Characterizing and Improving the Performance of Intel Threading Building Blocks

Gilberto Contreras, Margaret Martonosi
Princeton University
Motivation

- Chip Multiprocessors are the new computing platform.
  - 2 cores, 4 cores, 8 cores... Are we ready?
- Why is parallelism so challenging?

- Identify parallelism
- Annotation/ extraction of parallelism
- Mapping to cores

- Respond to:
  - OS effects
  - Thermal emergencies
  - Variability trends
  - Reliability issues
This work answers the following questions:

- What are some of the major sources of overheads?
- How do they impact overall parallelism performance?
- How can we improve parallelism performance?
Our work focus

• This talk will focus on the Intel Threading Building Blocks (TBB)
  - Task-based parallelization library for C++ applications
  - Support a wide range of parallelism types
  - Utilizes task stealing for load balancing

Methodology is applicable to other parallelism management approaches
Presentation Outline

• Description of TBB
  – Programming example
  – Task management in TBB

• Characterization Methodology
  – Measuring basic operations using simulation and real-system measurements
  – TBB overheads in PARSEC benchmarks
  – Performance of Task Stealing

• Improving TBB
  – Occupancy-based task stealing

• Summary and Conclusions
for (i=k+1; i<size; i++) {
    L[i][k] = M[i][k] / M[k][k];
    for (j=i+1; j<size; j++)
        M[i][j] = M[i][j] -
                 L[i][k]*M[k][j];
}
Reducing TBB Library Overhead?

- **Understand Overheads**
  - Creating tasks
    - `spawn()`
  - Assigning tasks to worker threads
    - `get_task()`
    - `queue_acquire()`
    - `wait_for_all()`
  - Stealing or rebalancing parallelism
    - `steal()`

- **Improve parallelism reorganization policies**
  - Employ smart redistribution policies
  - Make this as fast and as efficient as possible
Methodology

Benchmarks
- PARSEC
- Microbenchmarks

Intel Threading Building Blocks (TBB)
- Open source 2.0 version

Real CMP System
- 4-core AMD system (2 processors)
- 4GB RAM
- Linux 2.6
- Oprofile is used for performance counter measurements

Cycle-accurate CMP simulator
- 2-issue, in-order cores
- 32KB D$ (coherent), 32K I$
- 8MB shared L2 cache
- MSI directory-based coherence protocol
- Mesh network, 32b BW/ port/ cycle
Cost of Parallelism Management

Simulation Results (4-32 cores)

Cycles

Get Task  Spawn  Stealing (Successful)  Stealing (Unsuccessful)  Acquire Queue  Wait For All

Runtime activity

- 4 cores  8 cores  12 cores  16 cores  32 cores
TBB Overheads: PARSEC

**fluidanimate**

- **Average time per core**
  - P8: 0%
  - P12: 5%
  - P16: 10%
  - P25: 15%
  - P32: 20%

**swaptions**

- **Average time per core**
  - P8: 0%
  - P12: 5%
  - P16: 10%
  - P25: 15%
  - P32: 20%

**blackscholes**

- **Average time per core**
  - P8: 0%
  - P12: 5%
  - P16: 10%
  - P25: 15%
  - P32: 20%

**streamcluster**

- **Average time per core**
  - P8: 0%
  - P12: 5%
  - P16: 10%
  - P25: 15%
  - P32: 20%

**Synchronization**
- 41% 47%
- 34% 54%
Improving Stealing

- TBB utilizes random stealing as its victim selection policy
Occupancy-based Stealing

Random Stealing

Occupancy-based stealing

- Random stealing:
  - Random number
  - Stealing

- Occupancy stealing:
  - Scanning
  - Stealing
Performance of Occupancy-based Stealing

<table>
<thead>
<tr>
<th></th>
<th>Occupancy-Based</th>
<th>1-cycle scan</th>
<th>1-cycle scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal stealing</td>
<td>1-cycle stealing</td>
<td></td>
</tr>
<tr>
<td>Bitcounter</td>
<td>2.5%</td>
<td>2.7%</td>
<td>3.7%</td>
</tr>
<tr>
<td>LU</td>
<td>10%</td>
<td>9.7%</td>
<td>8.0%</td>
</tr>
<tr>
<td>Matmult</td>
<td>9.5%</td>
<td>19%</td>
<td>21.1%</td>
</tr>
</tbody>
</table>

- Smarter selection policies are desired
- High potential in overhead reduction
Conclusions

• Increasing usage of TBB makes it a prime candidate for in-depth characterization

• Parallelization libraries help, but tend to exhibit high (dynamic) overheads (>40% at 32 cores)

• Understanding software overheads is the first step in creating high-performance parallel systems

• We have presented a detailed characterization of the Intel Threading building Blocks and implemented occupancy-based stealing (19% performance over random stealing).
Thanks!
Summary

• Programmers require tools that allows them to take (fast) advantage of increasing core counts.

• Parallelization libraries help, but tend to exhibit high (dynamic) overheads (>40% at 32 cores)

• Understanding software overheads is the first step in creating high-performance parallel systems

• We have presented a detailed characterization of the Intel Threading building Blocks and implemented occupancy-based stealing (19% performance over random stealing).
Cost of Parallelism Management

- 4 core, 1.8GHz AMD system
- Oprofile configured to measure CPU_CLK_UNHALTED

- 1 to 32 core CMP simulator
- 2-issue, in-order cores
- Shared L2

Our goals:
1) Reduce per-event overheads
2) Improve rebalancing
Static versus Dynamic Management

PARSEC

- **fluidanimate**
- **swaptions**
- **blackscholes**

Graphs showing speedup versus number of cores for different benchmarks, comparing Static (pthread) and TBB approaches.