SOMA: An OpenMP Toolchain For Multicore Partitioning

E. Ruffaldi, G. Dabisias, F. Brizzi, G. Buttazzo

Scuola Superiore Sant’Anna

Pisa, Italy

ACM/SIGAPP Symposium on Applied Computing
April 6, 2016
Real-time systems are moving towards multicore architectures. The majority of multithread 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 constrains.
Objectives

Starting from a parallel C++ code, we aim to create:

- a way to **visualize** task concurrency and code structure as graphs.
- A **scheduling** algorithm, supporting multicore architectures and guaranteeing real-time constraints.
- A **run time support** for the program execution which guarantees the scheduling order of tasks.
State of the Art

StarPu\textsuperscript{1}
- Parallelization tool over heterogenous resources.
- Scheduler.
- Drawback: no timing guarantee.

RT-OpenMP\textsuperscript{2}
- Real-time OpenMP
- Drawback: mainly theoretical.

OMPSS\textsuperscript{3}(Barcelona Supercomputing Center)
- Asynchronous parallelism and data-dependency.
- Drawback: difficult to be extended.

\textsuperscript{1}C. Augonnet, et al.. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 2011.


Design Choices

Requirements

- Specification of the parallel tasks’ structure.
- Specification of the real-time parameters.
- Tool to instrument the code.
Design Choices

Requirements

- Specification of the parallel tasks’ structure.
- Specification of the real-time parameters.
- Tool to instrument the code.

**OpenMP**

- Minimal code overhead.

**Clang**

- Provides code analysis and source to source translation capabilities through AST traversal.
- Patched to support custom OpenMP pragmas: deadline and period.

Both are open source and supported by several vendors.
**Basic Example**

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

int main()
{
    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
    }
    #pragma omp sections
    {
        #pragma omp section
        {
            // do stuff (bar)
            work(bar);
        }
    }
    // implicit barrier
```

**Parallel code structure**
**SOMA**: Static OpenMP Multicore Allocator
Instrumentation for Profiling

Custom profiler to time OpenMP code blocks and functions.

- Extracted information: execution time, children execution time, caller identifier, for loop counter.

- Output as XML file.

