execution environment

Benchmarks using the IBM XL environment were run in a p690 32-way Power4 machine at 1.1 GHz with 128 GB of RAM. We used IBM's XLF compiler with the following flags:*-O3 -qipa=noobject -qsmp=omp*. The operating system was AIX 5.2. The XL runtime had implemented only the *DWB* policy. All experiments were run in exclusive mode.

Multi-block benchmark (Mblock)

To evaluate the *DWB* in different imbalance scenarios we have used the *mblock* benchmark using the *large* input data set. As we have described

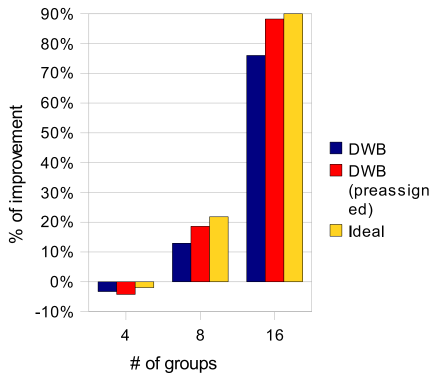


Figure 4.19: Mblock benchmark improvements (32 CPUs)

in Section 4.5.1, this input has different load imbalances depending on the number of groups: ranging from a 0% to a 60% of imbalance.

First, we wanted to find an "ideal" execution time for the benchmark so we could evaluate how far we where from it with *DWB*. With a simulation process we selected those combinations of groups and distributions of threads that seemed more likely to obtain a good performance result. Table 4.5 shows how many *candidate* distributions we tried for each number of groups.

Groups	**# of distributions tested**
4	19
8	19
16	19

Table 4.5: Number of tests to find the ideal for *mblock* (32 CPUs)

Figure 4.19 shows the improvements we obtain using *DWB* over the original version for the *mblock* application with input blocks defined above. We can observe that in the cases where the application is unbalanced (i.e. with 8 and 16 groups) *DWB* improves considerably the performance of the application. But, it seems there is still potential for improvement as shown by the *ideal* distribution. This could be because of two reasons: either the distribution found is not the optimal or because the overheads are too high.

To find it out we evaluated the *mblock* benchmark again but we used as the distribution of threads for the execution the distribution that was found

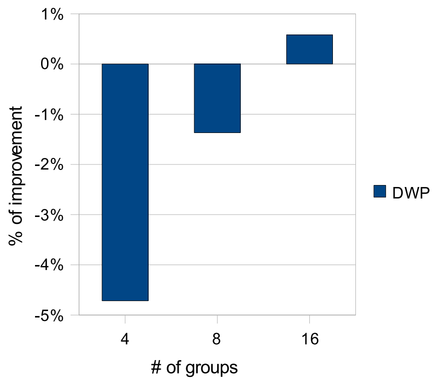


Figure 4.20: *SP-MZ* class A improvements (32 CPUs).

in a previous execution with *DWB*. This means, that instead of applying the decision in the same execution in this case we are doing it in the *next* one where we can disable all the profiling. The results are shown in Figure 4.19 with the *DWB (preassigned)* label. We can see that in this case *DWB* obtains an improvement close to the *ideal*. So, we can conclude that distribution it founds is quite good but the improvements, while important, are hampered because of the time to find the distribution and the profiling overheads.

Regular applications (***SP-MZ***)

As in the *MPI+OpenMP* evaluation, we used the *SP-MZ* benchmark to identify if it would be possible to use *DWB* as default strategy even in those cases where the application is well balanced. In the nested *OpenMP* version of *SP-MZ* there are no load balancing algorithms so only *DWB* was evaluated against the regular version.

Figure 4.20 shows the results for the execution with the class A input and 32 processors. We can see that the overhead introduced by *DWB* is at most 5% which we think is a reasonable maximum overhead. In the other cases the overhead is much lower.

Groups	# of distributions tested
4	12
8	28
16	13

Table 4.6: Number of tests to find the ideal for *BT-MZ* (32 CPUs)

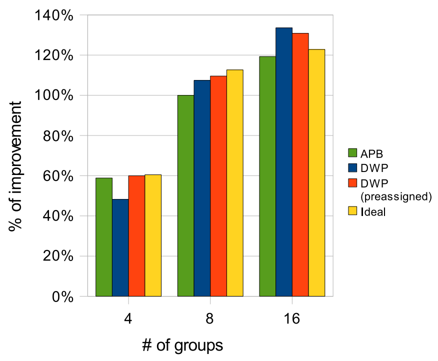


Figure 4.21: *BT-MZ* class A improvements (32 CPUs).

Irregular applications (*BT-MZ*)

As in the *MPI+OpenMP* evaluation, we used the *BT-MZ* benchmark to study how *DWB* behaves when the strategy is applied to applications with irregular distributions of data. In this case, we only evaluated the class A, as we know that the behavior of the classes A and B are very similar.

First, as in the *mblock* experiments, we wanted to find an "ideal" execution time for the benchmark so we could evaluate how far we where from it with *DWB*. With a simulation process we selected those combinations of groups and threads that seemed more likely to obtain a good performance result. Table 4.6 shows how many *candidate* distributions there were for each number of groups.

Then, we evaluated both our *DWB* proposal and the *Application Processor Balancing*(APB) algorithm that comes embedded with the *BT-MZ* benchmark. Our goal is to be as close to *APB* as possible.

Figure 4.21 shows the improvements of both methods for the *BT-MZ* benchmark with a class A input and 32 CPUs and different number of *outer* groups of parallelism. The *ideal* bars correspond to the best of the execution times from the *candidate* distributions. We can see that in all cases, except with two groups, the improvement we obtain with *DWB* is better than *APB*.

In the case of having only four *groups* of parallelism *DWB* does obtain an improvement close to a 50% but not as good as *APB* which improves the unbalanced application by a 59%. This difference could be due to profiling overheads.

To find it out, as in the *mblock* experiment, we evaluated the *BT-MZ* benchmark again but we used as the distribution of threads for the execution the distribution that was found in a previous execution with *DWB*. This means, that instead of applying the decision in the same execution in this case we are doing it in the *next* one where we can disable all the profiling. The results are shown in Figure 4.21 with the *DWB (preassigned)* label. We can see that in this case *DWB* obtains the same improvement as *APB* with four groups.

When comparing with the *ideal* numbers, we see that *DWB* is close to it in all cases and even in one it finds a distribution that it is better that one of our candidate distributions.

These results suggest that *DWB* does a good job in finding a good distribution of threads across the groups and this translates in great improvements in the application. Even so, some results seem to indicate that the profiling may have an impact in some situations and some extra optimizations may be needed.

4.5.4 Dynamic Processor Balancing vs Dynamic Weight Balancing

Execution environment

The SMP machine used for the experiments comparing *DPB* and *DWB* is a 64-way POWER5 machine (1656MHz processors) with 514 GB of RAM running AIX 5.3. We used IBM's XLF compiler with the following flags:*-O3 -qipa=noobject -qsmp=omp*. All experiments were run in exclusive mode. We implemented both, *DPB* and *DWB*, in IBM's XL runtime.

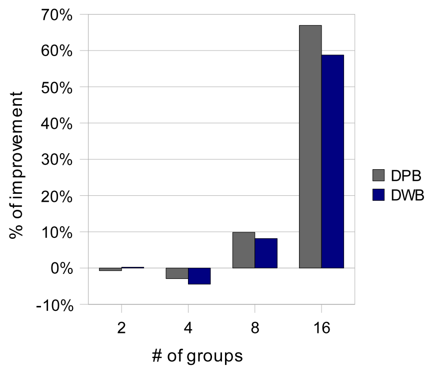


Figure 4.22: *mblock* improvements with the *large* input (32 CPUs).

Multi-block

We used two input sets of the *mblock* application to compare *DPB* and *DWB*: the *large* input set and the *large8* input set. As we described in section 4.5.1, these inputs generate imbalances that range, depending on the number of groups, from a 0% to a 88%.

Figure 4.22 shows the results obtained for the *mblock* benchmark with the *large* input data set and 32 CPUs. With two and four groups neither of the policies obtain any improvement but that was to expect as the amount of work for all the groups is the same. In these two balanced scenarios, the overhead of both policies is fairly similar as well between 1 to 3%, which seems reasonable. As the imbalance increases with the number of groups both policies start improving the performance by using better thread distributions. *DPB* obtains a better improvement in both cases as it achieves a better distribution.

Figure 4.23 shows the results obtained for the *mblock* benchmark with the *large8* input data set and 32 CPUs. In this case, the improvements are even larger in the unbalanced scenarios, as the imbalance is bigger, but it is the *DWB* policy that obtains a better distribution obtaining almost a 100% more of improvement in the case with 16 groups.

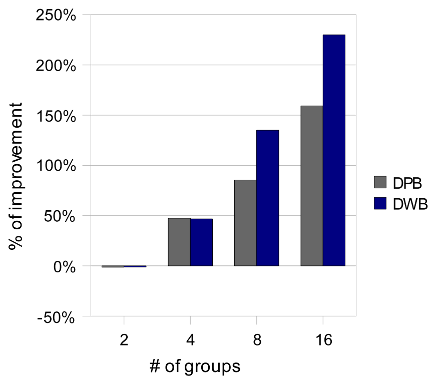


Figure 4.23: *mblock* improvements with the *large8* input (32 CPUs).

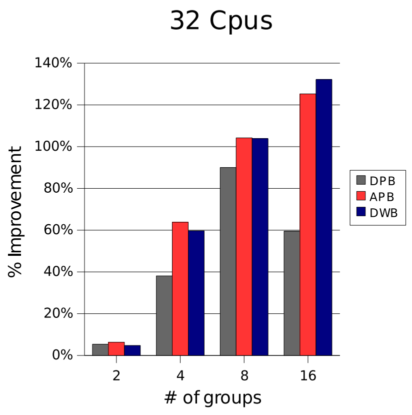


Figure 4.24: *BT-MZ* class A improvements (32 CPUs).

BT-MZ

We executed the *BT-MZ* benchmark with a class A input with both policies. The results are shown in Figure 4.24.

This results show that while *DWB* improvements are very close to our target goal (i.e. *APB*), *DPB* came short in all cases but with two groups. This is explained by the larger time that it takes *DPB* to find an stable distribution.

From these experiments we can conclude that *DWB* seems to obtain better improvements than *DPB* although there are some cases where *DPB* wins.

4.6 Conclusions

In this chapter, we have presented a method to compute at runtime a thread distribution that can be used in scenarios where a multilevel parallelization has been used in an application. We explain how to implement our algorithm for both, *MPI+OpenMP* applications and pure *OpenMP* applications with nested parallelism.

Our proposal works by gathering information, at runtime, about the time each group spent doing useful work. Based on these measures, a distribution policy computes a new thread distribution that is applied afterwards. We have implemented two distribution policies:

Dynamic Processor Balancing works by moving processors from the less loaded group to the most loaded one in each step of the computation.

Dynamic Weight Balancing works by distributing all the processors based on the discovered load of each of the groups.

Our evaluation results confirmed that the proposed runtime mechanism performs well obtaining improvements close to those of load balancing algorithms implemented with knowledge of the application. Even, when application is well balanced (as in the *SP-MZ* case) the mechanism shows a low overhead. This suggests that a runtime could use our mechanism as a default option for nested parallelism. Of the two policies, our evaluation suggests that the *Dynamic Weight Balancing* obtains in general better thread distributions than the *Dynamic Processor Balancing*.

One critical factor is the dynamic profiling overhead. The evaluation shows that although some times the policies find an appropriate distribution of threads the improvement obtained is reduced by the overheads of the mechanism. So, a more careful implementation of the profiling mechanism is very important for this approach to obtain better results.

The number of groups is a critical parameter in the performance of an application that was not taken into account in our work. Future work will deal with the problem of automatically deciding how many groups should be used at the outer level. This will allow an optimal exploitation of nested parallelism without programmer intervention.

Chapter 5

OpenMP Tasks evaluation

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'

Isaac Asimov

Russian-american writer (1920–1992)

Abstract

OpenMP recently added a task model to allow the parallelization of applications not based on loop parallelism. Within this new proposal independent units of work called tasks can be created. The details of the scheduling of these tasks are left open in the specification. But the exact scheduling decision may have a great impact on the performance of applications using tasks. In this chapter, we explore the new tasking model and present an evaluation of different scheduling strategies for it.

5.1 Introduction

OpenMP grew out structured around parallel loops and was meant to handle dense numerical applications. The simplicity of its original interface, the use of a shared memory model, and the fact that the parallelism of a program is expressed in directives that are loosely-coupled to the code, all have helped *OpenMP* become well-accepted today. However, the sophistication of parallel programmers has grown in the 10 years since *OpenMP* was introduced, and the complexity of their applications is increasing. Therefore, the new *OpenMP* 3.0 [Org08] added a new tasking model [ACD$^+$07], that we discussed in Chapter 2, to address this new programming landscape. The new directives allow the user to identify units of independent work, called `tasks`, leaving the scheduling decisions of how and when to execute them to the runtime system.

