StealTree: Low-Overhead Tracing of Work Stealing Schedulers

Jonathan Lifflander, Sriram Krishnamoothy, Laxmikant V. Kale

Presented by Yufei Chen

Background

- Async-Finish Syntax
- Work Stealing Schedulers
- Tracing and Performance Analysis

Async-Finish

- async S asynchronous executes statement S. In the meantime, the current thread continues.

  a starting of a lightweight thread

- finish S is used to synchronize with all the asynchronous activities that arise during the execution of S.

  wait until all asyncs spawned by S finish

Work Stealing Scheduler

- Dynamic load balancing strategy

- Assume computation begins with a single task
- terminates when task and all dependencies finish
- Each thread maintain a deque of tasks
- Two phases: working & stealing
  - working: executes tasks from local deque
  - stealing: steal from a "victim" thread from other end of the deque

Work Stealing – Help first

@async

- Continue to execute current task
- Push concurrent tasks to the deque

@finish

- Current task continuation pushed to deque
- Complete processing of nested tasks

```
@async(Task t, Cont this):
  deque.push(t);

@finish(Task t, Cont this):
  deque.push(this);
  process(t);

@taskCompletion:
  t = deque.pop();
  if(t) process(t);
  //else this phase ends

@steal(Cont c, int victim):
  c=attemptSteal(victim);
```

Work Stealing – Work first

@async

- Push currently executing task to the deque
- Execute the nested task identified

- Partially-executed task is ready for steal
- If no steals, sequential execution order

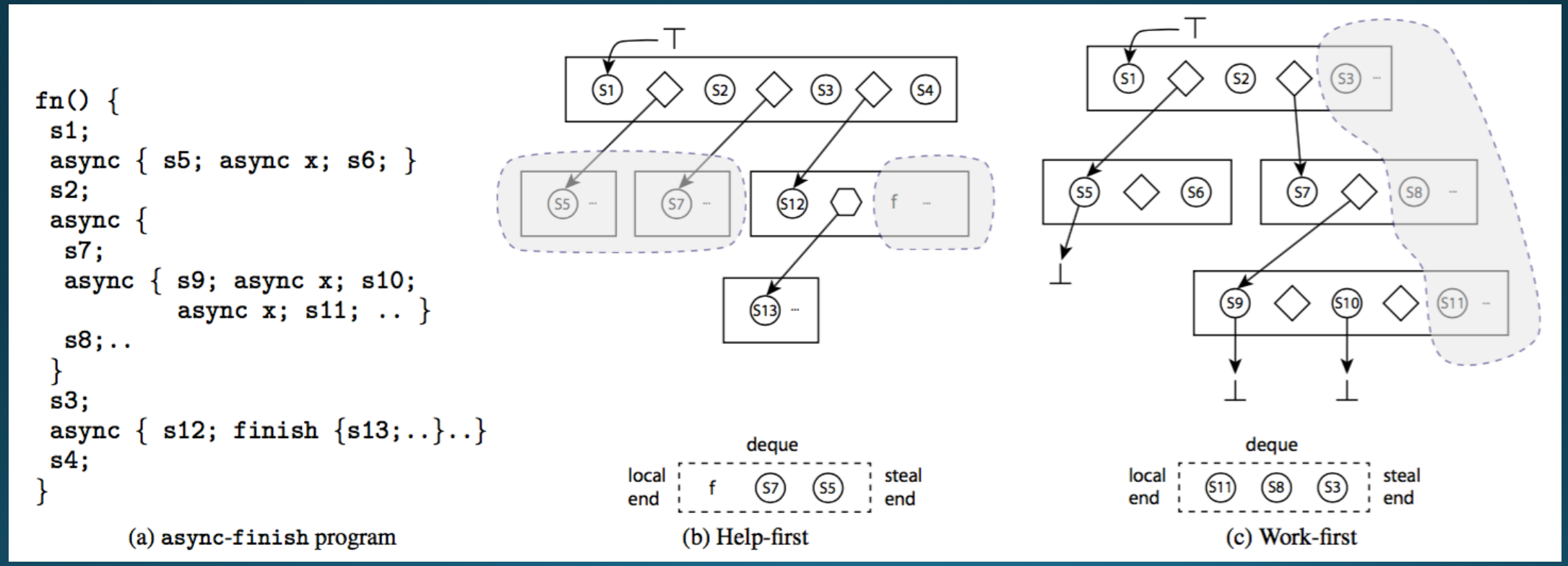
```
@async(Task t, Cont this):
  deque.push(this);
  process(t);

@finish(Task t, Cont this):
  deque.push(this);
  process(t);

@taskCompletion:
  //same as help first minus
  //some finish scope mgmt

@steal(int victim):
  //same as help–first
```

Tracing and Performance Analysis

- For work-first, steals can be recorded for the purposes of
  - Data-race detection
  - Optimizing transactional memory conflict detection
- Do not provide a general tracing algorithm, as locking required

