Automatic Generation of Performance/Memory Models for Parallel Scientific Applications

Van Bui
Argonne National Laboratory

In collaboration with:
Boyana Norris, Lois Curfman McInnes, and Li Li
Argonne National Laboratory
Barbara Chapman and Oscar Hernandez
University of Houston
Kevin Huck
University of Oregon
The Challenge

- How can we make performance instrumentation, collection, and analysis a more automated/efficient process??

- A “standard” performance profiling interface does not exist for several parallel programming models
Why a Standard Profiling Interface?

- Reduce the cost of tool development
- Performance measurement and analysis more efficient across platforms
- These APIs should be designed to facilitate instrumentation, collection, and analysis
OpenMP Runtime API for Profiling

- OpenMP ARB “sanctioned” performance monitoring interface for OpenMP
- Implemented inside the OpenMP runtime library
- Performance tools communicate with OpenMP runtime library through this interface
Performance Measurement and Analysis System

Components

- Performance Collector (measurement + analysis)
- Database Component
- PerfExplorer (performance and power analysis)
- Application components
System Components
Performance Visualization

![Performance Visualization Diagram](image)

<table>
<thead>
<tr>
<th>Performance Metrics</th>
<th>Region Profile</th>
<th>System Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.35 Total Time(sec)</td>
<td>0.00 MAIN_</td>
<td>0.00 Linux - x86_64</td>
</tr>
<tr>
<td>0.00 OMP Time(sec)</td>
<td>2.08 omp parallel @ft.cftf1_555</td>
<td>0.00 cookie.mcs.anl.gov</td>
</tr>
<tr>
<td>8.68 OMP Work Time(sec)</td>
<td>2.09 omp parallel @ft.cftf2_801</td>
<td>0.00 Process 0</td>
</tr>
<tr>
<td>0.00 OMP Barrier Time(sec)</td>
<td>2.07 omp parallel @ft.cftf3_846</td>
<td>1.05 Thread 0</td>
</tr>
<tr>
<td>0.52 Global Stalls</td>
<td>0.00 omp parallel @ft.checksum_876</td>
<td>1.05 Thread 1</td>
</tr>
<tr>
<td>0.10 Floating Point Inefficiency</td>
<td>0.09 omp parallel @ft.compute_indexmax</td>
<td></td>
</tr>
<tr>
<td>0.66 Memory Stalls</td>
<td>1.09 omp parallel @ft.compute_initial_c</td>
<td></td>
</tr>
<tr>
<td>0.50 Cycles per Instruction</td>
<td>1.04 omp parallel @ft.evolve_226</td>
<td></td>
</tr>
<tr>
<td>2.54 Cycles per Floating Point Instruction</td>
<td>0.23 omp parallel @ft.init_ui_191</td>
<td></td>
</tr>
<tr>
<td>1.11 Instructions per Cycle</td>
<td>0.00 omp parallel @print_results.fprint_re</td>
<td></td>
</tr>
<tr>
<td>0.39 Floating Point Instructions per Cycle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- Blue: Lower values signify better performance.
- Red: Higher values indicate areas for improvement.

**Image Notes:**
- The diagram shows a performance visualization tool, CUBE: rt.A.cube, which displays various performance metrics such as time, work time, and instructions per cycle.
- The system tree visualizes the execution path and the parallel regions in the program.
- The metrics include total time, OMP time, OMP work time, barrier time, global stalls, floating point inefficiency, memory stalls, cycles per instruction, and instructions per cycle.

---

*Argonne National Laboratory* 
*CS@UH*
Conclusions