Task Analysis with Score-P

27.06.2012 | Daniel Lorenz  
Jülich Supercomputing Centre  
CScADS Tools Workshop  
Snowbird, Utah
Overview

- Introduction to Score-P
- Task related performance issues
- Reconciling tasking with existing techniques
- Example analysis
- Future work
Score-P
Introduction to Score-P

- Common tools infrastructure:
  - Starting with Scalasca, Periscope, TAU, and Vampir
  - Open for other tools and groups
- SILC and LMAC projects funded by BMBF, Germany
  - Scalable Infrastructure for Automatic Performance Analysis of Parallel Codes (SILC)
  - Performance dynamics of massively parallel codes (LMAC)
- PRIMA project funded by DOE, US
  - Performance Refactoring of Instrumentation, Measurement, and Analysis Technologies for Petascale Computing
Score-P functionality

- Fundamental tool concepts:
  - Instrumentation (various methods), later sampling
  - Event trace recording
  - Profile generation
  - Online access to profiling data and execution control

- Parallelization methods:
  - MPI
  - OpenMP 3.0
  - Hybrid parallelism (and serial)

- More functionality in the future (Cuda, OmpSs, HMPP, Pthreads, …)

- Analysis tools kept separate on top of Score-P components
Score-P architecture

- **Target application**: MPI, OpenMP, hybrid, serial
- **Instrumentation**:
  - Call-path profiles (CUBE4 and TAU formats)
  - Event traces (OTF2 format)
- **Online interface**
- **Score-P measurement infrastructure**:
  - Hardware counter
  - Memory management
  - More ...
- **Instrumentation**:
  - Compiler
  - TAU instrumentor
  - OPARI 2
  - User
  - Binary instrumentor
  - MPI wrapper
- Tools:
  - Vampir
  - Scalasca
  - TAU
  - Periscope
Score-P availability

- Current release is version 1.0.2
  - New BSD license
  - The task profiling features of this presentation will be in the Score-P 1.1 release

- Download: http://www.score-p.org
Goals of the tasking support

- Analysis of task related performance issues
  - Task granularity
  - Task dependency analysis (under development in Scalasca)

- Reconcile existing techniques with tasking
  - No continuous instruction stream per thread anymore
  - Additional level of parallelism and code structure need to be represented

- Generic event model, used for multiple tasking systems
  - Currently, implementation for OpenMP tied tasks
  - OmpSs and HMPP support under development
Task related performance issues
Task granularity

- Tasking offers automated load balancing
  - But introduces task management overhead

- Tasks may be too small
  - The management overhead may cause performance loss
  - Task creation may become a bottleneck
  - Only a fraction of tasks may be too small
    - Especially when using recursive task creation structures
  - Identify problematic tasks
    - For recursive tasks: Where is the best cut-off?

- Tasks may be too large and too few
  - Reduction of the load balancing effects
  - Similar effects may happen with few, long dependency chains
What data shall we measure?

- We want to measure
  - runtime of tasks
  - task creation time and management overhead
  - Number of tasks
- Only a fraction of tasks may have performance issues
  - In the total sum, the effects might be leveled by other tasks
  - Additional statistical information (min, max, median, mean) might help recognizing an issue
How to identify problematic tasks

Provide possibilities to group tasks

- by constructs
- depending on certain parameters (e.g. recursion depth)
Reconcile tasking with existing techniques
Reconcile tasking with existing techniques

No continuous instruction stream per thread anymore

- Distinguish the event stream of each task
- Need to identify task instances
- Track task switches
- For OpenMP tied tasks, we can insert necessary instrumentation
Task data representation

- Additional level of parallelism and code structure

- For Scalasca/Score-P we want to integrate tasks into Cube call trees

- Where shall we place tasks in the call tree?
Display tasks in a Cube4 profile (1)

- Require that the inclusive time is the subtree’s sum of exclusive times
- Tasks must appear at execution point in the tree of the implicit task
  - Correct metric attribution
  - Other position may lead to
    - Negative times for exclusive execution time (and other metrics)
    - Appearance of false idle times at synchronization points

At execution point

```
1 !$ omp parallel region
  1 create task 1
  1 barrier
  6 task 1
```

At creation point

```
1 !$ omp parallel region
  -5 create task 1
  6 task 1
  7 barrier
```
Display tasks in a Cube4 profile (2)

