DWS: Demand-aware Work-Stealing in Multi-programmed Multi-core Architectures

Quan Chen, Long Zheng, Minyi Guo

Shanghai Jiao Tong University, China

PMAM 2014
Outline

• Background
• Problem & Motivation
• Demand-aware Work-Stealing (DWS)
• Evaluation
• Conclusions
Background

- Hardware: Multi-core/Many-core Architectures
- Scenario: Multiple parallel programs
Background-parallel programs

- Traditional parallel programs
  - **Hard** to adjust the number of threads at runtime

- Task-based parallel programs
  - Dynamic task scheduling

![Diagram showing work-sharing and work-stealing]

**Easy to adjust the number of workers**
Lock the central task pool when getting a task
Work-stealing
Problem & Motivation

- Aggressive feature of work-stealing
  - On a $k$-core computer, $k$ threads/workers are launched

- Existing solutions
  - Time-sharing - ABP yielding mechanism
  - Space-sharing - Equal-partitioning
Time-sharing

ABP yielding mechanism

- If a thread fails to steal a task, it goes to sleep

Active
- Thread 3
- Thread 2
- Thread 1

Sleep

Unfair resource allocation

Poor data locality
Space-sharing

- Equal-partitioning mechanism
  - If $m$ programs co-run on a $k$-core computer, each program is allocated $k/m$ cores.

Fair but inefficient

![Diagram of space-sharing](image)
Demand-aware Work-Stealing (DWS)

- Start from Equal-partitioning
- Dynamically balance cores at runtime
  - If $p_i$ cannot fully-utilized a core, it release the core
  - If $p_i$ has too many tasks, it tries to obtain more cores
A worker decides whether to release its core by itself

```
Algorithm 1: Work-stealing algorithm in DWS

Input: w, current worker

1  int failed_steals = 0; // num of failed steals
2  while work is not done do
3      if w is free then
4          if its task pool is not empty then
5              w obtains a task t from its own task pool;
6              failed_steals = 0;
7          else
8              w randomly selects v as victim worker;
9              if v has a non-empty task pool then
10                 w steals t from v;
11                 failed_steals = 0;
12              else
13                 failed_steals ++;
14                 if failed_steals > T_SLEEP then
15                     w goes to sleep;
16                     w waits to be woken up;
17                 end if
18          end if
19      end if
20      if t then
21         w executes t;
22      end if
23  end while
```

If a worker fails too many times (T_SLEEP) to steal a new task, it goes to sleep.
The coordinator decides whether to obtain more cores

- If a program has too many queued tasks, it should try to get some free cores

C1: The more queued tasks in a program, the more cores should the program obtain

C2: A program can take its allocated cores back

C3: A program cannot obtain the busy cores
C1: The more queued tasks in a program, the more cores should the program obtain.

How many: \[ N_w = \frac{N_b}{N_a} \]

<table>
<thead>
<tr>
<th>Num of active workers</th>
<th>( N_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of queued tasks</td>
<td>( N_b )</td>
</tr>
<tr>
<td>Num of free cores</td>
<td>( N_f )</td>
</tr>
<tr>
<td>Num of released cores</td>
<td>( N_r )</td>
</tr>
<tr>
<td>Num of cores expected</td>
<td>( N_w )</td>
</tr>
</tbody>
</table>
Coordinator - Which?

1. $N_w \leq N_f$
   - Randomly select $N_w$ free cores

2. $N_f < N_w \leq N_f + N_r$ \hspace{1cm} (C2)
   - Select $N_f$ free cores + its $(N_w - N_f)$ released core

3. $N_w > N_f + N_r$ \hspace{1cm} (C3)
   - $N_f$ free cores + its $N_r$ released cores

<table>
<thead>
<tr>
<th>Num of active workers</th>
<th>$N_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of queued tasks</td>
<td>$N_b$</td>
</tr>
<tr>
<td>Num of free cores</td>
<td>$N_f$</td>
</tr>
<tr>
<td>Num of released cores</td>
<td>$N_r$</td>
</tr>
<tr>
<td>Num of cores expected</td>
<td>$N_w$</td>
</tr>
</tbody>
</table>
Evaluation platform

- A Dual-socket Quad-core computer with Hyper-Threading Technology
- Each socket is a Quad-Core Intel Xeon E5620

<table>
<thead>
<tr>
<th>Hardware &amp; Configuration</th>
<th>Size/Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1/L2 cache size (each core)</td>
<td>256 KB/1MB</td>
</tr>
<tr>
<td>L3 cache size (each socket)</td>
<td>12 MB</td>
</tr>
<tr>
<td>Main memory size</td>
<td>32 GB</td>
</tr>
<tr>
<td>Operation system</td>
<td>Linux 2.6.32-38</td>
</tr>
</tbody>
</table>
Benchmarks

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-1</td>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>p-2</td>
<td>PNN</td>
<td>Polynomial Neural Network</td>
</tr>
<tr>
<td>p-3</td>
<td>Cholesky</td>
<td>Cholesky decomposition</td>
</tr>
<tr>
<td>p-4</td>
<td>LU</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>p-5</td>
<td>GE</td>
<td>Gaussian Elimination algorithm</td>
</tr>
<tr>
<td>p-6</td>
<td>Heat</td>
<td>Five-point heat distribution</td>
</tr>
<tr>
<td>p-7</td>
<td>SOR</td>
<td>2D Successive Over-Relaxation</td>
</tr>
<tr>
<td>p-8</td>
<td>Mergesort</td>
<td>Merge sort on 4E6 numbers</td>
</tr>
</tbody>
</table>

Calculate execution time:

\[ T_i = \frac{\sum_{r=1}^{a} t_{ir}}{a}, \quad T_j = \frac{\sum_{r=1}^{b} t_{jr}}{b} \]
Performance of DWS

DWS can significantly improve the performance of the benchmarks
Without the coordinator, the performance of the benchmarks is degraded
We should choose $T_{SLEEP} = k$ or $2k$ on a $k$-core computer.
Contributions & conclusions

• A modified work-stealing algorithm that enables a program to release the under-utilized cores.

• A coordinator to manage the workers. It enables a program to grab and use the under-utilized cores released by other programs.

• We have implemented DWS, which achieves a performance gain of up to 32.3% in the best cases compared to traditional work-stealing schedulers.
Thanks!

Questions?