In this chapter, we explore the expressivity of the new proposal while we prepare a number of different benchmarks that allows as to explore different possibilities about the scheduling policies of these new `tasks`. We extended the Nanos runtime [TMD$^+$07] with two scheduling strategies: a breadth-first approach and a work-first approach. We implemented several queueing and work-stealing strategies. Then, we evaluated combinations of the different scheduling components with the application we developed. We, also, evaluated how these schedulers behave if the application uses `tied tasks`, which have some scheduling restrictions, or `untied` ones, wich have no scheduling restrictions.

Our aim is to discover what issues will programmers have to deal with when they start using the new *OpenMP* task model.

5.2 Motivation and related work

The Intel *work-queueing* model [SHPT99] was an early attempt to add dynamic task generation to *OpenMP* . This proprietary extension to *OpenMP* allows hierarchical generation of tasks by nesting `taskq` constructs. Our Nanos group proposed `dynamic sections` [BDG$^+$04] as an extension to the standard `sections` construct to allow dynamic generation of tasks.

Finally, a committee from different institutions developed a task model [ACD$^+$07] for the *OpenMP* language that was included in the latest standard specification [Org08]. One of the things this task proposal leaves open is the scheduler of tasks that should be used and when tasks should be immediately executed.

Scheduling of tasks is a well studied field but their applicability to the *OpenMP* model remains an open question. There are two main scheduler families: those that use breadth-first schedulers (see for example the work from Narlikar [Nar99] and those that use work-first schedulers with work-stealing techniques (see for example Cilk [FLR98] and Acar et al. [ABB00]). Korch et al. [KR04] made a very good survey of different task pool implementations and scheduling algorithms and evaluated them with a radiosity application. Many of these works have found that work-first schedulers tend to obtain better performance results.

Besides scheduling, task granularity is a key factor to scale up a task parallel program. Dynamic aggregation is a common technique used by many parallel languages to increase the granularity of tasks.

Most languages use some kind of task inlining (or lazy creation) in order to increase the granularity of the tasks. Inlined tasks still keep the potential to spawn a full task if needed. The most common criteria to spawn a new task is based on the load of the processors (i.e. if some processor is idle) [TTY99,MKRHH90,FLR98,GSC96] but some other options using level based or priority schemes have been studied [LH95].

But even lazy task creation has problems with very fine grained tasks [MKRHH90] because there is still some overhead associated with task creation. The other option instead of inlining lazy tasks is serializing them to reduce even further the overhead. The problem with serializing tasks upon creation, as noted by Kranz et al. [KRHHM89], is that it can lead to load unbalance or deadlock because a wrong decision cannot be undone. The decision is taken by a cut-off function that decides whether continue creating parallelism or serialize the potential task. Some proposals try to achieve this by using as cut-off the load of the thread system [KRHHM89, CLP$^+$08].

Another proposal, uses the size of data structures [HLA94] to control task creation but it depends on the compiler understanding complex structures like lists, which is difficult in the C or Fortran languages. Aharoni et al. use the number of elements of the structure at run-time to control granularity [AFB92] but for non-uniform elements they also need compiler help. Rugina and Rinard generate a level based cut-off to serialize tasks in their automatic approach to divide and conquer algorithms [RR99].

But, it is unclear how all these algorithms will map into the *OpenMP* new task model as most of the previous work was in the context of recursive algorithms and where there were no scheduling restrictions at all. But the *OpenMP* task model allows to parallelize not only recursive applications but also applications that mix traditional work-sharing regions with task parallelism. Our goal in this work is to evaluate previous techniques in the context of *OpenMP* and understand which parameters will be the programmers need to tune in the future to obtain good performance from of their applications using the task model.

5.3 Task programming

We parallelized several applications with the new *OpenMP* tasking model. This allowed us to gain insight into the new model and have a pool of applications for evaluation purposes as there were no applications developed using *OpenMP* tasks at the moment of this work.

We have worked on applications across a wide range of domains (linear algebra, sparse algebra, servers, branch and bound, etc) to test the expressiveness of the proposal. Some of the applications (*multisort*, *fft* and *queens*) are originally from the Cilk project [FLR98], some others (*pairwise alignment*, *connected components* and *floorplan*) come from the Application Kernel Matrix project from Cray [CFLM04] and one (*sparseLU*) has been developed by us. These kernels were not chosen because they were the best representatives of their class but because they represented a challenge for the previous 2.5 *OpenMP* standard and were publicly available. We show the key parts of those applications and present how the new *OpenMP* task model enables new parallelization strategies.

In order to organize the presentation of the examples, we divide them into two subsections. First we describe situations showing how tasking allows one to express more parallelism (or to exploit it more efficiently) than current *OpenMP* worksharing constructs. Second, we describe situations in which tasking replaces the use of nested parallelism.

5.3.1 Worksharing versus tasking

In this subsection we illustrate some examples where the use of the new *OpenMP* tasks allows the programmer to express more parallelism (and thus obtain better performance) than could be expressed with *OpenMP* 2.5 worksharing constructs.

SparseLU

The `for` worksharing construct is able to handle load imbalance situations by using dynamic scheduling strategies. Tasking is an alternative option to parallelize this kind of loop, as shown in the code excerpt in Figure 5.1. In this code, the *if* statements that control the execution of functions *fwd*, *bdiv* and *bmod* for non-empty matrix blocks are the sources of load imbalance. One could use an *OpenMP* `for` worksharing construct with `dynamic` scheduling for the loops on lines 9, 14 and 21 and 23 (for the *bmod* phase one can either parallelize the outer, line 21, or the inner loop, line 23, with different load balance versus overhead trade-offs). On the other hand, if the inner loop is parallelized the iterations are smaller which allows a dynamic schedule to have better balance but the overhead of the worksharing is much higher. Using tasks, a single thread could create work for all those non-empty matrix blocks, achieving both load balance and low overhead in the generation and assignment of work.

It is interesting to note that, if the proposed extension included mechanisms to express point-to-point dependencies among tasks, it would be possible to express additional parallelism that exists between tasks created in lines 11 and 16 and tasks created in line 25. Also it would be possible to ex-

```
int sparseLU() {
  int ii, jj, kk;
  #pragma omp parallel
    #pragma omp single nowait
      for (kk=0; kk<NB; kk++) {
        lu0(A[kk][kk]);
        /* fwd phase */
        for (jj=kk+1; jj < NB; jj++)
          if (A[kk][jj] != NULL)
            /* only create tasks for non-empty blocks */
            #pragma omp task untied
              fwd(A[kk][kk], A[kk][jj]);
        /* bdiv phase */
        for (ii=kk+1; ii < NB; ii++)
          if (A[ii][kk] != NULL)
            /* only create tasks for non-empty blocks */
            #pragma omp task untied
              bdiv (A[kk][kk], A[ii][kk]);
        /* wait for previous tasks */
        #pragma omp taskwait
        /* bmod phase */
        for (ii=kk+1; ii < NB; ii++)
          if (A[ii][kk] != NULL)
            for (jj=kk+1; jj < NB; jj++)
              if (A[kk][jj] != NULL)
                /* only create tasks for non-empty blocks */
                #pragma omp task untied
                {
                  if (A[ii][jj]==NULL)
                    A[ii][jj]=allocate_clean_block();
                  bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
                }
        /* wait for all previous tasks */
        #pragma omp taskwait
      }
}
```

Figure 5.1: Main code of *SparseLU* with *OpenMP* tasks

press the parallelism that exists across consecutive iterations of the *kk* loop. Instead, the taskwait reduces parallelism to ensure those dependences are not violated.

Protein pairwise *alignment*

This application aligns all protein sequences from an input file against every other sequence. The alignments are scored and the best score for each pair is output as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. It uses the recursive Myers and Miller algorithm to align sequences [Gen97].

The outermost loop can be parallelized, but the loop is heavily unbalanced, although this can be partially mitigated with dynamic scheduling. Another problem is that the number of iterations is too small to generate enough work when the number of threads is large. Also, the loops of the different passes (forward pass, reverse pass, diff and tracepath) can also be parallelized but this parallelization is much finer so it has higher overhead.

```
/* all threads pick up some sequences */
#pragma omp for
  for (si = 0; si < nseqs; si++) {
    len1 = compute_sequence_length(si+1);
    /* compare to the other sequences */
    for (sj = si + 1; sj < nseqs; sj++) {
      /* create a task for each comparison */
      #pragma omp task
      {
        len2 = compute_sequence_length(sj+1);
        compute_score_penalties(...);
        forward_pass(...);
        reverse_pass(...);
        diff(...);
        mm_score = tracepath(...);
        if (len1 == 0 || len2 == 0) mm_score  = 0.0;
        else mm_score /= (double) MIN(len1,len2);
          /* printing in mutual exclusion */
        #pragma omp critical
          print_score();
      }
    }
  }
```

Figure 5.2: Main code of the pairwise *aligment* with tasks

In Figure 5.2 we show how we used *OpenMP* tasks to efficiently exploit the parallelism available in the inner loop in conjunction with the parallelism available in the outer loop, which uses a `for` worksharing construct. This breaks iterations into smaller pieces, thus increasing the amount of parallel work but at lower cost than an inner loop parallelization because they can be executed inmediately.

5.3.2 Nested parallelism versus tasking

In this subsection we illustrate some examples where the use of the new *OpenMP* tasks allows a programmer to express parallelism that in *OpenMP* 2.5 would be expressed using nested parallelism. The versions of these programs using nested OpenMP, while simple to write, usually do not perform well [ADH+07] because a variety of problems (load imbalance, synchronization overheads, . . .).

Multisort*, *FFT* and *Strassen

Multisort is a variation of the ordinary mergesort. It sorts a random permutation of n 32-bit numbers with a fast parallel sorting algorithm by dividing an array of elements in half, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. When the array is too small, a serial quicksort is used so the task granularity is not too small. To avoid the overhead of quicksort, an insertion sort is used for arrays below a threshold of 20 elements.

The parallelization with tasks is straight forward and makes use of a few `task` and `taskwait` directives (see figure 5.3).

FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm. *Strassen*'s algorithm for multiplication of large dense matrices uses hierarchical decomposition of a matrix. The structure of the parallelization of these two kernels is almost identical to the one used in multisort, so we will omit them.

```
void sort(ELM *low, ELM *tmp, long size) {
  if (size < quick_size) {
    /* quicksort when reach size threshold */
    quicksort(low, low + size - 1);
    return;
  }

  /* split into 4 pieces: A, B, C and D */
  quarter = size / 4;
  A = low; tmpA = tmp;
  B = A + quarter; tmpB = tmpA + quarter;
  C = B + quarter; tmpC = tmpB + quarter;
  D = C + quarter; tmpD = tmpC + quarter;

  /* create tasks to sort vector splits A, B, C and D */
  #pragma omp task untied
    sort(A, tmpA, quarter);
  #pragma omp task untied
    sort(B, tmpB, quarter);
  #pragma omp task untied
    sort(C, tmpC, quarter);
  #pragma omp task untied
    sort(D, tmpD, size - 3 * quarter);

  /* wait for all sort tasks to finish */
  #pragma omp taskwait

  /* create tasks to merge A with B and C with D */
  #pragma omp task untied
    merge(A, A+quarter-1, B, B+quarter-1, tmpA);
  #pragma omp task untied
    merge(C, C+quarter-1, D, low+size-1, tmpC);

  /* wait for AB and CD merge to finish */
  #pragma omp taskwait

  /* merge AB with CD */
  merge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
}
```

Figure 5.3: Sort function using *OpenMP* tasks

Floorplan

The *Floorplan* kernel computes the optimal floorplan distribution of a number of cells. The algorithm is a recursive branch and bound algorithm. The parallelization is straight forward (see figure 5.4). We hierarchically generate tasks for each branch of the solution space. But this parallelization has one caveat. In these kind of algorithms (and others as well) the programmer needs to copy the partial solution up to that point to the new parallel branches (i.e. tasks). Due to the nature of C arrays and pointers, the size of it becomes unknown across function calls and the data scoping clauses are unable to perform a copy on their own. To ensure that the original state does not disappear before it is copied, a task barrier is added at the end of the function. Other possible solutions would be to copy the array into the parent task stack and then capture its value or allocate it in heap memory and free it at the end of the child task. In all these solutions, the programmer must take special care.

N Queens

This program, which uses a backtracking search algorithm, computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an n x n chessboard such that none of the queens attacks any other.

In this application, tasks are nested dynamically inside each other. As in the case of floorplan, the state needs to be copied into the newly created tasks so we need to introduce additional synchronizations (see Figure 5.5) so the original state is alive when the tasks start so they can copy it.