- All tasks appear as children of the implicit task
- If tasks appear as children in other explicit tasks:
  - *Random execution order leads to incomparable call-tree structure*
  - *Call-tree may become extremely deep*
  - *You might end up with separate node for every task instance*
  - *Could lead to inconsistent call tree*

```
<table>
<thead>
<tr>
<th>Barrier</th>
<th>Task 1</th>
<th>Taskwait</th>
<th>Task 2</th>
<th>Taskyield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter barrier</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start task 1 =&gt; enter task 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter taskwait</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start task 2 =&gt; enter task2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter taskyield</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resume Task 1 =&gt; exit taskwait</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Display tasks in a Cube4 profile (3)

- A task may be suspended and resumed at another scheduling point
  - How do we count undividable metrics, e.g. visits?
    - Similar problem for min, max, sum of squares
  - First event of the resumed task is an exit event
  - We would need to copy the whole call stack of the task

- Solution
  - Leave stub node for task execution at execution point
  - Put task’s inner structure in a separate tree beside the implicit task
Call-tree example (main)

- 0.01053 main
- 0.00001 !$omp parallel @task_fibonacci.c:78
  - 0.00001 !$omp single @task_fibonacci.c:80
    - 0.00000 !$omp single sblock @task_fibonacci.c:80
      - 0.00001 fib
        - 0.00001 !$omp create task @task_fibonacci.c:48
        - 0.00000 !$omp create task @task_fibonacci.c:53
        - 0.10205 !$omp taskwait @task_fibonacci.c:58
          - 0.00374 !$omp task @task_fibonacci.c:53
          - 0.00481 !$omp task @task_fibonacci.c:48
    - 0.08185 !$omp implicit barrier
      - 0.15112 !$omp task @task_fibonacci.c:48
      - 0.06680 !$omp task @task_fibonacci.c:53
Call-tree example (tasks)
Analysis example
nqueens

- Code of the Barcelona OpenMP Tasking Suite (BOTS)
- Calculate the possibilities to place n queens on an nxn chess board
- BOTS provide multiple versions of the code
  - Analyze the version without cut-off
  - There is also an optimized version with a cut-off
- Runs performed on Juropa using a GNU compiler
Speedup of nqueens without cut-off (s)

Number of threads

- 0
- 0.2
- 0.4
- 0.6
- 0.8
- 1
- 1.2

- no cut-off
Profile comparison (execution time)

Profile of a run with one thread

Profile of a run with four threads

Sum of execution time of user code over all threads stays nearly the same
Profile comparison (execution time)

Profile of a run with one thread

- 108.18 \texttt{!omp task 2}
  - 56.11 \texttt{!omp create task 2}
  - 2.47 \texttt{!omp taskwait}
- 0.00 main
- 0.00 \texttt{!omp parallel}
  - 0.00 \texttt{!omp implicit barrier}
  - 0.00 \texttt{!omp atomic}
  - 0.00 \texttt{!omp single}
  - 47.68 \texttt{!omp taskwait}
    - 166.75 \texttt{!omp task 2}
    - 0.00 \texttt{!omp create task 2}

Profile of a run with four threads

- 112.43 \texttt{!omp task 2}
  - 322.47 \texttt{!omp create task 2}
  - 24.93 \texttt{!omp taskwait}
- 0.00 main
- 0.00 \texttt{!omp parallel}
  - 0.00 \texttt{!omp implicit barrier}
  - 0.00 \texttt{!omp atomic}
  - 0.00 \texttt{!omp single}
  - 186.80 \texttt{!omp implicit barrier}
    - 352.37 \texttt{!omp task 2}
    - 0.00 \texttt{!omp single}
    - 72.27 \texttt{!omp taskwait}
      - 107.46 \texttt{!omp task 2}
      - 0.00 \texttt{!omp create task 2}

Additional time due to management overhead
## Tasks by recursion level

<table>
<thead>
<tr>
<th>Depth level</th>
