A New Approach for Performance Analysis of OpenMP Programs

Xu Liu, John Mellor-Crummey, Mike Fagan
Rice University

http://hpctoolkit.org
Motivation

• Multicore is used everywhere
  – cell phones
  – laptops
  – supercomputers

• Threads per node grow rapidly
  – IBM Blue Gene/Q: 64 threads
  – IBM Power 7: 128 threads
  – Intel MIC: ~200 threads

• Taking advantage of massive threads is important
Programming model for multi-threading

- **Low level**
  - `pthread`

- **High level**
  - language based: Cilk
  - library based: Intel thread building block (TBB)
  - compiler based: OpenMP
Programming model for multi-threading

- **Low level**
  - pthread

- **High level**
  - language based: Cilk
  - library based: Intel thread building block (TBB)
  - compiler based: OpenMP
    - standardized
    - portable
Programming model for multi-threading

- Low level
  - pthread

- High level
  - language based: Cilk
  - library based: Intel thread building block (TBB)
  - compiler based: OpenMP
    - standardized
    - portable

OpenMP is the most popular programming model for multi-threading
OpenMP’s fork-join parallelism

parallel region

parallel region

parallel region
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance

Performance tool is needed
Previous work

- Instrumentation based tools: TAU, Scalasca
  - high measurement overhead
- Sampling based tools: Intel Vtune
  - no specific OpenMP support
- Hybrid tools: Sun Studio/Oracle Analyzer API for OpenMP
  - support low-overhead, sampling-based measurement
  - use large disk space for measurement data
  - insufficient support for statically-linked applications
  - insufficient mechanisms to blame root causes of performance losses
    - attributes waiting to “symptoms” rather than “causes”
Previous work

- Instrumentation based tools: TAU, Scalasca
  - high measurement overhead
- Sampling based tools: Intel Vtune
  - no specific OpenMP support
- Hybrid tools: Sun Studio/Oracle Analyzer API for OpenMP
  - support low-overhead, sampling-based measurement
  - use large disk space for measurement data
  - insufficient support for statically-linked applications
  - insufficient mechanisms to blame root causes of performance losses
    - attributes waiting to “symptoms” rather than “causes”

Goal: build a tool to overcome all the shortcomings in existing tools
Our approach

- A hybrid performance tool for OpenMP programs
  - quantify performance losses
  - pinpoint problematic code for optimization
  - low measurement overhead
  - support both dynamically and statically linked binaries

- With the feedback from our tool, we were able to get 1.1x-5.7x speedup for four well-known benchmarks
Overview of our tool

- **OpenMP tools API**
  - minimum set of support to tools
  - low overhead

- **HPCToolkit**
  - a leading sampling-based performance tool for parallel programs
  - call path profiler + static binary analyzer + GUI
  - on top of OpenMP tools API
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks
OpenMP tools API

- Maintain states
- Provide callbacks

```
Fork
Join
Fork
Join
Fork
Join
Fork
Join
```

- entrance callback
- exit callback
- state callback

- working
- idle
- overhead
OpenMP tools API

- Maintain states
- Provide callbacks

- Low overhead
- Try to make it into OpenMP standard
Enhancement of HPCToolkit

- Measure and attribute costs to full user-program calling contexts
  - unified view of calling contexts across all threads
- Shift blame for idleness from symptoms to causes
- Support both OpenMP profiling and tracing
Problem: separate views for different threads

Worker threads don’t know the full user-level context for work
Problem: separate views for different threads

Worker threads don’t know the full user-level context for work
Call stack snapshot in OpenMP threads

- Regions in gray have distributed calling contexts.
Call stack snapshot in OpenMP threads

regions in gray have distributed calling contexts
Call stack snapshot in OpenMP threads

regions in gray have distributed calling contexts
Call stack snapshot in OpenMP threads

regions in gray have distributed calling contexts
Call stack snapshot in OpenMP threads

regions in gray have distributed calling contexts
Call stack snapshot in OpenMP threads

regions in gray have distributed calling contexts
Call stack snapshot in OpenMP threads

Regions in gray have distributed calling contexts.

Too tiny.
Call stack snapshot in OpenMP threads

Online deferred context resolution

regions in gray have distributed calling contexts

too tiny
Results of deferred context construction

main._omp_fn.*: outlined functions correspond to <parallel region>
Enhancement of HPCToolkit

- Measure and attribute costs to full user-program calling contexts
- Shift blame for idleness from symptoms to causes
  - undirected blame: load imbalance, serialization
  - directed blame: lock and critical section contention
- Support both OpenMP profiling and tracing
Problem: meaningless hotspots

hotspot is `do_wait`, but don’t know why
Undirected blame

- *do_wait* is the symptom
- Causes are working threads
  - e.g. last thread has more work than other threads
- Blaming *do_wait* to working threads
Undirected blame

- *do_wait* is the symptom
- Causes are working threads
  - e.g. last thread has more work than other threads
- Blaming *do_wait* to working threads
Undirected blame

- *do_wait* is the symptom
- Causes are working threads
  - e.g. last thread has more work than other threads
- Blaming *do_wait* to working threads

```plaintext
Fork

               do_wait
               do_wait
               do_wait

Joint

2 idle threads
3 working threads
```
Undirected blame

- *do_wait* is the symptom
- Causes are working threads
  - e.g. last thread has more work than other threads
- Blaming *do_wait* to working threads
Code-centric view: \texttt{hypre\_BoomerAMGRelax}

20\% idleness and 80\% work in this parallel region
Serial Code in AMG2006 8 PE, 8 Threads

7/8 threads are idle: sequential code
LULESH running on 48 cores

Free in both LULESH benchmark and libgomp causes high idleness.

