NATIONAL TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

OpenMP extensions to support dependent work distributions

DIPLOMA THESIS

of

Ioanna N. Tsalouchidou

Supervisor: Nectarios Koziris
Associate Professor of N.T.U.A.

Athens,18 July 2011

National Technical University Of Athens
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

OpenMP extensions to support dependent work distributions

DIPLOMA THESIS

of

Ioanna N. Tsalouchidou

Supervisor: Nectarios Koziris
Associate Professor of N.T.U.A.

Approved by the committee on th of March 2011

....	....	....
Nectarios Koziris	Nikolaos Papaspyrou	Dimitrios Fotakis
Associate Professor NTUA	Assistant Professor NTUA	Lecturer NTUA

Athens, 18 July 2011

....

Ioanna N. Tsalouchidou
Electrical and Computer Engineer N.T.U.A.

Abstract

OpenMP is an Application Program Interface (API), which was first introduced in 1996 and since then it was adopted as the most popular standard for shared-memory parallel programing.It provides a portable and scalable model for the developers of multi-threaded applications.

OpenMP is still in the process of being extended, which means that in order to support each new feature changes should be done both in the front and back ends. These changes have to do with the new syntax that appears as well as the code generation that targets the runtime system. For this reason and for the needs of this thesis we use the Mercurium compiler and the Nanos runtime system which are designed to accommodate new concepts and extensions that we need to add in OpenMP.

In this thesis we propose extensions to allow the runtime detection of dependencies between for loops. With these extensions we are able to reduce the delays that are potentially caused from unbalanced workloads in the for loops and improve performance.

Keywords

OpenMP, OmpSs, Mercurium, Nanos, dependencies.

Contents

List of Figures

Acknowledgements

This work was developed in Universitat Politecnica de Catalunya (UPC) and Barcelona Supercomputing Center (BSC). The cooperation with the Computing Systems Laboratory of the National Technical University of Athens, Greece (NTUA) took place within the Erasmus/Socrates program.

First of all, I would like to thank my tutor in Barcelona, Professor Eduard Ayguade, for giving me the opportunity to work with the BSC group, for being extremely helpful from the first time of our contact and for trusting me by giving me this project. I would also like to thank Alejandro Duran for helping me with this project all this period of time and for being the person answering all my questions. Moreover, I would like to thank Roger Ferrer for being always there to help me and spend time for this project.

Also, I would like to thank my advisor in Athens, Professor Nektarios Koziris for supporting me to carry out my thesis abroad and for suggesting me Barcelona Supercomputing Center and for giving me the most valuable research orientation. I would also like to thank lecturer Georgios Goumas for introducing me to parallel programming, for helping me all this period of time, for believing in my work and for giving me the best advice in order to to complete this project successfully.

I would also like to thank my family and my friends for their tolerance and support all these years. But above all, I would like to thank my close friend and colleague, Stratis for his constant help and companionship. I want to thank him for being always close to me during the difficult times and because without his help this effort would be even harder for me to accomplish.

Chapter 1

Introduction

1.1 Multi-core Architectures

Despite the improvements and the steps that have been taken in order to reduce the size of single gates, problems such as the heat dissipation prevent us to raze even more the single-core clock frequency. This problem accompanied by the great need of improving the performance of some applications, led the researchers to design parallel architectures.

Single-core computational systems used to depend all their potential to improve performance at the parallelization on the level of instruction. The only effort towards utilizing parallelization was the concurrent use of the accelerator and the input/output devices. Nowadays, since the instruction-level parallelism cannot be exploited any further and the power consumption is of great importance, the parallelization is directed to the use of multiple accelerators and the effective use of them.

According to Moore's law [9], transistors available in accelerators are doubling each 18 to 24 months. This means that what is now important is to find effective ways in order to synchronize the accelerating units and to make them communicate rapidly and consistently. Furthermore, we should have a good control of the simultaneous references to the main memory and the input/output devices.

For the parallelization of the computations researchers have designed many different architectures and programming models that can in the first place increase the performance and then be eligible to expand with the addition of new accelerating units. Two models dominate the parallel architectures: shared memory and distributed memory which are further discussed in the forthcoming paragraphs.

1.1.1 Shared memory architectures

In the model of shared memory, processors have common access to global memory. The most important feature is the use of a single address space, which means that all processors have the same view of the memory. Each processor has his own level of cache memory and all processors are connected in the same network, typically a shared bus as it is shown in the figure 1.1.

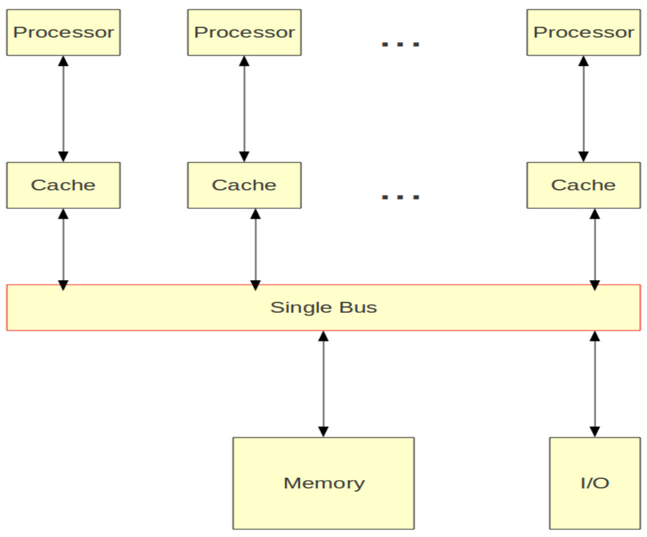

Figure 1.1: *Shared Memory Model.*

The communication between processors is done through reads and writes to the common data with the simultaneous update of each processor's individual cache. This means that the programming is being held easier and the execution can be done clearly, separated to the different units. On the other hand, with this type of architecture we face a major problem when a number of processors are trying to write in the global memory data, at the same time.

1.1.2 Distributed memory architectures

In the architecture of distributed memory there is no common memory space where all the processors have access. Each processor has its own local memory space and is the only one to have access on it as it is depicted in the figure 1.2. The exchange of information is done by sending and receiving explicit messages to and from the private memories of the processors. The sender sends its messages to the other processors, when this is needed, through the communication network.

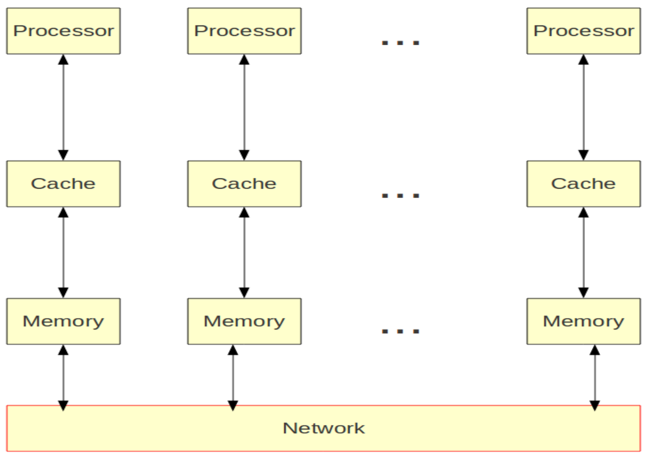

Figure 1.2: *Distributed Memory Model.*

Consequently, this model is not as user friendly as the previous one, because all the information that has to be exchanged should be exchanged manually. On the other hand, its main advantage is the fact that there is no need of a big common memory space that needs to be consistent and updated all the time as in the case of the shared memory model.

1.2 Parallel Programming and its difficulties

Now that multi-core architectures are widely used and parallel programming starts to replace the contemporary sequential programming, an other property starts concerning programmers. Concurrency, which refers to the simultaneous execution of instructions and the potential interaction of these computations with each other, raise new issues that do not exist in the sequential programming.

The concurrent use of shared resources can lead the outcome to be indeterminate. The first case of indeterminate situation is when two or more competing actions are each waiting the other to finish. This makes all of them wait and no one of them to continue. This case is named after *deadlock* [1] which is a common problem in multiprocessing of shared memory model, since there is widely used the idea of mutual exclusion, in order to avoid the simultaneous use of the same resource.

An other common problem of concurrent programming is the problem of *starvation*, which is quite similar to the *deadlock*. The problem of starvation rises when a process needs a resource in order to continue its execution, but is persistently denied to get the resource because of this resource being occupied by an other program.

Moreover, concurrency introduces new classes of potential problems that have not arose in conventional sequential programing. Some of the most common problems are the software bugs and especially the *race conditions*.

Apart from the problems that arise when we try to create a correct parallel code, there is always the problem of efficiency and performance which are the main reasons to use parallel programming. The problem of communication and synchronization of the processes among each other are some of the greatest obstacles in getting good parallel program performance. Moreover, synchronization and scheduling overhead may be some other reasons of getting low performance.

An other problem that is critical to performance is to balance the load of an applications' workload among the threads [3]. The solution to this problem is to minimize idle time of threads, which means idle cores and eventually waste of resource and extension of execution time . If we achieve to share the workload to all threads trying to avoid the work sharing overheads, we can have the minimum waste in computation time and finally achieve improvements in performance [6].

In fact the program performance follows Amdahl's law [8] which is the following:

$$\frac{1}{(1-P)+\frac{P}{N}}$$

where P is the proportion of the program that can be parallel and as a result *(1-P)* is the proportion of the program that remains sequential. The above formula estimates the maximum speedup that can be achieved when we use N number of processors.

The above law shows that the maximum speedup that can be achieved is not linear to the number of processors, which consists an other proof of the unsurpassed problems of parallel programming.

