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

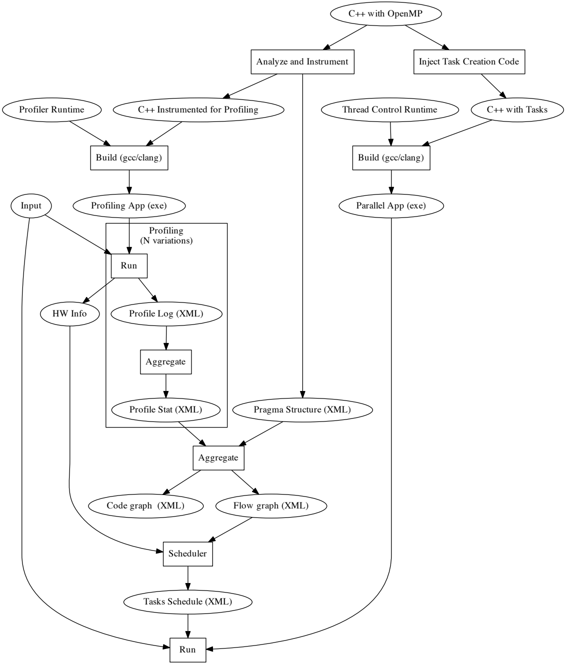
- Provides code analysis and source to source translation capabilities.
- Modularity and great efficency.
- Opensource and supported by several vendors like Google and Apple.

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

General Design

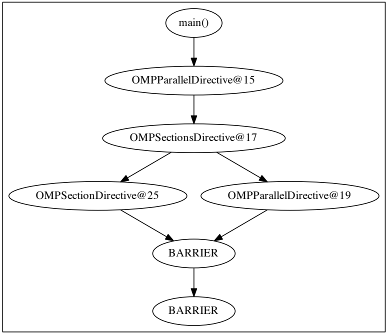
The framework takes as input a C++ code annotated with OpenMP.

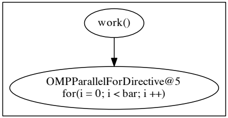
- The pragmas are extracted with all relevant informations using Clang and saved as XML .
- The input code is rewritten to perform profiling .
- The scheduler tool uses these informations to create a possible schedule.
- The input code is rewritten to allow execution according to the generated schedule.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

Simple Example

```
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);
            }
        }
    }
}
```





OpenMP

Multiple threads of execution perform tasks defined by directives.

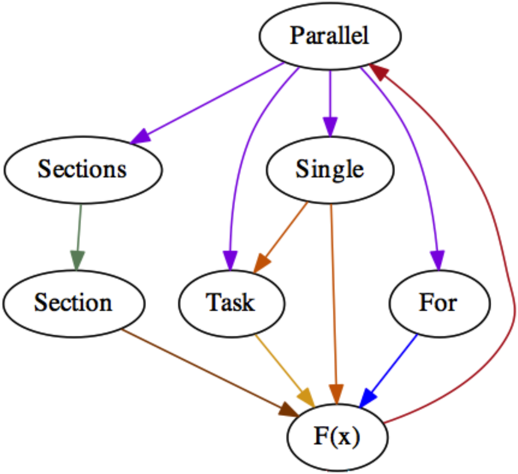
- Each directive applies to a block of C++ code embedded in a scope.
- Allows nested parallelism though nested directives.
- Clauses allow variables management.

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

Choosen subset for the framwork:

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

OpenMP



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.
  - Execution time, children execution time, caller identifier, for loop counter.
  - Output is saved in an XML file.

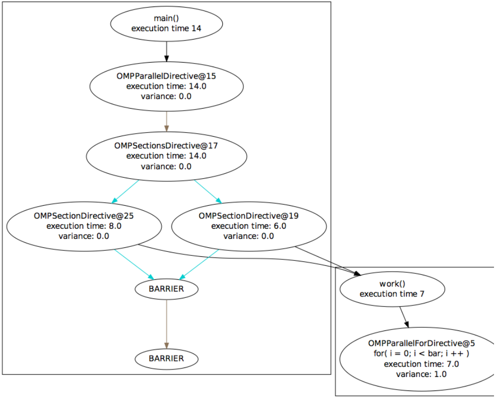
```
...
//#pragma omp parallel for
if( ProfileTracker profile_tracker = ProfileTrackParams(3, 5, bar - 0))
for (int i = 0; i < bar; ++i)
{
     //do stuff
}
...
//#pragma omp section
if( ProfileTracker profile_tracker = ProfileTrackParams(12, 25))
{
    //do stuff (bar)
    work(bar);
}
...
```

Graphs

- Pragmas are extracted from the source file and saved in an XML file.
- Profiler is executed N times and statistics are obtained.

Using these informations two pragma graphs are created.

- Flow Graphs: illustrates the parallel dependencies between tasks.
- Code Graph: illustrates the pragma nesting.



Scheduler

Intrumentation for profiling - Annotated example

Flow graph

Scheduler

Scheduler - Search tree

Scheduler - Constraints check

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