Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers

Jonathan Lifflander
Dept. of Computer Science
University of Illinois Urbana-Champaign
jliffl2@illinois.edu

Sriram Krishnamoorthy
Comp. Sci. & Math. Division
Pacific Northwest National Lab
sriram@pnnl.gov

Laxmikant V. Kale
Dept. of Computer Science
University of Illinois Urbana-Champaign
kale@illinois.edu

Abstract

Work stealing is a popular approach to scheduling task-parallel programs. The flexibility inherent in work stealing when dealing with load imbalance results in seemingly irregular computation structures, complicating the study of its runtime behavior. In this paper, we present an approach to efficiently trace async-finish parallel programs scheduled using work stealing. We identify key properties that allow us to trace the execution of tasks with low time and space overheads. We also study the usefulness of the proposed schemes in supporting algorithms for data-race detection and retentive stealing presented in the literature. We demonstrate that the perturbation due to tracing is within the variation in the execution time with 99% confidence and the traces are concise, amounting to a few tens of kilobytes per thread in most cases. We also demonstrate that the traces enable significant reductions in the cost of detecting data races and result in low, stable space overheads in supporting retentive stealing for async-finish programs.

Categories and Subject Descriptors D.2.5 [*Software*]: Software Engineering—Testing and Debugging; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

Keywords work-stealing schedulers; tracing; async-finish parallelism

1. Introduction

The increase in number of processor cores anticipated in both commodity and high-end systems has motivated the study of dynamic task parallelism. The structured parallel programming idioms simplify performance-portable programming and tackle problems related to frequent synchronization on large-scale systems.

Work stealing is a well-studied dynamic load balancing strategy with several useful characteristics—composability, understandable space and time bounds, provably efficient scheduling, etc. [5]. Several programming models support dynamic task parallelism using work stealing, including OpenMP 3.0 [3], Java Concurrency Utilities [11], Intel Thread Building Blocks [14], Cilk [5, 6], and X10 [15]. While several properties have been proven about work stealing schedulers, their dynamic behavior remains hard to analyze. In particular, the flexibility exhibited by work stealing in responding to load imbalances leads to less structured mapping of work to threads, complicating subsequent analysis.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

In this paper, we focus on studying work stealing schedulers that operate on programs using `async` and `finish` statements—the fundamental concurrency constructs in modern parallel languages such as X10 [15]. In particular, we derive algorithms to trace work stealing schedulers operating on async-finish programs.

Tracing captures the order of events of interest and is an effective approach to studying runtime behavior, enabling both online characterization and offline analysis. While useful, the size of a trace imposes a limit on what can be feasibly analyzed, and perturbation of the application's execution can make it impractical at scale. Tracing individual tasks in an async-finish program is a prohibitive challenge due to the fine granularity and sheer number of individual tasks. Such programs often expose far more concurrency than the number of computing threads to maximize scheduling flexibility.

In this paper, we derive algorithms to efficiently trace the execution of async-finish programs. Rather than trace individual tasks, we exploit the structure of work stealing schedulers to coarsen the events traced. In particular, we construct a steal tree: a tree of steal operations that partitions the program execution into groups of tasks. We identify the key properties of two scheduling policies—help-first and work-first [9]—that enables the steal tree to be compactly represented.

In addition to presenting algorithms to trace and replay async-finish programs scheduled using work stealing, we demonstrate the usefulness of the proposed algorithms in two distinct contexts—optimizing data race detection for structured parallel programs [13] and supporting retentive stealing without requiring explicit enumeration of tasks [12].

The following are the primary contributions of this paper:

- identification of key properties of work-first and help-first schedulers operating on async-finish programs to compactly represent the stealing relationships;
- algorithms that exploit these properties to trace and replay async-finish programs by efficiently constructing the *steal tree*;
- demonstration of low space overheads and within-variance perturbation of execution in tracing work stealing schedulers;
- reduction in the cost of data race detection using an algorithm that maintains and traverses the dynamic program structure tree [13]; and
- retentive stealing algorithms for recursive parallel programs, while prior work required explicitly enumerated task collections [12].