1.3 Motivation

Now that we know which problems cost as far as performance is concerned, we can manage to eliminate them. The reason of this project was first of all to introduce a way to reduce the idle time of threads between the tasks that are executed in parallel using OpenMP API.

Being too rigid, OpenMP does not allow the programmer to avoid load imbalance and express all the parallelization available, especially in the cases that we need to scale to systems that have large number of processors.

This project is an attempt to introduce a way of exploiting parallelization in cases that OpenMP, as it is currently implemented, cannot. It pursues to provide an easy way to balance the workload of threads in cases that there are dependent work distributions and finally to reduce the execution time.

1.4 Organization

The rest of the thesis is organized in six more chapters. Chapter 2, includes a detail description of OpenMP and some basic ideas and common structures. In chapter 3 we give an outline of OmpSs environment, which is the one that we use and extend for the needs of this thesis. In the next chapter, chapter 4, we give more details about the extensions that we have implemented and we analyze theoretically some examples. In chapter 5 we give a more in-depth description about the implementation of the extensions and we explain some of the basic structures that are implemented. In chapter 6 we describe the algorithm sparseLU that we use to evaluate the extensions and then we present the results of the evaluation of this algorithm and the results of some more artificial algorithms. Finally, in chapter 7 we conclude the thesis and we state some useful directions for futures work.

Chapter 2

The OpenMP environment

OpenMP[10], [11] is an Application Program Interface (API), that is widely used for writing multi-threaded applications for shared memory architectures. It combines compiler directives, library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. It provides a portable and scalable model but also a user-friendly and efficient approach to shared-memory parallel programming.

The main idea of OpenMP is that the already existing thread creates a certain number of slave-treads. After that, the task that needs to be executed is divided to a number of parts equal to the number of the threads that exist in total. As a result, each part of the initial task can be executed by one thread and all of them to be executed concurrently, with the runtime environment allocating threads to different processors.

The user of OpenMP explicitly specifies which should be the actions to be taken in order for the compiler and the runtime to run the program in parallel. Moreover, the user is responsible to produce an accurate program free of data-dependencies, data conflicts and deadlocks. This means that OpenMP implementations are not required to check for the restrictions that will cause a program to be classified as non-conforming. Finally, OpenMP is not expected to include any kind of compiler-generated automatic parallelization.

2.1 The OpenMP execution model

The basic execution entity of OpenMP is the thread. Each thread has its own stack and associated static memory and is managed from the OpenMP runtime system. In fact, each OpenMP program begins as a single thread of execution. This first thread, called the *initial thread* executes the program sequentially and persists throughout the whole execution.

When the initial thread encounters a `parallel construct` it creates (*forks*) a number of threads that *team of threads*. This team of threads, includes the initial thread, that now is named *the master thread* of the new team, and 0 or more other threads. The actual number of the threads that will be created is defined by the user when the num_threads clause is used.

Each one of the created threads has the obligation to execute one of the implicit tasks that are created. The code of these implicit tasks is defined by the code inside the parallel construct. Each task of this parallel construct is assigned to a different thread in the team of threads and it becomes tied. This means that this task will be always executed by the thread that was initially assigned to.

At the end of the parallel construct, there is an implicit barrier. Beyond this barrier only the master thread is allowed to continue the execution.The threads that were created by the initial thread synchronize and join to this single thread. Furthermore, there is no restriction in the number of the parallel constructs that can be specified in a single program.

During the execution inside the parallel construct the threads may encounter *worksharing constructs* or/and *task constructs*. Further on there will be a detailed description of the execution model and the syntax of both of the above cases as well as the parallel construct itself. All the above are depicted in figure 2.1.

Inside the parallel construct the team can encounter a `worksharing construct`. At that point, the work inside the construct is divided among the members of the team and executed cooperatively instead of being executed by every thread. The way that the worksharing construct is divided

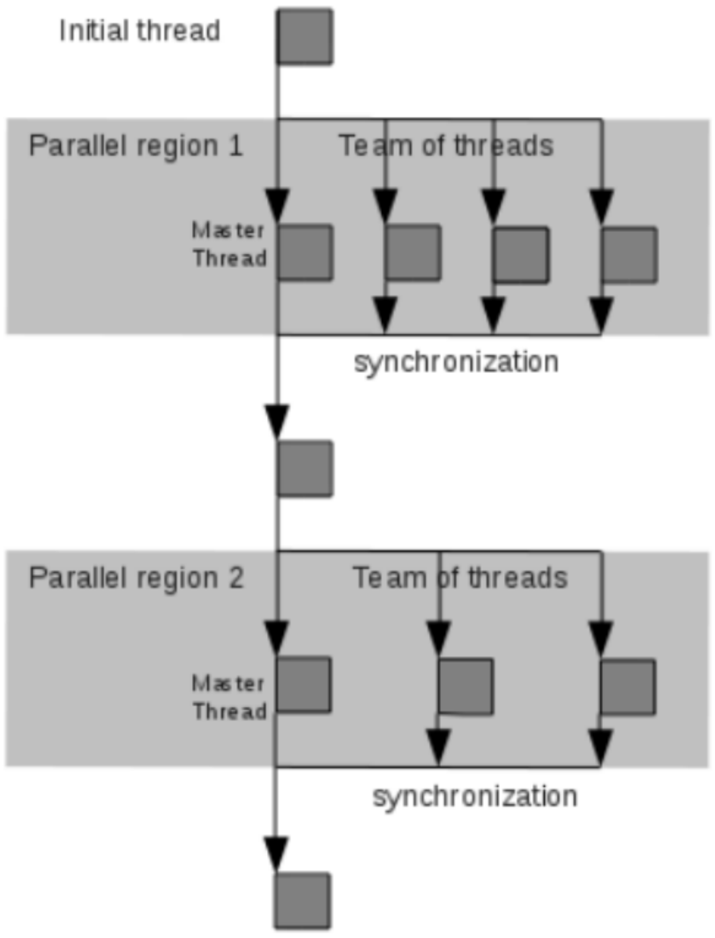

Figure 2.1: *Execution model*

in order to be shared to the threads, has to do with some instructions that the user states and differentiate the behavior of the threads.

When a thread encounters a `task construct`, a new task is generated. This new task may be executed by the encountering thread immediately or be delayed. If the execution of the task is suspended, the encountering thread can resume it if this thread is tied with this task or let an other thread to resume it in case of the thread being untied.

Moreover, there are available some synchronization constructs and some library routines in order to coordinate tasks and data access in parallel regions.

2.2 Description of the constructs

In this section we discuss some of the most important OpenMP directives relevant to this thesis.

First of all the OpenMP directives are specified by using the `#pragma` keyword and has the following syntax:

`#pragma omp` *directive-name [clause[[,]clause]...] new-line*

2.2.1 The parallel construct

The syntax of the parallel construct is the following :

`#pragma omp parallel` *[clause[[,]clause]...] new-line*

where the clause can be one of the following :

```
if
num_threads
default
private
firstprivate
shared
copyin
reduction
```

As we have already mentioned, when the initial thread encounters the parallel construct it creates a team of threads and becomes the master thread of this team. The region is executed by all the threads of the team, including the master thread. Furthermore the number of threads of the team remains stable during the execution of this specific parallel region.

Inside the parallel region each thread has its own identification number. Beginning from the number zero, which identifies in all cases the master thread, all threads get an incremental number. By using the library routine `omp_get_thread_num()` we can get the number of a specific thread.

The encountering thread creates also a number of tasks equal to the number of threads, inside the parallel region. The code of these tasks is determined by the parallel construct and each one of them gets tied with one thread. This and only this thread (which is tied with the specific task) is allowed to execute it.

At the end of the parallel region there is an implicit barrier, beyond which only the master thread can resume execution of the enclosing task region. In the case of nested parallel regions, the thread that encounters the second parallel region acts in the same way as we have already described and becomes the master of the new team of threads.

We also have to mention that a program that branches into or out of a parallel region is not supposed to be conforming. Finally, we should be careful not to depend the programs on the ordering of the evaluation of the clauses, or on any side effect of the evaluation clauses.

2.2.2 The worksharing construct

With the worksharing constructs the execution of a region is destributed among the members of the team that encounter this construct. The worksharing construct does not have an explicit barrier. However, a barrier is implied at the end of the region. Exception is when the `nowait` clause is specified. The nowait clause removes the implied barrier and now the treads can proceed to the instructions following the worksharing region without waiting for the other members of the team to finish with their own part of the region

In OpenMP we define the following four worksharing constructs.

loop construct

`sections` construct

`single` construct

`workshare` construct

2.2.2.1 Loop construct

The loop construct denotes that the iterations of the loops will be distributed to the threads of the team (that already exist and execute the parallel region) and they will be executed in parallel. The syntax of the loop construct is the following :

`#pragma omp for` *[clause[[,]clause]...] new-line*

Where the *clause* can be one of the following :

```
private
firstprivate
lastprivate
reduction
schedule
collapse
ordered
nowait
```

We have to note that a loop region always binds with the innermost parallel region and the threads that execute the iterations of the loop are only those that execute the parallel region. At the end of the for loop, there is an implicit barrier. In case of a *nowait* clause this implicit barrier does not exist anymore.

A for loop has a number of iterations, lets hypothesize that this number is N and begin from 0 to N-1. When the loop is not executed in parallel, the logical numbering denotes the sequence in which the iterations will be executed. In the case of the parallel execution the way that the iterations will be distributed to the threads is defined by the `schedule` clause. Each thread binds with and executes a *chunk* which is a contiguous non-empty subset of the loop and that has to be an invariant integer expression with always a positive value.

