Contents

Abstract

The aim of this thesis is to create a framework for guaranteeing *real-time* constraints on parallel *OpenMP* C++ code. The framework provides a static schedule for the allocation of tasks on system threads and a run-time support for the *real-time* execution. In order to do so the original source code is first instrumented, then profiled and finally rewritten by means of clang. Performance results are provided on a Computer Vision application.

0.1 Chapter's structure

Chapter 1

Introduction

1.1 Motivation, context and target application

The last years have seen the transition from single core architectures towards multicore architectures, mainly in the desktop and server enviroment. Lately also small devices as smartphones, embedded microprocessors and tablets have started to use more than a single core processor. The actual trend is to use a lot of cores with just a reduced instruction set as in *general purpose GPUs*.

Also *real time* systems are becoming more and more common, finding their place in almost all aspects of our daily routines; this systems often consist of several applications executing concurrently on shared resources. The main differences between this systems and a system designed to achive high performance can be summarized as follows:

- *Real time* programs need strict timing guarantees, while high performance programs try to achive the lowest possible computation time, usually in a best effort manner.
- *Real time* programs need to be predictable; in principle it could be that a real time program could finish almost always before its deadline on a high performance system, but it could be that in some exceptional cases, due to execution preemption, context switches, concurrent resource access . . . the program does not finish in time. To solve this, it may happen that the mean execution time of the real time program grows, but the program becomes also predictable, in the sense that it always finishes within its deadline.
- High performance systems need to "scale" well when the architecture becomes more powerful, while *real time* systems need just to satisfy the timing contrains, even with no performance gain.

The most relevant drawback of actual real time systems is that most of them are usually made to exploit just one single computing core, while their capabilities demand is growing. Applications like Computer Vision, Robotics, Simulation, Video Encoding/Decoding, Software Defined Radios,... have the necessity to process in parallel more tasks to achive a positive feedback for the user. This bring to two possible solutions:

- Find new and better scheduling algorithms to allocate new tasks using the same single core architecture
- Upgrade the processing power by adding new computing cores or by using a faster single core.

The first solution has the disadvantage that, if the computing resources are already perfectly allocated, it is not possible to find any better scheduling for the tasks to make space for a new job. A faster single core is also often not feasible, given the higher power consumption and temperature; this aspect is very relevant in embedded devices. The natural solution to the problem is to exploit the new trend toward multicore systems; this solution has opened a new research field and has brought to view a lot of new challenging problems. Given that the number of cores is doubling according to the well known *Moores law*, it is very important to find a fast and architecture independent way to map a set of *real time* tasks on computing cores. With such a tool, it would be possible to upgrade or change the computing architecture in case of new *real time* jobs, just scheduling them on the new system.

1.2 Objective

To overcome the problems stated before, the described tool must have four fundamental features:

- An easy *API* for the programmer to specify the concurrency between *real time* tasks together with all the necessary scheduling parameters (deadlines, computation times, activation times ...)
- A way to visualize task concurrency and code structure as graphs.
- A *scheduling algorithm* which supports multicore architectures, adapting to the specific platform.
- A *run time support* for the program execution which guarantees the scheduling order of tasks and their timing contrains.

1.3 Supporting parallelism in C/C++

In the last years the diffusion of multi-core platforms has rapidly increased enhancing the need of tools and libraries to write multi-threaded programs.

Several major software companies developed their own multi-thread libraries like Intel with Thread Building Block (TBB) and Microsoft with Parallel Patterns Library (PPL). There are also several open-source libraries like Boost and OpenMP. With the release of C++11 standard also the standard C++ library supports threading.

Lately appeared also automatic parallelization tools. These softwares allow to automatically transform a sequential code into an equivalent parallel one, some of the best examples are YUCCA and the Intel C++ compiler.

Lastly it is worth to mention briefly GPU, that have become accessible to programmers with the release of tools like Nvidia's CUDA or the open-source OpenCL. These libraries allow the user to easily run code on the GPU; of course the program to run must have some particular features because GPU are still not general purpose.

In this thesis there was the necessity to find two multi-thread libraries, one used by the input program to describe its parallel sections and the other to run the input program with the static custom schedule.

The first library had to satisfy the following requirements. It has to allow to transform easily a given sequential realtime code into a parallel one, without upsetting too much the original structure of the code; given that these kind of code have been usually deeply tested and must meet strict requirements.

What stated above is important also because part of the thesis is to extract informations from the parallel code, such as to distinguish between two parallels regions of code and understand precedences and dependencies between blocks of code. These informations are vital to be able to transform the code in a set of tasks and to provide to the scheduler algorithm the most accurate parameters. Furthermore the parallel code has to be instrumented both to profile it and to divide it into tasks for the final execution. The analysis of the various frameworks didn't involved their execution performance as the aim of the thesis is to use the library calls just to create a database containing the information regarding the parallel structure of the code.

For these reasons OpenMP has soon been considered the best choice. OpenMP behaviour is described in the chapter **??**, here will be roughly presented some of its features, that are necessary to understand the reason of the choice.

First of all OpenMP is minimally invasive as it just adds annotations inside the original sequential code, without any needs of changing its structure. OpenMP works embedding pieces of code inside scopes and adding annotations to each of these scopes by means of the pragma construct. The scopes automatically identify the distinct blocks of code (tasks) and also give immediately some information about the dependencies among the differents blocks. The use of pragmas is very convenient as they are skipped by the compiler if not informed with a specific flag. This implies that, given an OpenMP code, to run it sequentially is just enough to not inform the compiler of the presence of the OpenMP. This feature will be usefull to profile the code. Finally OpenMP is well supported by the main compilers and it has a strong and large community developing it, including big companies such as Intel.

The second library had instead opposite requirements as it has to be highly efficient. For this reason has been chosen the C++ standard library. Since the release of the C++11 standard, the C++ standard library was provided with threads, mutex, semaphore and condition variables. This library has the drawback of being not easy to use when coming to complicated and structured tasks, but on the other side it is fully customizable as it allows to instantiate and use directly system threads. It provides the chance to directly tuning the performance of the overall program and differently from the other parallelization tools the *std* library allows to allocate each task on a specific thread.

1.4 The OpenMP standard

Jointly defined by a group of major computer hardware and software vendors, *OpenMP* is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from desktop to the supercomputer.

The *OpenMP API* uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by *OpenMP* directives. The *OpenMP API* is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full *OpenMP* support library) and as sequential programs (directives ignored and a simple *OpenMP* stubs library). An *OpenMP* program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially, as if enclosed in an implicit task region, called an initial task region, that is defined by the implicit parallel region surrounding the whole program.

If a construct creates a data environment after an *OpenMP* directive, the data environment is created at the time the construct is encountered. Whether a construct creates a data environment is defined in the description of the construct. When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. The code for each task is defined by the code inside the parallel construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. Each directive uses a number of threads defined by the standard or it can be set using the function call *void omp_set_num_threads(int num_threads)*. In this project this call is not allowed and the thread number for each directive is managed separately. There is an implicit barrier at the end of each parallel construct; only the master thread resumes execution beyond the end of the parallel construct, resuming the task region that was suspended upon encountering the parallel construct. Any number of parallel constructs can be specified in a single program.

