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

Giacomo Dabisias, Filippo Brizzi

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 devloped framework has the following characteristics.

- 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 contrains.

Design Choice: OpenMP and Clang

OpenMP

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

Clang

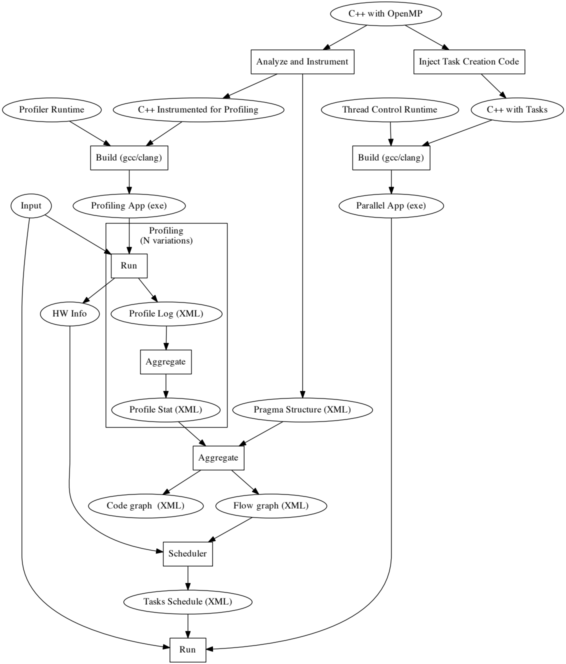
- Provides code analysis and source to source translation capabilities.
- Modularity and great efficency.
- Opensource and supported by several vendors like Google and Apple.

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

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.


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.
- Agumented Flow Graph : enhances the Flow Graph with the profiling informations and the function calls.

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

OpenMP

Multiple threads of execution perform tasks defined by directives.

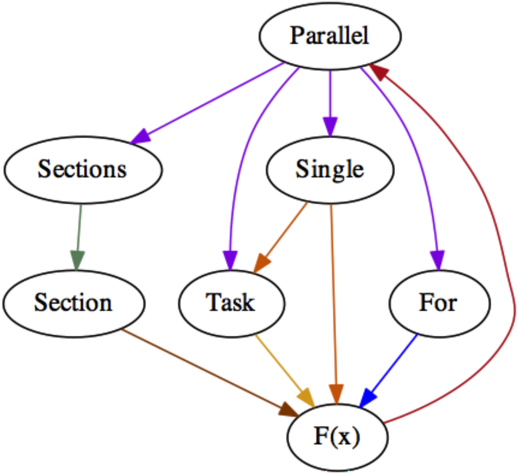
- Each directive applies to a block of C++ code embedded in a scope.
- Allows nested parallelism though nested directives.
- Clauses allow variables management.

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

Choosen subset for the framwork:

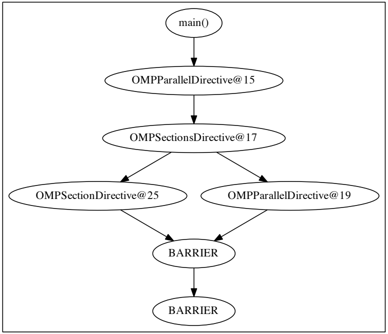
- Control directives : parallel, sections, single.
- Working directives : task, section, for.

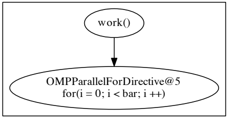
OpenMP



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);
            }
        }
    }
}
```





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

```cpp
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
...
```

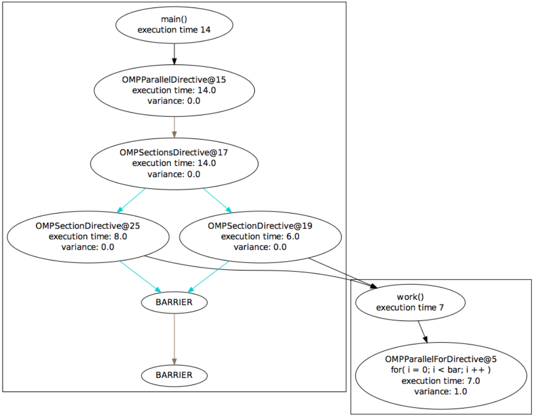
Instrumentation for Profile

- 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.

```
...
//#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.
- 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

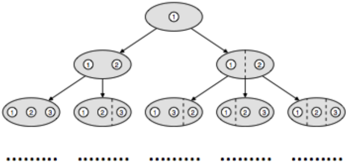
The scheduler takes as input the Agumented Flow Graph and the program's deadline.

- 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 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

```
//#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

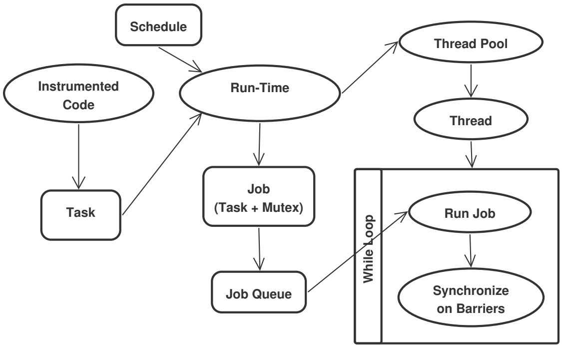
The run-time parses the generated XML schedule.

- 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 - Runt-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.



Scheduler - (Cetto & Chetto)

Final execution

Final execution - Intrumentation

Final execution - Run-time

Fianl execution - (thread pool)

Final execution - (multiple job queues)

Final execution - (synchronization)

General structure

General structure -(graph of the test code)

Results

Results - (some tables and graphs)

Results - (total completion time)

Results - (service time - boxplot)

Results - (Jitter)

Results - Comments