Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers

Jonathan Lifflander*, Sriram Krishnamoorthy†, Laxmikant V. Kale*

{jliffl2, kale}@illinois.edu, sriram@pnnl.gov

*University of Illinois Urbana-Champaign
†Pacific Northwest National Laboratory

June 19, 2013
Motivation

- Structured parallel programming (e.g. async-finish) idioms have proliferated
  - Examples: OpenMP 3.0, Java Concurrency Utilities, Intel TBB, Cilk, X10

- Work stealing is often used to schedule them:
  - Well-studied dynamic load balancing strategy
  - Provably efficient scheduling
  - Understandable bounds on time and space
Tracing

- Where and when each task executed
- Captures the order of events and is effective for online and offline analysis
- Challenges
  - The size may limit what can be feasibly analyzed
  - It may perturb the application’s execution making it impractical
- Applications
  - Replay
  - Performance analysis
  - Data-race detection, retentive stealing, ...
Trace Sizes Using the Steal Tree
Approach

- For `async-finish` programs tracing individual tasks is not feasible
  - Often these programs expose far more concurrency than the number of threads
    - Fine granularity
    - Sheer number of tasks
- Rather than trace individual tasks, exploit the structure of the scheduler to coarsen the events traced
- We identify key properties of two scheduling policies:
  - Help-first: expose more concurrency by expanding tasks in the current scope before executing a task
  - Work-first: depth-first traversal of the code (Cilk)
Example async-finish Program

```javascript
fn() {
  s1;
  async {
    s5;
    async w;
    s6;
  }
  s2;
  finish {
    s7;
    async x;
    s8;
    async y;
    s9;
    async z;
    s10;
  }
  s3;
  async { s11; }
  s4;
}
```
Example `async-finish` Program

```
fn() {
  s1;
  async {
    s5;
    async w;
    s6;
  }
  s2;
  finish {
    s7;
    async x;
    s8;
    async y;
    s9;
    async z;
    s10;
  }
  s3;
  async {
    s11;
  }
  s4;
}
```
Enqueue all asyncs in the current **level** until a **finish** is reached.
Help-first Scheduling Policy

- Enqueue all asyncs in the current level until a finish is reached
Help-first Scheduling Policy

- Enqueue all asyncs in the current level until a finish is reached

---

**Example Program**

```javascript
fn() {
  s1;
  async {
    s5;
    async w;
    s6;
  }
  s2;
  finish {
    s7;
    async x;
    s8;
    async y;
    s9;
    async z;
    s10;
  }
  s3;
  async { s11; }
  s4;
}
```

**Snapshot of Execution**

1. `s1`, `s2`, `s3`
2. `s7`, `s8`, `s9`, `s10`
3. `s3`, `s7`, `s8`, `s9`, `s10`, `z`

**Deque**

- `steal end`
- `local end`
Theorem (5.8 in the paper):

The tasks executed and steal operations encountered in each working phase can be fully described by (a) the level of the root in the total ordering of the steal operations on the victim’s working phase, and (b) the number of tasks and step of the continuation stolen at each level.
Help-first Scheduling Policy

With levels and counters

Snapshot of Execution

Deque

Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers
Help-first Scheduling Policy

A steal occurs (annotated \( c2 \))

Snapshot of Execution

Deque

Steal end

local end
Help-first Scheduling Policy

Another steal occurs (annotated $c_2$); **Steal Tree** **before the steal**

### Snapshot of Execution

```
level

1
  s1
  s2
  s3
  c2

2
  s7
  s8
  s9
  s10

3
  Z
```

```
I : c
level  counter
```

### Deque

```
local end
steal end
```

### Steal Tree

```
c1 1:0 2:0
```
Help-first Scheduling Policy

Another steal occurs (annotated $c_2$); Steal Tree *after the steal*

### Snapshot of Execution

<table>
<thead>
<tr>
<th>Level</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1, s2, s3</td>
</tr>
<tr>
<td>2</td>
<td>s7, s8, s9, s10</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
</tr>
</tbody>
</table>