It is very important to notice that *OpenMP*-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non conforming. Also the developed tool will only accept well written programs, whitout checking if they are *OpenMP*-compliant. The *OpenMP* specification makes also no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output statements (or routines) using the provided synchronization constructs or library routines; this assumption is also maintained in the developed tool.

In C/C++, *OpenMP* directives are specified by using the #**pragma** mechanism provided by the C and C++ standards. Almost all directives start starts with #**pragma omp** and have the following grammar:

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

A directive applies to at most one succeeding statement, which must be a structured block, and may be composed of consecutive #pragma preprocessing directives.

It is possible to specify for each variable, in an *OpenMP* directive, if it should be private or shared by the threads; this can be done using the clause attribute *shared(variable)* or *private(variable)*
There is a big variety of directives which permit to express almost all computational patterns; for this reason a restricted set has been choosen in this project. Real time applications tend to be composed by a lot of small jobs, with only a small amount of shared variables and a lot of controllers. Given this, the following *OpenMP* directives have been choosen:

- #**pragma omp parallel** : all the code inside of this block is executed in parallel by all the available threads. Each thread has its variables scope defined by the appropriate clauses.

- #**pragma omp sections** : this pragma opens a block which has to contain section directives; it has always to be contained inside a #**pragma omp parallel block**. There is an implicit barrier at the end of this block synchronizing all the section blocks which are included.

- #**pragma omp section** : all the code inside of this block is executed in parallel by only *one* thread.

- **#pragma omp for** : this pragma must precede a for cycle. In this case the *for loop* is splitted among threads and a private copy of the looping variable is associated to each. This pragma must be nested in a **#pragma omp parallel** directive or can be expressed as **#pragma omp parallel for** without the need of the previous one.

- **#pragma single** : this pragma must be nested inside a **#pragma omp parallel** and means that the code block contained in it must be executed only by a single thread

- **#pragma task** : this pragma must be nested inside a **#pragma omp parallel** and means that all the possible threads will execute in parallel the same code block contained in it. In the developed tool this structure is not allowed. The allowed structure instead is composed by a number of **#pragma task** nested inside a **#pragma single** block. The semantic of this contruct is the same as having **#pragma omp section**s inside **#pragma omp sections**.

The considered pragma set can be splitted into two groups:

- A first set composed of **#pragma omp parallel**, **#pragma omp sections** and **#pragma omp single** which are "control" pragmas, meaning that they are used to organize the task execution.

- A second set containing **#pragma omp section**, **#pragma omp task** and **#pragma omp for** which represent "jobs", since they contain the majority of the computation code.

OpenMP imposes that pragmas belonging to the second group must always be nested inside a control pragma and that no pragmas can be nested inside them. It is still possible to overcome this rule by invoking a function, which contains pragmas, inside one of the pragmas contained in the first group; however to make this approach work it is necessary to set the *OMP_NESTED* environment variable by invoking the function call *omp_set_nested(1)*.

With this subset of *OpenMP* it is possible to create all the standard computation patterns like *Farms*, *Maps*, *Stencils* ...

OpenMp synchronization directives as **#pragma omp barrier** are not supported for now; only the synchronization semantic given by the above directives is ensured.

1.5 Clang as LLVM frontend

Clang [5] is a compiler front-end for the C, C++ and Objective-C programming languages. It relies on LLVM as its back-end.

A compiler front-end is in charge of analyzing the source code to build the intermediate representation (IR) which is an internal representation of the program. The frontend is usually implemented as three phases: lexing, parsing, and semantic analysis. This helps to improve modularity and separation of concern and allows programmers to use the frontend as a library in their projects.

The IR is used by the compiler backend (LLVM in the case of Clang), that transforms it into machine language, operating in three macro phases: analysis, optimization and code generation.

The Clang project was started by Apple and was open-sourced in 2007. Nowadays its development is completely open-source and besides Apple there are several major software companies involved, such as Google and Intel.

Clang is designed to be highly compatible with GCC, Its command line interface is similar to and shares many flags and options with GCC. Clang was chosen for the development of the thesis over GCC for three main reasons. Clang has proven to be faster and less memory consuming in many situations [6]. Clang has a modular, library based architecture. This structure allows the programmer to easily embed Clang's functionalities inside its own code. Each of the libraries that forms Clang has its specific role and set of functions; in this way the programmer can simply use just the libraries he needs, without having to study the whole system. On the other side GCC design makes difficult to decouple the front-end from the rest of the compiler. The third and most important Clang's feature is that it provides the possibility to perform code analysis, extract informations from the code and, most important, to perform source-to-source transformation.

Clang was not the only possibility out of GCC, also the Rose Compiler and Mercurium were viable options.

The strength of Clang, is in its implementation of the Abstract Syntax Tree (AST). Clang's AST is different from ASTs produced by some other compilers in that it closely resembles both the written C++ code and the C++ standard.

The AST is accessed through the *ASTContext* class. This class contains a reference to the *TranslationUnitDecl* class which is the entry point into the AST (the root). It also provides the methods to traverse the AST.

Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types. Many of these hierarchies have several layers and bifurcations so that the whole AST is composed by hundreds of classes for a total of more than one hundred thousand lines of code. Basic types derive mainly from three main disjoint classes: *Decl*, *Type* and *Stmt*.

As the name suggests the classes that derive from the *Decl* type represent all the nodes matching piece of code containing declaration of variables (*ValueDecl*, *NamedDecl*, *VarDecl*), functions (*FunctionDecl*), classes (*CXXRecordDec*l) and also function definitions.

The Clang's AST is fully type resolved and this is afforded using the *Type* class which allows to describe all possible types (*PointerType*, *ArrayType*).

Lastly there is the *Stmt* type which refereres to control flow (*IfStmt*) and loop block of code (*ForStmt*, *WhileStmt*), expressions (*Expr*), return command (*ReturnStmt*), scopes (*CompoundStmt*), etc..

Together with the above three types there are other "glue" classes that allow to complete the semantic. The most remarkable ones are: the *TemplateArgument* class, that, as the name suggests, allows to handle the template semantic and the *DeclContex* class that is used to extend *Decl* semantic and that will be shown later.

To built the tree the nodes are connected to each other; in particular a node has references to its children. For example a *ForStmt* would have a pointer to the *CompoundStmt* containing its body, as well to the *Expr* containing the condition and the *Stmt* containing the initialization. Special case is the *Decl* class that is designed not to have children thus can only be a leaf in the AST. There are cases in which a *Decl* node is needed to have children, like for example a *FunctionDecl*, which has to refer to the *CompoundStmt* node containing its body or to the list of its parameters (*ParmVarDecl*). The *DeclContext* class has been designed to solve this issue. When a *Decl* node needs to have children it can just extend the *DeclContext* class and it will be provided with the rights to points to other nodes.

