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

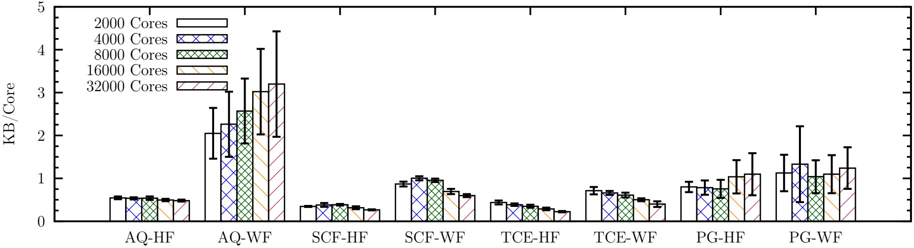
Motivation

- 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

```
 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

- ⊤ Root of Computation
- ● Sequential Block
- ◆ Asynchronous Task
- ⬢ Finish Scope
- ○ Continuation

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

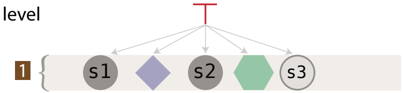
Help-first Scheduling Policy

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

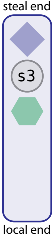
Example Program

```
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



Deque



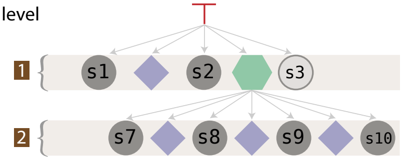
Help-first Scheduling Policy

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

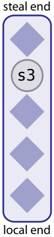
Example Program

```
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



Deque



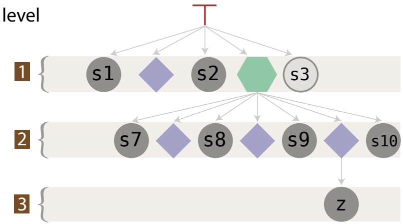
Help-first Scheduling Policy

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

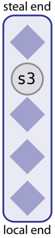
Example Program