Another issue is the need to count all the solutions found by different tasks. One approach is to surround the accumulation with a critical directive but this would cause a lot of contention. To avoid it, we used `threadprivate` variables. In this way, all threads can acumulate the solutions they find. The variables are reduced within a `critical` directive to the global variable at the end of the parallel region.

```
void add_cell(int id, coor FOOTPRINT, ibrd BOARD,
        struct cell *CELLS) {
  int  i, j, nn, area;
  ibrd board;
  coor footprint, NWS[DMAX];

  for (i = 0; i < CELLS[id].n; i++) {
    nn = compute_possible_locations(id, i, NWS, CELLS);
    /* for all possible locations */
    for (j = 0; j < nn; j++) {
      /* create a task for each possible configuration */
      #pragma omp task private(board, footprint,area) \
        shared(FOOTPRINT,BOARD,CELLS) untied
      {
        /* copy parent state*/
        struct cell cells[N+1];
        memcpy(cells,CELLS,sizeof(struct cell)*(N+1));
        memcpy(board, BOARD, sizeof(ibrd));
        compute_cell_extent(cells,id,NWS,j);
        /* if cell cannot be layed down, prune search */
        if (! lay_down(id, board, cells)) goto _end;
        area = compute_new_footprint(footprint,FOOTPRINT,
                        cells[id]);
        /* if last cell */
        if (cells[id].next == 0) {
          if (area < MIN_AREA)
            #pragma omp critical
              if (area < MIN_AREA)
                save_best_solution();
        } else if (area < MIN_AREA)
          /* only continue if area is smaller to best area,
             otherwise prune */
          add_cell(cells[id].next, footprint, board,cells);
      _end:;
      }
    }
  }
  /* This taskwait ensures parent state remains alive
     for child's to copy it*/
  #pragma omp taskwait
}
```

Figure 5.4: *Floorplan* kernel with *OpenMP* tasks

```
void nqueens_par(int n, int j, char *a, int d)
{
   int i;

   /* spawn a task for each possible solution */
   for (i = 0; i < n; i++) {
    #pragma omp task
    {
      /* allocate a temporary array and copy <a> into it */
      char* b = alloca((j + 1) * sizeof(char));
      memcpy(b, a, j * sizeof(char));
      b[j] = i;
      if (ok(j + 1, b)) {
        nqueens(n, j + 1, b, d+1);
      }
    }
   }

  /* This taskwait ensures parent state remains alive
      for child's to copy it*/
   #pragma omp taskwait
}
```

Figure 5.5: N Queens kernel with *OpenMP* tasks

5.4 Task scheduling strategies

We extended the Nanos research runtime library [TMD+07] with two families of task schedulers: breadth-first schedulers and work-first schedulers. These schedulers implement the restrictions about scheduling of `tied tasks` (i.e. tied tasks can only be scheduled on the thread to wich they are tied to). Also, we implemented several cut-off strategies that we explain in this section.

5.4.1 Breadth-first scheduling

Breadth-first scheduling (BF) is a naive scheduler in which every task that is created is placed into the team pool and execution of the parent task continues. So, all tasks in the current recursion level are generated before a thread executes tasks from the next level.

Initially, tasks are placed in a team pool and any thread of the team can grab tasks from that pool. When a task is suspended (e.g. because a `taskwait`), if it is a `tied task` it will go to a private pool of tasks of the thread that was executing the tasks. Otherwise (i.e an `untied task`), it will be queued into the team pool.

Threads will always try to schedule first a task from their local pool. If it is empty then they will try to get tasks from the team pool.

We implemented two access policies for the task pools: LIFO (i.e. last queued tasks will be executed first) and FIFO (i.e. oldest queued tasks will be executed first).

5.4.2 Work-first scheduling

Work-first scheduling (WF) tries to follow the serial execution path hoping that if the sequential algorithm was well designed it will lead to better data locality.

The WF scheduler works as follows: whenever a task is created, the creating task (i.e. the parent task) is suspended and the executing thread switches to the newly created task. When a task is suspended (either because it created a new one or because some synchronization) the task is placed in a per thread local pool. Again, this pool can be accessed in a LIFO or FIFO manner.

When looking for tasks to execute, threads will look on their local pool. If it is empty, they will try to steal work from other threads. In order to minimize contention we used a strategy where a thread traverses all other threads starting by the next thread (i.e. thread 0 starts trying to steal from thread 1, thread 1 from thread 2, ... and thread n from thread 0). When stealing from another thread pool, to comply with *OpenMP* restrictions, a task that has become tied to a thread cannot be stolen (note that a `tied task` that has not been yet executed can be stolen). The access to the victim's pool can also be LIFO or FIFO.

We also implemented a stealing strategy that first tries to steal the parent task of the current task. If the parent task cannot be stolen (i.e. because is either already running or waiting on some synchronization) then the default stealing mechanism is used.

The Cilk scheduler [FLR98] pertains to this family of schedulers. In particular, it is a work-first scheduler where access to the local pool is LIFO, tries to steal the parent task first and otherwise steals from another thread pool in a FIFO manner.

5.4.3 Cutting off

In order to reduce the size of the runtime structures and, also, reduce the overheads associated to creating tasks, the runtime can decide to start executing tasks immediately. This is usually referred as cutting off.

This is particularly important with breadth-first scheduling as it tends to generate a large number of tasks before executing them. In work-first scheduling the number of tasks that exist at a given time is not so large but it may grow over time because of tasks being suspended at synchronization points.

It is important to note that tasks that are executed immediately because of a cut-off policy are different than the ones that get executed immediately with the work-first scheduler. When cutting off, the new task does not go through to the whole creation process and in many aspects forms part of the creating tasks (e.g. cannot be suspended on its own).

We have implemented several simple but effective cut-off policies:

Max number of tasks (max-task) The total number of tasks that can exist at the same time is computed as a factor of the number of *OpenMP* threads (i.e. $k * num\_threads$). Once this number is reached new tasks are executed immediately. When enough tasks finish, tasks will be created again. In our implementation, we use a default value for k of 8.

Max task recursion level (max-level) When a new task is created, if it has more ancestors than a fixed limit l then the new task is executed immediately. Otherwise it can be created. In our implementation, we use a default value for l of 4.

Max number of ready tasks (max-ready) When a new task is created, if the number of ready tasks in the system is more than a certain limit l the task is executed immediately. Otherwise, the task can be created.

In our implementation, we use a default value for l of 1 per number of *OpenMP* threads.

Workload based (load-based) When a new task is created, the system checks how many *OpenMP* threads are idle at that moment. If the number is greater that zero the task is created. Otherwise it is executed immediately.

5.5 Evaluation

5.5.1 Applications

We have used the applications described in Section 5.3 to evaluate the different scheduler and cut-off policies from Section 5.4.

In all applications (except *Alignment*) we marked all tasks as *untied* and we removed any kind of manual cut-off that was there from the programmer to leave total freedom to the scheduler. The *Alignment* application makes heavy use of `threadprivate` and, because of that, we could not mark the tasks as `untied`.

5.5.2 Methodology

We evaluated all the benchmarks on an SGI Altix 4700 with 128 processors, although they were run on a cpuset comprising a subset of the machine to avoid interferences with other running applications.

We compiled all applications with our Mercurium compiler [BDG$^+$04] using gcc with option -O3 as the backend. The serial version of the application was compiled with gcc -O3 as well. The speed-ups were computed using the serial execution time as the baseline and using the average execution time of 5 executions.

We have executed all applications with different combinations of schedulers. Table 5.1 summarizes the different schedules we have used in the evaluation, their properties (see Section 5.4 for details) and the name we will be using to refer to them in the next sections.

Scheduler Name	Scheduler Type	Pool Access	Steal Access	Steal Parent
bff	breadth-first	FIFO	-	-
bfl	breadth-first	LIFO	-	-
wfff	work-first	FIFO	FIFO	No
wffl	work-first	FIFO	LIFO	No
wflf	work-first	LIFO	FIFO	No
wfll	work-first	LIFO	LIFO	No
cilk	work-first	LIFO	FIFO	Yes

Table 5.1: Summary of schedules used in the evaluation

For each schedule we have run the applications without using any cut-off and then using the cut-offs we had implemented: the *max-task*, *max-level*, *max-ready* and *load-based* cut-offs.

Then, we wanted to know how the restrictions of `untied tasks` affected the performance that can be obtained with the different schedulers. So, we have also tried for those combinations that were best from each application but with all tasks `tied` (we control this via an environment variable that the runtime checks to see if it needs to honor the `tied tasks` semantics or not).

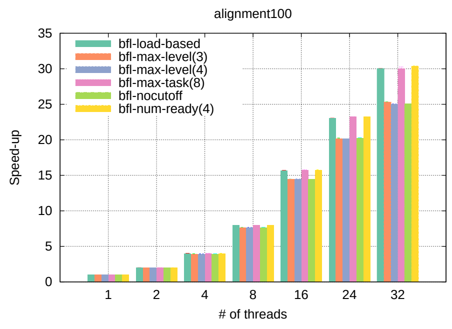
5.5.3 Results

In this section we present several lessons we have learned about task scheduling from this evaluation.

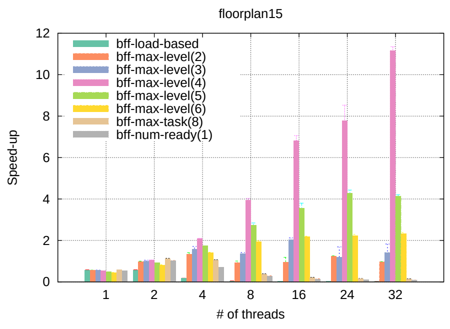
Lesson 1: Cutting off: yes, but how?

Initially, we wanted to know whether using a cut-off function would be useful and if so which one would be most useful. We evaluated all the applications (some of them with different inputs) with different cut-offs and schedulers and also with no cut-off function.

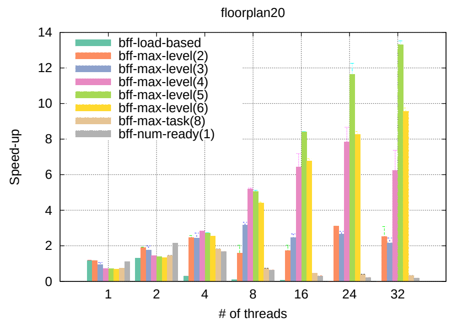
Figure 5.6 shows the speed-up of four of the applications (*Alignment*, *Floorplan*, *SparseLU* and *Multisort*) . For each of them, we show some representative cut-off strategies and also the performance without a cut-off for a fixed scheduler policy (the best one for each case). The number between parenthesis in the cut-off name indicates a value of n for that cut-off.



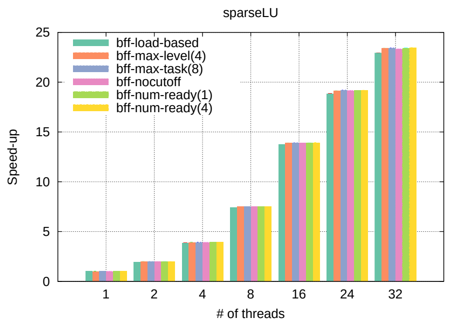
(a) Alignment (100 sequences)



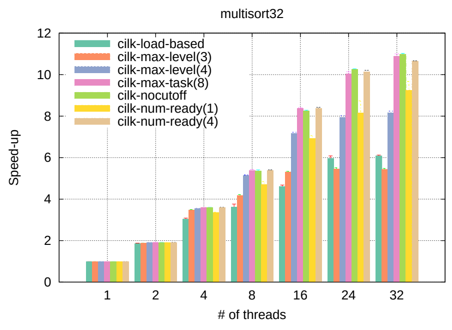
(b) Floorplan (15 cells)



(c) Floorplan (20 cells)



(d) SparseLU (50 blocks of 100x100 elements)



(e) Multisort (32M complex numbers)

Figure 5.6: Performance of difference cutoffs

We can see that in both *SparseLU* and *Multisort* the use of a cut-off strategy does not improve over not using one at all. In all other applications, the use of a cut-off does help to obtain a better performance.

In the case of the *Alignment*, the gain obtained is a good 20% and that is the case where there is less improvement. In *Floorplan*, not using a cut-off is so bad that we could not even finish the executions because our CPU time expired.

Another observation is that the best cut-off strategy depends on the application. For example, in *Alignment* the best strategy is a *max-task* strategy but in *Floorplan* a *max-level* strategy works best. Choosing a wrong strategy can, in fact, be worst than no using a cut-off strategy at all as it happens in *Multisort*.

But, even for the same application, not always the same cut-off is going to be the best. For example, in *Floorplan* we can see that if there are 15 sequences as input an strategy cutting off at level 4 works best but if there are 20 sequences then it works best to cut off at level 5.

So, while it seems clear that using a cut-off technique is a key factor in most cases to obtain a good performance, the decision of which to use remains unclear because it depends on the exact application and the input data.