There are other two classes that are worth speaking about: *SourceLocation* and *SourceManager* class. The *SourceLocation* class allows to map a node to the source code. The SourceManager instead provides the methods to calculate the location of each node. These classes are very powerful as they allow to retrieve both the start and the end position of a node in the code, giving the exact line and column number. For example given a *ForStmt*, the *SourceManager* is able to provide the line number of where the stmt starts and ends, but also the column number where the loop variable is declared or where the increment is defined.

To traverse the AST the Clang provides the *RecursiveASTVisitor* class. This is a very powerful and quite easy to learn interface that allows the programmer to visit all the AST's nodes. The user can customize this interface in such a way it will trigger only on nodes he is interested about; for example the methods *VisitStmt()* or *VisitFunctionDecl()* are called each time a node of that type is encountered. Each AST's node class contains getter methods to extract informations out of the code. For example a *Stmt* class has a method to know what kind of *Stmt* is the node, as *IfStmt*, *Expr*, *ForStmt*, etc.. In turn *ForStmt* class provides methods to find out the name of the loop variable, it's initial value and the loop condition.

To better understand how the Clang'AST is structured, Code 1.1 and 1.2 contain a simple code and the associated AST.

```cpp
class A {
public:
  int x;
  void set_x(int val) {
    x = val * 2;
  }
  int get_x() {
    return x;
  }
};
int main() {
  A a;
  int  val = 5;
  a.set_x(val);
}
```

Code 1.1: Simple code.

```
TranslationUnitDecl
|-CXXRecordDecl <clang_ast_test.cpp:2:1, line:13:1> class A
| |-CXXRecordDecl <line:2:1, col:7> class A
| |-AccessSpecDecl <line:3:1, col:7> public
| |-FieldDecl <line:4:2, col:6> x 'int'
| |-CXXMethodDecl <line:5:2, line:7:2> set_x 'void (int)'
| | |-ParmVarDecl <line:5:13, col:17> val 'int'
| | `-CompoundStmt <col:22, line:7:2>
| |   `-BinaryOperator <line:6:3, col:13> 'int' lvalue '='
| |     |-MemberExpr <col:3> 'int' lvalue ->x
| |     | `-CXXThisExpr <col:3> 'class A *' this
| |     `-BinaryOperator <col:7, col:13> 'int' '*'
| |       |-ImplicitCastExpr <col:7> 'int' <LValueToRValue>
| |       | `-DeclRefExpr <col:7> 'int' lvalue ParmVar 'val' 'int'
| |       `-IntegerLiteral <col:13> 'int' 2
| |-CXXMethodDecl <line:9:2, line:11:2> get_x 'int (void)'
| | `-CompoundStmt <line:9:14, line:11:2>
| |   `-ReturnStmt <line:10:3, col:10>
| |     `-ImplicitCastExpr <col:10> 'int' <LValueToRValue>
| |       `-MemberExpr <col:10> 'int' lvalue ->x
| |         `-CXXThisExpr <col:10> 'class A *' this
| |-CXXConstructorDecl <line:2:7> A 'void (void)' inline
| | `-CompoundStmt <col:7>
| `-CXXConstructorDecl <col:7> A 'void (const class A &)' inline
|   `-ParmVarDecl <col:7> 'const class A &'
`-FunctionDecl <line:15:1, line:21:1> main 'int (void)'
  `-CompoundStmt <line:15:12, line:21:1>
    |-DeclStmt <line:17:2, col:5>
    | `-VarDecl <col:2, col:4> a 'class A'
    |   `-CXXConstructExpr <col:4> 'class A' 'void (void)'
    |-DeclStmt <line:18:2, col:14>
    | `-VarDecl <col:2, col:13> val 'int'
```

```
|   `-IntegerLiteral <col:13> 'int' 5
`-CXXMemberCallExpr <line:20:2, col:13> 'void'
  |-MemberExpr <col:2, col:4> '<bound member function type>' .set_x
  | `-DeclRefExpr <col:2> 'class A' lvalue Var 'a' 'class A'
  `-ImplicitCastExpr <col:10> 'int' <LValueToRValue>
    `-DeclRefExpr <col:10> 'int' lvalue Var 'val' 'int'
```

Code 1.2: Clang AST of the simple code.

Clang supports the insertion of custom code inside the input one through the *Rewriter* class. This class provides several methods that allow, specifying a *SourceLocation*, to insert, delete and replace text; it also allows to replace a *Stmt* object with another one. The programmer cannot know a priori the structure of the input source code, so the best way to insert the custom text, in the correct position, is during the parsing of the AST. It is in fact possible to access each node's start and end *SourceLocations* reference, to transform them in a line plus column number and insert the text at the end of the line or at line above or below, as needed by the program.

The inserted text and its position are stored during the parsing of the AST in a buffer inside the *Rewriter* object; when the parsing is completed the buffer's data is inserted in the code generating a new file.

Clang's support to pragmas and OpenMP is really recent. Intel provided an unofficial patched version of the original Clang, which fully supports the OpenMP 3.3 standard, in July 2013 and the patch has not yet been inserted in the official release. Although it is not an official release Intel has worked inline with the Clang community principle and design strategies and it also produced a good Doxygen documentation of the code. This patch works jointly with the Intel OpenMP Runtime Library [7], which is open-source.

For what concern the support to generic pragmas the only remarkable work, that goes close to this goal is the one of Simone Pellegrini. He indeed implemented a tool (Clomp [8]) to support OpenMP pragmas in Clang. Clomp is implemented in a modular and layered way; this implies that the same structure can be easily used to support customized pragmas.

Chapter 2

Design

2.1 The framework

The framework takes as input a C++ source code annotated with *OpenMP* and translates each pragma block in a task. After that the tool searches for the best possible schedule that satisfies the tasks timing contrains. The source code is then executed with the given schedule and the help of a newly produced run-time support.

The developed tool works accordingly to the following steps:

- the *AST*, Abstract Sintax Tree, of the source code is created using *Clang*. From this all the relevant information of each *OpenMP* pragma are extracted and inserted in a properly formatted *XML* file.

- Each pragma in the source code is substituted with a proper profiling function call. The execution of the new code produces a log file which includes, for each pragma, timing informations.

- The new source code and the pragma *XML* file are given as imput to a second tool written in *Python*. This tool parses the *XML* file and creates a graph which represents the parallel execution flow of the tasks. After that it executes the given profiled source code N times creating statistics of the execution. The graph, enhanced with the new profiling information, is saved as a new *XML* file

- A scheduling algorithm is run on the created graph to find the best possible scheduling sequence accordingly to the profiling information. The found scheduling is then checked to be compatible with the precedence contraints given by the *OpenMP* standard and, in case, a *XML* schedule file is created.

- The source code is rewritten substituting to each pragma a proper code block for the creation of the tasks. During the execution each task is passed to the run-time support which allocates it accordingly to the previously creted schedule.

Picture 2.1 gives a visual representation of the framwork.