There are several types of schedule:

- static
- dynamic
- guided
- automatic
- runtime

The first one the static schedule is declared as: `schedule (static ,` *chunk_size* `)`. This means that the iterations are separated into *chunk_size* number of chunks and assigned to the threads of the team in the roundrobin way. When a *chunk_size* is not specified the iterations are divided in the way that is more approximately equal in size.

The dynamic schedule, which is declared as: `schedule(dynamic,` *chunk_size* `)`, denotes that the chunks that are created based to the *chunk_size* are distributed and binded to the threads of the team, in the way that the threads request. Each thread request a chunk, and as soon as it finishes executing it, it can request an other one until no chunks are available for execution. Each chunk of the loop contains *chunk_size* iterations, except for the last one that it can have less than *chunk_size*. In case that the *chunk_size* is not specified the default number is 1.

As far as the guided schedule is concerned, the threads act in the same way as the dynamic schedule when the `schedule (dynamic,` *chunk_size* `)` is specified. The difference is that the for a *chunk_size* of 1 the size fo each chunk is proportional to the number of the unassigned iterations if we divide them by the number of threads in the team decreasing to 1. If the *chunk_size* is l (where l is greater than 1), the size of each chunk is determined in the same way with the obligation that the chunks should not contain fewer than l iterations (except for the last chunk to be assigned, which may have fewer than l iterations). When the *chunk_size* is not specified it gets 1 by default.

The auto schedule , specified by the `schedule (auto)` let the compiler and/or the runtime to decide how to distribute the iterations to the threads. A *chunk_size* must not be specified.

Finally the runtime schedule, specified by `schedule (runtime)`, let the decision to be taken when the run time the schedule and the chunk size are taken from the run-sched-var Internal Control Variables (ICV). A *chunk_size* must not be specified as well.

2.2.3 The task construct

With the `task` construct the user can define an explicit task. This `task` region binds with the innermost parallel region. The syntax of the task construct is the following:

`#pragma omp task` *[clause[[,]clause]...] new-line*

structured block

where a clause can be :

- if
- untied
- default (shared/none)
- private (*list*)
- firstprivate (*list*)
- shared (*list*)

As we have already described, when a thread encounters a `task` construct, a task is created according to the code of the structured block. The thread that first encounters the task construct can either execute it immediately of delay the execution for later or even let an other thread to assign with the task and execute it later.

When a thread starts executing a `task` region it can temporarily suspend the execution and resume it later. I the task is not declared as untied the thread that started the execution should finish it, otherwise an other thread of the team can finish executing this task.

When a program branches into or out of a `task` region then it is not conforming. Also the program should not depend to any side effects or to the ordering of the evaluation of the clauses.

2.3 Task-For construct comparison

Programmers that are using OpenMP, have to decide which parallel construct to use in order to parallelize a region. This is not always a trivial decision as there are many differences that are not always obvious. For example, if we want to parallelize a for-loop we should examine whether is better to use the for construct or the parallel construct. What we should have in mind is the way that the tasks are created and how they are distributed on the threads.

Now we will describe in detail three different occasions of the for-loop and the task construct.

- The `task construct` has a for loop in the *structured block*

  ```
  #pragma omp task
    for(i=0;i<N;i++)
      f();
  ```

  Lets hypothesize that we have two threads. One of them is the master thread, which creates a task that includes the whole for loop. After the task is created both threads are eligible to execute the task depending on their availability.

  The first figure (Figure 2.2) shows that the master thread creates the task that it is subject to the execution. In the second figure (Figure 2.3) the second thread, the one with *thread_number* = 1, continues the execution since the master thread is not available to execute the task that had created. In the second instance of the figure we can see also that during the execution of the task there is some time that is lost between before the execution starts and after the execution finishes. This time is depicted with the more gray color before and after the actual execution of the for loop.

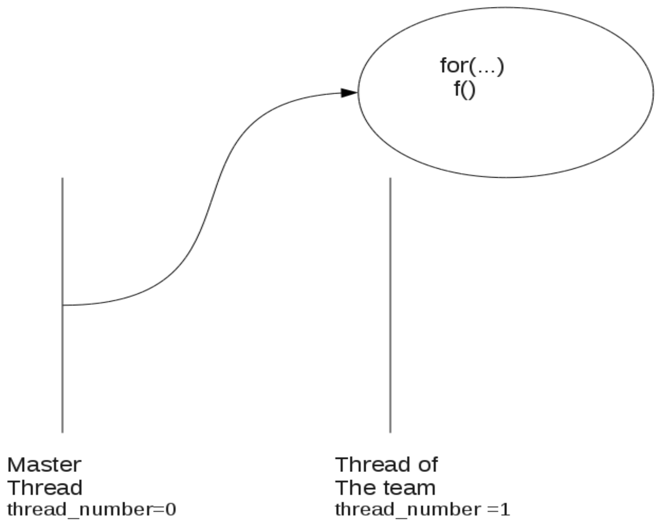

Figure 2.2: *Task construct with for loop as structure block: creation of the tasks.*

- The `task construct` inside a for loop.

```
for(i=0;i<N;i++)
  #pragma omp task
      f();
```

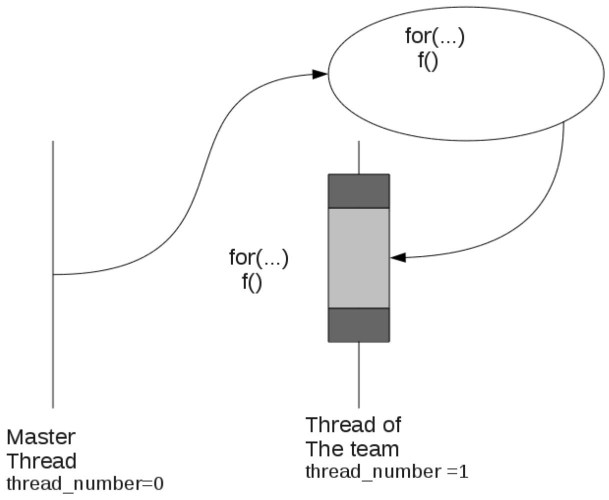

Figure 2.3: *Task construct with for loop as structure block: execution of the tasks.*

Lets hypothesize that we have again two threads. The first one that encounters first the for loop, starts executing it. Inside the for loop there is a `task` construct. As a result the thread encounters N times the `task` construct and creates N tasks. When the N number of tasks are created then they are distributed in the two threads and start being executed.

Figure 2.4 and figure 2.5 are two instances which depict the two hypothetical threads and how they create and execute the tasks. In the first instance, figure 2.4, the encountering thread creates N number of

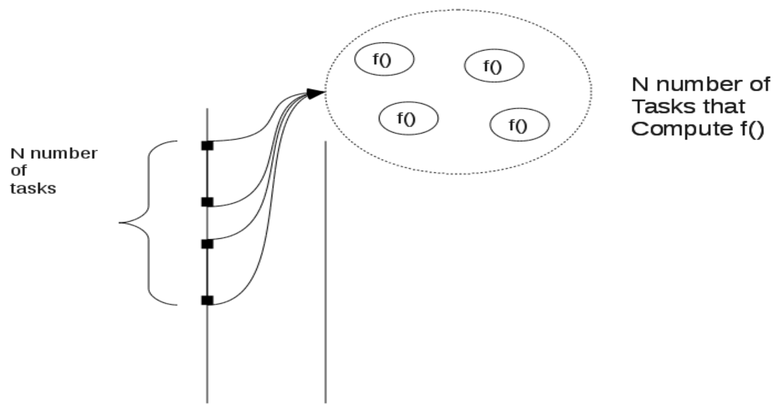

Figure 2.4: *Task construct inside a for loop: creation of the tasks.*

tasks each one of them will have the code of the `task` construct, which is the call of the function f.

After the creation, these tasks will have to be distributed to the threads depending their availability. In the second figure 2.5 it is also depicted the execution intervals that happen because of the movements of the tasks to the threads.

- A for loop inside a for loop construct.

```
#pragma omp for
  for(i=0;i<N;i++)
      f();
```

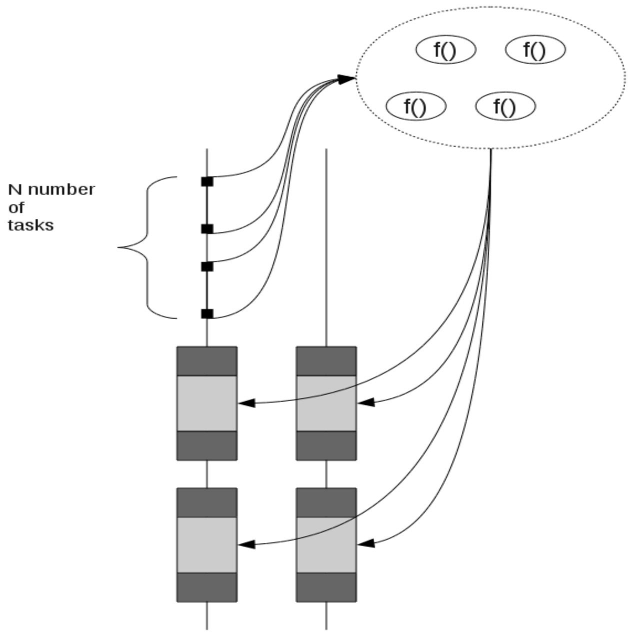

Figure 2.5: *Task construct inside a for loop: execution of the tasks.*

Lets consider again the same hypothetical case of the two threads. When one of these two threads encounters a for construct, a task is created. This task is being separated in two tasks, as it is shown in figure 2.6, each one of which has the half number of iterations of the initial task, which in this case is N/2.

After that separation the tasks are distributed to the existing threads as it is shown in figure 2.7.

