



UNIVERSITÀ DI PISA



Scuola Superiore
Sant'Anna

di Studi Universitari e di Perfezionamento

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 achieve a lower computation time in a best effort manner.

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

Objectives

The aim of this work is to create:

- ▶ An easy API 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 constrains.

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 efficiency.
- ▶ Opensource and supported by several vendors like Google and Apple.

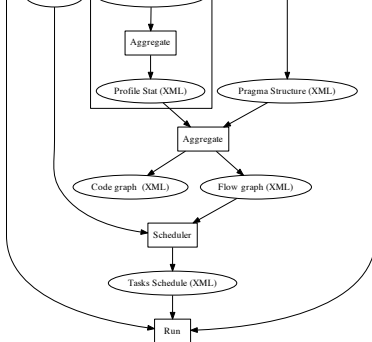
In July 2013 Intel released a patched version of Clang which fully supports the OpenMP 3.3 standard.



General Design

The framework 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.



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.
- ▶ Agumented Flow Graph: enhances the Flow Graph with the profiling informations and the function calls.
- ▶ Schedule graph: enhances the Agumented Flow Graph with scheduling informations.

The graphs are stored using Python objects and visualized using Graphviz.



OpenMP

Multiple threads of execution perform tasks defined by directives.

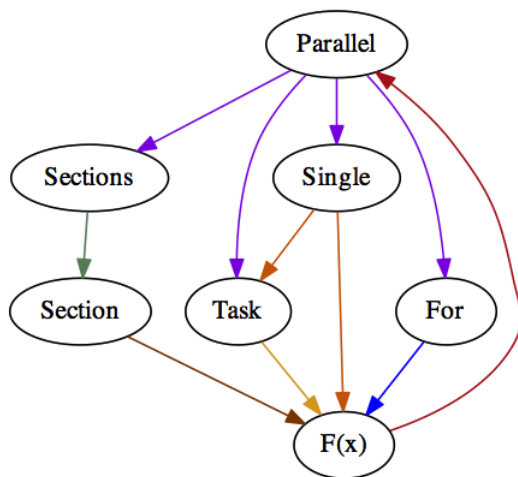
- ▶ Each directive applies to a block of C++ code embedded in a scope.
- ▶ Allows nested parallelism through nested directives.
- ▶ Clauses allow variables management.

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

Chosen subset for the framework:

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

OpenMP

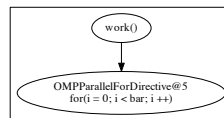
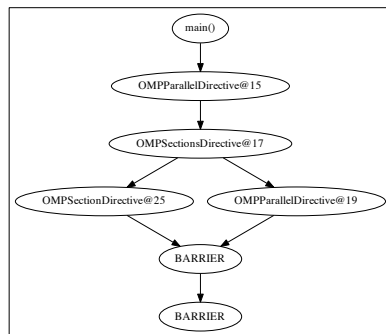


Simple Example

```

1 void work(int bar){
2     #pragma omp parallel for
3     for (int i = 0; i < bar; ++i)
4     {
5         //do stuff
6     }
7 };
8 int main(int argc, char* argv[]) {
9     int bar;
10    #pragma omp parallel private(bar)
11    {
12        #pragma omp sections
13        {
14            #pragma omp section
15            {
16                //do stuff (bar)
17                work(bar);
18            }
19            #pragma omp section
20            {
21                //do stuff (bar)
22                work(bar);
23            }
24        }
25    }
26 }

```



Clang

Clang and OpenMP:

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

```

1 class A {
2 public:
3     int x;
4     void set_x(int val) {
5         x = val * 2;
6     }
7     int get_x() {
8         return x;
9     }
10 };
11 int main() {
12     A a;
13     int val = 5;
14     a.set_x(val);
15 }

```

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
|   ...

```

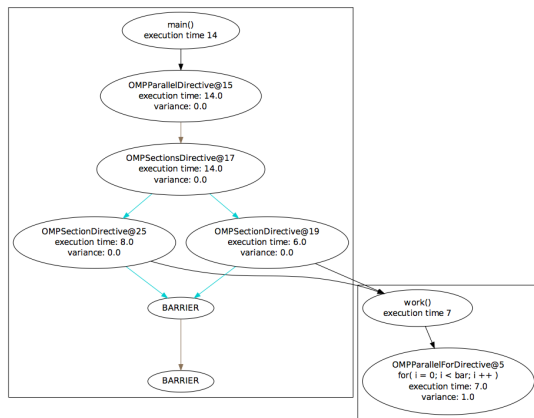
Instrumentation for Profile

- ▶ Creation of a custom profiler to time 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.