### Deque

- Steal end
- local end

### Steal Tree

- $c_1$: 1:1, 2:0
- $c_2$: 1:0, 1:1, 2:1, 2:2
Help-first Scheduling Policy

Another steal occurs (annotated \( c^3 \)); **Steal Tree** after the steal

---

**Snapshot of Execution**

- Level 1:
  - Task `s1`
  - Task `s2`
  - Task `s3` (annotated as \( c^3 \))

- Level 2:
  - Task `s7`
  - Task `s8`
  - Task `s9`
  - Task `s10`

- Level 3:
  - Task `Z`

**Deque**

- Task `s3`
- Task `c3`
- Task `c1`
- Task `c2`
- Task `c3`

**Steal Tree**

- Steal end
- Local end

---

Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers

Jonathan Lifflander

16 / 29
Help-first Scheduling Policy

Another steal occurs (annotated $c_4$); Steal Tree after the steal

Snapshot of Execution

Deque

Steal Tree

Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers
Theorem (6.3 in the paper):

- The tasks executed and steal operations encountered in each working phase can be fully described by (a) the level of the root in the total ordering of the steal operations on the victim’s working phase, and (b) the step of the continuation stolen at each level.
Implementation

- Shared-memory
  - Cilk (work-first)
  - Results on POWER7 (64 cores, 128 hyper-threaded)

- Distributed-memory
  - Work stealing using active messages
  - Implemented and evaluated for both work-first and help-first
  - Results on Titan at ORNL (Cray XK6)
## Empirical Results

### Shared and distributed memory

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllQueens</td>
<td>nq = 14, sequential cutoff 8</td>
</tr>
<tr>
<td>Heat</td>
<td>nt = 5, nx = 4096, ny = 4096</td>
</tr>
<tr>
<td>Fib</td>
<td>n = 43</td>
</tr>
<tr>
<td>FFT</td>
<td>n = 67108864</td>
</tr>
<tr>
<td>Strassen</td>
<td>n = 4096</td>
</tr>
<tr>
<td>NBody</td>
<td>iterations = 15, nbodies = 8192</td>
</tr>
<tr>
<td>Cholesky</td>
<td>n = 2048, z = 20000</td>
</tr>
<tr>
<td>LU</td>
<td>n = 1024</td>
</tr>
<tr>
<td>Matmul</td>
<td>n = 3000</td>
</tr>
</tbody>
</table>

### Distributed-memory

| AQ          | nq = 19, sequential cutoff 10                      |
| SCF         | 128 beryllium atoms, chunk size 40                  |
| TCE         | $C[i, j, k, l] + = A[i, j, a, b] \times B[a, b, k, l]$ |
| O-blocks    | 20 14 20 26, V-blocks 120 140 180 100               |
| PG          | 13K sequences                                       |
Execution Time Ratio

→ Shared-memory
Execution Time Ratio

Distributed-memory
Storage Overhead

→ Shared-memory
Storage Overhead

→ Distributed-memory
Utilization Graphs

Cilk LU

Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers
Applications

→ Two distinct contexts

- Data-race detection: *Scalable and precise dynamic datarace detection for structured parallelism* (Raman et al., [PLDI’12])
- Retentive stealing: *Work stealing and persistence-based load balancers for iterative overdecomposed applications* (Lifflander et al., [HPDC’12])
Data-race Detection

- As the program executes, the DPST (dynamic structure program tree) is built in parallel (see the PLDI’12 paper)
  - Used to determine the relationships between `async` and `finish` statements
  - The DPST is traversed to determine if two tasks *may execute in parallel*
  - The LCA (lowest common ancestor) must be found
  - This involves traversing up the DPST, until the common ancestor is found
- **Applying the STEAL TREE:**
  - Use the STEAL TREE to shorten the LCA traversal
DPST Traversal Percent Reduction

Using the STEAL TREE

Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers
Jonathan Lifflander
Concluding Remarks

- Framework for compactly tracing work stealing schedulers
- Applications
  - Performance analysis
  - Data-race detection
  - Retentive stealing