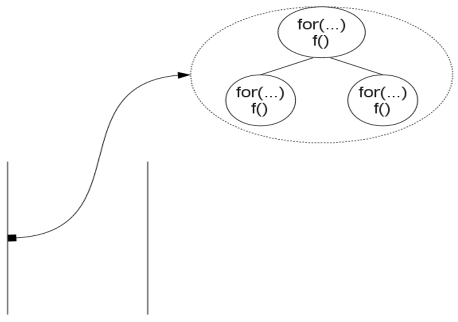

Figure 2.6: *For loop inside a for construct: creation of the tasks.*

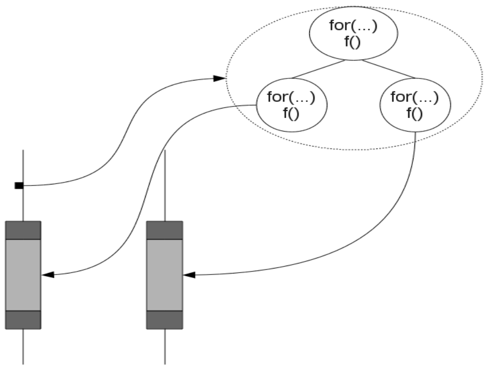

Figure 2.7: *For loop inside a for construct: execution of the tasks.*

Chapter 3

The OmpSs environment

The OmpSs [5] programming model is an attempt to extend the current OpenMP standard including some of its features and progressing in many more. The objective of OmpSs is to include features that will make OpenMP simpler and more portable and introduce a way to migrate applications to homogeneous and heterogeneous architectures.

The already existing model of OpenCL tries to unify the programming models for architectures based on hardware accelerators. It is an attempt to ensure portability, low-level access to the hardware and performance portability. On the other hand it provides a great amount of low level details which makes it difficult to use for those that are not familiar with these type of programming models.

In contrast, the user of OmpSs needs only to decompose his application into tasks and outline the usage of the variables in each task that has created. For these variables the user has to describe whether they are inputs, outputs, privates etc. After this declaration the user does not have any other responsibilities and from that point on, the runtime system is responsible to organize the way that tasks will be executed and to detect the possible dependencies. Furthermore, the runtime is responsible to organize how the work will be distributed at the heterogeneous set of processors and manage the memory hierarchy in a transparent and portable way.

From the programming model point of view, in order to express more successfully the dependencies between the tasks and to implement locality-aware dataflow scheduling policies, we need to find a stable way to express the dependencies between the tasks at runtime. As far as the execution model is concerned, the fork/join model of OpenMP as well as the parallel construct are rejected. Thus asynchronous dataflow execution engines are implemented that hide the complexities and dynamic heterogeneities in the core and memory architectures as well as adapt to dynamic changes in resource availability and workloads.

3.1 The execution model

In order to describe a little further the OmpSs execution model we should point out the difference in how the threads are created in OpenMP and OmpSs. As we have already described, OpenMP follows the fork/join model. On the other hand OmpSs follows the thread-pool model which means that all the treads exist since the beginning of the execution. When the execution starts only one of these threads, the master thread executes the code. The rest of the threads remain stand-by to execute the work when it is available. In OmpSs the term pool still refers to a team of threads as in the case of OpenMP. The only difference between the two notions is that in OmpSs the team is divided in two kind of threads: the master thread and the worker threads.

Using the usual OpenMP worksharing constructs or task constructs, the master thread in this case can generate work for the other threads. Moreover, having the possibility of nesting the worksharing constructs, which is an other difference between openMP and OmpSs, all threads can eventually become work generators.

From all the above, we can understand that as we already have all the threads from the beginning of the execution, there is no need for the parallel construct. Furthermore, being possible to have all the constructs nested, there is no actual need tho implement the nested teams of threads. For all the above, the parallel construct is rejected.

3.2 The dataflow extensions-The task construct

In this section we describe the task construct, an extension of OmpSs that is already implemented and integrated to the programming model. The syntax of the construct is as follows [7]:

```
#pragma omp task [omp_clauses] [ss_clauses]
task_block
```

The `ss_clauses` can be one of the following: `input, output, inout and inout-set`. These four clauses are accompanied by an expression that leads to evaluate a set of *lvalues*. Later they are used by the runtime system in order to build the dependency graph for the tasks.

- input clause: The task that has an input clause which evaluates to an lvalue cannot be executed since a task that has been already created has an output clause that evaluates to the same lvalue and has not finished its execution.
- output clause: The task that has an output clause which evaluates to an lvalue, cannot be executed since a task that has been already created has an input or an output clause that evaluates to the same lvalue and has not finished its execution.
- inout clause: In this case we consider the inout clause that evaluates to an lvalue as if it was input clause and output clause applying to the same lvalue.
- inout-set clause: In this case we consider that the inout-set that evaluates to a lvalue is equivalent to an inout that evaluates to the same rvalue with the exeption that it will not create dependencies with other tasks with an inout-set clause which evaluates to the same lvalue.

3.3 The implementation

OmpSs is composed of the Mercurium compiler, which is a source-to-source compiler, and the Nanos++ runtime library. In this paragraph we give some information about these two components as well as the scheduler component and we provide a good insight on the role of each one of them.

3.3.1 Mercurium

The first basic component of the OmpSs, the compiler, is restricted to recognize the constructs [5]. After recognizing them the rest of its role is to transform them into calls in order to be used by the runtime library.

One type of transformations that the compiler makes is to change the dataflow clauses, that we have already describe, to expressions that will be evaluated during the execution of the program. The idea is that these expressions will be passed to the runtime library as addresses of memory that will be used afterwords to construct the dependency graph.

Furthermore, the Mercurium compiler creates code to enable the runtime to execute the applications to a range of target devices. The target devices can be, exept for the SMP, also cuda and cell. When a task is associated with a target or a target directive is liked to a task which is different from the SMP the representation of this task is passed to a handler which is responsible for distributing the work to the devices.

3.3.2 Nanos++

The second component of the OmpSs programming model, the Nanos++ is a runtime library for task-based programming models [5]. The unit that the Nanos runtime library is based on is the WorkDescriptor which is a unit of work that creates instances of a subclass that describes the work to be done.

WorkDescriptors are created by the compiler both for tasks constructs and for worksharings and contains data for the environment of their execution but also for the dependencies related to them. Each one of the work descriptors is related to a list which contains information about the devices that a task can run to. More specifically each element of the list refers to a device and contains information, about this specific device, which is needed to run the code related to the work descriptor.

The runtime system also maintains a graph where the tasks are linked together according to the dependencies they have. Each time the compiler creates a work descriptor it is submitted to this graph in order to have its dependencies checked. For simplicity, efficiency reasons and in order to avoid deadlocks the graph that is created is the one of each task and contains information about the dependencies of its children. Finally, in order to detect the dependencies the runtime keeps the addresses of all the arguments when dependency clauses exist.

Chapter 4

Design of loop dependences for OpenMP

In this chapter we describe in detail the extensions of the OmpSs which were implemented for the needs of this thesis. The extensions are related to the *for loops* and the way to express and implement the dependencies among them. The proposal of this thesis is to extend the for-loop construct that already exists in OpenMP with some additional clauses that are used to retrieve the dependencies among the for loops and also the dependencies among the for-loops and the tasks.

More specifically, we can say that when a for-loop needs a variable that is previously computed by an other for-loop then we say that it has an *input* dependence and we express it with the *input* clause. Accordingly, when a for-loop is computing a variable that is planned to be used later from a task of an other for-loop we can say that it has an *output* dependence and we express it with an *output* construct. In the last case when a for-loop needs a variable that is computed in a previous task block or a for-loop block and also generates a variable that is needed later on for other blocks then we say that it has an input-output dependence. If this input-output dependence applies to the same variable, then we express the input-output dependence with an *inout* clause.

4.1 The syntax of the construct

The syntax of the construct is equivalent to the task constructs which we described in the previous chapter. More specifically the construct is :

```
#pragma omp for  [omp_clauses] [ss_clauses]
  for-block
```

Where the ss_clauses can be one of the following:

1. **input** : The for-loop that has an *input* clause which evaluates to an *lvalue* cannot be executed since an other for-loop that has been already created has an output clause that evaluates to the same *lvalue* and has not finished its execution.

2. **output** : The for-loop that has an *output* clause which evaluates to an *lvalue*, cannot be executed since an other for-loop that has been already created has an input or an output clause that evaluates to the same *lvalue* and has not finished its execution.

3. **inout** : In this case we consider the *inout* clause that evaluates to an *lvalue* as if it was input clause and output clause applying to the same *lvalue*.

The syntax of the above clauses is shown below:

- input (data-reference-list)
- output (data-reference-list)
- inout (data-reference-list)

As described above, the dependencies of the for loops are described as data reference lists, which is more generic than an actual variable list. With the data reference list we mean the list of variables and references to sub-objects, that can also include array element references.

For a better understanding of how the *ss_clauses* are used we can see the following example (Listing 4.1) and the graph 4.1.

Listing 4.1: Outline of dependencies

```
int main ()
{
...
#pragma omp for output (var1,var3)
        for (...)
        {...}
#pragma omp for output (var2) inout(var3)
        for (...)
        {...}
#pragma omp for input (var2, var3)
        for (...)
        {...}
#pragma omp for input (var1)
        for (...)
        {...}
...
}
```

4.2 The types of dependencies

We can define two general categories to use these clauses. The first one is to apply them to the whole loop and not care about the induction variable and the second one is to explicitly define that the dependence of the loop is related to the induction variable. The first case describes **coarse grain dependencies** whereas the second case **fine grain dependencies**.

4.2.1 Coarse grain dependencies

In the first case, we imply that the whole loop is dependent on an other loop, if the clause is input or inout, or that the loop makes a dependence to an other loop if the clause is the ouput. In this case we consider the loop as if it was a task where we apply the dependencies we described in the previous chapter. As a result, a for-loop cannot be executed if it has an input dependence and the variables that this reference is applied to are not previously computed.