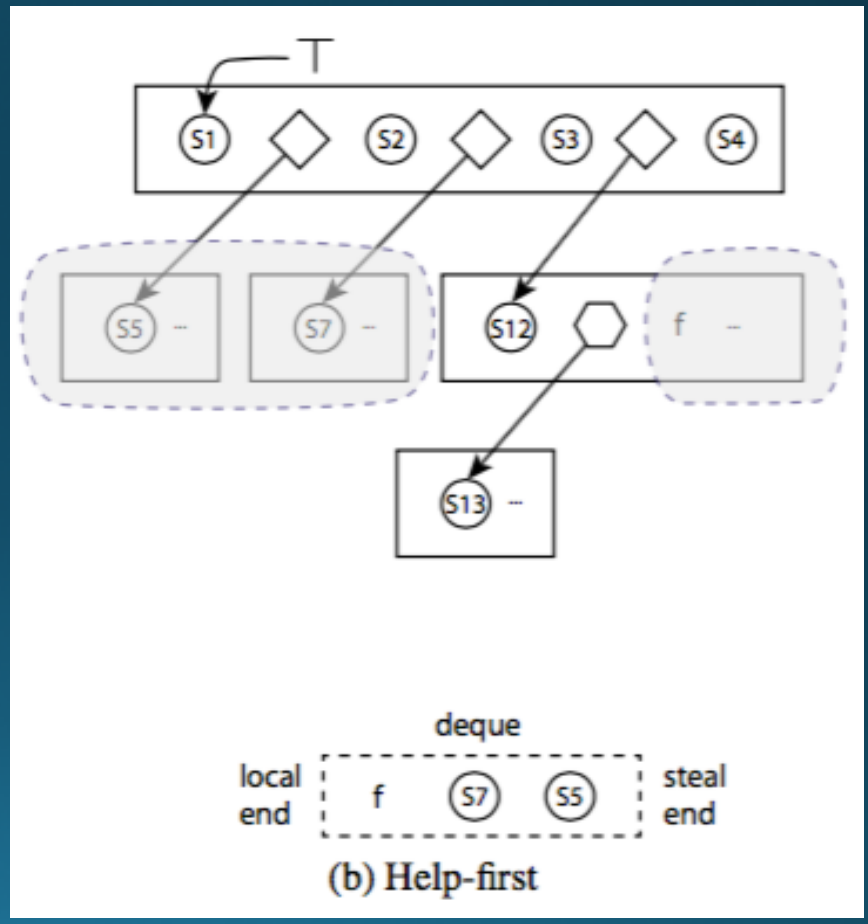
- A general tracing algorithm doesn't require global sync or locking

Problem Statement



Tracing Help-First Schedulers

- Recap
  - When @async, new tasks pushed onto the deque, current task continues
  - Children of these new tasks has same finish scope as the parent
  - When @finish, current task's continuation pushed onto the deque
  - The "new task" executed in new finish scope
- Steps in the outermost task (s1…s4) processed before any other steps

