A Framework for static allocation of parallel OpenMP code on multi-core platforms

Giacomo Dabisias, Filippo Brizzi

Supervisors: E. Ruffaldi, G. Buttazzo

Università degli studi di Pisa,
Scuola Superiore Sant'Anna
Pisa,Italy

February 28, 2014

Context and motivations

Real-time systems are moving towards multicore architectures. The majority of multithread/core libraries target high performance systems.

- Real-time applications need strict timing guarantees and predictability.

Vs

- High performance systems try to achive a lower computation time in a best efford manner.

There is no actual automatic tool which has the advantages of HPC with timing contrains.

Objectives

The aim of this work is to create:

- An easy way to specify the concurrency between real- time tasks and scheduling parameters.
- 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.

Design Choices: OpenMP and Clang

OpenMP

- Minimal code overhead.
- Well spread standard.
- Opensource and supported by several vendors like Intel and IBM.

Clang

- Provides code analysis and source to source translation capabilities.
- Modularity and great efficency.
- Opensource and supported by several vendors like Google and Apple.

In July 2013 Intel released a patched version of Clang which fully supports the OpenMP 3.3 standard.

OMPSS

Devoleped at the Barcelona Supercomputing Center (BSC) in 2011.

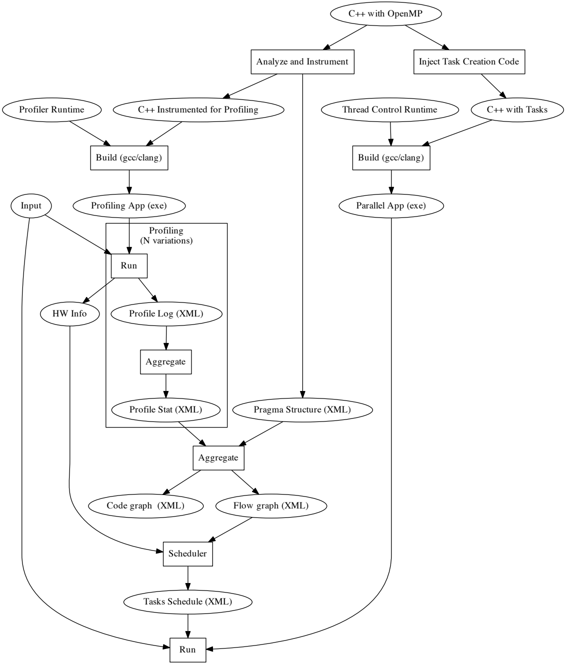
- Alternative to Clang.
- Extends OpenMP with new directives to support asynchronous parallelism.
- New directives extending accelerator based APIs like CUDA or OpenCL.
- It is based on the Mercurium compiler which provides source to source transformation tools.

OMPSS has not been chosen because its development is limited to the BSC.

General Design

The framework (Soma) takes as input a C++ code annotated with OpenMP.

- The pragmas are extracted with all relevant informations using Clang and saved as XML.
- The input code is rewritten to perform profiling.
- The scheduler tool uses these informations to create a possible schedule.
- The input code is rewritten to allow execution according to the generated schedule.
- The code is then executed with a custom run-time support.


C++ with OpenMP
Analyze and Instrument
Inject Task Creation Code
Profiler Runtime
C++ Instrumented for Profiling
Thread Control Runtime
C++ with Tasks
Build (gcc/clang)
Build (gcc/clang)
Input
Profiling App (exe)
Parallel App (exe)
Profiling
(N variations)
Run
HW Info
Profile Log (XML)
Aggregate
Profile Stat (XML)
Pragma Structure (XML)
Aggregate
Code graph (XML)
Flow graph (XML)
Scheduler
Tasks Schedule (XML)
Run

Graphs

The framework creates three types of graphs to visualize and work on the extracted data

- Code Graph: represents the nested structure of the pragmas in the source code.
- Flow Graph: represents the parallel execution flow and the synchronization barriers.
- Augumented Flow Graph: enhances the Flow Graph with the profiling information and the function calls.
- Schedule Graph: enhances the Augumented Flow Graph with scheduling informations.

The graphs are stored using XML and visualized using Graphviz.

OpenMP

Multiple threads of execution perform tasks defined by directives.