```c
...  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 // #pragma omp parallel for
2 3 if (ProfileTracker profile_tracker = ProfileTrackParams(3, 5, bar - 0))
4 5 for (int i = 0; i < bar; ++i)
6 7 {  // do stuff
8 9 }
10 11 // #pragma omp section
12 13 if (ProfileTracker profile_tracker = ProfileTrackParams(12, 25))
14 15 {  // do stuff (bar)
16 work(bar);
17 } 
18 ...
```
The profiled code is executed $N$ times and statistics are obtained. Profile statistics can be associated to different input arguments.
The input is the profiling XML with the tasks’ deadline and period.

- The problem is \textit{NP-complete}
  - all possible schedules have to be checked,
  - high computational load.
- It is possible to set a \textbf{fixed} amount of \textit{computation time}.
- Scheduler \textit{parallel version}: better results in a fixed amount of time.

Output as XML file with the instructions for the real-time execution.
Scheduler: Algorithm

The scheduler assigns each task to a flow using a tree. Each flow will be allocated to a different virtual processor (thread).

- The algorithm splits each pragma for block.
- When a leaf is reached (complete schedule), the algorithm checks if the current solution is better than the previous one.
The produced schedule does not account for **precedence relations**.

- For each task we set:
  - the deadline starting from the last one;
  - the arrival time 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.
Instrumentation for Real-Time Execution

Pragma block $\rightarrow$ Custom task.

- Pragma code block is embedded in a **function call**.
  - Nested function declaration not allowed in C++.
  - Declare the function in a **scoped class**.

- Out of scope variables are caught.

- The nested pragma structure is not changed.

- Each *for* statement is rewritten in order to allow it to be split.
Real-Time Execution

- Final Executable
  - Tasks
  - XML Schedule

- Run-Time Support
  - Job (Task + Mutex + Thread ID)
  - Job Queue
  - Thread Pool
  - Thread
  - Run Job
  - Synchronize
    - While Loop
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.
Face recognition algorithm in OpenCV using Multiscale Cascade Detector (Viola Jones algorithm).

- **Input are two stereo camera videos.**
- **Frames are dispatched in blocks of N frames.**
Results

- Test on an Intel i7@3.2 GHz with 6 cores and HT running Linux Kernel 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, 1280x720, 1920x1080
## Results: Execution Times

<table>
<thead>
<tr>
<th></th>
<th>Sequential $T_{seq}[s]$</th>
<th>OpenMP $T_c(n)[s]$</th>
<th>$\epsilon(n) = \frac{T_{seq}}{nT_c(n)}$</th>
<th>SOMA $T_c(n)[s]$</th>
<th>$\epsilon(n) = \frac{T_{seq}}{nT_c(n)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>480p(4)</td>
<td>750</td>
<td>195</td>
<td>0.96</td>
<td>195</td>
<td>0.96</td>
</tr>
<tr>
<td>720p(4)</td>
<td>3525</td>
<td>921</td>
<td>0.96</td>
<td>921</td>
<td>0.96</td>
</tr>
<tr>
<td>1080p(4)</td>
<td>8645</td>
<td>2271</td>
<td>0.95</td>
<td>2270</td>
<td>0.95</td>
</tr>
<tr>
<td>480p(6)</td>
<td>-</td>
<td>133</td>
<td>0.94</td>
<td>134</td>
<td>0.93</td>
</tr>
<tr>
<td>720p(6)</td>
<td>-</td>
<td>627</td>
<td>0.94</td>
<td>629</td>
<td>0.93</td>
</tr>
<tr>
<td>1080p(6)</td>
<td>-</td>
<td>1536</td>
<td>0.94</td>
<td>1539</td>
<td>0.94</td>
</tr>
<tr>
<td>480p(12)</td>
<td>-</td>
<td>98</td>
<td>0.64</td>
<td>92</td>
<td>0.68</td>
</tr>
<tr>
<td>720p(12)</td>
<td>-</td>
<td>427</td>
<td>0.69</td>
<td>426</td>
<td>0.69</td>
</tr>
<tr>
<td>1080p(12)</td>
<td>-</td>
<td>1043</td>
<td>0.69</td>
<td>1035</td>
<td>0.70</td>
</tr>
</tbody>
</table>
## Results: Mean Service Time

Mean service time (gap between the delivery of a parsed image) in seconds.

- **SOMA variance < OpenMP variance**

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Sequential mean $T_s$</th>
<th>Sequential mean var</th>
<th>OpenMP mean $T_s$</th>
<th>OpenMP mean var</th>
<th>SOMA mean $T_s$</th>
<th>SOMA mean var</th>
</tr>
</thead>
<tbody>
<tr>
<td>480p(4)</td>
<td>0.2823</td>
<td>0.0014</td>
<td>0.2966</td>
<td>0.0014</td>
<td>0.2919</td>
<td>0.0004</td>
</tr>
<tr>
<td>720p(4)</td>
<td>1.3263</td>
<td>0.0087</td>
<td>1.3955</td>
<td>0.0087</td>
<td>1.3884</td>
<td>0.0009</td>
</tr>
<tr>
<td>1080p(4)</td>
<td>3.2524</td>
<td>0.0101</td>
<td>3.4399</td>
<td>0.0101</td>
<td>3.4369</td>
<td>0.0075</td>
</tr>
<tr>
<td>480p(6)</td>
<td>-</td>
<td>0.0016</td>
<td>0.3038</td>
<td>0.0016</td>
<td>0.3023</td>
<td>0.0006</td>
</tr>
<tr>
<td>720p(6)</td>
<td>-</td>
<td>0.0111</td>
<td>1.4241</td>
<td>0.0111</td>
<td>1.4206</td>
<td>0.0064</td>
</tr>
<tr>
<td>1080p(6)</td>
<td>-</td>
<td>0.0238</td>
<td>3.4906</td>
<td>0.0238</td>
<td>3.4983</td>
<td>0.0197</td>
</tr>
<tr>
<td>480p(12)</td>
<td>-</td>
<td>0.1421</td>
<td>0.4223</td>
<td>0.1421</td>
<td>0.4148</td>
<td>0.0044</td>
</tr>
<tr>
<td>720p(12)</td>
<td>-</td>
<td>0.0862</td>
<td>1.9426</td>
<td>0.0862</td>
<td>1.9228</td>
<td>0.1334</td>
</tr>
<tr>
<td>1080p(12)</td>
<td>-</td>
<td>0.3956</td>
<td>4.7394</td>
<td>0.3956</td>
<td>4.6915</td>
<td>0.6277</td>
</tr>
</tbody>
</table>
Results - Comments

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

- Almost same performance.
- SOMA has a lower service time variance → more predictable.
- Low overhead as OpenMP.

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

- More realistic test cases will be tested.
Future Steps

Creation of **custom pragmas** and clauses.

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

Better **scheduler heuristics**.

- Save time by early pruning.

Implement a **probabilistic profiling** step.

- Some functions may not be called.

Add the possibility to extend the concept to **heterogeneous computing**.
Thank you!

▶ Questions?

Email: fi.brizzi@sssup.it