2. Background and Related Work

2.1 Async-Finish Parallelism

An `async` statement identifies the associated statement as a task, the basis unit of concurrent execution. A task identified by the

`async` statement can be executed in parallel with the enclosing task, referred to as the parent task. A `finish` statement identifies the bounds of concurrency. All computation enclosed by a `finish` statement, including nested concurrent tasks, are required to complete before any statement subsequent to the finish can be executed.

This async-finish parallelism model, supported in X10, enables both fully strict and terminally strict computations. In fully strict computations, exemplified by Cilk, a `finish` statement implicitly encloses all execution in a task, requiring all tasks transitively nested within a task to complete before it returns. The async-finish model extends the fully strict model by supporting *escaping asyncs*, allowing a task to return before its nested concurrent tasks need to complete.

2.2 Work Stealing Schedulers

We shall assume that the computation begins with a single task and terminates when the task and its descendents complete execution. Concurrent tasks can be executed in parallel by distinct threads or processes. Each thread maintains a local deque of tasks and alternates between two phases until termination. In the *working phase*, each thread executes tasks from its local deque. When no more local work is available, a thread enters the *stealing phase* to steal from a victim's deque. A stealing thread (a *thief*) attempts to steal a task until work is found or termination is detected. The thief pushes the stolen task onto its local deque and enters a new working phase. Each thread pushes and pops tasks from the *local end* of its deque, while the thief steals from the other end (*steal end*) of the victim's deque.

We consider two scheduling policies (outlined in Figure 1) for async-finish programs identified by Guo et al. [9]. In the *work-first* policy, a thread, upon encountering an `async` or a `finish` statement, pushes the currently executing task onto the deque and begins to execute the nested task identified. A thief can steal the partially-executed task (a *continuation*) once it is pushed onto the deque. In the absence of steal operations, this policy mirrors the sequential execution order and has been shown to exhibit efficient space and time bounds.

In the *help-first* policy, the working thread continues to execute the current task, pushing any encountered concurrent tasks onto the deque. Encountering a `finish` statement, the current task's continuation is pushed onto the deque to complete processing of the tasks nested within the finish scope. Finish scopes constrain the help-first scheduler, requiring the tasks in the finish scope to be processed before tasks sequentially following the finish scope can be spawned. This scheduling policy was shown to speedup work propagation by Guo et al. [9].

2.3 Tracing and Performance Analysis

Several papers have noted for work-first schedulers that steals can be recorded for the purposes of maintaining series-parallel relationships using the SP-hybrid algorithm—for data-race detection [4, 10] and for optimizing transactional memory conflict detection [2]. However, these do not provide a general tracing algorithm and require a global lock to store "threads" of work. We provide a general tracing algorithm that does not require global synchronization or locking. We are not aware of any prior work on tracing the more complex help-first scheduling policy. The limitations of the prior work in tracing work stealing schedulers is evidenced by the fact that some of the most recent work on data-race detection for async-finish programs [13] does not exploit the steal relationship—a beneficial approach that we demonstrate later in this paper.

Work stealing has typically been studied empirically (e.g., [1, 12]). Tallent and Mellor-Crummey [16] presented blame shifting to relate lack of parallel slack in work stealing to code segments.

```
@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);
```

(b) Help-first scheduler

```
@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
```

(a) Work-first scheduler

Figure 1. Basic actions in the two scheduling policies

They assume global information about the number of active and idle workers.

3. Notation

We define a *continuation step* (or simply *step*) to be the dynamic sequence of instructions with no interleaving `async`, `finish`, `at`, or `when` statements. Each continuation step will be executed by exactly one thread and cannot be migrated between threads during execution.