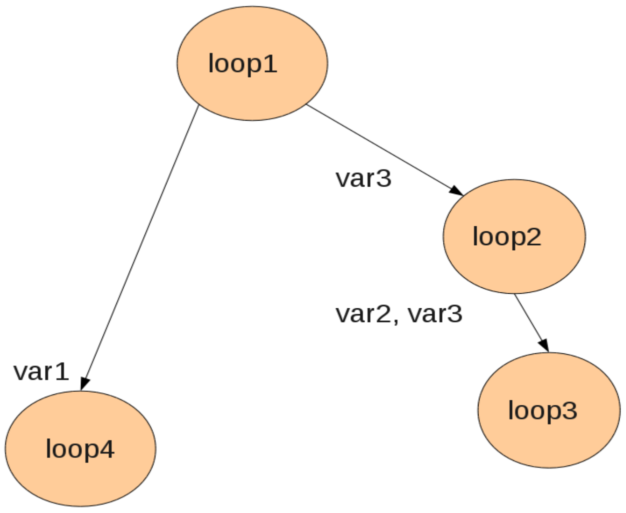

Figure 4.1: *Dependence graph.*

For example in the case of the Listing 4.2, we imply that the whole third loop is dependent of the first and the second loop, because the b matrix is dependent of the results of the first loop and the c matrix is dependent of the results of the second loop.

and to express the dependencies graphically we can see figure 4.2:

As a result we expect the third loop not to be executed before the b and the c matrices are computed. However, the first and the second loop are independent which means that the second loop can be executed before the first one, provided that there are threads that are not in use by the first loop.

A better understanding of the actual use of the above idea can be the Listing 4.3.

In this case there is an unbalanced workload on the first loop that makes it

Listing 4.2: Coarse grain dependencies: example 1

```
void main(){
...
#pragma omp for output (b)
        for (i=0;i<10;i++){
                b[i]++;
        }

#pragma omp for output (c)
        for (i=0;i<10;i++){
                c[i]++;
        }

#pragma omp for input (b, c)
        for (i=0;i<10;i++){
                b[i]=...
                c[i]=...
        }
...
#pragma omp taskwait
}
```

Listing 4.3: Coarse grain dependencies: example 2

```
void main(){
...
#pragma omp for output (b)
        for (i=0;i<10;i++){
            for (j=0;j<100000;j++)
                b[i]++;
        }

#pragma omp for output (c)
        for (i=0;i<10;i++){
            for (j=0;j<3;j++)
                c[i]++;
        }

#pragma omp for input (b, c)
        for (i=0;i<10;i++){
                b[i]=...
                c[i]=...
        }
...
#pragma omp taskwait
}
```

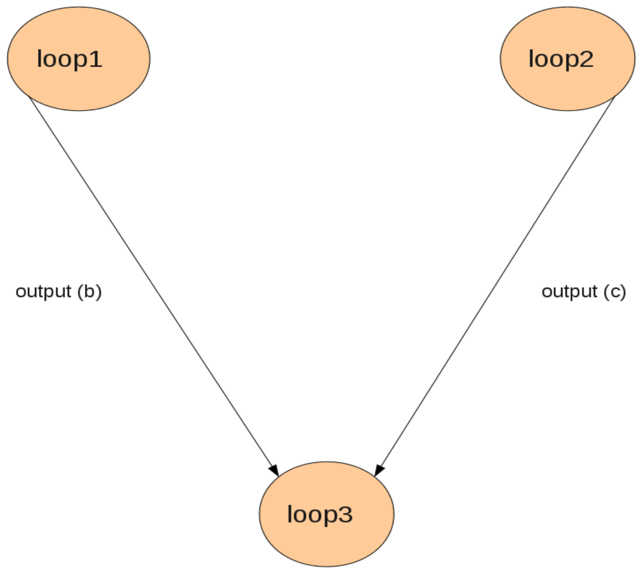

Figure 4.2: *Dependence graph for the coarse grain example .*

finish its execution really slow. In this case and as long as there are available threads there is no need to wait the first loop to finish execution. As a result the free threads continue with the execution of the second loop and thus we raise the level of parallelism.

4.2.2 Fine grain dependencies

The second category is to express the dependencies according to the induction variable of the matrices that possibly exist as dependent variables. In this case, the expression is applied to the loop for each iteration and it is evaluated as if each chunk of the loop was a task with the defined dependencies to refer to the specific chunk ranges. More specifically, if a loop has an input dependence from a previous one, but in this case only for a few iterations and the loop that has the related output dependence has already

Listing 4.4: Fine grain dependencies: example 1

```
void main(){
...

#pragma omp for output (b[0:4])
        for (i=0;i<5;i++){
                b[i]++;
        }

#pragma omp for output (c[0:4])
        for (i=0;i<5;i++){
                c[i]++;
        }

#pragma omp for input (b[0:4], c[0:4])
        for (i=0;i<10;i++){
                b[i]=...
                c[i]=...
        }
...
#pragma omp taskwait
}
```

computed these iterations, then the input-dependent loop can start being executed conditionally that there are available threads.

For example in the case of Listing 4.4 the third loop is again dependent of the first and the second loop, but this time with the difference that the b and the c matrices of the third loop is dependent only in the first five elements and not in the following.

If we want to show the dependence graph between the for-loops, figure 4.3 we have to describe the two phases: The phase one describes the dependencies between the loops for the first five iterations and the phase two shows that for the rest of the iterations there are no dependencies between the loops.

Considering now the following case, where the loops are not balanced, we can understand better the actual use of the Listing 4.5.

In this case the third loop has a dependence on the first five elements of

Listing 4.5: Fine grain dependencies: example 2

```
void main(){
...

#pragma omp for output (b[0:4])
        for (i=0;i<5;i++){
            for (j=0;i<10000;i++)
                b[i]++;
        }

#pragma omp for output (c[0:4])
        for (i=0;i<5;i++){
            for (i=0;i<10000;i++)
                c[i]++;
        }

#pragma omp for input (b[0:4], c[0:4])
        for (i=4;i<10;i++){
                b[i]=3*i
                c[i]=4*i
        }
...
#pragma omp taskwait
}
```

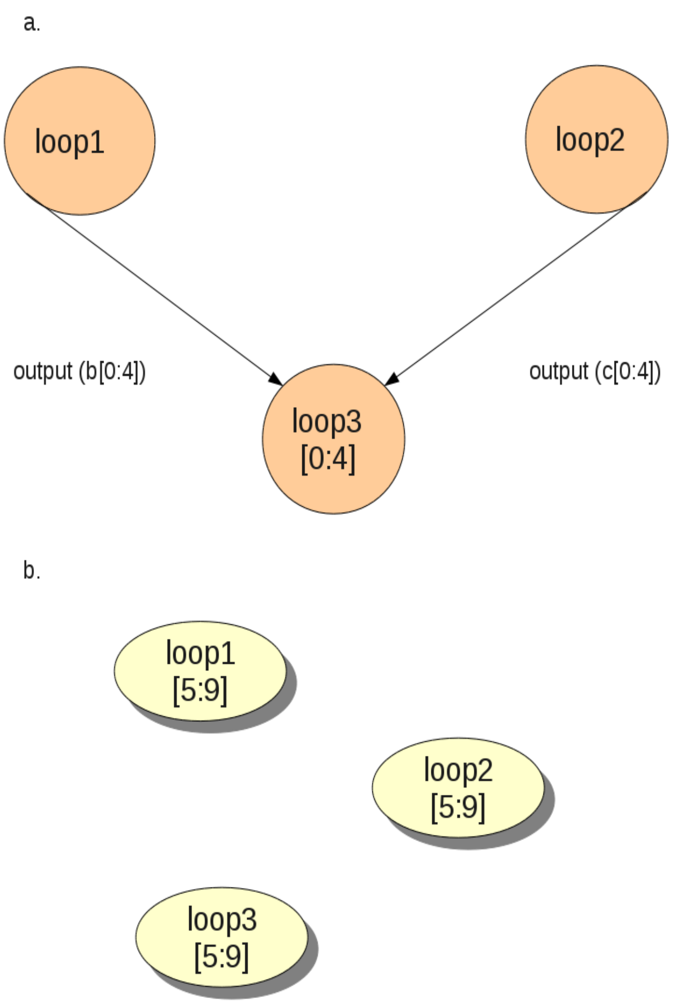

Figure 4.3: *Dependence graph for the fine grain example.*

b and c matrices. However, there is no need to wait for the calculation of these five elements since the induction variable of the third for loop begins to increment from four which means that only one of the elements of the matrices are dependent to the previous loops. So there is no actual dependence in the elements that the induction variable is greater than four and as a result these iterations can start being executed before the end of the execution of the firs two loops.

In this case, by expressing the dependencies relative to the induction variable the level of the parallelism is again raised. If this was not the case, and we expressed it only with the coarse grain construct then the third loop should be stalled and executed after the first two loops have finished their execution.

4.3 SparseLU example

In order to suport the above with a less artificial example we use the SparseLU algorithm [5], which computes an LU matrix factorization.

4.4 Description of the algorithm SparseLU

As we have already mention, SparseLU computes the factorization of an LU blocked matrix. The matrix is organized as a hypermatrix with pointers to the actual blocks of data. The implementation includes the use of four basic functions. The main functions are the *lu0, fwd, bdiv* and *bmod*[7].

In Listing 4.6 is the main code of the sequential SparseLU kernel.