4x speedup after changing to Google’s tcmalloc.
Directed blame

- Thread waiting at lock is the symptom
- Cause is the lock holder
- Blame lock waiting to lock holder

Fork → lockwait → Join

- acquire lock
- release lock
Directed blame

- Thread waiting at lock is the symptom
- Cause is the lock holder
- Blame lock waiting to lock holder

```
F o r k

-lockwait-

J o i n
```

accumulate samples indexed by the lock address

acquire lock
release lock
Directed blame

- Thread waiting at lock is the symptom
- Cause is the lock holder
- Blame lock waiting to lock holder

Fork

lockwait

Join

acquire lock
release lock

accumulate samples indexed by the lock address

lock holder takes these samples at lock release
Directed blame

- Thread waiting at lock is the symptom
- Cause is the lock holder
- Blame lock waiting to lock holder

```
<table>
<thead>
<tr>
<th>Fork</th>
<th>lockwait</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire lock</td>
<td>lock holder takes these samples at lock release</td>
<td></td>
</tr>
<tr>
<td>release lock</td>
<td>accumulate samples indexed by the lock address</td>
<td></td>
</tr>
</tbody>
</table>
```
Lock blame shifting for NAS UA

- Lots of locks
- 8.4% of execution time waiting for locks
- 34% of lock waiting due to locks acquired at highlighted call site
Reducing lock contention for NAS UA

- Use `omp_test_lock`
- Defer the lock acquisition to the next iteration
- Eliminate most lock contention time
Enhancement of HPCToolkit

- Measure and attribute costs to full user-program calling contexts
- Shift blame for idleness from symptoms to causes
- Support both OpenMP profiling and tracing
Tracing: AMG 2006 (solver phase)
Tool performance

- Measurement overhead
  - OpenMP tools API
  - HPCToolkit scalability
- Insightful optimization feedbacks
Benchmarks

- Four case studies
  - AMG2006
    - one of Sequoia benchmarks from LLNL
    - running on 48 threads
  - LULESH
    - an application benchmark from LLNL
    - uses work-sharing parallel regions without nesting and tasking
    - 48 threads
  - BT-MZ.B
    - BT in multi-zone NPB with workload B
    - uses nested parallel regions without tasking
    - 32 threads: 4 for outer region and 8 for inner region
  - HEALTH
    - a benchmark in Barcelona tasking benchmarks
    - uses tasking: more than 17 million tasks
    - 8 threads, using medium input
Profiling overhead

- OpenMP tools API implemented in GNU OpenMP (GOMP)
- HPCToolkit takes 200 samples/s/thread

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>GOMP</th>
<th>GOMP + tools API</th>
<th>GOMP+tools API + profiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2006</td>
<td>54.02s</td>
<td>+0.15%</td>
<td>+5.0%</td>
</tr>
<tr>
<td>LULESH</td>
<td>402.34s</td>
<td>+0.05%</td>
<td>+3.6%</td>
</tr>
<tr>
<td>NAS BT-MZ</td>
<td>32.10s</td>
<td>+0.16%</td>
<td>+6.6%</td>
</tr>
<tr>
<td>HEALTH</td>
<td>71.74s</td>
<td>+0.64%</td>
<td>+3.5%</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>Problem</td>
<td>Optimization</td>
<td>Improvement</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>AMG2006</td>
<td>high idleness in parallel regions</td>
<td>use dynamic scheduling for load balance</td>
<td>1.12x</td>
</tr>
<tr>
<td>LULESH</td>
<td>serialization in <em>free</em></td>
<td>switch to tcmalloc</td>
<td>3.93x</td>
</tr>
<tr>
<td>NAS BT-MZ</td>
<td>parallelize insignificant work</td>
<td>eliminate necessary parallelism</td>
<td>1.09x</td>
</tr>
<tr>
<td>HEALTH</td>
<td>high lock contention for task queue</td>
<td>coarsen task granularity</td>
<td>5.65x</td>
</tr>
</tbody>
</table>
Summary

- **OpenMP tools API -- we are involved in designing it**
  - incurs very little runtime overhead
  - supports efficient sampling-based measurement tools
  - supports root cause analysis of performance bottlenecks
  - try to make it into OpenMP standard

- **HPCToolkit -- we developed it**
  - provides low-overhead profiling and tracing
  - supports all OpenMP features
  - delivers insights into performance of OpenMP codes

- **Now enhanced HPCToolkit works on IBM Blue Gene/Q**