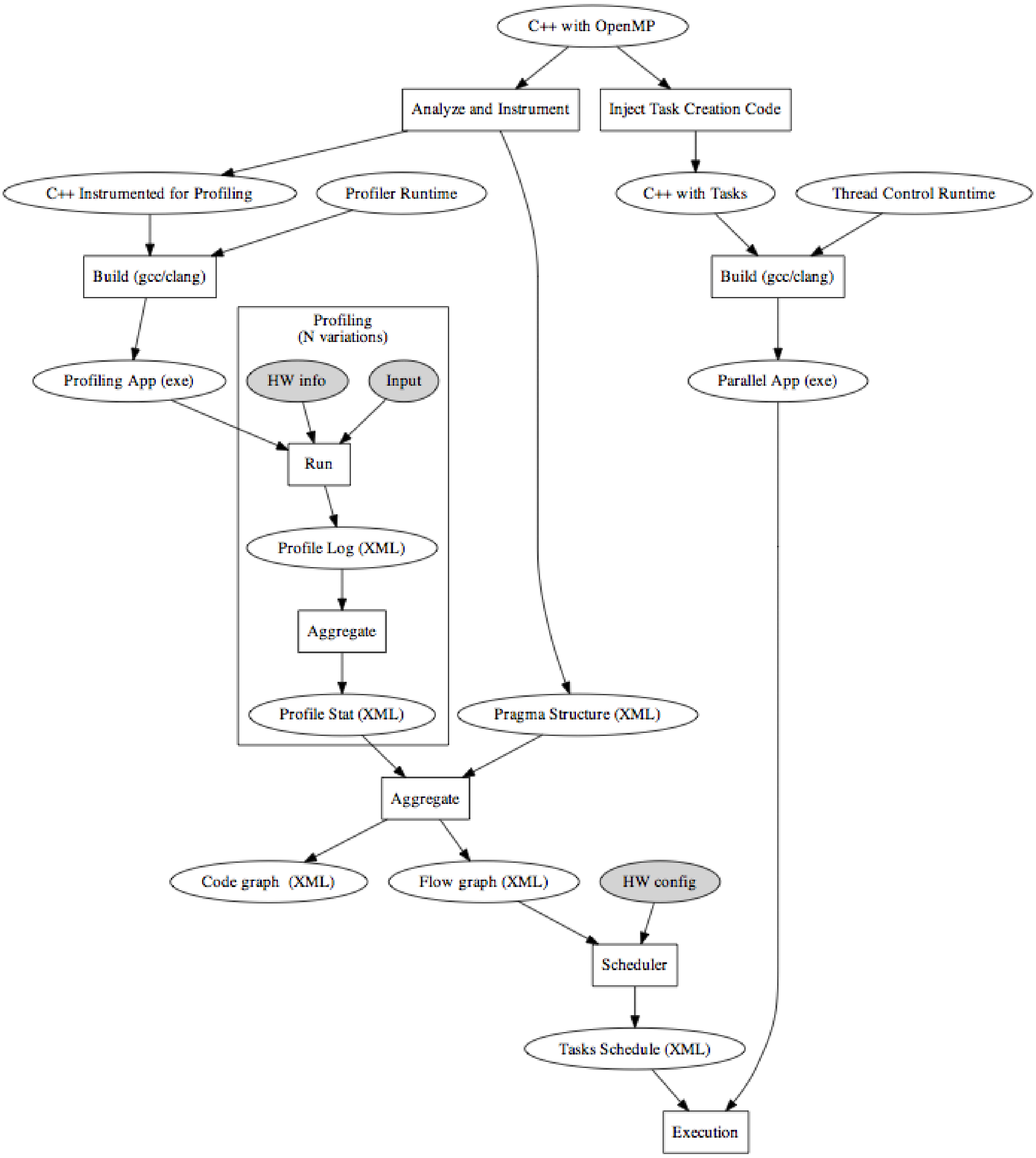

Figure 2.1: The famework structure

2.2 A simple example

A simple example has been developed in order to show how the tool works in each step. Given the *OpenMP* semantic described before, the two **#pragma omp section** are executing in parallel abd synchronize at the end of the **#pragma omp parallel private(bar)**. The clause *private(bar)* makes the *bar* variable private to each thread in order to have no race condition. To execute the code with some timing constrains some code has been added inside of the for loop which is not relevant for the explaination purpose.

```
#include <omp.h>

int work(int bar){
    #pragma omp parallel for
    for (int i = 0; i < bar; ++i)
    {
       //do stuff
    }
    return 0;
};

int main(int argc, char* argv[]) {
    int bar;
    #pragma omp parallel private(bar)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                //do stuff (bar)
                work(bar);
            }

            #pragma omp section
            {
                //do stuff (bar)
                work(bar);
            }
        }
    }
    return 0;
```

```
}
```

Code 2.1: Sample code

2.3 Analysis

2.3.1 Code

In this chapter will be presented and justified the choices that have been made during the parsing of the input source code; in particular which features have been extracted and why.

First of all will be presented how are translated OpenMP pragma in the Clang's AST. Our framework targets only a small part of the OpenMP environments, in particular only *parallel*, *sections* (*single*), *section* (*task*) and *for* pragmas. These pragmas have the common property that are all transformed into *Stmt* nodes in the AST. Each of these pragmas is represented in the AST by a specific class: *OMPParallelDirecetive*, *OMPSectionsDirective* and so on. All these class inherit from the *OMPExecutableDirective* class which in turn derives from the *Stmt* class.

These classes has three main functions, one to know the name of the directive associated to it, one to retrieve the list of the clauses associated to the pragma and the last to get the stmt associated to the pragma. Based on this last function the above directives can be divided into two groups, the first containing the *for* pragma and the second all the others. The difference between the two groups is that the *for* pragma has associated a *ForStmt*, while the other have associated a *CompoundStmt*. All the clauses derived from a common primitive ancestor which is the *OMPClause* class.

A real-time program, to be scheduled, needs to provide some informations about its time constraints, in particular the deadlines; these data can be provided in a separate file or directly inside the code. In this framework has been chosen the second approach and it has been done using the OpenMP clauses. The standard clauses clearly don't allow to specify the deadline of a pragma, so has been created a patch at the standard Clang to support of the *deadline* clause. This patch can be further enhanced to support other custom clauses, such as the activation time or the period of the pragma.

The thesis's framework parses the source code customizing the *RecursiveASTVisitor* interface; in particular it overrides two methods: *VisitFunctionDecl()* and *VisitStmt*. Each time the parser comes up with a *FunctionDecl* object or a *Stmt* object it invokes the associated custom function *VisitFunctionDecl()* adds all objects representing a function definition to a *FIFO* list. At the end of the parsing this list will contain the definition of all the functions in the input source code. *VisitStmt()* instead triggers on each stmt, it checks the type and if it is an *OMPExecutableDirective* node it adds it to another *FIFO* list, that at the end will contain all the pragmas. The two lists have the property that the order of their elements is given by the positions of the nodes in the source code, the smaller the starting line of a node the smaller its position in the list. This property is granted by the fact that the input code is parsed top down.