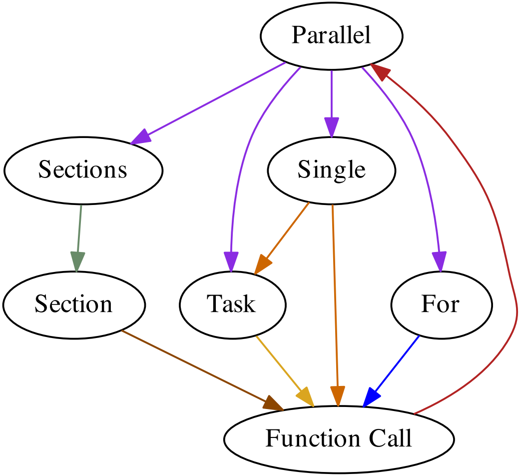
- Each directive applies to a block of C++ code embedded in a scope.
- Allows nested parallelism though nested directives.
- Clauses allow variables management.

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

Choosen subset for the framework:

- Control directives : parallel, sections, single.
- Working directives : task, section, for.

OpenMP - Hierarchy

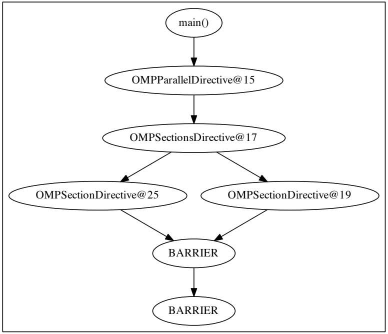


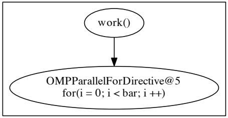
Nested structure of OpenMP pragma's

Simple Example

```c
void work(int bar){
    #pragma omp parallel for
    for (int i = 0; i < bar; ++i)
    {
        //do stuff
    }
};
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);
            }
        } //implicit barrier
    } //implicit barrier
}
```

First generated Flow Graph





Clang

The strength of Clang lies in its implementation of the Abstract Syntax Tree (AST).

- Closely resembles both the written C++ code and the C++ standard.
- Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor.
- Hundreds of classes for a total of more than one hundred thousand lines of code.

Clang - AST

To traverse the AST, Clang provides the RecursiveASTVisitor class.

- Very powerful and easy to learn interface
- Possibility to create a custom visitor that triggers only on specific nodes.

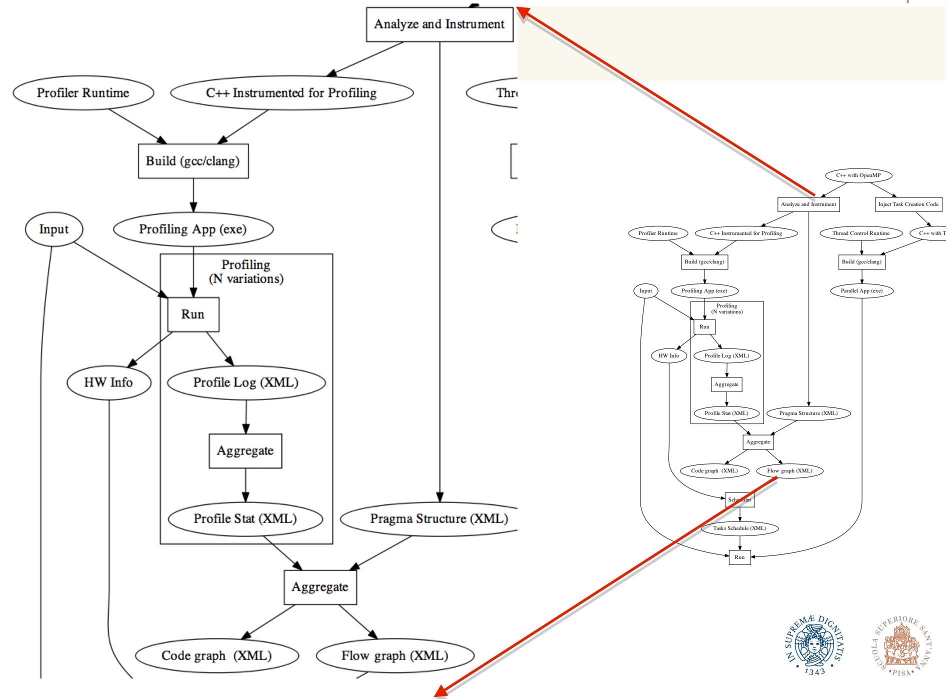
Clang supports the insertion of custom code through the Rewriter class.

- Allows insertion, deletion and replacement of code.
- Operations are performed during the AST visit.
- A new source file with all the modifications is generated at the end of the visit.

