Contents

Abstract

0.1 Objective

0.2 Chapter's structure

0.3 Motivation, context and target application

The last years have seen the transition from single core architectures towards multicore architectures in the desktop and server enviroment mainly. 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; the actual problem is that most of this systems are made to exploit usually just one single computing core, while their capabilities are growing. 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 cannot 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 following *Moores law*, it is very important to find a fast and architecture independent way to map a set of *real time* jobs on computing cores. Given 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.

0.4 Supporting parallelism in C/C++

0.5 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 application 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 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.

With this semantic 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.

0.6 Clang as LLVM frontend

Chapter 1

Design

1.1 The framework

1.2 A simple example

1.3 Analysis

1.3.1 Code

1.3.2 Parallelism

1.4 Intrumentation for profiling

1.5 Profiling

1.6 Schedule generation

1.7 Instrumentation for the execution

1.8 Run-time support

Chapter 2

Implementation

2.1 Scheduling XML schema

2.2 Instrumentation for Profiling

2.3 Profiling implementation

2.4 Schedule generating tool

2.5 Instrumentation for the execution

2.6 Run-time support

Chapter 3

Performance evaluation

3.1 A computer vision application

3.2 Results with statistics

Chapter 4

Conclusions

4.1 Achieved results

4.2 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