Once all the pragmas are in the list, the tool inspects each node, extracting information and saving them in a custom class. The newly created objects are used to build a pragmas tree. Since an input code can have multiple functions containing OpenMP pragmas and at static time it is not possible to understand where and when these functions will be called, the framework builds different pragmas tree for each function containing pragmas. It is possible to know, for each function, at which line its body starts and ends and so it is possible to match each pragma to the correct function. The tree structure is given by the nested architecture of the OpenMP pragmas, that has been described in chapter 1.4. The built of the tree is quite simple and straightforward as there are several properties that come to handy. The extracted pragmas in the list are ordered according to their starting line, so pragmas belonging to the same function are continuous. Every time a pragma is popped from the list, its starting line is checked, if it belongs to the same function of the previous node it is added to the current tree, otherwise it will be the root of a new tree. Another property is that a pragma is a child of another pragma, only if it is nested inside it; to be nested a node must have its starting line greater and its ending line smaller than the other one. The last property, that still comes from the ordered pragma list and from the definition of nested, is that a node can be nested only in its previous node (in the list) or in the father of the previous node, or in the father of the father and so on.

During the creation of the trees each AST node is transformed in a custom object that will carry only the information useful for the framework.

- Pragma type: parallel, sections, for, etc.
- Start line and end line of the statement associated with the pragma.
- A reference to the object containing the information of the function where the pragma is defined.
- A reference to the original AST node.
- The list of the pragma's clauses and of the variables involved.
- The list of its children nodes and a reference to its parent node.
- In case the node is of type *for* or *parallel for* it contains the reference to another object that contains all the information of the for:
  - the name of the loop var, its type and initial value.
  - The name of the condition variable, or the condition value.

- The increment.

The framework support only the parsing of for with a simple structure:

$$\begin{aligned}&parameter = value \mid var\\ &c\_op \;= < \mid > \mid <= \mid >=\\ &i\_op \;= ++ \mid -- \mid += \mid -= \mid *=\\ &for([type]\ var \;=\; parameter;\ var\ c\_op\ parameter;\ var\ i\_op\ [parameter])\end{aligned}$$

The *ForStmt* class fully supports the C++ For semantic, this means that it would be possible for the framework to support any kind of For declaration. It has been chosen to support only a basic structure because the effort required to expand the semantic it's very high and, with some slightly modification to the code, it is possible to support almost any possible scenarios. For example a for declaration like this:

```
for (int i = foo(); i < bar*baz; i ++)
```

can be translated as:

```
int init_val = foo();
int cond_val = bar*baz;
for(int i = init_val; i < cond_val; i ++)
```

becoming understandable by the framework.

Once all the pragmas have been translated and put in a tree, the new data structures are translated in the xml format. Each object is described either by a *Pragma* tag or by a *Function* tag. The two tags contains a list of other tags, one for each of the variables contained in the tree's objects. The semantic of the xml allows also to translate perfectly the original tree structure. This is done nesting *Pragma* tags one inside the other. The outermost tags are of type *Function*. Each function is the root of a tree so it will contain one or more *Pragma* tags. In turn each *Pragma* tag, if has children in the original tree, will contain other *Pragma* tags. Code 2.2 represent e portion of the xml generated from the sample code in paragraph 2.2.

```
<File>
    <Name>omp_test.cpp</Name>
    ...
    <Function>
        <Name>main</Name>
        <ReturnType>int</ReturnType>
```

```
        <Parameters>
            <Parameter>
                <Type>int</Type>
                <Name>argc</Name>
            </Parameter>
            <Parameter>
                <Type>char **</Type>
                <Name>argv</Name>
            </Parameter>
        </Parameters>
        <Line>12</Line>
        <Pragmas>
            <Pragma>
                <Name>OMPParallelDirective</Name>
                <Options>
                    <Option>
                        <Name>private</Name>
                        <Parameter>
                            <Type>int</Type>
                            <Var>bar</Var>
                        </Parameter>
                    </Option>
                </Options>
                <Position>
                    <StartLine>15</StartLine>
                    <EndLine>30</EndLine>
                </Position>
                <Children>
                    <Pragmas>
                        <Pragma>
                            <Name>OMPSectionsDirective</Name>
                            <Position>
                                <StartLine>17</StartLine>
                                <EndLine>29</EndLine>
                            </Position>
                            <Children>
                                <Pragmas>
                                    <Pragma>
                                        <Name>OMPSectionDirective</Name>
```

```
                                        <Position>
                                            <StartLine>19</StartLine>
                                            <EndLine>22</EndLine>
                                        </Position>
                                    </Pragma>
        ...
</File>
```

Code 2.2: XML file of the pragma structure of Code 2.1.

This xml file will then be passed to the scheduler algorithm, that will add a semantic to each node to build a parallelization graph, that will be used to create the tasks' schedule. The original trees are not discarded and they will be used to produce the final code, during a following parsing of the code.

2.3.2 Parallelism

Using the previously created *XML* file, which contains all the pragmas present in the source code, two different graphs are created. The first one reflects the pragmas structure in the source code, while the second one displays the execution flow of the different pragma blocks. Each pragma is represented by a node which contains all the relevant informations. All nodes derive from a general *Node* class; the most relevant attributes are the following:

- ptype : represents the type of the pragma.
- start_line : represents the code line where the pragma block starts.
- children : a list of all the children pragmas.
- parents : a list of all the pragma parents.
- time : the execution time of the pragma.
- variance : the variance of the execution time.
- deadline : the deadline of the task.
- arrival : the arrival time of the task.

Depending on the specific pragma special classes are derived like *For_Node* in case of a ***#pragma omp for*** or ***#pragma omp parallel for*** or *Fx_Node* in case of a function node.

To create the first graph the tool starts parsing the *XML* file and creating a proper object for each encountered pragma. It is important to notice that also pragmas which are not actually executed will be inserted in the graphs.

The second graph is created taking care of the execution semantic given by *OpenMp*. Again the *XML* file is parsed and an object is created for each pragma. Each object is then connected with the proper ones and if necessary fake *Barrier* nodes are added to guarantee the synchronization given by the standard. This special nodes are added whenever a "control" pragma is encountered; this is due to the fact that this type of pragmas use to have more than one children, creating a sort of diamond topology, which have to synchronize at the end of the pragma block fig:2.2.

2.4 Visual graph generation

To visualize the code structure, parallel code execution and the function call graph, three different type of graphs have been generated, each containing a series of nodes which are connected through undirected archs. The first node of each graph displays the function name along with the total computation time. For each function in the source code a different graph is created in two different formats; for visualization a *PDF* file, while a *DOT* file is created so that the graph can be manipulated with other tools. The code structure graph, simply called code graph, shows how pragmas are nested inside each other. Each node displays relevant informations as pragma type, starting line, execution time and variance. The parallel code execution graph, called flow graph, shows which nodes can be executed in parallel; some simple rules apply in this case to understand the execution flow:

- a node can execute only after all the parents have completed.
- All nodes which have a single parent in common can execute in parallel (this is shown by having the same color for arches which can execute in parallel).
- All nodes have to synchronize on barrier nodes.