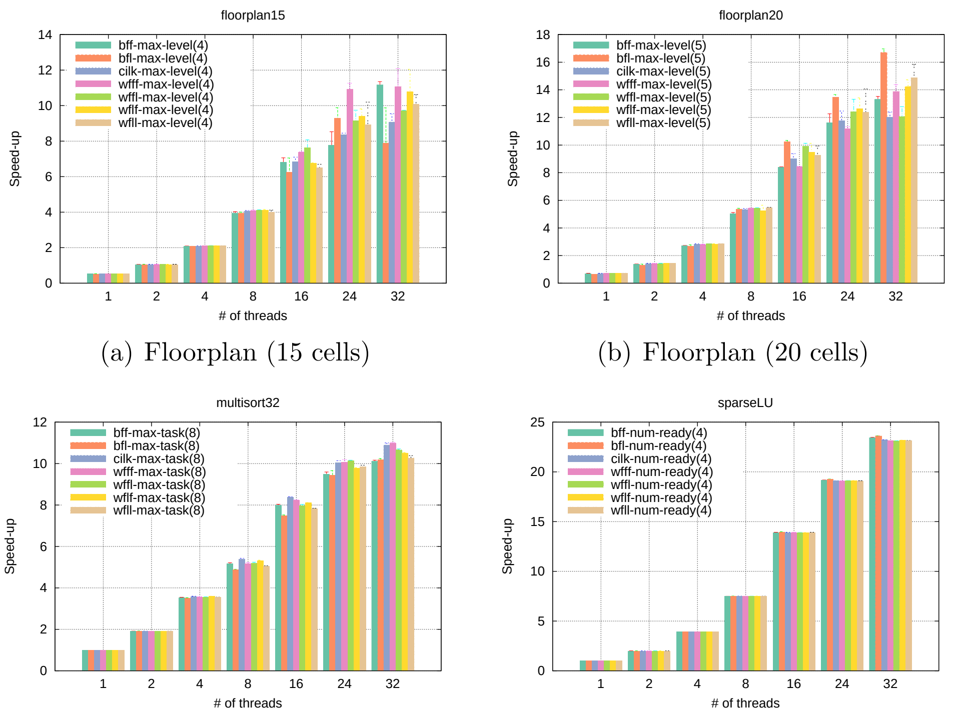
Lesson 2: Work-first schedulers work best?

Figures 5.7 and 5.8 show the speed-up obtained with different schedulers with a fixed cut-off strategy (the one that works the best).

In the *Floorplan* application (Figures 5.7(a) and 5.7(b)), with few threads all the different schedulers seem to obtain about the same performance. As the number of threads increases to 32, the work-first scheduler *wfff* obtains the best performance with an input of 15 cells. With an input of 20 cells, it is *bfl* the scheduler that obtains the best speed-up.

In the *multisort* application (Figure 5.3) two work-first schedulers obtain a slightly better performance than the rest (*cilk* and *wfff*). In this application, all work-first schedulers obtain a better speed-up than the breadth-first schedulers. This is because of the better use of locality of work-first schedulers.

In *sparseLU* (Figure 5.7(d)), all the scheduler obtain roughly the same speed-up although as the number of threads is increased, we can see that *bfl* and *bff* obtain a slightly better speed-up.



(a) Floorplan (15 cells)

(b) Floorplan (20 cells)

(c) Multisort (32M integers)

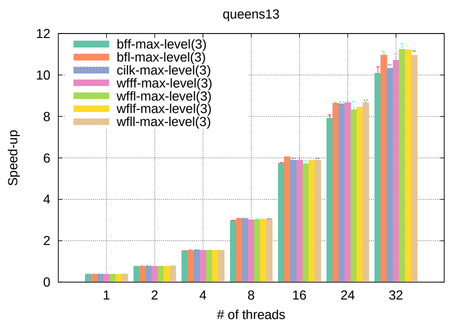
(d) sparseLU (50 blocks of 100x100 elements)

Figure 5.7: Speed-ups with different schedulers

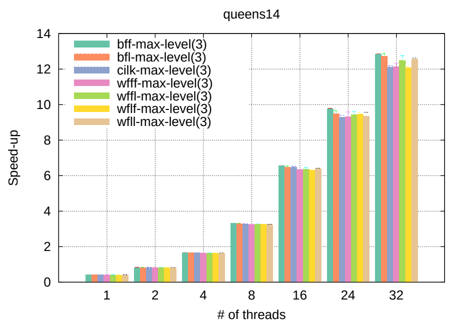
In the *N Queens* application with a board of size 13 (Figure 5.8(a)), the *wffl* scheduler obtains the best speed-up but other work-first schedulers (*wflf* and *wfll*) and the *bfl* get almost the same performance. With a board of size 14 (Figure 5.8(b)), the best speed-up is obtained by the *bff* and *bfl* schedulers. The *wffl* and *wfll* work-first schedulers obtain almost the same speed-up.

In *Strassen* (Figure 5.8(c)), all work-first schedulers obtain a slightly better performance than breadth-first schedulers. Among them, the *cilk* and *wfff* schedulers obtain the best results but only by a small margin.

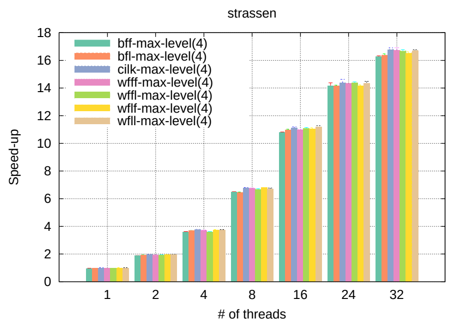
In *FFT* (Figure 5.8(d), again, all work-first schedulers get a slight better speed-up than breadth-first schedulers with the *cilk* scheduler obtaining the best speed-up among work-first schedulers.



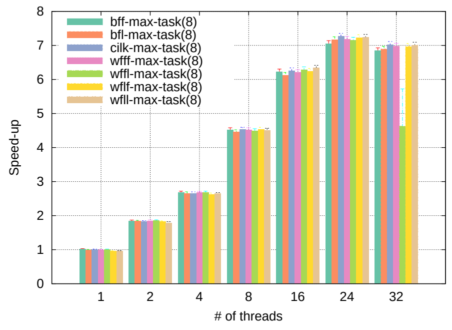
(a) N Queens (13x13 board)



(b) N Queens (14x14 board)



(c) Strassen (1280x1280 matrix)



(d) FFT (32M integers)

Figure 5.8: Speed-ups with different schedulers

Overall, we can see that there is not much of a difference between the different schedulers. Work-first schedulers tend to get a slightly better performance because they exploit data locality better but the difference in performance is very small compared to the impact that the election of a cut-off strategy has. The main reason behind this is that, because of cut-offs, most of the tasks of an application are executed immediately and the scheduler decision has no effect over them. Moreover, the task executed immediately do so in a manner that is very close to work-first schedulers.

Lesson 3: Deep-first schedulers should be the default

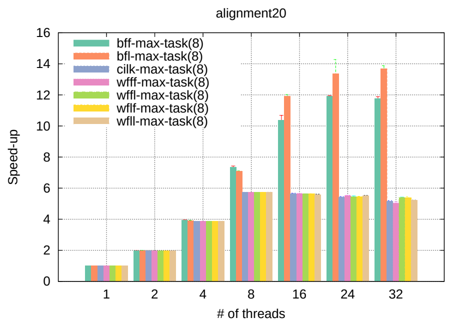
Figures 5.9 and 5.10 show how the same schedulers perform when the tasks are `tied` instead of `untied` as in the previous section.

We can observe, that for the *Alignment* application (Figures 5.9(a) and 5.9(b)), while with a few threads all schedulers seem to get about the same performance, both breadth-first schedulers outperform work-first schedulers as the number of threads grows. Among the breadth-first schedulers, *bfl* obtains a little bit more of performance than *bff*.

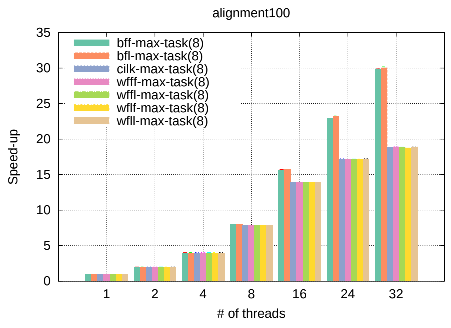
In the *Floorplan* application (Figures 5.9(c) and 5.9(d)), work-first scheduler performance is even worse. None of them is able to improve over the serial version. Meanwhile, both breadth-first schedulers scale up. In this application, the best breadth-first scheduler depends on the input: so while with an input of 15 cells, *bff* works best, with the 20 cells input, it is *bfl* the scheduler that obtains a better performance.

We can quickly see, that in the remaining applications, the same pattern holds true: if the tasks are `tied`, the performance of the work-first schedulers is severely degraded to the point that no speed-up is obtained. The reason behind this behavior is because work-first schedulers switch to the newly created tasks and no other thread can pick that task (because it is `tied`). Thus, the amount of created tasks is severely reduced. In most cases, there are no eligible tasks at all for other than the first thread which explains why execution time remains constant.

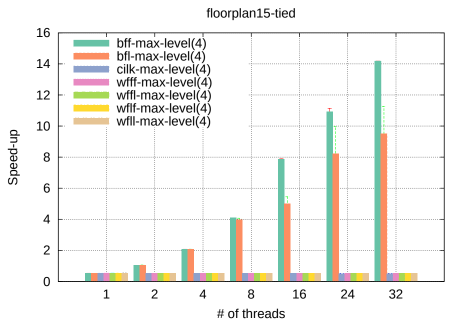
In the light of these results, it seems a wise choice for a compiler (or runtime library) to choose a breadth-first scheduler because the overall performance with `tied` and `untied` tasks is much better than work-first schedulers. While both bread-first schedulers obtain a very similar performance, it seems that *bfl* is slightly better most of the times. This happens in the *alignment* application either 20 (Figure 5.9(a)) or a 100 sequences (Figure 5.9(b)), in *Floorplan* with 20 cells (Figure 5.9(d)), in *sparseLU* (Figure 5.10(a)), in *Nqueens* with both boards sizes (Figures 5.10(b) and 5.10(c)). In contrast, *bff* only clearly wins in *Floorplan* with a 15 cells input (Figure 5.9(c)) and it seems to be slightly better for *FFT* (Figure 5.10(e)).



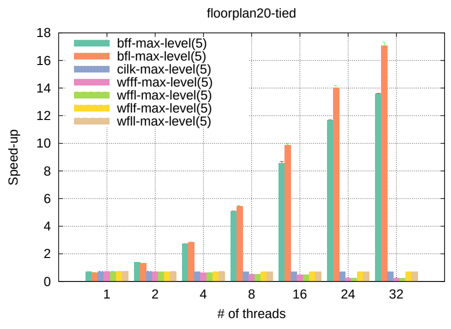
(a) Alignment (20 sequences)



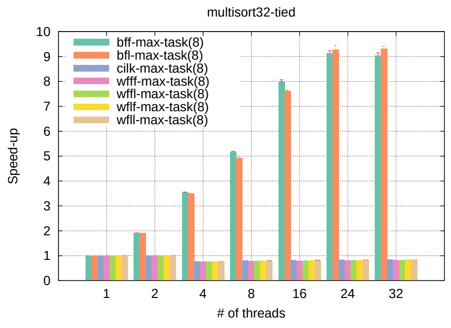
(b) Alignment (100 sequences)



(c) Floorplan (15 cells)

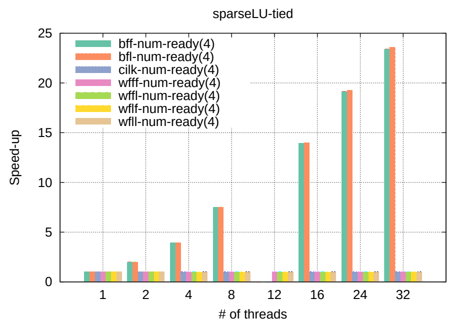


(d) Floorplan (20 cells)

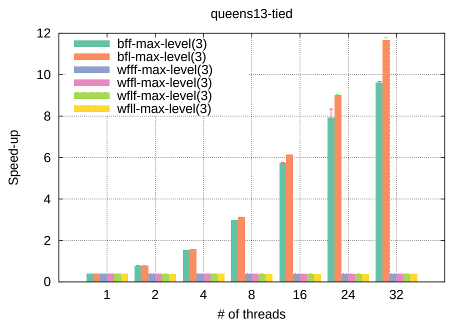


(e) Multisort (32M integers)

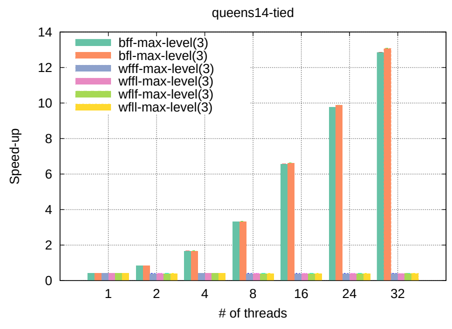
Figure 5.9: Speed-ups with tied tasks



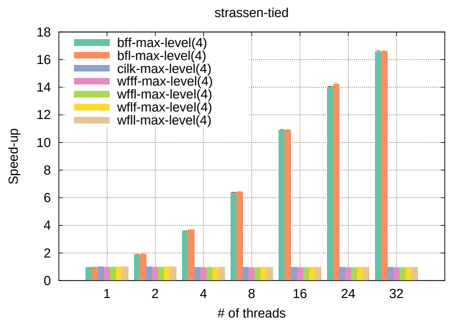
(a) sparseLU (50 blocks of 100x100 elements)



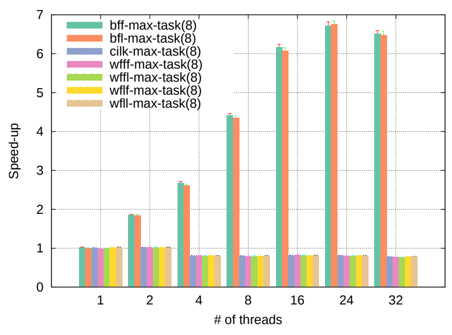
(b) N Queens (13x13 board)



(c) N Queens (14x14 board)



(d) Strassen (1280x1280 matrix)



(e) FFT (32M complex numbers)

Figure 5.10: Speed-ups with tied tasks

5.6 Conclusions

We have developed several applications from very different domains making use of the new *OpenMP* tasking model which has allowed both to gain insight inside the new model and have a group of benchmarks to conduct further experimentation.