```
1  ...
2  //#pragma omp parallel for
3  if( ProfileTracker profile_tracker = ProfileTrackParams(3, 5, bar - 0))
4  for (int i = 0; i < bar; ++i)
5  {
6      //do stuff
7  }
8  ...
9  //#pragma omp section
10 if( ProfileTracker profile_tracker = ProfileTrackParams(12, 25))
11 {
12     //do stuff (bar)
13     work(bar);
14 }
15 ...
```

Profile

- ▶ The profiled code is executed N times and statistics are obtained.
- ▶ It is possible to use different arguments for the execution.
- ▶ An edge is added for each function call that contains pragmas.



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

Scheduler

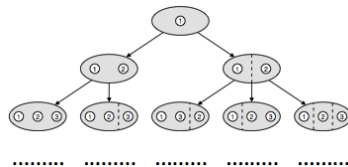
The scheduler takes as input the Agumented Flow Graph and the program's deadline.

- ▶ 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.

Final Execution - Instrumentation

Each pragma block is transformed in a custom task.

- ▶ Each pragma code block is embedded in a function call.
- ▶ The function is member of a class defined inside the scope of 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

```
1  // #pragma omp section
2  {
3      class Nested : public NestedBase {
4      public:
5          virtual shared_ptr<NestedBase> clone() const {
6              return make_shared<Nested>(*this);
7          }
8          Nested(int pragma_id, int & bar) :
9              NestedBase(pragma_id), bar_(bar) {}
10         int & bar_;
11
12         void fx(int & bar){
13             //do stuff (bar)
14             work(bar);
15             launch_todo_job();
16         }
17         void callme() {
18             fx(bar_);
19         }
20     };
21     shared_ptr<NestedBase> nested_b = make_shared<Nested>(19, bar);
22     if (ThreadPool::getInstance()->call(nested_b))
23         todo_job_.push(nested_b);
24 }
```

Final Execution - Run-time

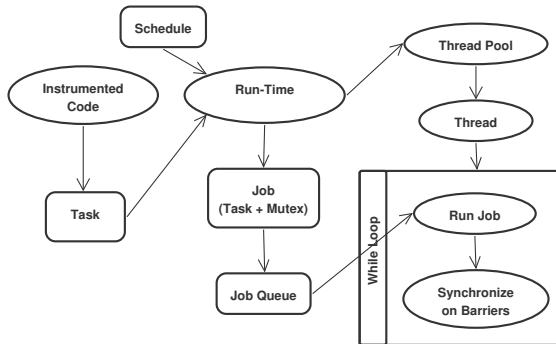
The run-time parses the generated XML schedule.

- ▶ 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 - Runtime

Each thread behaves as follows

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



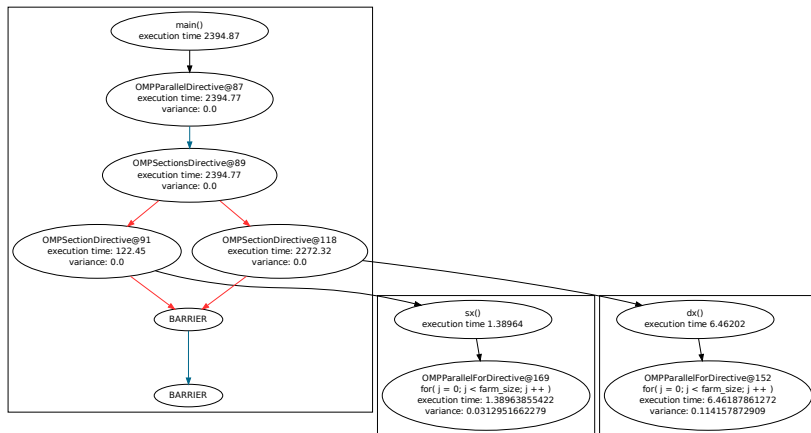
General structure

Face recognition algorithm in OpenCV.

- ▶ Takes as input two videos to simulate the human view.
- ▶ Frames are dispatched in blocks of N frames.
- ▶ All faces are detected 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).

General structure



Results

- 6 cores with hyperthreading.
- Statistics are calculated over 5 executions.

	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	<i>variance</i>	T_s	<i>variance</i>	T_s	<i>variance</i>
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

	Sequential	OpenMP		Soma	
	<i>mean T_s</i>	<i>mean T_s</i>	<i>mean var</i>	<i>mean T_s</i>	<i>mean var</i>
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 - boxplot)

Results - (Jitter)

Results - Comments