In the call graph each node invoking a function containing pragmas is connected to the function subgraph by a directed arch fig:2.2; the execution flow continues after the function call terminates and resumes in the children of the caller node. The semantic of the execution is the same as the one of the flow graph.

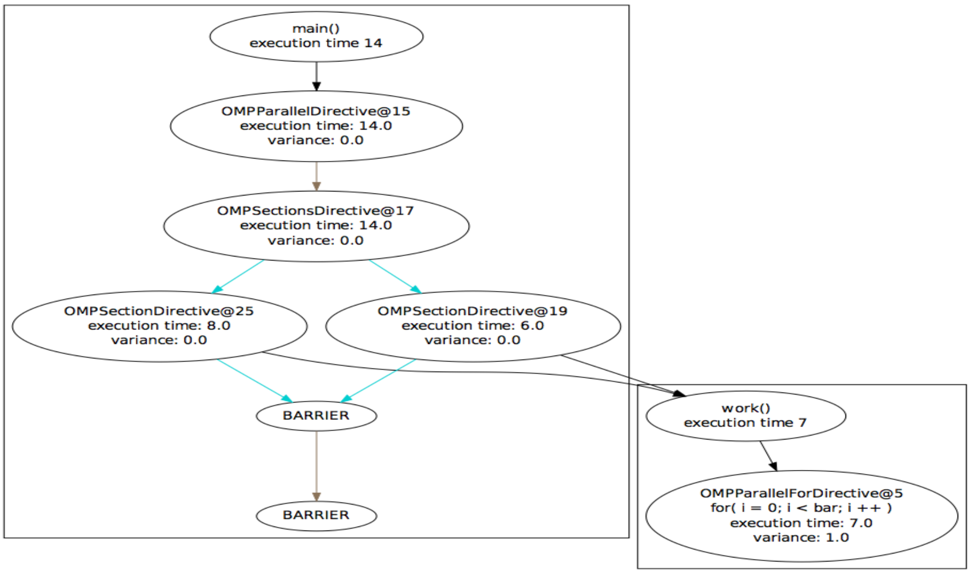

Figure 2.2: call graph example

2.5 Intrumentation for profiling

To produce a good schedule the framework needs informations about the tasks, in particular their computation time, their relations and precedences. In paragraph 2.1 we have seen how the pragmas are extracted and their structure; what was left is to retreive the computation time of each task and the functions' call graph. The only way to get these informations is to profile at runtime the sequential version of the input code; to have the sequential version, given that the code is parallelized with OpenMP, is enough to compile it without the *fopenmp* flag.

To be profiled the code needs to be instrumented. In the original code are added calls to a run-time support which calculates the time of the tasks, tracks the caller id of each function and stores them in a log file.

The instrumentation is performed during the first parse of the code, when the pragma statements are collected; the instrumentation does not depend on the semantic of each pragma and makes no distinction between functions and pragma. The idea is that, each time during the parsing a pragma or a call to a function is met, a call to the runtime support is inserted. As we have seen in paragraph 2.1, both functions and pragmas have associated either a *CompoundStmt* or a *ForStmt* node. A *CompoundStmt* has the characteristic that it always represents a block of code enveloped in a couple of curly brackets (a scope). In C++ variables declared inside a scope are locally to it, so are destroyed when the scope ends; the idea is to insert in the original code, at the beginning of the scope, a call to a custom object constructor, that starts a timer. When the scope ends the inserted object is destroyed and its destructor called; the destructor stops the timer and saves the time in a log file.

The tasks can be nested to each other, this means that an outermost task computation time contains the computation time of its subtasks; in other words, the sum of all the tasks' computation time could exceed the total computation time of the program. To obviate at this problem has been designed a method so that each task can keep track of the computation time of its children so that it is possible to obtain its effective computation time. This method allows also to keep track of the caller identity of each pragma or function; the caller is always either another pragma or a function.

This method works as follows: there is a global variable that stores the identity of the current pragma or function in execution. Each time a pragma or a function starts its execution the profiler object is allocate and its constructor invoked. The function puts a reference of the pragmafunction, where it is defined, in the global variable and saves the old value as it identifies its caller. When the object is destroyed the destructor is invoked and it communicates to its caller its computation time, so that the other task can increment the variable containing the children computation time. Before ending the destructor swap again the value of the global variable, passing to it the identifier of its caller. In case of a For task the profiler evaluates the number of iterations; this is very important because it helps the scheduler algorithm to decide how much to split the For in the final parallel execution. This evaluation is done subtracting the initial value from the ending value and dividing for the increment. This method is not perfect because it may happen that the values of the loop variable or of the conditional variable are changed inside the For body, changing the number of iterations; however the framework's target applications are real-time programs, so it is very unlikely to find dynamic For block. A possible solution to this problem would be to create a new variable, initialized to zero, that it is incremented by one at each iteration and when the For completes its value is caught and stored in the log file. At the end the log file will contain for each task:

- The total time of the task, from when it was activated since it terminates.
- The time of all its nested tasks, to calculate the effective time just perform the difference with the total time.
- The identifier of the pragma or function that called the task.
- In case of For task the number of iteration.

Code 2.3 shows the log file of the code 2.1.

```
<LogFile>
  <Hardware NumberofCores="4" MemorySize="2000"/>
```

```
  <Pragma fid="3" pid="5" callerid="3" elapsedTime="6" childrenTime="0"
   loops="6"/>
  <Function fid="3" callerid="19" elapsedTime="6" childrenTime="6"/>
  <Pragma fid="12" pid="19" callerid="17" elapsedTime="6" childrenTime="6"
   />
  <Pragma fid="3" pid="5" callerid="3" elapsedTime="8" childrenTime="0"
   loops="8"/>
  <Function fid="3" callerid="25" elapsedTime="8" childrenTime="8"/>
  <Pragma fid="12" pid="25" callerid="17" elapsedTime="8" childrenTime="8"
   />
  <Pragma fid="12" pid="17" callerid="15" elapsedTime="14" childrenTime="
   14"/>
  <Pragma fid="12" pid="15" callerid="12" elapsedTime="14" childrenTime="
   14"/>
  <Function fid="12" elapsedTime="14" childrenTime="14"/>
</LogFile>
```

Code 2.3: XML file of the pragma structure of Code 2.1.