We have implemented several scheduling policies, based on breadth-first and work-first families, and cut-off strategies for this tasking model and evaluated them with the aforementioned applications.

The evaluations shows that the impact of the cut-off function in the performance of an application can be very large as it allows to reduce the overheads associated with the creation of more tasks than needed. But, the appropriate cut-off to use is dependent on the application. This leaves the user no choice but to either do some analysis of what might be better for his application or just follow a try-and-error approach. In the next chapter, we will introduce a self-tuned technique that will choose the right cut-off without user intervention by gathering information about the tasks at runtime.

The evaluation of the schedulers shows that while with `untied` tasks work-first schedules work slightly better that breadth-first schedules the selection of a correct cut-off strategy has a much bigger impact. On the other hand, with `tied` tasks breadth-first schedulers seriously outperform work-first schedulers. This seams to indicate that the correct default scheduler should be some kind of breadth-first scheduler as `tied` tasks are the default in *OpenMP*.

Chapter 6

Self-tuned task granularity cut-off

A mind all logic is like a knife all blade

Rabindranath Tagore

Bengali poet (1861–1941)

Abstract

In task parallel languages an important factor in achieving good performance is the use of a cut-off strategy that reduces the number of tasks created. Only enough tasks need to be created to keep all the threads busy but and at the same time it is interesting to avoid very fine grained tasks. Unfortunately, the best cut-off strategy its usually dependent on the application structure or even the input data of the application.

In this chapter, we describe a new cut-off technique that, using information from the application collected at runtime, decides which tasks should be cut-off to improve the performance of the application.

6.1 Introduction & Motivation

Our evaluation of different scheduling policies and cut-off strategies (that allow to dynamically increase the granularity of tasks) in Chapter 5 showed that both are important decisions to obtain a good performance from an application using the *OpenMP* task model. But, of the two, choosing the right cut-off strategy is by far the more critical decision. Many times the cut-offs are hand-coded into the application by the programmer. This critical decision can be time consuming and difficult to made as it may depend on variable factors like the input data of the application.

We propose a runtime mechanism, that we call Adaptive Tasks Cutoff (ATC), which dynamically decides the most convenient cut-off for the application. This technique is based on profiling information collected at runtime in order to discover the granularity of the tasks created by the application and cut-off them appropriately.

We have implemented our proposal in the Nanos *OpenMP* environment (although the mechanism is not based in any *OpenMP* or Nanos dependent feature) and evaluated it with a set of diverse applications and benchmarks in an Altix system. Our results show that ATC, in most cases, is able to cut-off tasks adequately to achieve a good speed-up without the intervention of the programmer.

6.2 Profiling tasks

To adaptively coalesce *OpenMP* tasks by employing a cut-off to prune excess parallelism, we needed to gather information about the application at runtime. To obtain this information we have implemented a dynamic profiler in our Nanos runtime [TMD+07]. The Nanos runtime is a research *OpenMP* runtime which implements most the major features. Nanos uses user-level threads, called nano-threads [Pol93], on top of POSIX threads which are created when the application starts. For each *OpenMP* task that is spawned, a nano-thread is eagerly created. But, if a task is executed immediately only a small context is allocated in the nano-thread stack and the same nano-thread that encountered the task executes it. Consequently, multiple *OpenMP* tasks can be executed by a single nano-thread.

We are interested in two kinds of information: the amount of work in each user specified task and the amount of work each nano-thread (or real task) has done.

In particular, we want to know from a given node in a recursive tree how much total work there is in the subtree spanning from that node (including the node itself). Note that regular loops with tasks are just a particular case of a recursive tree (i.e. with just one level of depth). This will allow us to predict how much work a future task created at the same level will do.

We keep track of the time each user *OpenMP* task spends running. As we are interested in the computational load we disable the timers at synchronization points (such as `taskwait`). We have two timer counters associated with each *OpenMP* task: one for the work load of the task itself and another for the time of all its descendants (i.e. its spanning tree). Then, we have another time counter for the total execution time of a nano-thread.

When an *OpenMP* task finishes, it processes its profile information. This processing consists of three steps:

1. The task adds its time to the total time of the nano-thread.

2. The task adds its time to the tree time of its *OpenMP* parent task. Note, that as the *OpenMP* parent task might be being executed by a different thread this update needs to be protected by mutual exclusion.

3. The task updates a shared depth-level indexed structure where we keep the average computational load of the subtree spanning from a given depth level in the recursive tree. This operation also needs to be performed under mutual exclusion.

As this post-processing can have a large impact in the application we have implemented three different profiling modes:

Full mode In this profiling mode, we collect all the information so we have a very accurate description of the application behavior.

Minimal mode In this profiling mode, we only collect the total time of the nano-thread. Thus, the overhead of the profiling is minimal as there are no updates from other threads in the same memory locations (which require extra synchronization). The problem with this mode is that it does not obtain enough information to feed or algorithm.

Adaptive mode In this profiling mode, the application starts in full profiling mode but it progressively switches to minimal mode to avoid overheads. When enough samples (a runtime parameter currently defined to 100) are collected at a given depth-level profiling in that level is switched to minimum.

The information obtained from this profiling is used to predict the behavior of future tasks by our adaptive cut-off algorithm.

6.3 Adaptive tasks cut-off (ATC)

In this section we present our adaptive tasks cut-off (ATC). The cut-off uses information obtained from the profiler to decide whether or not to allow the creation of a new task. ATC does not need any source code modification from the user. It will be invoked by the runtime whenever an OpenMP task is about to be created to decide whether to prune it or not.

6.3.1 Design objectives

In order to design our cut-off we have tried to achieve the following objectives that in our opinion will maximize the application performance:

1. Obtain profiling information quickly. For the cut-off to be effective, we need to obtain information from the profiler quickly. This means, we need to force threads to do a depth first execution as soon as possible to obtain information about the spanning trees.

2. Generate enough tasks for all threads. Obviously, we do not want threads to be idle unless absolutely necessary (*i.e.*, unless creating tasks

for them is too expensive). The goal is to generate enough tasks so the scheduler can avoid load imbalances by giving work to all the threads.

3. Do not allow an unbounded number of tasks. We want to limit the number of task created in order to reduce the resources allocated to the runtime.

4. Avoid fine-grain tasks. Doing this we try to reduce the overheads associated with task creation and only create tasks when their computational load pays off the overhead of creation. By reducing the overhead we improve the performance of the application.

Several of these objectives are contradictory (e.g. objectives #1 and # 2). We have tried to balance them adequately in the design of the algorithm.

6.3.2 The ATC algorithm

Initially, as the cut-off still does not have information from the profiler we have a decision process that, based in our previous observations [DCA08], in most cases it will maximize our design objectives. Tasks are allowed to be created if the following two conditions are met:

1. There are fewer ready tasks than twice the number of threads in a given level. This condition serves two purposes: first, it forces the threads to go deeper into the tree so the profiler get information (objective #1); second, it restricts the number of tasks to a bound number (objective #3) while allowing a minimum number of tasks to ensure that all threads will have work (objective #2).

2. The depth-level is less than a certain limit (defined to be 4 for us). This condition tries to be conservative in allowing the creation of deep tasks (which tend to be fine-grain, objective #4) at this stage of the execution.

Note that by combining both conditions, we only generate the top level tasks that are the most promising however we do not generate too many of them. Note that combining both condition, we are only generating top level tasks which are the most promising, but we do not generate so many of them that we later regret our decision.

Once the profiler has gathered information about a level an estimation of computational load that the task would have is computed. The estimation, currently, is the average computational load of all the sub-trees spanning from the level of the tree that has been profiled (which cannot be obtained with the minimal profile mode). This estimation assumes that all tasks of a given level will have a similar behavior. It is a very simple approach that can be changed in the future to use more powerful prediction techniques.

```
level = current level + 1
if ( level is closed ) then
  not create
else
  estimate = estimate_task_size(level)

  if no estimate then
    if ready_tasks[level] < 2 * number of threads
       and level < default_max_level then
      create
    else
      not create
    fi
  else
    if estimate > mininum_size and
      total_ready_tasks < 4 * number of threads then
      create
    else
      not create
    fi
  fi
fi
```

Figure 6.1: Adaptive task cut-off pseudo-code

If the predicted grain size is smaller than a certain value (we use 1 millisecond[1]) then the task will not be created (objective #4). Otherwise, it can be created if there are not enough ready tasks for the threads to execute. This, again, ensures some bound on the number of tasks (objective #3) but generates sufficient parallelism for the threads (objective #2).

The overhead of making this decision can be very large in applications with very fine grain tasks. As an optimization, to reduce it, we allow the

[1]This value was obtained through microbenchmarking of task creation

Application	**Description**	**Notes**
Strassen	Matrix multiplication	
N Queens	Solves the n-queens problem	Branch and bound
Multisort	Sorts an array of integers	
sparseLU	LU matrix factorization	
Floorplan	Computes a floor plan distribution	Branch and bound
Alignment	Aligns protein sequences	Tasks are tied

Table 6.1: Applications summary

profiler in the adaptive mode to mark a level as *closed.* When all the samples of a level have been collected the profiler checks which is the estimated time for that level, and as it will be constant in the future (because no more samples will be collected), if the estimation determines the grain is to small the level is *closed.* This allows the cut-off to make a decision with just a comparison. Note, that closing a level does not preclude that all tasks in deeper levels are cut off.

6.4 Evaluation

6.4.1 Applications

For the evaluation of our cut-off proposal we used the applications described in Chapter 5. We summarize them in table 6.1.

In all applications (except *Alignment*) we marked all tasks as *untied* and we removed all manual cut-offs from the programmer leaving total scheduling freedom. The *Alignment* application makes heavy use of `threadprivate` and, because of that, we could not mark the tasks as `untied`.

6.4.2 Experimental setup

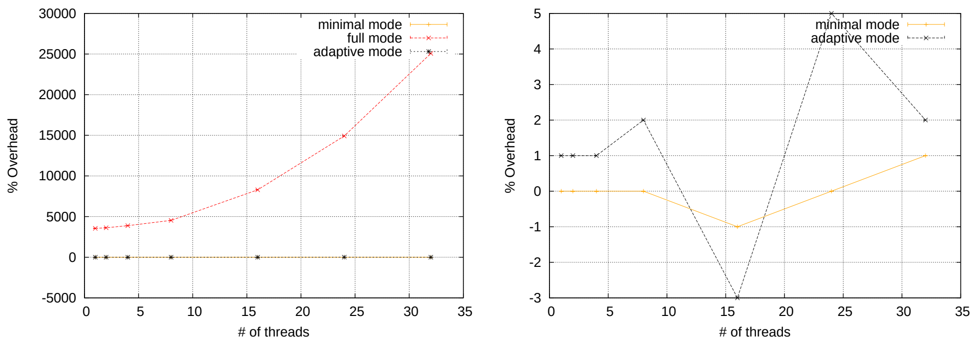
We evaluated all the benchmarks on an SGI Altix 4700 inside a CPU partition (with its own memory) of 32 processors to avoid interferences with other running applications.

We compiled all applications with our Mercurium compiler [BDG$^+$04] using gcc with option -O3 as the backend. The serial version of the application was compiled with gcc -O3 as well. The speed-ups were computed using the serial execution time as the baseline and using the average execution time of 5 executions.

6.4.3 Profiler impact

The first question we want to address is how much overhead the profiler introduced. This will allow us to determine if it was worth the additional complexity of enabling and disabling the profiler instead of continuously running the profiler.

We used *NQueens* for this experiment because it has the finest-grained tasks of all the benchmarks. If the overhead of the profiler is low enough for *NQueens* it will most likely have a low impact for the others applications with coarser tasks.



(a) All modes (b) Only minimal and adaptive modes

Figure 6.2: Overhead of Queens (board of 13x13) with different profiling modes

Figure 6.2 shows the overheads of the different profile methods (compared against a non-profiled version) with a chess board of size 13x13. We can see that, while only profiling the nano-thread execution (the minimal profiling mode) time has almost no impact, profiling the whole tree structure (full profiling mode) severely reduces the performance obtained due to the overhead of the profiling. But, limiting the amount of samples we collect to

Name	Description
work-stealing	Work-first scheduler with work-stealing.
bff	Breadth-first scheduler (FIFO order).
bfl	Breadth-first scheduler (LIFO order).

Table 6.2: Summary of used schedulers

Name	Description
max-level=N	Tasks are cut off based on their depth (N is the level where they are cut).
max-tasks=N	Tasks are cut off based on the total number of tasks in the system (N per num-threads is the number of allowed tasks).
num-ready=N	Tasks are cut off based on the number of ready tasks in the system (N per num-threads in the number of allowed ready tasks).
load-based	Tasks are only created if there is an idle thread at creation time.