<th>Mean time</th>
<th>Sum</th>
<th>Number of tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>23.6 µs</td>
<td>0.0003 s</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>17.4 µs</td>
<td>0.0034 s</td>
<td>196</td>
</tr>
<tr>
<td>2</td>
<td>13.4 µs</td>
<td>0.0293 s</td>
<td>~2,000</td>
</tr>
<tr>
<td>3</td>
<td>10.6 µs</td>
<td>0.2019 s</td>
<td>~19,000</td>
</tr>
<tr>
<td>4</td>
<td>8.05 µs</td>
<td>1.086 s</td>
<td>~135,000</td>
</tr>
<tr>
<td>5</td>
<td>5.97 µs</td>
<td>4.520 s</td>
<td>~750,000</td>
</tr>
<tr>
<td>6</td>
<td>4.23 µs</td>
<td>14.31 s</td>
<td>~3,400,000</td>
</tr>
<tr>
<td>7</td>
<td>2.93 µs</td>
<td>34.25 s</td>
<td>~11,700,000</td>
</tr>
<tr>
<td>8</td>
<td>1.98 µs</td>
<td>61.56 s</td>
<td>~31,000,000</td>
</tr>
<tr>
<td>9</td>
<td>1.35 µs</td>
<td>83.01 s</td>
<td>~61,000,000</td>
</tr>
<tr>
<td>10</td>
<td>0.94 µs</td>
<td>83.48 s</td>
<td>~89,000,000</td>
</tr>
<tr>
<td>11</td>
<td>0.69 µs</td>
<td>62.42 s</td>
<td>~91,000,000</td>
</tr>
<tr>
<td>12</td>
<td>0.51 µs</td>
<td>32.26 s</td>
<td>~63,000,000</td>
</tr>
<tr>
<td>13</td>
<td>0.26 µs</td>
<td>7.145 s</td>
<td>~27,000,000</td>
</tr>
</tbody>
</table>
Tasks by recursion level  

Mean task creation time approx. 0.85 µs

<table>
<thead>
<tr>
<th>Depth level</th>
<th>Mean time</th>
<th>Sum</th>
<th>Number of tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>23.6 µs</td>
<td>0.0003 s</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>17.4 µs</td>
<td>0.0034 s</td>
<td>196</td>
</tr>
<tr>
<td>&gt;5 % Overhead</td>
<td>2</td>
<td>13.4 µs</td>
<td>~ 2,000</td>
</tr>
<tr>
<td>&gt;10 % Overhead</td>
<td>4</td>
<td>8.05 µs</td>
<td>~135,000</td>
</tr>
<tr>
<td>&gt;20 % Overhead</td>
<td>6</td>
<td>4.23 µs</td>
<td>~3,400,000</td>
</tr>
<tr>
<td>&gt;100 % Overhead</td>
<td>11</td>
<td>0.69 µs</td>
<td>~91,000,000</td>
</tr>
<tr>
<td>7</td>
<td>2.93 µs</td>
<td>34.25 s</td>
<td>~11,700,000</td>
</tr>
<tr>
<td>8</td>
<td>1.98 µs</td>
<td>61.56 s</td>
<td>~31,000,000</td>
</tr>
<tr>
<td>9</td>
<td>1.35 µs</td>
<td>83.01 s</td>
<td>~61,000,000</td>
</tr>
<tr>
<td>10</td>
<td>0.94 µs</td>
<td>83.48 s</td>
<td>~89,000,000</td>
</tr>
<tr>
<td>11</td>
<td>0.69 µs</td>
<td>62.42 s</td>
<td>~91,000,000</td>
</tr>
<tr>
<td>12</td>
<td>0.51 µs</td>
<td>32.26 s</td>
<td>~63,000,000</td>
</tr>
<tr>
<td>13</td>
<td>0.26 µs</td>
<td>7.145 s</td>
<td>~27,000,000</td>
</tr>
<tr>
<td>Depth level</td>
<td>Mean time</td>
<td>Sum</td>
<td>Number of tasks</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>0</td>
<td>23.6 µs</td>
<td>0.0003 s</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>17.4 µs</td>
<td>0.0034 s</td>
<td>196</td>
</tr>
<tr>
<td>&gt;5 % Overhead</td>
<td>13.4 µs</td>
<td>0.0293 s</td>
<td>~2,000</td>
</tr>
<tr>
<td>&gt;10 % Overhead</td>
<td>8.05 µs</td>
<td>1.086 s</td>
<td>~135,000</td>
</tr>
<tr>
<td>3</td>
<td>10.6 µs</td>
<td>0.2019 s</td>
<td>~19,000</td>
</tr>
<tr>
<td>4</td>
<td>5.97 µs</td>
<td>4.520 s</td>
<td>~750,000</td>
</tr>
<tr>
<td>5</td>
<td>4.23 µs</td>
<td>14.31 s</td>
<td>~3,400,000</td>
</tr>
</tbody>
</table>

- Let us target less than 5% management overhead per task
- 210 tasks may be too little for proper load balancing
- Upper levels do not contribute significant amount of execution time
- Tasks in last level will grow due to merge with children
- Compromise: Cut-off at level 3
Resulting speedup

Number of threads
Future Work

- Currently, only OpenMP tied tasks are supported
  - Ongoing work on HMPP and OmpSs support
  - Hopefully, a new OpenMP tools interface provides necessary information to support untied tasks, too

- Trace analysis of tasks with Scalasca
  - Extend for additional patterns
  - Task dependency analysis