2.6 Profiling

The priviously instrumented code is first executed N times, which is given as input parameter, using as arguments the data contained in a specific text file. At each iteration the algorithm produces, for each function and pragma, their execution time and, in case of a #pragma omp for or #pragma omp parallel for, also the number of executed cycles. This data is gathered during the N iterations and then the mean value of the execution time, executed loops and variance for each node is produced and saved in a log file. Ex:

```
<Log_file>
  <Hardware>
    <NumberofCores>4</NumberofCores>
    <MemorySize>2000</MemorySize>
  </Hardware>
  <Function>
    <FunctionLine>3</FunctionLine>
    <Time>7.0</Time>
    <Variance>1.0</Variance>
    <CallerId>[19, 25]</CallerId>
    <ChildrenTime>7.0</ChildrenTime>
  </Function>
```

```
...
  <Pragma>
    <FunctionLine>12</FunctionLine>
    <PragmaLine>25</PragmaLine>
    <Time>8.0</Time>
    <Variance>0.0</Variance>
    <Loops>-1462062072.0</Loops>
    <CallerId>['17']</CallerId>
    <ChildrenTime>8.0</ChildrenTime>
  </Pragma>
  <Pragma>
    <FunctionLine>12</FunctionLine>
    <PragmaLine>19</PragmaLine>
    <Time>6.0</Time>
    <Variance>0.0</Variance>
    <Loops>-1462062072.0</Loops>
    <CallerId>['17']</CallerId>
    <ChildrenTime>6.0</ChildrenTime>
  </Pragma>
...
```

The new data is added to the flow graph previously produced to be used later in the scheduling algorithm. This graph is then saved as *XML* file by saving nodes and edged separately, giving each a unique identifier.

```
<File>
  <Name>source_exctractor/test_cases/thesis_test/omp_test.cpp</Name>
  <GraphType>flow</GraphType>
  <Function id="30">
    <Name>work</Name>
    <ReturnType>int</ReturnType>
    <Parameters>
      <Parameter>
        <Type>int</Type>
        <Name>bar</Name>
      </Parameter>
    </Parameters>
    <Line>3</Line>
    <Time>7.0</Time>
```

```xml
    <Variance>1.0</Variance>
    <Callerids>
      <Callerid>19</Callerid>
      <Callerid>25</Callerid>
    </Callerids>
    <Nodes>
      <Pragma id="58">
        <Name>OMPParallelForDirective</Name>
        <Position>
          <StartLine>5</StartLine>
          <EndLine>8</EndLine>
        </Position>
        <Callerids>
          <Callerid>3</Callerid>
        </Callerids>
        <Time>7.0</Time>
        <Variance>1.0</Variance>
      </Pragma>
    </Nodes>
    <Edges>
      <Edge>
        <Source>30</Source>
        <Dest>58</Dest>
      </Edge>
    </Edges>
  </Function>
```

2.7 Schedule generation

The problem of finding the best possible schedule on a multicore architecture is known to be a *NP* hard problem. Given N tasks and M computing cores, the problem consists of creating K, possibly lower than M, execution flows in order to assign each task to a single flow. Each flow represents a computing core onto which the task should be run. To find a good solution a recursive algorithm has been developed which, by taking advantage of a search tree fig:2.3, tries explore all possible solutions, pruning "bad" brenches as soon as possible. Often the algorithm could not finish in a reasonable time due to the big number of possible solutions; to solve this problem a timer has been added to stop the computation after a certain amount of time given as input.

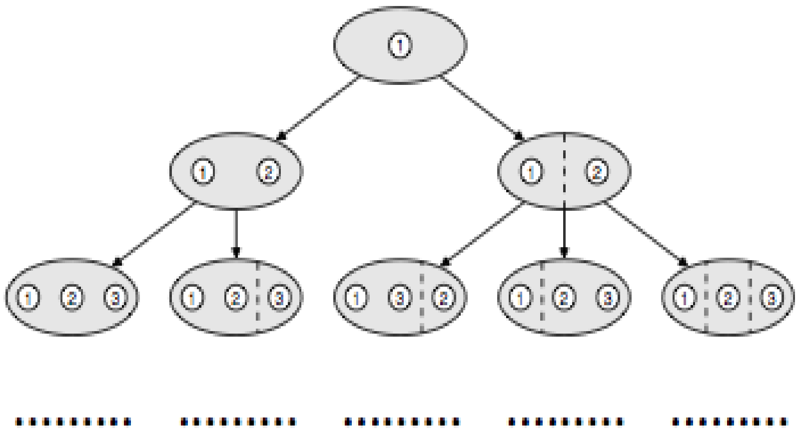

Figure 2.3: search tree

At each level of the search tree a single task is considered; the algorithm inserts the task in each possible flow, checks if the partial solution is feasible and, if affermative, continues until all tasks have been set arriving to a leaf. To check if the partial solution is feasible the algorithm calculates the cost of the actual solution and compares it with the best solution found so far, then it checks that the number of created flows is less than a predefined number and that the timer has not expired; if all this requirements are met, the brench continues its execution, otherwise it is pruned. After that all task are set, if the requirements are fullfilled, the actual solution is compared with the optimal found so far and, in case, the actual one will become the new optimal solution. To calculate if a solution is better than another a simple heuristic has been used: the cost of a task is it's computation time, each flow has as cost the summation of all the costs of the containing tasks and the cost of a set of flows (solution or partial solution) is the maximum of the costs of the flows. Given this metric a solution is better than another if it has a lower cost. Having a low flow cost means that the flows are well balanced; it is also important to notice that the algorithm is working in a breadth-first manner so that the number of flows is conservative, meaning that the lowest possible number is used to find the best solution. It is possible to easily add any number of pruning and cost metrics to improve the actual search algorithm

There is a small variation of the algorithm when a task containing a *#pragma parallel for* or *#pragma for* is encountered. In this case the algorithm tryes to split the for loop as much as possible creating new tasks which are added to the task list. First the task is divided in two tasks and they are added to the task list, then the task is splitted in three checking this solution and so on until arriving to the number of available cores. The execution time of each task will be updated accordingly to the number of sub tasks in which it was splitted.

A parallel version of this algorithm has also been developed in order to check more solutions in the same time. It is important to remember that in *Python*, even if more threads are created, there is only a single interpreter, so all the threads execution is serialized; to avoid this problem the tool creates different processes, each with its own *Python* interpreter. Given that the algorithm requires a lot of shared and private data which is updated at each computation step, the parallelisation of the algorithm would have been extremely complex, so an easier approach has been used. The same sequential algorithm is executed in parallel using for each process a randomized input order of the tasks. In this way each execution will produce all possible solutions in a different order; in any case after a certain amount of time all the processes will find all possible solutions, but with a timing contrain it is very likely that more solutions are checked, in the same time, then in the sequential version. The algorithm terminates returning an optimal solution in the sequantial case and K solutions in the parallel version; in this case the solutions are then compared and the best one is choosen as scheduling sequence.
It is important to notice that such a sequence could in principle not be schedulable, since the algorithm does not take care of precedence relations, but tries only to find the cheapest possible allocation. To check if the solution is feasible a second algorithm has been implemented following a modified version of the the parallel Chetto&Chetto algorithm [2].

This algorithm works in two phases, the first one sets the deadline for each task, while the second one sets its arrival time. To set the deadline, the algorithm sets the deadline of all the task with no predecessors to the expected deadline; then recursivly it sets the deadline of all task wich have all their successors deadline set by calculating the minimum of the difference between the computation time of the successor and the deadline of the successor.