Table 6.3: Summary of used cut-offs.

one hundred and then disable the full profiling (adaptive mode) reduces this impact to a minimum barely noticeable (with the overhead being at most around 5%).

Therefore, the adaptive profiling mode is the one we have used to obtain the information for our adaptive cut-off mechanism.

6.4.4 Cut-off evaluation

Methodology

We wanted to compare our adaptive cut-off with the best cut-off for each of the applications. Firs we looked for the best schedule and cut-off pair. We executed all the applications with the cross-product of combinations of schedulers and cut-offs (Table 6.2 and Table 6.3 summarize the schedulers and cut-offs we have used). For those cut-off that worked well we tried several variations with different parameters. We then fixed the schedule to the one obtained in the previous step and we also tried variations of the cut-offs that seemed to work worst to try to find the worst one.

For some applications we did this experiments exploring different input sizes to see how our algorithm performs with variations of the input data.

Table 6.4 summarizes the best and worst cut-off and the schedule (which was the one that obtained the best speed-up) used for application and input. Keep in mind that many times there are several cut-offs which are close to the best and worst ones. We consider this values as the bounds to measure the success of our adaptive cut-off: we would like to be as close as possible to the best cut-off and as far away as possible from the worst one.

Scenario	**Schedule**	**Best cut-off**	**Worst cut-off**
Multisort (32 million integers)	work-stealing	max-tasks=8	maxlevel=3
Alignment (20 sequences)	bfl	max-tasks=8	load-based
Alignment (100 sequences)	bfl	max-tasks=8	num-ready=4
Strassen (1280x1280 matrix)	work-stealing	max-level=4	num-ready=4
Floorplan (15 cells)	bff	max-level=4	num-ready=1
Floorplan (20 cells)	bff	max-level=5	num-ready=1
Queens (13x13 board)	bfl	max-level=3	max-tasks=8
Queens (14x14 board)	bfl	max-level=3	max-tasks=8
SparseLU (50 blocks of 100x100 elements)	bfl	num-ready=4	load-based

Table 6.4: Summary of best and worst cut-offs

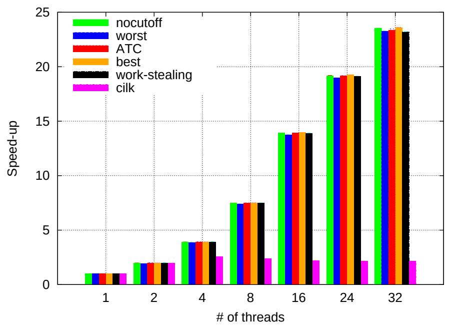
Results

In the following results, we present for all the experiments the speed-up obtained without using any cut-off, with the best cut-off, with the worst cut-off, with our adaptive cut-off (labeled ATC) and with a work-stealing scheduler (with no cut-off and no lazy creation) for comparison purposes. For those applications (*multisort*,*nqueens* and *sparseLU*) that we have the corresponding Cilk code we also evaluated them. For each application we show the speed-up obtained and also the number of tasks created at each depth level with the different cut-offs plus the potential number of tasks at each level (i.e. the number of OpenMP tasks defined by the user). This allows us to observe the differences in behavior among the different cut-offs.

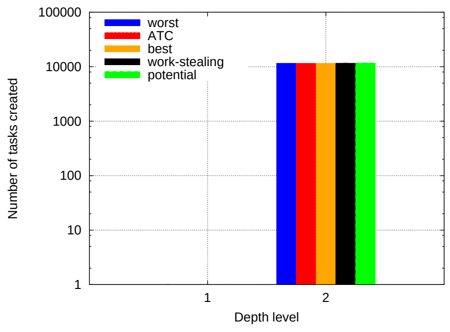
Figure 6.4 and Figure 6.3 show the results for the *Multisort* and *SparseLU* applications respectively. We can see that ATC does a good job and there is almost no difference in speed-up with the best cut-off as the number of tasks

Label	Meaning
nocutoff	Best schedule with no cut-off
worst	Best schedule with worst cut-off
ATC	Best schedule with ATC
best	Best schedule with best cut-off
work-stealing	Work-first schedule with no cut-off
cilk	Cilk version

Table 6.5: Summary of labels used in the evaluation



(a) Speed-ups



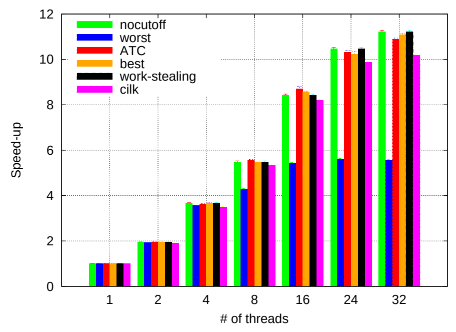
(b) Number of tasks created with 32 CPUs

Figure 6.3: Results for sparseLU (50 100x100 blocks)

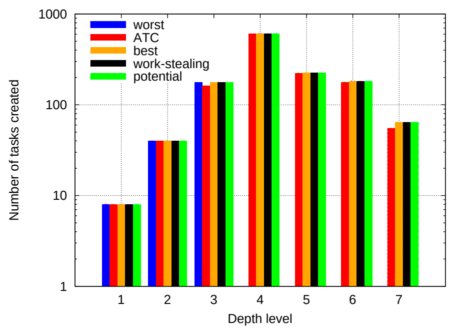
created at each level are roughly the same. Both applications have coarse grain tasks and benefit the most by applying some bound to the number of tasks created.

Figure 6.5 presents the performance obtained for the *Strassen* application. While ATC does not do a bad job, it opens up one more level (the 5th level) than the best cut-off and that reduces the achieved speed-up. We have checked the profile information and the size of the tasks at depth 5 are coarse enough to be executed. So, probably there are other factors we are not accounting for (e.g. amount of synchronization) that affect the identification of the right cut-off point. ATC is much better than some of the other cut-offs so we still see this as a positive result because is much closer to the best one than to the worst one.

Figure 6.6 and Figure 6.7 we show the results for the *Alignment* bench-

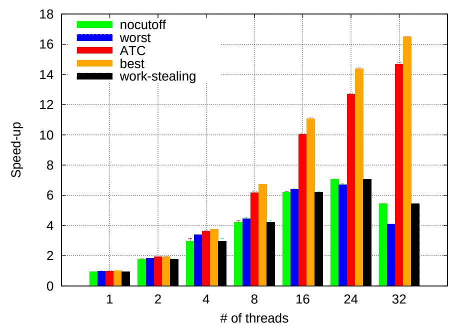


(a) Speed-ups

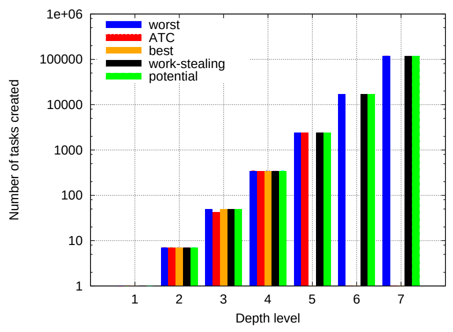


(b) Number of tasks created with 32 CPUs

Figure 6.4: Results for Multisort (32 millions integers)



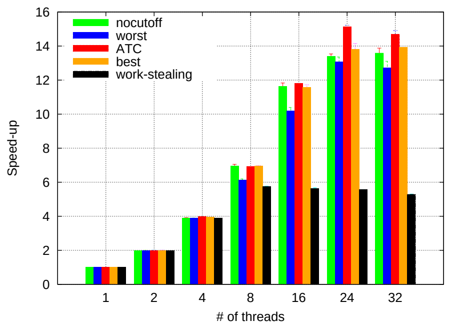
(a) Speed-ups



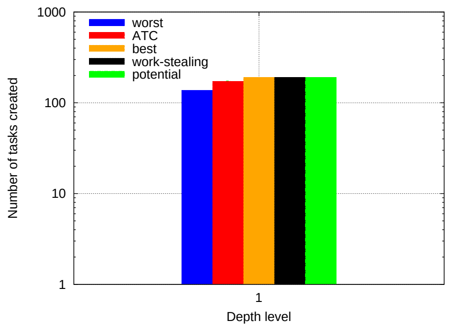
(b) Number of tasks created with 32 CPUs

Figure 6.5: Results for Strassen (size 1280x1280)

mark with two data sets: a small one of 20 sequences and a much bigger on of 100 sequences. In both cases, ATC makes good decisions and controls the number of tasks created better than the best cut-offs. The size of the tasks with the small data set is much smaller but ATC does a good job dealing with them. Note also than the gap between the best and worst cut-offs is very narrow for this application. In fact, because *alignment* uses `tied tasks`, unlike in other applications, the critical factor is the schedule being used and not as much the cut-off. For example, note how, with both inputs, that when no cut-off is used both work-stealing and the best schedule (*bfl* in this case) have the same amount of tasks to schedule. But, because `tied task` do not interact well with work-stealing the performance obtained is much lower.

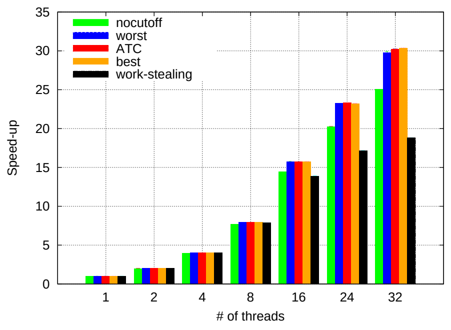


(a) Speed-ups

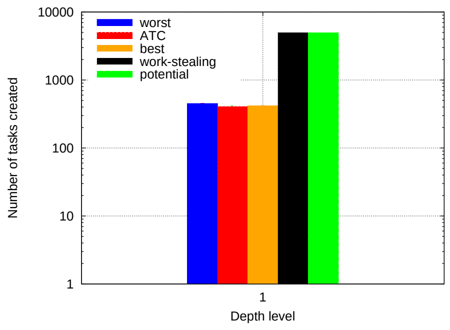


(b) Number of tasks created with 32 CPUs

Figure 6.6: Results for Alignment (20 sequences)



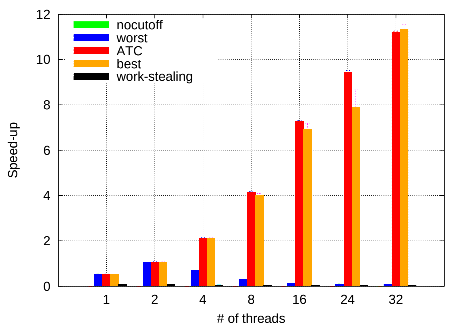
(a) Speed-ups



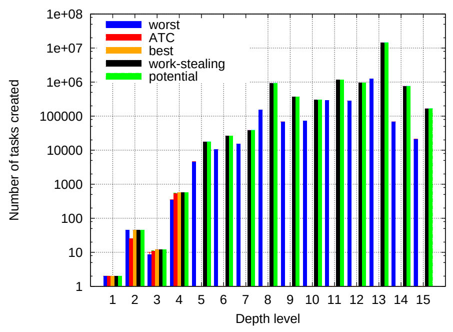
(b) Number of tasks created with 32 CPUs

Figure 6.7: Results for Alignment (100 sequences)

The results for the *Floorplan* benchmark are shown in Figure 6.8 and 6.9. We have also tried two input sets: one with 15 cells and another with 20 cells. The size of the problem increases exponentially with the size of the input and the number of potential tasks at some levels is over 40 million which causes severe performance problems for all the versions that do not have a cut-off (note that even the worst cut-off reduces drastically the number of created tasks) and we run out of CPU time in many executions. That is why there are no results for the *nocutoff* or *work-stealing* versions in many cases. *Floorplan* is a branch and bound algorithm and, as such, it is highly irregular in the size and number of tasks created at each level (see in Figure 6.9(b) for the potential number of tasks at each level). Because of this, with

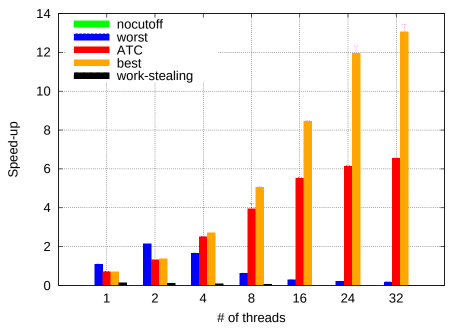


(a) Speed-ups

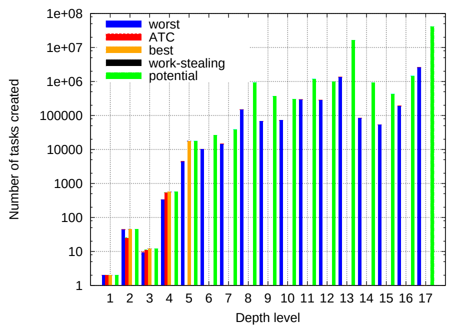


(b) Number of tasks created with 32 CPUs

Figure 6.8: Results for Floorplan (15 cells)



(a) Speed-ups

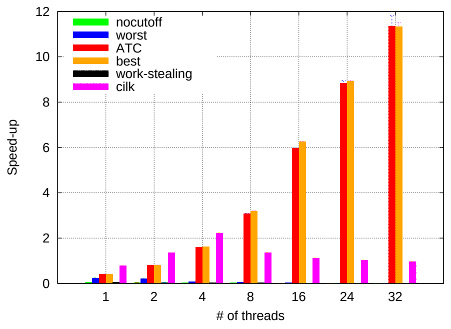


(b) Number of tasks created with 32 CPUs

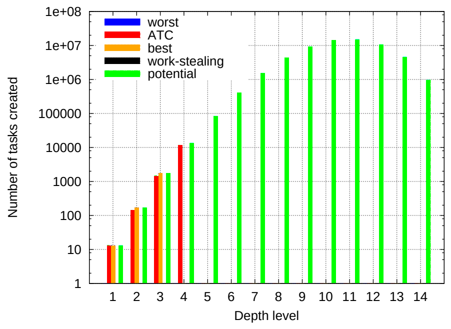
Figure 6.9: Results for Floorplan (20 cells)

one of the inputs ATC cuts the creation of tasks too soon which unbalances the application resulting in a less than optimal speedup. The scenario where there are many top level branches but a few of them accumulate most of the work load is, probably, the worst for our simple prediction technique. Even so, ATC manages to perform much better than the worst cut-offs (including several level-based cut-offs) and even the worst cut-offs reduce the amount of created tasks by a large amount. This allows the executions to finish which does not always happen with no cut-off.