Clang - 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
...
```



Analyze and Instrument
Profiler Runtime
C++ Instrumented for Profiling
Build (gcc/clang)
Input
Profiling App (exe)
Profiling
(N variations)
Run
HW Info
Profile Log (XML)
Aggregate
Profile Stat (XML)
Pragma Structure (XML)
Aggregate
Code graph (XML)
Flow graph (XML)
C++ with OpenMP
Analyze and Instrument
Inject Task Creation Code
Profiler Runtime
C++ Instrumented for Profiling
Thread Control Runtime
Build (gcc/clang)
Build (gcc/clang)
Input
Profiling App (exe)
Parallel App (exe)
Profiling
(N variations)
Run
HW Info
Profile Log (XML)
Aggregate
Profile Stat (XML)
Pragma Structure (XML)
Aggregate
Code graph (XML)
Flow graph (XML)
Tasks Schedule (XML)
Run

Instrumentation for Profile

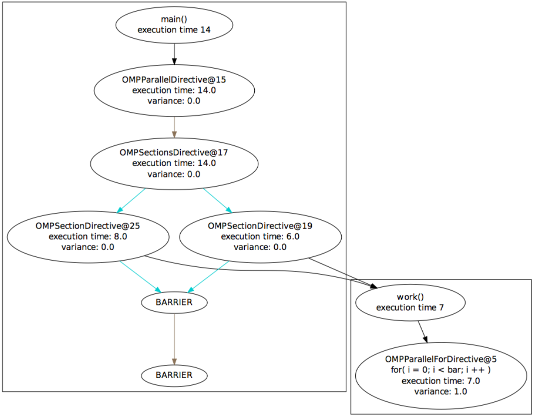
Creation of a custom profiler to time OpenMP pragma code blocks and functions. No existing profiling tool allows this operation.

- Code is instrumented with calls to a custom run-time support.
- Extracted information: execution time, children execution time, caller identifier, for loop counter.
- Output is saved in an XML file.
- Allows to create the Augumented Flow Graph containing all the call information.

```
...
//#pragma omp parallel for
if( ProfileTracker profile_tracker = ProfileTrackParams(3, 5, bar - 0))
for (int i = 0; i < bar; ++i)
{
     //do stuff
}
...
//#pragma omp section
if( ProfileTracker profile_tracker = ProfileTrackParams(12, 25))
{
    //do stuff (bar)
    work(bar);
}
...
```

Profile

- The profiled code is executed N times and statistics are obtained.
- Profile statistics can be associated to different input arguments.
- An edge is added for each function call that contains pragmas.



Possible improvement: probabilistic analysis of the function calls and execution times.


C++ with OpenMP
Analyze and Instrument
Inject Task Creation Code
Profiler Runtime
C++ Instrumented for Profiling
Thread Control Runtime
C++ with Tasks
Build (gcc/clang)
Build (gcc/clang)
Input
Profiling App (exe)
Parallel App (exe)
Profiling
(N variations)
Run
HW Info
Profile Log (XML)
Aggregate
Profile Stat (XML)
Pragma Structure (XML)
Aggregate
Code graph (XML)
Flow graph (XML)
Scheduler
Tasks Schedule (XML)
Run

Scheduler

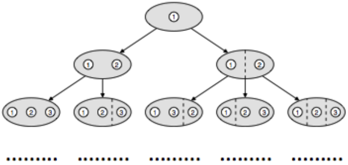
The scheduler takes as input the Augumented Flow Graph and the program's deadline.

- Since the problem is *NP*-complete, all possible schedules have to be checked.
- It is possible to set a fixed amount of computation time.
- A parallel version of the scheduler has been developed which achieves better results in a fixed amount of time.

The final schedule is saved as XML file which specifies for each pragma the thread identifier.

Scheduler - Algorithm

The scheduler assigns each task to a flow using a search tree. Each flow will be allocated to a different thread.



- All tasks are stored in an unordered list.
- The scheduler extracts one task at a time adding it to the current flows and to a new flow.
- The algorithm splits each pragma for node.
- The scheduler recurs on each flow pruning it as soon as possible.
- When a leaf is reached, the algorithm checks if the current solution is better then the previous one.

Scheduler - Feasibility

The produced schedule does not account for precedence relations. A modified version of Chetto&Chetto (1990) has been used to check the feasibility.

- All deadlines are set for each task starting from the last one.
- All arrival times are set for each task starting from the first and accounting for precedence relations.
- If all deadline are positive and each arrival time is less then the corresponding deadline the schedule is produced.


C++ with OpenMP
Analyze and Instrument
Inject Task Creation Code
Profiler Runtime
C++ Instrumented for Profiling
Thread Control Runtime
C++ with Tasks
Build (gcc/clang)
Build (gcc/clang)
Input
Profiling App (exe)
Parallel App (exe)
Profiling
(N variations)
Run
HW Info
Profile Log (XML)
Aggregate
Profile Stat (XML)
Pragma Structure (XML)
Aggregate
Code graph (XML)
Flow graph (XML)
Scheduler
Tasks Schedule (XML)
Run

Final Execution - Instrumentation

Each pragma block is transformed in a custom task.

- Each pragma code block is embedded in a new function call.
- Nested function declaration is not allowed in C++.
- Solved declaring the function in a local class defined inside the pragma block.
- All the variables used in the pragma block, but declared outside are passed to the class's constructor.
- The nested pragma structure is not changed.

Each for is rewritten in order to allow it to be splitted.

Final Execution - Instrumentation Example