In the second phase the algorithm sets all the arrival times of task with no predecessors to zero; after that it recursivly sets the arrival time of all tasks, which have the arrival time of all predecessors set, by calculating the maximum between all the arrival time of the predecessors, belonging to the same flow, and the deadline of all the tasks which are assigned to a different flow. This is due to the following fact:
let τ_j be a predecessor of τ_i, written as $\tau_j \to \tau_i$, with arrival time a_i and let F_k be the flow τ_i belongs to. If $\tau_j \in F_k$, then the precedence relation is already enforced by the previously assigned deadlines. So it is sufficient to ensure that task τ_i is not activated before τ_j. This can be achived by ensuring that :

$$a_i \geq a_i{}^{prec} = \max_{\tau_j \to \tau_i, \tau_j \in F_k} \{a_j\}.$$

If $\tau_j \notin F_k$, we cannot assume that τ_j will be allocated on the same physical core as τ_i, thus we do not know its precise finishing time. Hence, τ_i cannot be activated before τ_j's deadline d_j, that is:

$$a_i \geq d_i^{prec} = \max_{\tau_j \to \tau_i, \tau_j \notin F_k} \{d_j\}.$$

The algorithm checks then that all the deadlines and arrival times are consistent and in case produces the scheduling schema.

2.8 Instrumentation for the execution

In this paragraph will be presented the design strategies that have been used to instrument the input code to make it run accordingly to the schedule produced by the framework.

The framework needs to be able to isolate each task and execute it in the thread specified by schedule; to do so new lines of code are added in the original code to transform the old pragma in a collection of real atomic independent tasks. In this phase the functions are not considered as tasks and they won't be affected by the instrumentation. This is due to the fact that functions have no parallel semantic themselves and they can be simply executed by the tasks that invoke them, without affecting the semantic and improving the efficiency.

The idea of this phase is to transform each pragma block of code into a function, that will be called by the designated thread. One possibility was to take the code of the pragma, remove it from the function where it is defined and put it in a newly generated function; this way may be feasible with the Clang's tools but it is very complicated because of the presence of nested pragmas.

The other possibility, the one used in the framework, is to exploit, once again, the property of the pragmas to be associated with a scope. In C++ it is possible to define a class inside a function if the class is contained in a scope. Exploiting this property each pragma code has been enveloped inside a class declaration; in particular it constitutes the body of a function defined inside the new class.

In the case of a *for* pragma the framework needs to perform some additional modifications to the source code. Usually a For is splitted on more threads in the final execution so the For declaration has to be changed to allow the iterations to be scattered between different threads. In the For declaration are added two variables: an identifier, to distinguish the different threads and the number of threads concurring in the execution of the For. Here an example:

```
for(int i = begin; i < end; i ++)
```

becomes

```
int id; //incremental identifier of the task
int num_threads; // number of threads concurring in the execution of the
    for;
for(int i = begin + id * (end - begin) / num_threads; i < (id + 1) * (end
    - begin) / num_threads; i ++)
```

so if $num\_threads = 4$, $begin = 0$, $end = 16$, each thread will execute four iterations, in particular, the third thread, with $id = 2$ (identifier starts always from zero) will execute:

```
int new_begin = 0 + 2 *(16 - 0) / 4;
int new_end = 0 + (2 + 1) * (16 - 0) / 4;
for(int i = 8; i < 12; i ++)
```

After the definition of the class, at the end of the scope, the framework adds a piece of code that instantiates an object of the created class and pass it to the run-time support. The object will be collected by the designated thread which will invoke the custom function that contains the original code, running it.

This approach does not change the structure of the code, in particular nested pragmas remain nested; this means that there will be classes definition inside others classes, more precisely there will be tasks inside other tasks. This may seems a problem because it creates dependencies between tasks, not allowing a fully customizable schedule, but this is not true. According to the OpenMP semantics each task is not fully independent to the others, there can be precedences in the execution, but this approach grants that if two tasks can be run in parallel there will be no dependencies between them. To understand this we have to remind the OpenMP structure illustrated in paragraph 1.4, where it is explained that two pragmas containing computation code can be related only if in two different functions and so they won't be nested in the source code.

2.9 Run-time support

In this chapter will be presented how has been designed the run-time support for the execution of the final program. The aim of the run-time is to instantiate and manage the threads and to control the execution of the tasks. In particular it must allocate each task in the correct threads and must grant the precedence constraints between tasks setting the semaphores order. The runtime must be very fast to grant that the time constraints of the tasks are always satisfied. For this reason the runtime does no time consuming computations and all its allocation decisions are made based on what is written in the schedule. All the complicated calculation to decide the tasks allocation has been already done by the scheduler algorithm before the program execution and the produced schedule is taken as input by the program.

Now will be show step by step the execution of the runtime. First of all it parses the schedule file extracting all the informations and storing them in its own variables. It then instantiates a threads pool as large as specified in the schedule and it creates a job queue for each of thread.

The main program invokes the run-time support passing to it the object containing the function to be executed. The run-time embeds the received object in adhoc class, that includes the methods and variables needed to perform synchronization on that task. The created job is inserted in a vector common to all threads; at this point the runtime searches in the schedule to find which thread has been designated to run that job and puts an identifier of the job in that thread's job queue. In case of a For task the runtime has to execute some additional steps. Usually a For task is splitted on more threads, so the run-time has to duplicate the task for each thread involved; each copy is initialized with an incremental identifier, starting from zero and it also receives the total number of threads concurring at the execution of the task. These values are mandatory to inform each thread which iterations of the For has to execute.

Each thread executes an infinite loop. At the beginning of each iteration the thread checks if its queue contains a references to a job, in that case it pulls the first identifier and uses it to retrieve the real job in the vector of jobs and executes it. When the job ends the thread checks the schedule to see if it has to wait for other tasks to complete. Then the thread notifies that the job it was executing has been completed, so that any other thread waiting on that job can continue its executions. The common jobs' vector is needed because it allows to share information of a task between all the threads, in particular it is mandatory to perform task synchronization. In Code 2.1 the *Sections* task at line 16, once has launched its children tasks, has to wait for them to complete in order to finish. This rule is true for each "control" pragma that has children.

Chapter 3

Implementation

3.1 Scheduling XML schema

3.2 Instrumentation for Profiling

3.3 Profiling implementation

3.4 Schedule generating tool

3.5 Instrumentation for the execution

3.6 Run-time support

Chapter 4

Performance evaluation

4.1 A computer vision application

4.2 Results with statistics

Chapter 5

Conclusions

5.1 Achieved results

5.2 Future development

Bibliography

[1] Giorgio Buttazzo, Enrico Bini, Yifan Wu. *Partitioning parallel applications on multiprocessor reservations.* Scuola Superiore Sant'Anna, Pisa, Italy

[2] Giorgio Buttazzo, Enrico Bini, Yifan Wu. *Partitioning real-time applications over multi-core reservations.* Scuola Superiore Sant'Anna, Pisa, Italy

[3] Ricardo Garibay-Martinez, Luis Lino Ferreira and Luis Miguel Pinho, *A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems*

[4] Antoniu Pop (1998). *OpenMP and Work-Streaming Compilation in GCC.* 3 April 2011, Chamonix, France

[5] http://clang.llvm.org/

[6] http://clang.llvm.org/features.html#performance

[7] https://www.openmprtl.org/

[8] https://github.com/motonacciu/clomp/