Contents

Abstract

0.1 Objective

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 instrumented, profiled and rewritten by means of clang. Performance results are provided on a Computer Vision application.

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, 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 consists several applications executing concurrently on shared resources. The most relevant drawback is that most of this systems are usually made to exploit just one single computing core, while their capabilities demand is 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 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* jobs 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.
The solution needs three fundamental features which are supported by the described tool:

- An easy *API* for the programmer to specify the cuncurrency between *real time* tasks together with all the necessary parameters (deadlines, arrival times, activation times ...)
- The creation of a *scheduling algorithm* which supports multicore architectures.
- A *run time support* for the program execution which guarantees the scheduling order and the timing contrains.

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

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. The new data is added to the flow graph previously produced to be used later in the scheduling algorithm.

1.6 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 nodes, the problem consists of creating K, possibly lower than M, execution flows in order to assign each task to a single flow. To find a good solution a recursive algorithm has been developed which, by taking advantage of a search tree fig:1.1, tries to prune the 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.

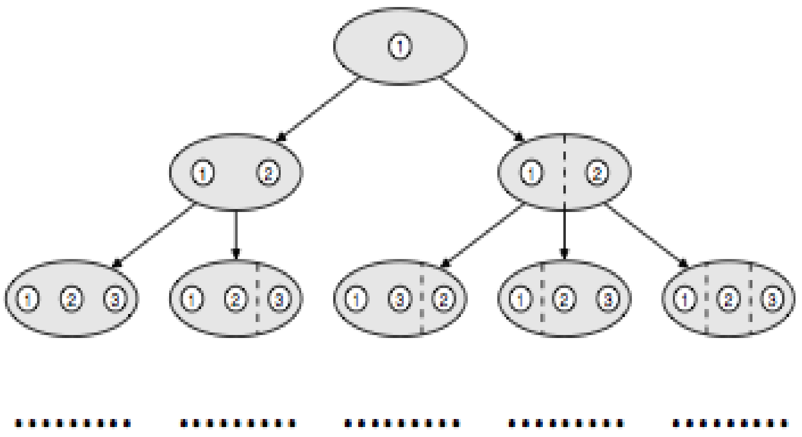

Figure 1.1: search tree

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 untill 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 now expired; if all this requirements are met, the brench continues its execution, else the brench 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 pos-

sible 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 untill 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 solution 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 easyer 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 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 likly to check more solutions than 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.

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