In the above algorithm, when the *lu0* is computed the rest of the instances of *fwd* and *bdiv* can be executed in parallel. Moreover, when a pair of instances of *fwd* and *bdiv* then one instance of *bmod* is allowed to be executed. On the other hand there are dependencies between each instance of bmod of one iteration of kk loop and the instances of lu0, fwd, bdiv and bmod in the next iteration of the same loop.

With the above analysis of the dependencies of the algorithm, we can use the extensions of OmpMP in order to exploit the available parallelism. For example we can use the for construct in order to distribute the work in the loops on the lines 11 and 16. We should also apply loop distribution to isolate the loop that executes the multiple instances of function *bdiv* and exploit the parallelism that exists between the instances of functions *bdiv* and *fwd*.

The main part of the code using only the extension of OpenMP would look like the Listing 4.7.

Listing 4.6: SparseLU

```
int sparseLU(){
  int ii, jj, kk;
  /*Allocation of the blocks A[ii][jj]*/s
  for (kk=0;kk<NB;kk++){

    lu0(A[kk][kk])

    for(jj=kk+1; jj<NB; jj++){
      if (A[kk][jj]!=NULL)
        fwd(A[kk][kk],A[kk][jj]);
    }
   for (ii=kk+1,ii<NB; ii++)
      if(A[ii][kk]!=NULL){
      bdiv(A[kk][kk], A[ii][kk]);
      for(jj=kk+1; jj<NB; jj++)
          if (A[kk][jj]!=NULL)
            {
              if (A[ii][jj]==NULL)
                A[ii][jj]=alocate_clean_block();
              bmod(A[ii][kk],A[kk][jj], A[ii][jj]);
            }
      }
   }
}
```

Listing 4.7: SparseLU without dependencies

```
int sparseLU(){

  int ii, jj, kk;

  /*Allocation of the blocks A[ii][jj]*/

  for (kk=0; kk<NB; kk++) {
      lu0(A[kk][kk]);

  #pragma omp for nowait
      for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
                fwd(A[kk][kk], A[kk][jj]);

  #pragma omp for
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL)
            bdiv (A[kk][kk], A[ii][kk]);

  #pragma omp for private(jj)
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL)
            for (jj=kk+1; jj<NB; jj++)
              if (A[kk][jj] != NULL) {
                  if (A[ii][jj]==NULL)
                    A[ii][jj]=allocate_clean_block();
                  bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
              }
  }

}
```

The graph that follows, 4.4, can give us a better insight of the dependencies between the tasks of the algorithm. It shows the dependencies between the functions lu0, fwd, bdiv and bmod in one iteration of kk loop.

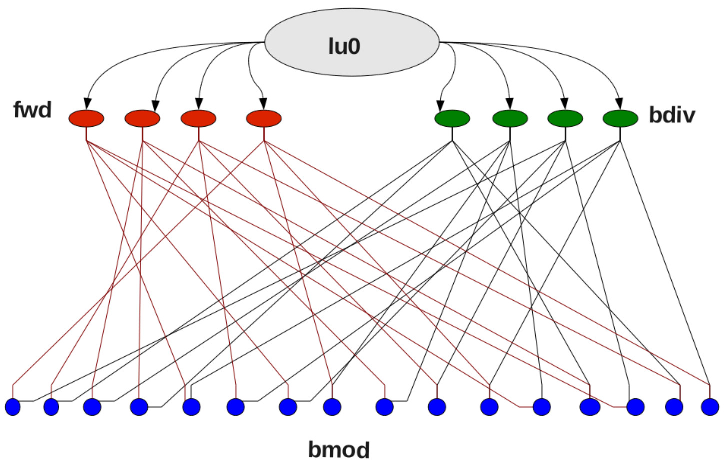

Figure 4.4: *Dependencies between SparseLU functions in one kk iteration.*

From this graph we can understand that there is no need to execute first the fwd function and then the bdiv function, since in one kk iteration we don't have any dependencies between them. In fact, the A[kk][kk] matrix is not a matrix to be computed inside the fwd or bdiv functions but only in lu0. Thus we can state that matrix A[kk][kk] can be only input(A[kk][kk]) and not output(A[kk][kk])

In the same way we can define whether the rest of the dependencies while computing matrix A are input, output or inout and then introduce the extensions proposed in this thesis. With the fine grain dependencies of the matrix A then we get the Listing 4.8

Listing 4.8: SparseLU with fine grain dependencies

```
int sparseLU(){

  int ii, jj, kk;

  /*Allocation of the blocks A[ii][jj]*/

    for (kk=0; kk<NB; kk++) {

#pragma omp task inout(A[kk][kk]) schedule(static)
      lu0(A[kk][kk]);

#pragma omp for input(A[kk][kk])
              inout(A[kk][jj]) schedule(static)
      for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
                fwd(A[kk][kk], A[kk][jj]);

#pragma omp for input(A[kk][kk])
              inout(A[ii][kk]) schedule(static)
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL)
            bdiv (A[kk][kk], A[ii][kk]);

      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL){
#pragma omp for input(A[kk][jj],A[ii][kk])
              inout(A[ii][jj]) schedule(static)
            for (jj=kk+1; jj<NB; jj++)
              if (A[kk][jj] != NULL) {
                  if (A[ii][jj]==NULL)
                    A[ii][jj]=allocate_clean_block();
                  bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
              }
                  }
  }

#pragma omp taskwait

}
```

4.5 Restrictions

In both coarse grain and fine grain dependencies we implemented the above only when the schedule is static. For the rest of the cases we were prevented by the structure and the nature of the current runtime system runtime. Furthermore, for the same reasons we were prevented to implement the above when the chunk is zero and not include the case when the chunk in non zero.

Finally, we have to mention that in case there is no need of using the ouput clause, then we consider it equal as if it had a OpenMP *nonwait* clause on it.

Chapter 5

Implementation

In this chapter we present the main implementation steps of the Nanos++ runtime system, with its classes and its methods as it is currently implemented (Figure5.1) [2] [4].

- Nanos++ runtime system as it is currently implemented, contains a **Dependency** class which encapsulates any necessary information in order to define dependencies. This class consists of the address field, which is a reference to the address that is used to identify a dependency. It also contains the input and output field which determine if the storage pointed by address is read/ written by the user of the dependency respectively.

- The runtime contains a a domain that is called **Dependencies Domain**. This represents a place that objects with dependencies (*DependableObjects*) can be submitted. Each one of the *WorkDescriptors* have associated a *DependenciesDomain* to which they can submit objects. They represent new tasks with dependencies or sets of dependencies that should be synchronized in order to have their dependencies satisfied.

- The dependency graph is formed by **Dependable Objects** which represent the entities that have dependencies. When a Dependable Object

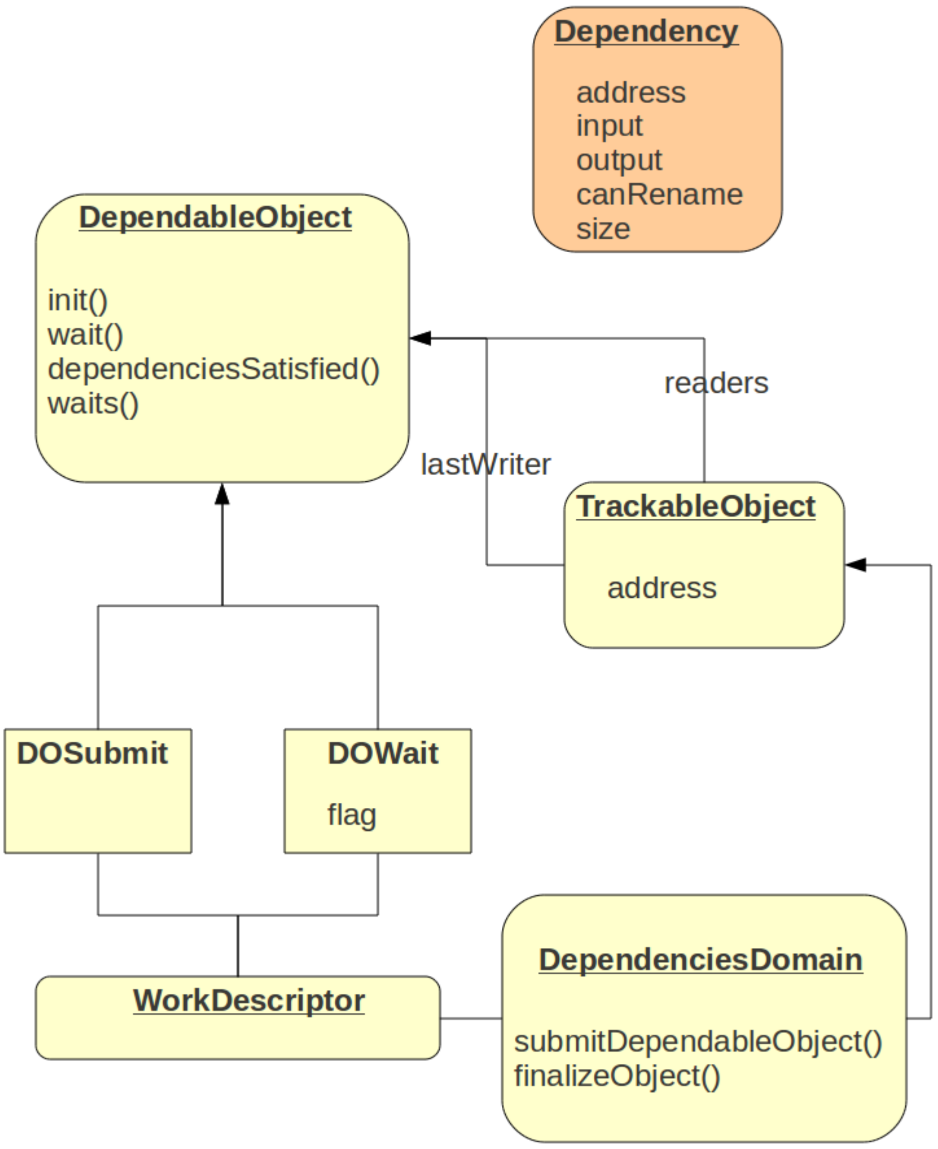

Figure 5.1: *Nanos++ dependencies system design.*

is added to the Dependencies domain, its dependencies are checked using their address as an identifier. When a Dependable Object ends the finalizeObject() is invoked in the Dependencies Domain in order to have the status of the dependencies updated.