Figure 6.10 and 6.11 show the results for the *NQueens* benchmark with two inputs: a 13x13 board and a 14x14 board. Again, the size of the problem increases exponentially with the size of the input. Also, being a backtracking

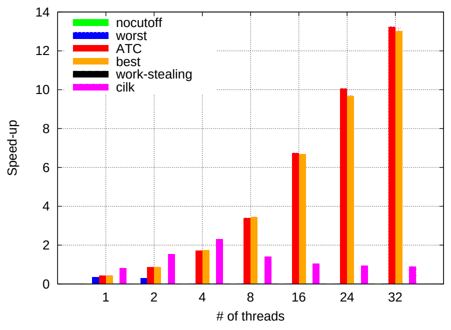


(a) Speed-ups

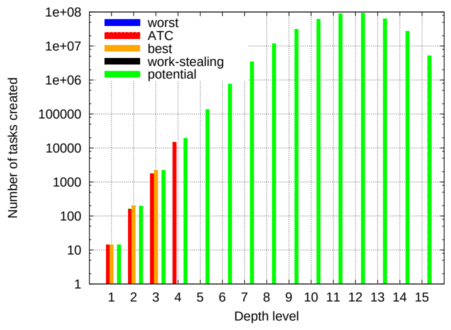


(b) Number of tasks created with 32 CPUs

Figure 6.10: Results for N Queens (13x13 board)



(a) Speed-ups



(b) Number of tasks created with 32 CPUs

Figure 6.11: Results for N Queens (14x14 board)

algorithm it is very irregular. However, as the pruning increases in a regular way as we progress down the task tree ATC does a good job in detecting it. And we can see it obtains the same performance as the best cut-off as they both cut task creation at level 3. The performance of the cut-offs not based on the depth level was so poor we were not able to obtain a result with more than a few CPUs because we ran out of CPU time (1 hour maximum) for a baseline run of a few minutes. The number of tasks generated for those cases are in the order of millions like in *Floorplan* but with even smaller granularity. Because the tasks are so fine, this benchmark allows us to verify that, with the optimizations we performed, the cut-off overhead can be kept to a minimum.

Overall, ATC makes very good decisions. In all but two cases ATC does find an near-optimal cut-off and, in those two cases, its decisions are much better than some of the worst decision that a user can make. Note that in many scenarios when the level-based cut-offs obtain good results the ones based on number of tasks do not (and vice versa) while ATC works well overall. We think this results prove that is a good cut-off technique to be used to save time to the average user that does not have (or does not want) the time to explore all cut-off possibilities to find the optimal.

6.5 Conclusions and future work

We have presented an adaptive technique for task parallel languages, that we call Adaptive Tasks Cut-off (ATC), to reduce the number of created tasks. Tasks that are cut-off have no chance to be spawned (even lazily) thus reducing the overhead. This is particularly in the case of very fine grain tasks where we have seen that even lazy creation might be to costly (*e.g.*, N Queens). ATC uses dynamic profiling of the application to estimate the granularity of the tasks that are being created. The profiling is progressively switched off dynamically to reduce the overhead it causes.

Our evaluation, with a set of applications with very different properties, shows that ATC, in most cases, correctly discovers the granularity of the tasks and it decides an appropriate cut-off. In all cases it behaves much better than other possible cut-offs that can be selected by inexperienced (o careless) users. This suggests that ATC is a good option for both a naïve user and as a default in a parallel runtime.

Although we have implemented the ATC cut-off in the context of *OpenMP*, we think that the technique is general enough, and by no means tailored to *OpenMP*, so it can be useful in other languages with support for task parallelism (particularly with fine grain tasks).

At present, our profiler and cut-off estimation is naïve and should be extended in several ways. First, it should enable one to characterize different tasks separately so that decisions for different tasks can be made independently. Second, in some applications the same kind of task may change its behavior multiple times through the lifetime of the application. Once the detailed sampling is disabled, the profiler can use the information obtained in the minimal mode to verify that the behavior is consistent with the prediction. If an inconsistency is found then the detailed profiling can be enabled again to collect samples again. Last, the prediction technique should be extended to deal better with unbalanced codes like *Floorplan*.

Other lines of future work would try to extend the evaluation with more applications. Particularly with application that mix both `tied` and `untied` tasks. It would also be interesting to study how the quality of the prediction relates to the number of sample. Another direction we would like to investigate is the resource usage by the different cut-offs because although the performance may be similar in some cases, the number of concurrent tasks (and thus the system resources usage) can be quite different. This might be important in environment where resources are scarce.

Chapter 7

Conclusions and Future Work

A life spent doing mistakes is not only more honorable but more useful than a life spent doing nothing

Carl Sagan

American astronomer (1934–1996)

Abstract

This chapter discusses the conclusions that can be extracted from this thesis work as well as different research possibilities for the future, to further explore the topic of parallel self-tuned runtime libraries .

7.1 Conclusions of this thesis

In this thesis we have presented several mechanisms that based on information gathered at execution time, adapt different parameters related to the exploitation of parallelism. From our work we can extract some general conclusions:

7.1.1 Adaptive algorithms

The overall objective of this thesis was to prove that is feasible to develop adaptive self-tuned algorithms for different aspects of the execution of parallel applications. We have done so for three of them: parallel loop scheduling, thread distribution in multilevel parallelism applications and task granularity control in task applications.

The evaluation results of the different techniques show that these algorithms obtain in most cases a performance close to a version that was hand-tuned by the programmer.

This is encouraging as it means that the time and knowledge that most users need to use parallel systems can be effectively reduced. A user may still need to deal himself with complex aspects if he wants to obtain the maximum performance out of a machine (just as sometimes programmers need to write in assembler to obtain the maximum performance). But, if it is just the odd case and not the general case it is a big step forward to improve the productivity of parallel systems.

Also, while we have developed these mechanisms in the context of the *OpenMP* language we believe that the general idea is applicable to any parallel language and that most of these mechanisms would have an easy translation to other parallel languages.

7.1.2 Profiling

From our experience in this work, we can identify one of the most important design factors: the profiling overhead. If the profiling overhead is too hight it does not matter how good the decisions of the algorithm are because they are never going to pay off. This is particularly important if we want to monitor all the lifetime of the application to check for changes in behavior and not make just a single decision at the beginning of the execution. So, special care needs to be taken in the design and implementation of the profiling.

One key idea is that usually the algorithms need very detailed information to take a decision but not so much information to verify afterwards that the decision was correct. This opens the door to implementing in the runtime library several profiling levels of detail with different trade-off ratios between detail and overhead.

These profiling mechanisms must be controlled by the self-tuned algorithms depending on the current state of its decisions. When they need detailed information, they can switch high-detailed profiling on but when minimum information is needed they can use a low-detailed low-overhead profiling mode.

7.1.3 Language design

We hope these results increase the trend of parallel languages that focus in specifying the parallelism of the application but leave freedom to the runtime library to adjust factors of the parallel execution.

Some of the language features, based on our experiences with *OpenMP* that help the development of self-tuned features are:

Region definitions The existence of regions of code (whether they are parallel regions, task regions or worksharing regions) help the runtime library to have entities that can be easily identified and properties attached to them. Otherwise, tricks like the DPD used in Chapter 4 to identify regions in *MPI* need to be used.

Malleability It is important for the language constructs to be able to adapt to changes in the execution parameters, like for example the number of threads, as the execution goes.

Separation of definition and execution The runtime library can only do so much if the language fixes how the parallel exploitation needs to happen. We are not arguing against having ways for the advanced user to specify exactly how things should be executed. But, having by default flexibility in the execution of the parallelism defined in the application allows to help the novel user to get decent performance with a relative effort.

7.2 Future work

Several questions remain open at the end of this thesis. They are outlined here as lines to be explored by future research:

7.2.1 More self-tuning

We have focused in the adaptation of three different aspects of the parallel exploitation but, of course, several other for which there is no self-tuned algorithm exist. As, an example, we list a few here:

Nested parallel regions execution When multiple parallel regions are present in an application the programmer needs to choose whether to use a single level of parallelism (and which of the available ones) or multiple levels. Some work to make this decision automatically already exists [DCL04] but only works for balanced loop nests.

Number of groups In applications with nested parallelism, we have seen that is important to choose a good thread distribution across the different *outer* groups of parallelism. But, as important, is to choose the right number of groups. So far, this is still a burden on the programmer.

Blocking size An important optimization to improve the performance of parallel algorithms (and also serial ones) is applying loop blocking. While the compiler can make the transformations easily for the user it needs to know the size of the block. This parameter depends on application and architectural factors and could be determined by the runtime library .

7.2.2 Integration

One of the most obvious questions not answered by this thesis is how multiple adaptive techniques can work together in a single runtime library .

This is a challenge that has two main issues:

- First, enough information needs to be collected for all the algorithms but we need to keep the profiling low. To achieve this we could design the self-tuned algorithms so they maximize the amount of information they share (and thus minimize the amount we need to obtain from the profiling). Or, we could the design the profiler so it only gathers a few metrics concurrently but then rotate which metrics are profiled over time. This may reduce the profiling overhead but it may increase the time needed to gather information so it remains to be seen if its useful.

- Second, there is a problem of conflictive decisions. So, for example, if we were deciding a thread distribution and at the same time trying to adapt a loop scheduler executed by those threads neither of the algorithms would probably take a wrong decision: a proper order of the decisions needs to be found. It will be important as well that the information obtained from the first algorithm is propagated to next ones.

7.2.3 Compiler and runtime library coordination

In our work, all our algorithm have used almost exclusively information that the runtime library gathered at runtime. While this approach guarantees that the algorithms will work even if we had a simple compiler or a complex application not suited for today's state-of-the-art analysis techniques, the compiler usually has a lot of information about the application that if coded appropriately in the application it can be very useful to the runtime library to take decisions.

Even, if the compiler information is not very accurate (e.g. obtained through a model) it would still be useful as the initial input to the adaptive algorithms. If it was inaccurate then it would later be refined with the information gathered by the runtime. This could allow to improve the initial decisions of the self-tuned algorithms and reduce the warm-up effect that we have observed in some situations.

But the coordination between the compiler and the runtime libraries can also take other forms. It is very common that the compiler can generate multiple optimized versions of a code but it does not have the information to chose one over the other. For example, a compiler can generate both a serial and a parallel version of an *OpenMP* `parallel` construct but it does not know with how many threads the application will run.

In cases like this, we propose the compiler could generate the different optimized versions and the runtime library can take the decision of which of them to execute depending of runtime factors. We have started to use this approach successfully in our task applications to reduce the overheads of the cut-offs even further by having a version of the code without runtime calls that gets invoked by the runtime library when it knows that further tasks will not be created.

7.2.4 Related information

Another interesting way to reduce the warm-up time is to find ways in that the profiled information and algorithm decisions from one unit (i.e. a parallel region, task region, function, . . .) is used by another. For example, a possible option would to identify loops with the same pattern. This would allow to reuse the scheduling decisions across several loops thus reducing both the warm-up time and the profiling impact (as we could probably switch faster to a low-overhead profile mode).

7.2.5 Hardware counters and new architectures

In our methods the main metric we have used to take decisions was the execution time of profiled regions but it would interesting to explore how to incorporate into the algorithms information obtained through hardware counters. This is particularly interesting with emerging architectures where

Bibliography

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The Data Locality of Work Stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[ABD$^+$03] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, R. Silvera, and X. Martorell. Is the Schedule Clause Really Necessary in OpenMP? In M.J. Voss, editor, *Proceedings of the International Workshop on OpenMP Applications and Tools 2003*, volume 2716 of *Lecture Notes in Computer Science*, pages 69–83, June 2003.

