Contents

Abstract

0.1 Objective

0.2 Chapter's structure

Chapter 1

Introduction

1.1 Motivation, context and target application

1.2 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 1.3, 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.

1.3 The OpenMP standard

1.4 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, in picture xxx is presented a simple code and the associated AST.

```
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);
}
```

```
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'
```

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 text, delete text 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 to insert the custom text, in the correct position, the best way is to perform the rewriters during the parsing of the AST. It is in fact possible to access to each node's start and end *SourceLocations* references, to transform them in a line plus column number and insert text at the end of the line, 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 strategy and it also produced a good Doxygen documentation of the code. This patch works jointly with the Intel OpenMP Runtime Library [cite], 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) to support OpenMP pragmas in Clang, but it is implemented in a modular and layered way; this implies that the same structure can be easily used to support customized pragmas. [cite]

Chapter 2

Design

2.1 The framework

2.2 A simple example

2.3 Analysis

2.3.1 Code

2.3.2 Parallelism

2.4 Intrumentation for profiling

2.5 Profiling

2.6 Schedule generation

2.7 Instrumentation for the execution

2.8 Run-time support

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