The DependableObject class provides four virtual methods that are used by the DependenciesDomain in order to provoke some actions. These methods are the *init()* method, which is used when a DependableObject is submitted. It is invoked by the DependenciesDomain and initializes the DependableObjects' internal members when this is necessary. The second method is the *wait()* method which is also invoked by the DependenciesDomain after the DependableObject has already been submitted. When all dependencies are satified in a DependableObject the *dependenciesSatisfied()* method is invoked by the DependenciesDomain. Finally, the last method of the DependableObject is the *waits()* method. This method is used by the DependenciesDomain to know whether the *wait()* method waits untill dependencies are satisfied. If so, the only action during DependableObject submission is adding it as successor for the tasks updating its dependencies.

- There are also two other entities, the **DOSubmit** and the **DOWait**. The first one represents a task which is submitted to the dependency system and its action takes place when its dependencies are satisfied. When a DOWait is submitted to a domain, the thread executing submitDependableObject() will add the DOWait to the dependency system and wait until its dependencies are satisfied before returning control to the caller.

- **WorkDescriptor** which represent the runtimes' basic unit of work, has a DependenciesDomain where DependableObjects can be submitted. Each WorkDescriptor is submitted to the dependency system of its parent's DependenciesDomain. The class of the **SlicedWD** (*Sliced WorkDescriptor*) derives also from *WorkDescriptor*, a mechanism that allows WorkDescriptors to be decomposed in a number of several WorkDescriptors. A Sliced WordDescriptor is always related to a Slicer and a SlicerData. Each Slicer defines the work descriptor behavior and each SlicerData keeps all the data that is needed to separate the work.

Above we have described the basic outline of the Nanos runtime system. In order to implement the extensions we have already described, we have to change both the Mercurium compiler and the Nanos runtime system.

In the following sections there will be a detailed description of the implementation of both the two cases.

5.1 Coarse grain loop dependencies

As we have already mentioned, Mercurium is a source to source compiler. As a result the first step in order to implement the coarse grain dependencies was to introduce a function that captures the type of the dependencies and continues with creating one new structure. This structure passes the information about what kind of dependencies we face, if for example it is input, output or inout.

The first case as we have already described refers to the dependencies that include the whole variable, as shown in code number 5.1.

In this case and for the first dependent block, which is the one in code number 5.2

Listing 5.2: First dependent block

```
#pragma omp for output (b) schedule (static)
        for (i=0;i<1;i++){
                for(j=0;j<100000000;j++)
                        b[i]++;
                }
```

the code that will be produced in order to express the dependencies will be the one in 5.3:

Listing 5.1: Coarse Grain Dependencies

```
#pragma omp for output (b) schedule (static)
        for (i=0;i<1;i++){
                for(j=0;j<100000000;j++)
                        b[i]++;
                }
#pragma omp for output (c) schedule (static)
        for (i=0;i<4;i++){
                for(j=0;j<10;j++)
                        c[i]++;
                }
#pragma omp for input (b, c) schedule (static)
        for (i=0;i<4;i++){
                b[i]++;
                c[i]++;
                }
#pragma omp taskwait
```

Listing 5.3: Code produced for the first dependent block

```
 nanos_dependence_t _dependences[1] = {{
                (void **) &ol_args->b_0,
                ((char *) (b) - (char *) ol_args->b_0),
                {
                    0,
                    1,
                    1
                },
                sizeof(int [100])
            }};
```

Where we declare the number of dependencies, which in this case is one *nanos_dependence_t _dependences[1].* Then the variable that is dependent which in this case is the int b[100] and it is declared as *(void **) &ol_args-¿_0* . The sturcture:

0,

1,

1

declares that there is no input dependence (0), there is one output dependence (1) and if the dependence can be renamed (1), which is not currently used. Finally we declare which is the size of the valuable which in this case is declared by the *sizeof(int [100])*

For the next block of dependencies the code that will be produced will be quite the same as the one we have already described. For that we will explain the code that is produced in the final block of dependencies 5.4:

Listing 5.4: Third block of dependencies

```
#pragma omp for input (b, c) schedule (static)
        for (i=0;i<4;i++){
                b[i]++;
                c[i]++;
                }
```

that produces the code 5.5, which describes in the same way the dependencies with the difference that it integrates in one structure both of them.

After creating the above for all the coarse grain dependencies of the for loops, the next step is to submit them. After that is the runtime that takes the initiative.

In this first case with the coarse grain dependencies, there was no need to change any part of the runtime system, since all the structures needed by the runtime were already implemented for the *task* dependencies. [7]

5.2 Fine grain loop dependencies

On the other hand, in the second part of the implementation we have to change both the Mercurium compiler and the runtime system, in order to support fine grain loop dependencies.

The first step of the implementation was to create a function that fills the dependencies instead of the structure that used to be created in the coarse grain dependencies.

Listing 5.5: Code produced for the third block of dependencies

```
nanos_dependence_t _dependences[2] = {
               {
                   (void **) &ol_args->b_0,
                   ((char *) (b) - (char *) ol_args->b_0),
                   {
                       1,
                       0,
                       1
                   },
                   sizeof(int [100])
               },
               {
                   (void **) &ol_args->c_0,
                   ((char *) (c) - (char *) ol_args->c_0),
                   {
                       1,
                       0,
                       1
                   },
                   sizeof(int [100])
               }
           };
```

If for example we have the same piece of code, this time introducing fine grain dependencies, code 5.6:

Listing 5.6: Fine Grain Dependencies

```
int b[100];
int c[100];
#pragma omp for output (b[i]) schedule (static)
        for (i=0;i<1;i++){
                for(j=0;j<100000000;j++)
                        b[i]++;
        printf("2_thread=%d\n",omp_get_thread_num());

                }

#pragma omp for output (c[i]) schedule (static)
        for (i=0;i<2;i++){
                printf("3_thread=%d\n",omp_get_thread_num());
                for(j=0;j<100000000;j++)
                        c[i]++;

                }
#pragma omp for input (b[i], c[i]) schedule (static)
        for (i=0;i<4;i++){
                printf("4_thread=%d\n",omp_get_thread_num());
                }
#pragma omp taskwait
```

the code that would be produced by the compiler would now have the following functions:

1. for the first block of dependencies the **_compute_dep_0(int i_start, int i_end, _nx_data_env_0_t *ol_args, nanos_dependence_t *result)** function that will be created 5.7: which describes the dependencies and is called in the main function of the code.

2. For the second block of dependencies the function **_compute_dep_1(int i_start, int i_end, _nx_data_env_1_t *ol_args, nanos_dependence_t *result)** function will be created and it will have the same structure as the above.

Listing 5.7: Code generated from the first dependent block

```
void _compute_dep_0(....)
{
  result[0].address = (void **) &ol_args->b_0;
  result[0].flags.input = 0;
/*0 in case of output dependence and
1 in case of input or inout dependence*/
  result[0].flags.output = 1;
/*0 in case of input dependence and
1 in case of output or inout dependence*/
  result[0].flags.can_rename = 1;/*not used now*/
  result[0].offset = ...
  result[0].size = (i_end - i_start) * sizeof(int);
}
```

3. Finally, the last function for the last block of dependencies that will be created, **void _compute_dep_2(int i_start, int i_end, _nx_data_env_2_t *ol_args, nanos_dependence_t *result)**, will have the format of the code 5.8:

After creating the above functions that fill out the dependencies of the whole program, and after calling them in the main() function of the produced code, its the runtime that takes the initiative.

Now we will describe the changes that were made in the runtime system in order to complete the implementation of the second part. These changes concern only the static schedule when the chunk is zero, which means that the runtime decides in how many pieces to separate the iterations of the for loop.

At first we extended the structure of the slicer in order to add the fields of the fine grain dependencies and the computation of the dependencies. After that, we divide the WorkDescriptor that computes the for loop and create Sliced WorkDescriptors each one of which will describe the work for some specific iterations.

The struct in code 5.9 is the one that we use to compute the dependencies

Listing 5.8: Code generated from the second dependent block

```
 void _compute_dep_2(...)
{
    result[0].address = (void **) &ol_args->b_0;
    result[0].flags.input = 1;
    result[0].flags.output = 0;
    result[0].flags.can_rename = 1;
    result[0].offset = ...
    result[0].size = (i_end - i_start) * sizeof(int);
    result[1].address = (void **) &ol_args->c_0;
    result[1].flags.input = 1;
    result[1].flags.output = 0;
    result[1].flags.can_rename = 1;
    result[1].offset = ...
    result[1].size = (i_end - i_start) * sizeof(int);
}
```

of each loop, that is initialized by other structs that extract the information from the code that is created by the compiler.

Listing 5.9: Dependencies struct

```
 typedef struct {
   int _lower;  /**< Loop lower bound */
   int _upper;  /**< Loop upper bound */
   int _step;   /**< Loop step */
   int _chunk;  /**< Slice chunk */
   int _sign;   /**< Loop sign 1 ascendant, -1 descendant */
   int _num_of_dep; /**< the number of dependencies */
   nanos_compute_deps_fun_t _compute_deps;
   /**< pointer to the function that fills the dependencies*/
} nanos_slicer_data_for_internal_t;
```

The final change took place in the SlicerDataFor(), which is a Nanos++ object that is used in worksharing loops with the static policy. The change happen to the submit() method which now submits the sliced work descriptor with its dependencies 5.10.