```
 //#pragma omp section
{
    class Nested : public NestedBase {
    public:
        virtual shared_ptr<NestedBase> clone() const {
          return make_shared<Nested>(*this);
        }
        Nested(int pragma_id, int & bar) :
            NestedBase(pragma_id), bar_(bar) {}
        int & bar_;

        void fx(int & bar){
            //do stuff (bar)
            work(bar);
            launch_todo_job();
        }
        void callme() {
            fx(bar_);
        }
    };
    shared_ptr<NestedBase> nested_b = make_shared<Nested>(19, bar);
    if(ThreadPool::getInstance()->call(nested_b))
        todo_job_.push(nested_b);
}
```

Final Execution - Run-time

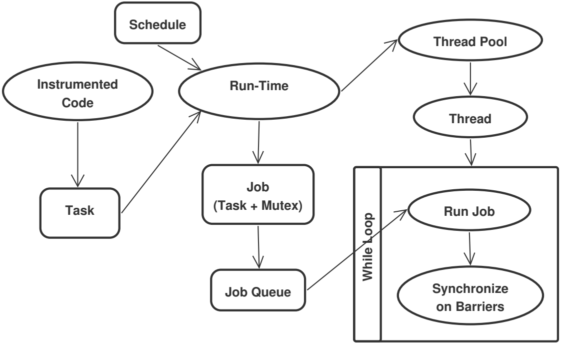
The run-time parses the generated XML schedule.

- Instantiates all the requested threads using the standard thread library.
- Creates a work queue for each thread.
- When invoked it receives a task and extracts from the schedule the designated execution thread.
- The task is enhanced with synchronization variables.
- The task is pushed in the corresponding working thread queue.

Final Execution - Run-time

Each thread behaves as follows

- An infinite loop is executed polling the its work queue.
- After work completion all the necessary synchronizations are perfomed before continuing with the next task.



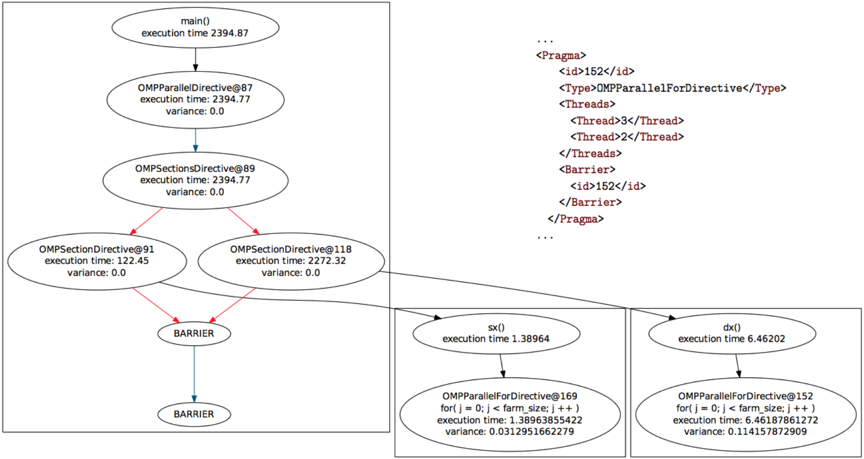
Test - General Structure

Face recognition algorithm in OpenCV.

- Takes as input two videos to simulate a stereo camera system.
- Frames are dispatched in blocks of N frames.
- All faces are detected, using a Cascade Detector Multiscale, and circled in each frame.
- All frames are saved on disk.

Three different qualities of the same videos have been used to test the algorithm (480p, 720p, 1080p).

Augumented Flow Graph for Evaluation