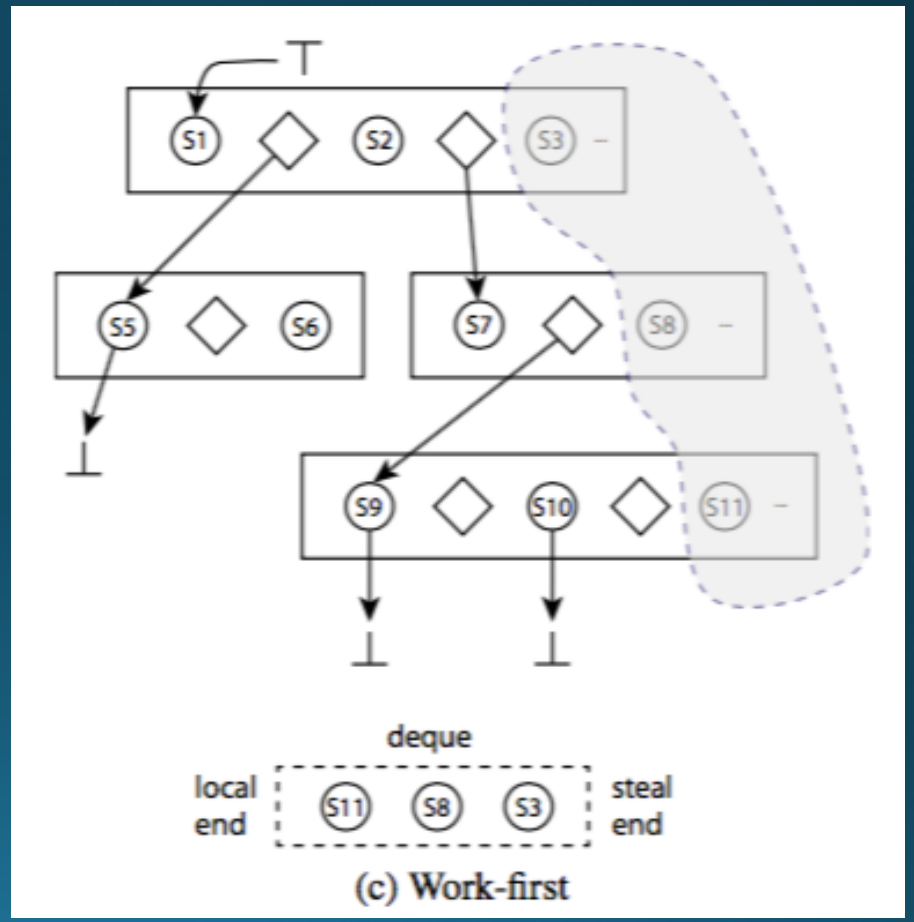


Tracing Help-First Schedulers

- Observation
  - Two tasks are in the **same immediately enclosing finish scope** if the closest finish scope that encloses each of them **also encloses their common ancestor**.
- Lemma (some highlights)
  - A task at a level is processed only after all its younger siblings in the same immediately enclosing finish scope are processed.
  - A task is stolen at a level only after all its older siblings in the same immediately enclosing finish scope are stolen.
  - At most one partial continuation is stolen at any level and it belongs to the last executed task at that level.

Tracing Work-First Schedulers

- Recap
  - When @async, push currently executing step's successor onto the deque
  - Execution continues with the first step in the new task
- Root continuation – level 0
- New tasks – level l+1 created on level l
- For levels from 0… l-1, exactly one continuation



Tracing Work-First Schedulers

- Observation
  - The deque, with l tasks in it, consists of **one continuation** at levels 0 through l – 1 with 0 at the steal-end

    and **the continuation** at level i **spawned by the step at level i – 1**.
- Lemma
  - When a continuation is stolen at level l (a) at least one task has been stolen at each level 0 through l – 1, (b) no additional tasks are created at level l.

Tracing and Replay

- Common data structures and level management

```
struct ContinuationHdr {
  int level; //this task's async level
  int step; //this continuation's step
};
struct Task : ContinuationHdr { ... };
struct Cont : ContinuationHdr { ... };
struct WorkingPhaseInfo {
  int victim; // victim stolen from
  vector<int> stepStolen; //step stolen at
      each level, init to −1
  vector<int> thieves; // list of thieves
};
```

```
struct WorkerStateHdr {
  //state for each working phase
  vector<WorkingPhaseInfo> wpi;
};

WorkerStateHdr wsh[NWORKERS]; //one
    per worker

//initializing computation's first task
@init(Task initialTask):
  initialTask.victim=−1;//victim set to −1
```

```
//start of working phase with continuation c
@startWorkingPhase(Cont c):
  c.level = 0; // level of starting frame

//spawning task t when executing task 'this'
@async(Task t, Cont this):
  t.level=this.level+1;
  t.step=0; this.step+=1;

//spawning task t in new finish scope when
    executing task 'this'
@finish(Task t, Cont this):
  t.level = this.level + 1; this.step += 1;
```

Tracing Algorithm: Work-First

- **Augment** the **steal operation** to track the steal relationship and **construct the steal tree.**
- wsh: WorkerStateHdr (one per worker)
- wpi: vector<WorkingPhaseInfo>

```
@steal(Cont c, int victim, int myrank):
  wsh[victim].wpi.back().stepStolen[c.level] = c.step;
  wsh[victim].wpi.back().thieves.push_back(myrank);
  WorkingPhaseInfo phase;
  phase.victim = victim;
  wsh[myrank].wpi.push_back(phase);
```

Tracing Algorithm: Help-First

- **Augment** the **steal operation** to track the steal relationship and **construct the steal tree.**
- wsh: WorkerStateHdr, wpi: vector<WorkingPhaseInfo>
- Note that Help-First allows **multiple** steals at each level
  - Override the WorkingPhaseInfo to store the number of tasks stole at each level (HelpFirstInfo)

```
@steal(Cont c, int victim, int myrank):
  // c is the continuation stolen by the thief
  if c.step == 0: // this is a full task
    wsh[victim].wpi.back().nTasksStolen[c.level] += 1;
  else: //this is a partial continuation
    wsh[victim].wpi.back().stepStolen[c.level] = c.step;
  wsh[victim].wpi.back().thieves.push_back(myrank);
  WorkingPhaseInfo phase;
  phase.victim = victim;
  wsh[myrank].wpi.push_back(phase);
```

Replay

- Powered by the traces collected
- During replay
  - Each thread executes the working phases assigned to it in order
  - Inform the creation of the thief to a spawned stolen task, instead of pushing to deque
  - Each thread executes its set of subtrees

Applications

- Utilizing the methods for Tracing and Replay,
- (an example) Data-race Detection
  - Building a **dynamic program structure tree** (DPST)
  - **Captures** the relationships between **async** and **finish** instances
  - Built at runtime, insert nodes into the tree in parallel
  - **Internal Nodes**: async and finish instances
  - **Leaf Nodes**: continuation steps
  - An implementation by Raman introduced in paper with
    - O(1) time inserting nodes into the DPST without synchronization

Thank you