This happens to every thread that executes the current loop except for

Listing 5.10: Submit

```
if ( _numDeps>0){
nanos_dependence_t *deps =(nanos_dependence_t *)
alloca( _numDeps * sizeof (nanos_dependence_t));
_compute_deps(nli->lower,nli->upper,nli,deps);
work.getParent()->submitWithDependencies(*slice,_numDeps,deps);
}
else{...}
```

the parent thread that finally submits the whole work descriptor again with the dependencies 5.11:

Listing 5.11: Parent thread submit

```
work.convertToRegularWD();
work.tieTo( (*team)[first_valid_thread] );
nli = ( nanos_loop_info_t * ) work.getData();

if ( _numDeps>0){
 nanos_dependence_t *deps =(nanos_dependence_t *)
 alloca( _numDeps * sizeof (nanos_dependence_t));
 _compute_deps(nli->lower,nli->upper,nli,deps);
 work.getParent()->submitWithDependencies(work,_numDeps,deps);
}
else {...}
```

Chapter 6

Evaluation

In order to evaluate the proposal of this thesis we used an artificial benchmark, that allow us to test the appropriateness and performance of the extension.

The benchmark 1 that we used is the following 6.1:

Listing 6.1: Benchmark 1

```
int main ()
{
  int a[1000000], i, j;

for ( i = N; i > 0 ; i-- ){
  a[i] = 0;
  for ( j = 0; j < i*100; j++ )
      a[i]++;
  printf("1.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
}

for ( i = N; i > 0; i-- )
  printf("2.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());

return 1;
}
```

In order to see the results of Benchmark 1 we run it in all three cases, without dependencies, with coarse grain dependencies and with fine grain dependencies.

We run the three versions using 1, 2, 3, 4, 6, 10, 12, 14, 18, 24 and 48 cores, in order to see the improvement that cause the versions of the benchmark with the dependencies. Then we divide the execution time of each one of the dependent benchmarks with the execution time of the benchmark that does not have dependencies.

6.0.1 No dependencies

Listing 6.2: Benchmark 1 with without dependencies

```
int main ()
{

int a[1000000], i, j;

#pragma omp for schedule(static) private(j)
for ( i = N; i > 0 ; i-- )
{
  a[i] = 0;
  for ( j = 0; j < i*100; j++ )
      a[i]++;

  printf("1.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
}

#pragma omp for schedule(static)
for ( i = N; i > 0; i-- )
  printf("2.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());


#pragma omp taskwait

return 1;

}
```

6.0.2 Coarse grain dependencies

Listing 6.3: Benchmark 1 with coarse grain dependencies

```
int main ()
{
...
#pragma omp for schedule(static) output(a) private(j)
for ( i = N; i > 0 ; i-- ){
   a[i] = 0;
   for ( j = 0; j < i*100; j++ )
      a[i]++;
 printf("1.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
}
#pragma omp for schedule(static) input(a)
for ( i = N; i > 0; i-- )
 printf("2.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
...
}
```

6.0.3 Fine grain dependencies

Listing 6.4: Benchmark 1 with fine grain dependencies

```
int main ()
{
...
#pragma omp for schedule(static) output(a[i]) private(j)
for ( i = N; i > 0 ; i-- ){
   a[i] = 0;
   for ( j = 0; j < i*100; j++ )
      a[i]++;
  printf("1.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
}
#pragma omp for schedule(static) input(a[i])
for ( i = N; i > 0; i-- )
  printf("2.a[%d]=%dthread=%d\n",i,a[i],omp_get_thread_num());
...
}
```

6.0.4 Numbers and Graphs

	$nodeps$	$coarsegdeps$	$finegdeps$	$nodeps/cgdeps$	$nodeps/fgdeps$
1	0.0088	0.0087	0.0088	1.0054	0.9981
2	0.0074	0.0069	0.0069	1.0737	1.0714
4	0.0079	0.0070	0.0051	1.1312	1.5331
6	0.0069	0.0083	0.0073	0.8308	0.9473
10	0.0153	0.0115	0.0083	1.3303	1.8445
12	0.0201	0.0144	0.0088	1.3939	2.2854
14	0.0438	0.0223	0.0424	1.9601	1.0315
18	0.0435	0.0258	0.0464	1.6845	0.9385
24	0.0429	0.0357	0.0477	1.2028	0.8991
48	0.0375	0.0363	0.0291	1.0332	1.2889

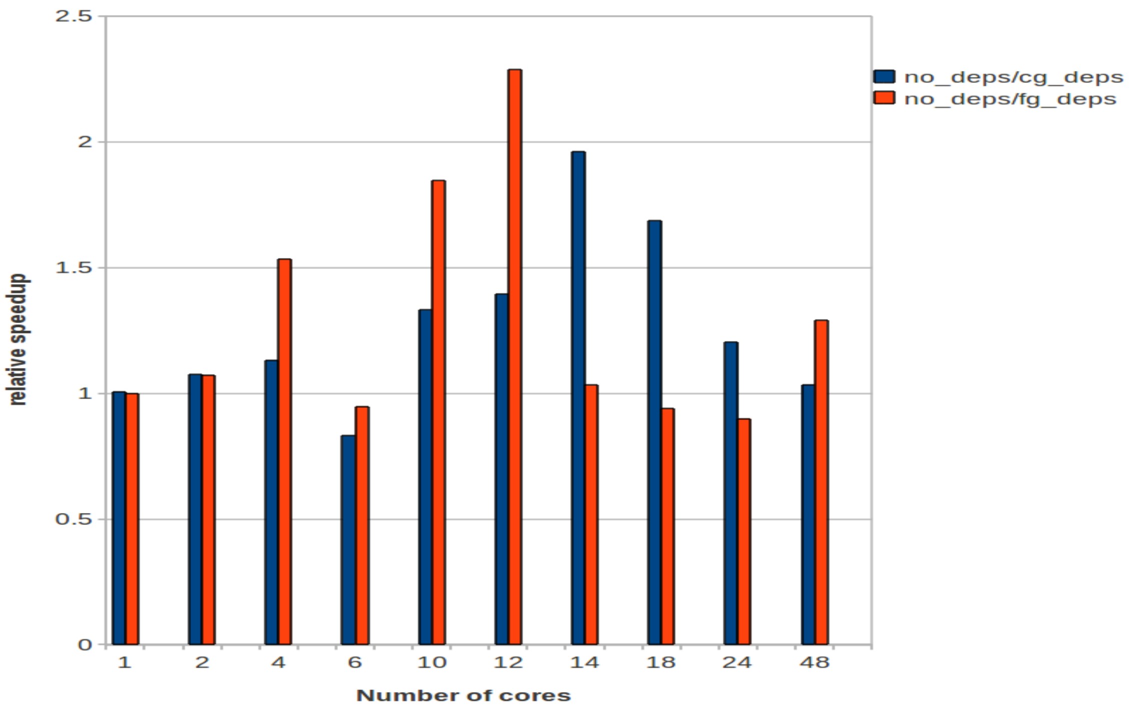

Figure 6.1: *Speed up for the first benchmark.*

Chapter 7

Conclusions and Future Work

OpenMP enables programmers to use simple constructs in order to parallelize their code. Unfortunately, OpenMP sometimes can be extremely rigid and not give the programmer the freedom that is needed to improve the efficiency even more. That is the reason why we have the need to extend it with more features.

These extensions are implemented as part of an other system called OmpSs. From all the above analysis, we realize that with the extensions that are proposed by OmpSs, the programmer can have more choices of exploiting even more regions. However, there is a long way in order to complete this idea, as the extensions for this project are related only with a specific scheduling policy, the static scheduling. This restriction does not give the programmer all the freedom that he needs, but above all does not let this idea show its full potential.

In the future there plans to implement the above ideas in all other types of scheduling, which makes us feel optimistic in bigger improvements and even better efficiency.

References

[1] Common multicore programming problems: Part 1 -Threads, data races, deadlocks and live locks. http://www.eetimes.com/design/embedded/4007217/ Common-multicore-programming-problems-Part-1-Threads-data-races-deadlocks-and-live-locks?pageNumber=0.

[2] Discussions on Dependences implementations. `https://pm.bsc.es/projects/nanox/wiki/Dependences`.

[3] Load Balance and Parallel Performance. `http://www.codeproject.com/KB/solution-center/LoadBalance.aspx?display=Mobile`.

[4] Nanos++ Dependency System Tutorial. `https://pm.bsc.es/projects/nanox/raw-attachment/wiki/MiniPresentationsScheduling/Dependencies.pdf`.

[5] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Javier Bueno, Alex Duran, Yoav Etsion, Montse Farreras, Foger Ferrer, Jesús Labarta, Vladimir Marjanovic, Lluis Martinell, Xavier Martorell, Josep M. Perez, Judit Planas, Alex Ramirez, Xavier Teruel, Ioanna Tsalouchidou, and Mateo Valero. Hybrid/Heterogeneous Programming with OmpSs and its Hardware/Software Implications. 2011.

[6] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007.

[7] Alejandro Duran, Roger Ferrer, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Jornual of Parallel Programming*, 37:292–305, 04/2009 2009.

[8] Mark D. Hill, Michael R. Marty. Amdahl's Law in the Multicore Era.

[9] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, 1965.

[10] OpenMP Architecture Review Board. Openmp application program interface. Specification, 2008.

[11] The openmp api specification for parallel programming. `http://www.openmp.org`.