```
⊤ 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



Deque



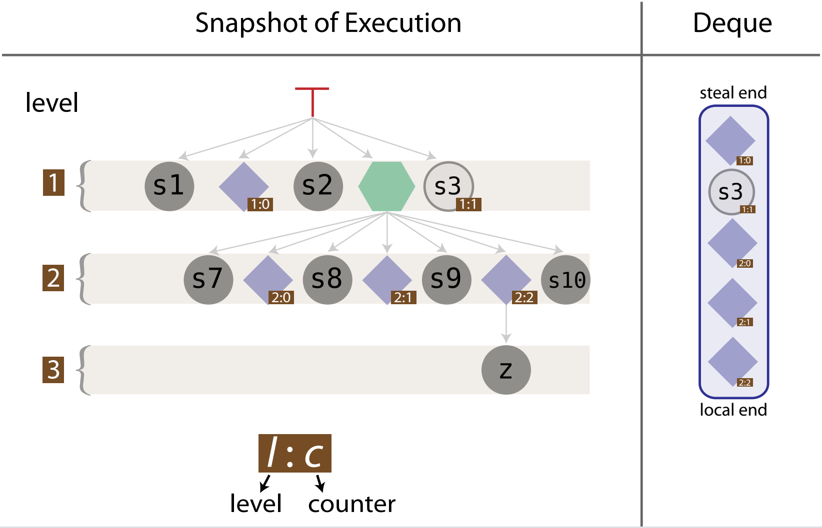
Help-first Scheduling Policy

→ Observation

- 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.*

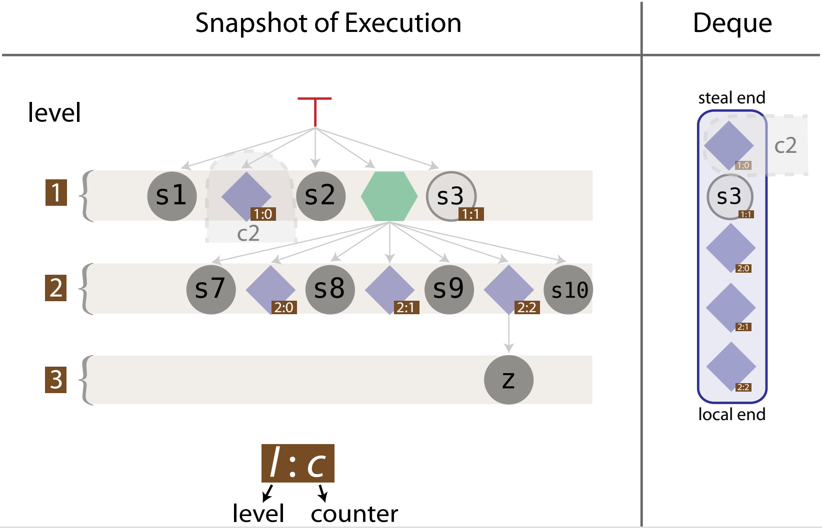
Help-first Scheduling Policy

→ With levels and counters



Help-first Scheduling Policy

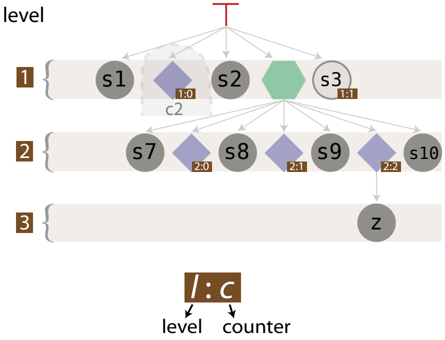
→ **A steal occurs (annotated $c2$)**

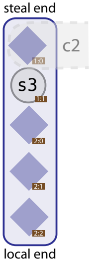


Help-first Scheduling Policy

→ **Another steal occurs (annotated $c2$); STEAL TREE *before* the steal**

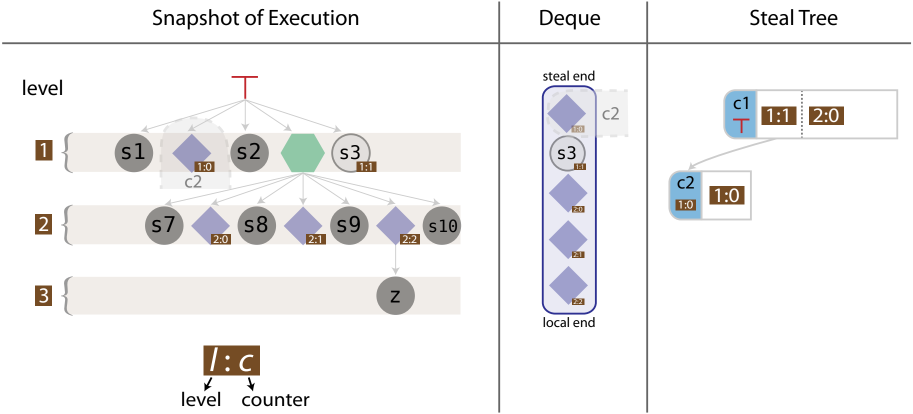
Snapshot of Execution	Deque	Steal Tree





Help-first Scheduling Policy

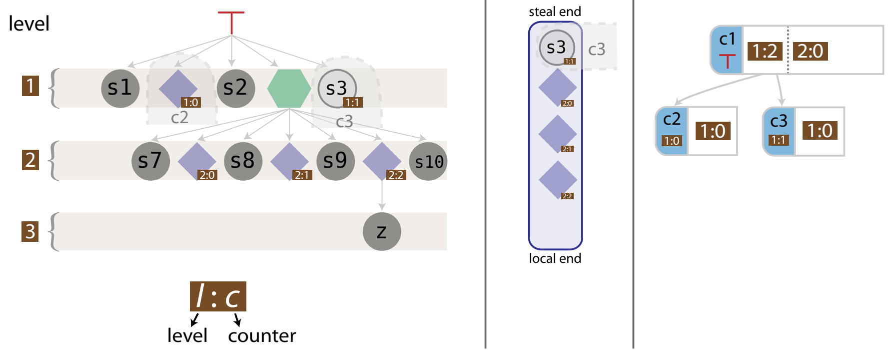
→ **Another steal occurs (annotated $c2$); STEAL TREE *after* the steal**