[ACD$^+$07] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In *Proceedings of the 3rd International Workshop on OpenMP*, Beijing, China, June 2007.

[ADE$^+$01] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *Lecture Notes in Computer Science*, 2104:1 – 10, 2001.

[ADH$^+$07] Eduard Ayguadé, Alejandro Duran, Joe Hoeflinger, Federico Massaioli, and Xavier Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[AFB92] Gad Aharoni, Dror G. Feitelson, and Amnon Barak. A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs. *Journal of Functional Programming*, 2(4):387–405, 1992.

[AGMJ04] E. Ayguadé, M. Gonzàlez, X. Martorell, and G. Jost. Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2004.

[AML^{+}99] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzàlez, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *Proceedings of the 1999 International Conference on Parallel Processing*, September 1999.

[aMS00] Dieter an May and Stephan Schmidt. From a Vector Computer to an MP-Cluster - Hybrid Parallelization of the CFD Code PANTA. In *Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[BBB^{+}91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[BBF^{+}01] Steve Behling, Ron Bell, Peter Farrell, Holger Holthoff, Frank O'Connell, and Will Weir. The POWER4 processor introduction and tuning guide. Technical Report SG24-7041-00, International Technical Support Organization, November 2001. ISBN 0738423556.

[BDG^{+}04] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP 2004*, October 2004.

[BFL^{+}01] A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, and C. Walshaw. Dynamic multi-partitioning for parallel finite element applications. *Parallel Computing*, 27(7):869–881, 2001.

[Bli02] R. Blikberg. Parallelizing AMRCLAW by Nesting Techniques. In *Proceedings on the 4th European Workshop on OpenMP (EWOMP2002)*, September 2002.

[BS99] R. Blikberg and T. Sörevik. Nested Parallelism: Allocation of Processors to Tasks and OpenMP Implementation. In *Proceedings of the 1999 International Conference on Parallel Processing*, September 1999.

[BS03] R. Blikberg and T. Sørevik. Nested parallelism in OpenMP. In *ParCo 2003*, 2003.

[Bul98] J. Mark Bull. Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In *European Conference on Parallel Processing*, pages 377–382, 1998.

[BYS^{+}01] T. Boku, S. Yoshikawa, M. Sato, C. G. Hoover, and W. G. Hoover. Implementation and Performance Evaluation of SPAM Particle Code with OpenMP-MPI Hybrid Programming. In *Proceedings of the 3rd European Workshop on OpenMP (EWOMP2001)*, September 2001.

[CE00] F. Cappello and D. Etiemble. MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks. In *Supercomputing'00*, Dallas, TX, 2000.

[CFLM04] B. Chamberlain, J. Feo, J. Lewis, and D. Mizell. An Application Kernel Matrix for Studying the Productivity of Parallel

Programming Languages. In *W3S Workshop - 26th International Conference on Software Engineering*, pages 37–41, May 2004.

[CLP$^+$08] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frédéric Arzel, and Nathalie Drach. A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Programs. In *Proceedings of the First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[CML05] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-Driven Processor Allocation. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):599–611, 2005.

[con05] The UPC consortium. UPC language specifications v1.2, May 2005.

[Cor01] IBM Corporation. POWER4 System Microarchitecture, October 2001.

[Cor04] Intel Corporation. Intel Itanium 2 Processor Reference Manual, May 2004.

[CSW$^+$08] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Utility-based Placement of Dynamic Web Applications with Fairness Goals. In *11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008)*, Salvador Bahia, Brazil, 2008.

[DCA08] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th International Workshop on OpenMP (To appear)*, 2008.

[DCL04] A. Duran, J. Corbalán, and J. Labarta. Runtime adjustment of parallel nested loops. In *Proceedings of the International*

Workshop on OpenMP Applications and Tools 2004, Lecture Notes in Computer Science, May 2004.

[DMSC99] D. Dent, G. Mozdzynski, D. Salmond, and B. Carruthers. Implementation and Performance of OpenMP in ECMWF's IFS Code. In *Proceedings of Fifth European SGI/Cray MPP Workshop*, Bologna, Italy, September 1999.

[DR00] Francis H. Dang and Lawrence Rauchwerger. Speculative Parallelization of Partially Parallel Loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.

[FCL01] F. Freitag, J. Corbalan, and J. Labarta. A Dynamic Periodicity Detector: Application to Speedup Computation. In *15th International Parallel and Distributed Processing Symposium*, pages 6 pp.–, April 2001.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[For95] Message Passing Interface Forum. The MPI message-passing Interface Standard, May 1995.

[GAL$^+$99] M. Gonzàlez, E. Ayguadé, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In *Proceedings on the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.

[GAM$^+$02] M. Gonzàlez, E. Ayguade, X. Martorell, J. Labarta, and Phu V. Luong. Dual-level Parallelism Exploitation with OpenMP in Coastal Ocean Circulation Modeling. In *Proceedings of the International Workshop on OpenMP: Experiences and Implementations (WOMPEI 2002)*, 2002.

[Gen97] Gene Myers and Sanford Selznick and Zheng Zhang and Webb Miller. Progressive multiple alignment with constraints. In *RECOMB '97: Proceedings of the first annual international conference on Computational molecular biology*, pages 220–225, New York, NY, USA, 1997. ACM.

[GOM$^+$01] M. Gonzàlez, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta, and N. Navarro. OpenMP Extensions for Thread Groups and Their Run-Time Support. *Lecture Notes in Computer Science*, 2017:324–338, 2001.

[GSC96] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996.

[Hen00] DS Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 10–10, Dallas, TX, 2000.

[HL94] Babak Hamidzadeh and David J. Lilja. Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing. In *International Conference on Parallel Processing*, pages 39–46, 1994.

[HLA94] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. Using the Run-time Sizes of Data Structures to Guide Parallel-thread Creation. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 79–90, New York, NY, USA, 1994. ACM.

[HT99] W. Huang and D. Tafti. A parallel Computing Framework for Dynamic Power Balancing in Adaptive Mesh Refinement Applications. In *Computational Fluid Dynamics 99*, 1999.

[hyp] Hyper-threading technology.
http://developer.intel.com/technology/hyperthread/.

[JdW06] Haoqiang Jin and Rob F. Van der Wijngaart. Performance Characteristics of the Multi-zone NAS Parallel Benchmarks. *J. Parallel Distrib. Comput.*, 66(5):674–685, 2006.

[JJJT03] H. Jin, G. Jost, D. Johnson, and W. Tao. Experience on the Parallelization of a Cloud Modelling Code Using Computer-Aided Tools. Technical report, NASA Ames Research Center, March 2003. NAS-03-006.

[JJY$^+$03] Haoqiang Jin, Gabriele Jost, Jerry Yan, Eduard Ayguadé, Marc Gonzàlez, and Xavier Martorell. Automatic Multilevel Parallelization using OpenMP. *Sci. Program.*, 11(2):177–190, 2003.

[KMB00] Philippe Kloos, Fabrice Mathey, and Philippe Blaise. OpenMP and MPI programming with a CG algorithm. In *Proceedings of the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[KNP06] Arun Kejariwal, Alexandru Nicolau, and Constantine D. Polychronopoulos. History-aware Self-Scheduling. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 185–192, Washington, DC, USA, 2006. IEEE Computer Society.

[KR04] Matthias Korch and Thomas Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurr. Comput. : Pract. Exper.*, 16(1):1–47, 2004.

[KRHHM89] D. A. Kranz, Jr. R. H. Halstead, and E. Mohr. Mul-T: a High-Performance Parallel Lisp. *SIGPLAN Not.*, 24(7):81–90, 1989.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[LH95] H-W. Loidl and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 8–10, 1995. Springer-Verlag.

[LR99] P. Lanucara and S. Rovida. Conjugate-Gradient Algorithms: an MPI-OpenMP Implementation on Distributed Shared Memory Systems. In *Proceedings on the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.

[Luc92] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of ACM SIGPLAN'92 Conference on Programming Language Desing and Implementation*, pages 220–211, 1992.

[MCN$^+$00] Xavier Martorell, Julita Corbalán, Dimitrios S. Nikolopoulos, Nacho Navarro, Eleftherios D. Polychronopoulos, Theodore S. Papatheodorou, and Jesús Labarta. A Tool to Schedule Parallel Applications on Multiprocessors: The NANOS CPU Manager. *Lecture Notes in Computer Science*, 1911:87–??, 2000.

[MKB00] Fabrice Mathey, Philippe Kloos, and Philippe Blaise. OpenMP Optimisation of a Parallel MPI CFD Code. In *Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[MKRHH90] Eric Mohr, David A. Kranz, and Jr. Robert H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 185–197, New York, NY, USA, 1990. ACM.

[ML94] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 5(4):379–400, 1994.

[Nar99] Girija J. Narlikar. Scheduling Threads for Low Space Requirement and Good Locality. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 83–95, New York, NY, USA, 1999. ACM.

[Org08] OpenMP Organization. OpenMP Application Program Interface, v. 3.0, May 2008.

[par] http://www.cepba.upc.edu/paraver.

[PDM$^+$98] M. Papermaster, R. Dinkjan, M. Mayfield, P. Lenk, B. Ciarfella, F. O'Connell, and R. DuPont. POWER3: Next Generation 64-bit PowerPC Processor Design. Technical report, IBM, October 1998.

[Pol93] C.D. Polychronopoulos. Nano-threads: Compiler Driven Multithreading. In *4th International Workshop on Compilers for Parallel Computing*, December 1993.

[RHJ07] Luiz De Rose, Bill Homer, and Dean Johnson. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *13th International Euro-Par Conference*, pages 150–159, 2007.

[RR99] Radu Rugina and Martin Rinard. Automatic Parallelization of Divide and Conquer Algorithms. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–83, New York, NY, USA, 1999. ACM.

[SE94] S. Subramaniam and D.L. Eager. Affinity Scheduling of Unbalanced Workloads. In *SuperComputer'94 Conference Proceedings*, 1994.

[SHF92] E. Schonberg S.F. Hummel and L.E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, 1992.

[SHPT99] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible Control Structures for Parallellism in OpenMP. In *1st European Workshop on OpenMP*, September 1999.

[SKK00] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning. In *EuroPar-2000*. Springer, 2000.

[Smi99] Lorna Smith. EPCC Development and Performance of a Hybrid OpenMP/MPI Quantum Monte Carlo Code. In *Proceedings of the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.

[Smi00] Lorna Smith. Mixed Mode MPI/OpenMP Programming. Technical report, UK High-End Computing, 2000.

[SSOB02] H. Shan, J.P. Singh, L. Oliker, and R. Biswas. A Comparison of Three Programming Models for Adaptive Applications on the Origin2000. *Journal of Parallel and Distributed Computing*, 62(2):241–266, 2002.

[SWC+07] Malgorzata Steinder, Ian Whalley, David Carrera, Ilona Gaweda, and David Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *10th IEEE/IFIP Symposium on Integrated Management (IM 2007)*, Munich, Germany, 2007.

[Taf] D.K. Tafti. Computational power balancing, Help for the overloaded processor.

[TMD+07] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for OpenMP Tasks in Nanos v4. In *CAS Conference 2007*, October 2007.

[TN93] T.H. Tzen and L.M. Ni. Trapezoid Self-scheduling Scheme for Parallel Computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[TTSY00] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In Sandhya Dwarkadas, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers, 5th International Workshop, LCR 2000*, volume 1915 of *Lecture Notes in Computer Science*. Springer, 2000.

[TTY99] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating Futures into Calling Standards. *SIGPLAN Not.*, 34(8):60–71, 1999.

[TTYF04] S. Tabirca, T. Tabirca, L.T. Yang, and L. Freeman. Evaluation of the Feedback Guided Dynamic Loop Scheduling (FGDLS) Algorithms. *IEICE TRANSACTIONS on Information and Systems*, 87(7):1829–1833, 2004.

[USR02] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[WLW$^+$07] XiaoYing Wang, DongJun Lan, Gang Wang, Xing Fang, Meng Ye, Ying Chen, and QingBo Wang. Appliance-Based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[YL94] Kelvin K. Yule and David J. Lilja. Categorizing Parallel Loops Based on Iteration Execution Time Variances. Technical Report HPPC-94-13, University of Minnesota, 1994.

[ZBC$^+$04] Y. Zhang, M. Burcea, V. Cheng, R. Ho, V. Cheng, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *Proc. of PDCS-2004: International Conference on Parallel and Distributed Computing Systems*, 2004.

[ZSA04] Guansong Zhang, Raúl Silvera, and Roch Archambault. Structure and Algorithm for Implementing OpenMP Workshares. In Barbara M. Chapman, editor, *WOMPAT*, volume 3349 of *Lecture Notes in Computer Science*, pages 110–120. Springer, 2004.

[ZUR04] Guansong Zhang, Priya Unnikrishnan, and James Ren. Experiments with Auto-Parallelizing SPEC2000FP Benchmarks. In *LCPC*, pages 348–362, 2004.

[ZV05] Yun Zhang and Michael Voss. Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CA, USA*. IEEE Computer Society, 2005.