```
...
<Pragma>
    <id>152</id>
    <Type>OMPParallelForDirective</Type>
    <Threads>
      <Thread>3</Thread>
      <Thread>2</Thread>
    </Threads>
    <Barrier>
      <id>152</id>
    </Barrier>
  </Pragma>
...
```

Test Objectives

System framework evaluation

- Evaluate the instrumented program's correctness.
- Compare the OpenMP and Soma completion time for performance evaluation.
- Measure framework's overhead.
- Check system's predictability.

Results

- Test on a Intel i7@3.2 GHz with 6 cores and HT running Linux 3.8.0.
- Statistics are calculated over 5 executions.
- Tested with three different scheduler configurations: 4, 6 and 12 cores.
- Video properties:
  - 2 people in each.
  - 1 minute length.
  - 24 FPS.
  - Resolutions : 640x360 (230400px), 1280x720 (921600px), 1920x1080 (2073600px)

Results - Execution Times

	Sequential	OpenMP		Soma	
	$T_{seq}[s]$	$T_c(n)[s]$	$\epsilon(n) = \frac{T_{seq}}{nT_c(n)}$	$T_c(n)[s]$	$\epsilon(n) = \frac{T_{seq}}{nT_c(n)}$
480p(4)	750	195	0.96	195	0.96
720p(4)	3525	921	0.96	921	0.96
1080p(4)	8645	2271	0.95	2270	0.95
480p(6)	-	133	0.94	134	0.93
720p(6)	-	627	0.94	629	0.93
1080p(6)	-	1536	0.94	1539	0.94
480p(12)	-	98	0.64	92	0.68
720p(12)	-	427	0.69	426	0.69
1080p(12)	-	1043	0.69	1035	0.70

Results - Service Time

Service time in second of each thread (gap between the delivery of a parsed image).

- Soma variance < OpenMP variance
- Video quality 720p, 6 cores.

Thread	Sequential		OpenMP		Soma	
	T_s	*variance*	T_s	*variance*	T_s	*variance*
0	1.3263	0.1968	1.4226	0.0092	1.4987	0.0065
1	-	-	1.4225	0.0090	1.4297	0.0065
2	-	-	1.4225	0.0098	1.4298	0.0065
3	-	-	1.4257	0.0125	1.4117	0.0067
4	-	-	1.4256	0.0129	1.4111	0.0060
5	-	-	1.4256	0.0129	1.4117	0.0060

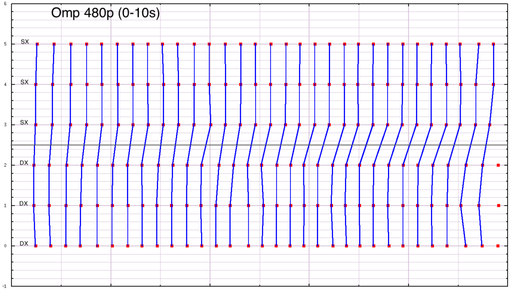
Performances remain consistent changing the video quality.

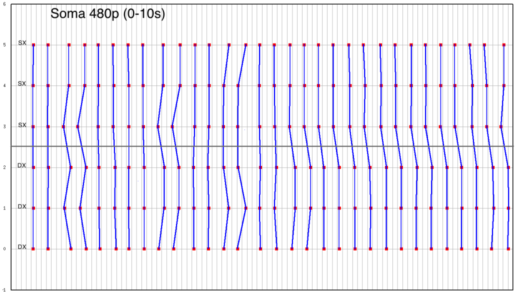
Results - Mean Service Time

Mean service time between cores for each video quality, changing the core number.

	Sequential	OpenMP		Soma	
	mean T_s	*mean* T_s	*mean var*	*mean* T_s	*mean var*
480p(4)	0.2823	0.2966	0.0014	0.2919	0.0004
720p(4)	1.3263	1.3955	0.0087	1.3884	0.0009
1080p(4)	3.2524	3.4399	0.0101	3.4369	0.0075
480p(6)	-	0.3038	0.0016	0.3023	0.0006
720p(6)	-	1.4241	0.0111	1.4206	0.0064
1080p(6)	-	3.4906	0.0238	3.4983	0.0197
480p(12)	-	0.4223	0.1421	0.4148	0.0044
720p(12)	-	1.9426	0.0862	1.9228	0.1334
1080p(12)	-	4.7394	0.3956	4.6915	0.6277

Results - Jitter





- Time between output of a frame for each of the 6 threads.
- Soma self realigning.
- OpenMP increasing delay between sx and dx.

Results - Comments

All the results of the framework are comparable with the OpenMP results.

- Almost same performance.
- The framework achieves a lower service time variance $\rightarrow$ more predictable.
- Low overhead.

The framework achieved the two main requested properties to work with real-time applications.

Future Directions

Creation of custom pragmas and clauses.

- Too many pragmas
- No possibility to specify real-time constrains

Better scheduler heuristics.

- Save time by early pruning.

Implement a probabilistic profiling step.

- Some functions could not be called.

Add the possibility to extend the concept to GPU and heterogeneous programming.