Help-first Scheduling Policy

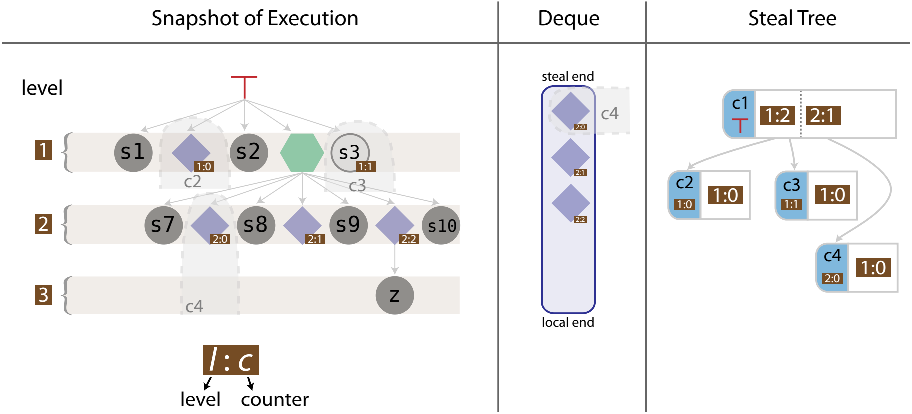
→ **Another steal occurs (annotated $c3$); Steal Tree *after* the steal**

Snapshot of Execution | Deque | Steal Tree



Help-first Scheduling Policy

→ **Another steal occurs (annotated $c4$); STEAL TREE *after* the steal**



Work-first Scheduling Policy

- 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)

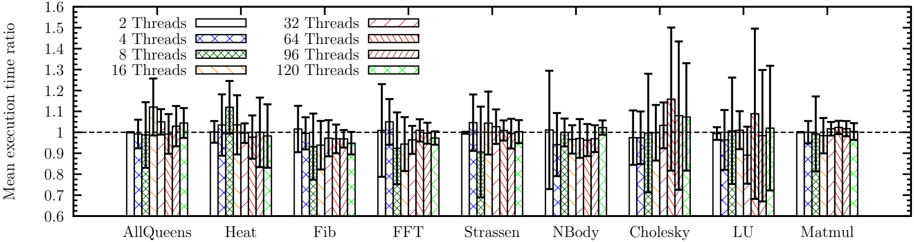
Empirical Results

→ Shared and distributed memory

Shared-memory	
Benchmark	Configuration
AllQueens	nq = 14, sequential cutoff 8
Heat	nt = 5, nx = 4096, ny = 4096
Fib	n = 43
FFT	n = 67108864
Strassen	n = 4096
NBody	iterations = 15, nbodies = 8192
Cholesky	n = 2048, z = 20000
LU	n = 1024
Matmul	n = 3000
Distributed-memory	
AQ	nq = 19, sequential cutoff 10
SCF	128 beryllium atoms, chunk size 40
TCE	$C[i,j,k,l] += A[i,j,a,b] * B[a,b,k,l]$ O-blocks $20\ 14\ 20\ 26$, V-blocks $120\ 140\ 180\ 100$
PG	13K sequences

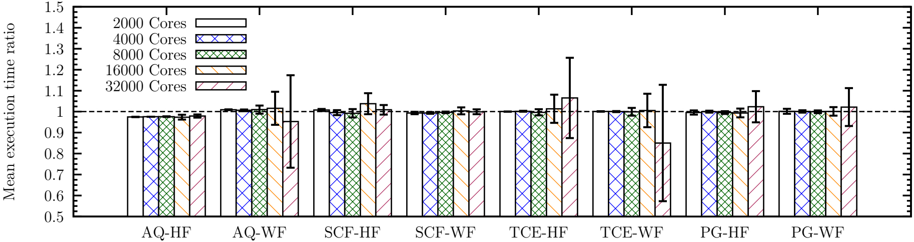
Execution Time Ratio

→ Shared-memory



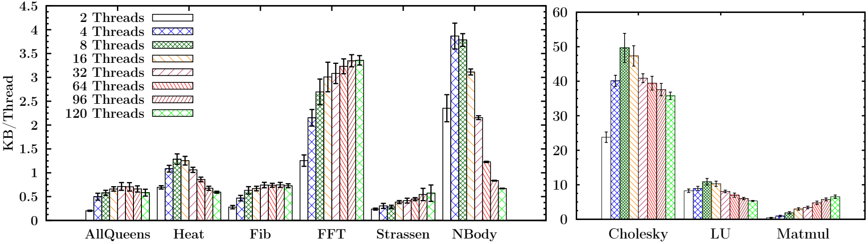
Execution Time Ratio

→ Distributed-memory



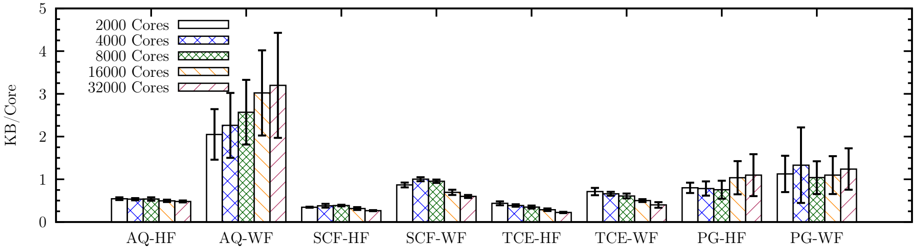
Storage Overhead

→ Shared-memory



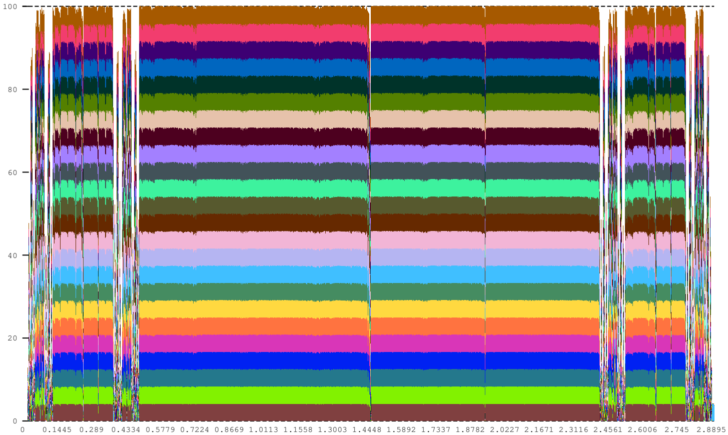
Storage Overhead

→ Distributed-memory



Utilization Graphs

→ Cilk LU



Applications

→ **Two distinct contexts**

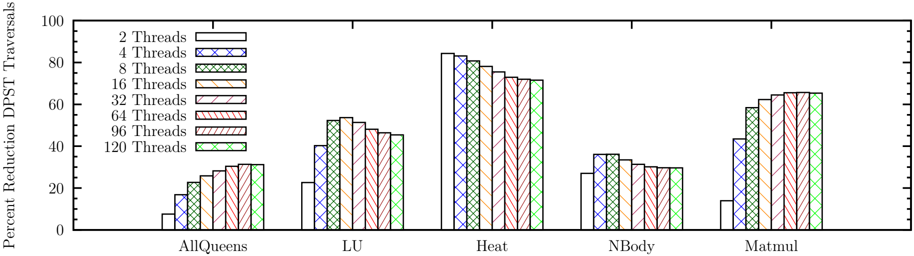
- Data-race detection: *Scalable and precise dynamic datarace detection for structured parallelism* (Raman et al., [PLDI'12])
- Retentive stealing: *Work stealing and persistence-based load balancers for iterative overdecomposed applications* (Lifflander et al., [HPDC'12])

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 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



Concluding Remarks

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