A continuation is the portion of execution, in terms of continuation steps, reachable from a given continuation step. In other words, a continuation represents the remainder of the execution that begins with the given continuation step. A task is a continuation marked by an `async` or `finish` statement, representing all steps executed in the task. In shared-memory environments, all continuations of a task occupy the same storage and transform the state from one step to the next. The term continuation is used in places where a task is used, except when we intend to specifically distinguish a task from a continuation. A partial continuation is a continuation that represents a proper subset of the computation represented by a task.

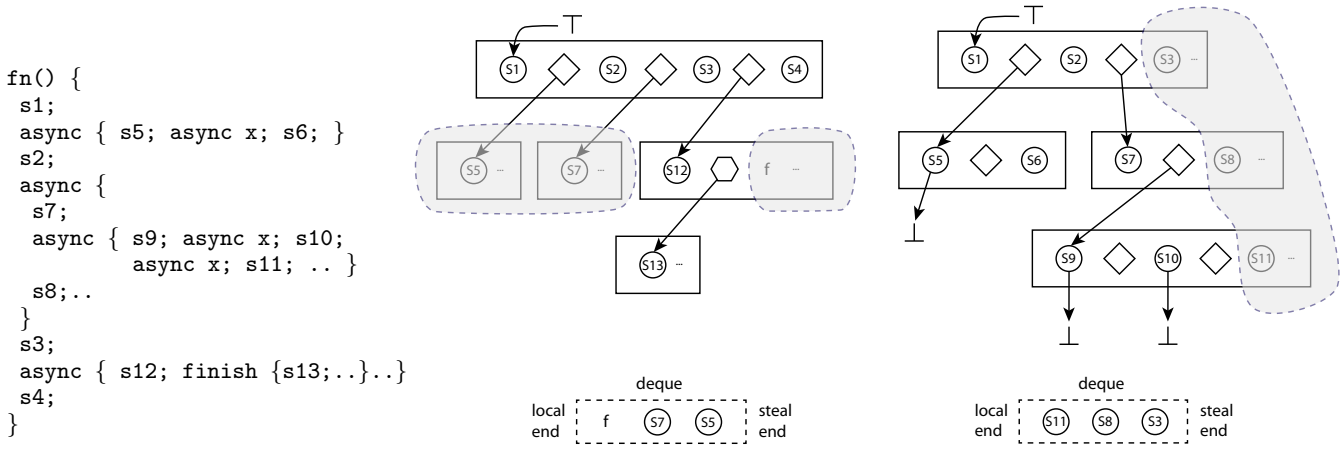
A task is said to spawn another task when the task's execution encounters an `async` statement. Without changing program semantics, we treat a `finish` statement synonymously with a `finish async`. Thus a `finish` statement leads to the spawning of a task as well. All child tasks spawned from a given task are referred to as siblings and are ordered from left to right. We also refer to a task's left and right siblings, if they exist.

Each task is associated with a level. The initial task in a working phase is at level 0. A level of a spawned task is one greater than that of the spawning task.

4. Problem Statement

An example async-finish program is shown in Figure 2a. In the figure, `s1`, `s2`, etc. are continuation steps. Tasks spawned using `async` statements need to be processed before the execution can proceed past the immediately enclosing `finish` statement. `async` statements that are not ordered by a `finish` statement (e.g., the `async` statements enclosing `s5` and `s9`) can be executed in any order. The objective is to compactly track the execution of each continuation step.

The execution in each worker is grouped into phases with each phase executing continuation steps in a well-defined order, starting from a single continuation. In each working phase, the computation begins with a single continuation step and involves the execution of all steps reached from the initial step minus the continuations that were stolen. The tracing overheads (space and time) can be



(a) `async-finish` program (b) Help-first (c) Work-first

Figure 2. An example async-finish parallel program and a snapshot of its execution. Legend: $\top$ represents the root task; $\bot$ represents a sequential task; a circle represents a step; a diamond represents an `async` statement; a hexagon represents a `finish` statement; '..' represents a continuation; an arrow represents a spawn relationship; a rectangular box represents a task; and the shaded region represents the region that is stolen. The deque depicted at the bottom has a *steal end* that is accessed by thieves and a *local end* that is accessed by the worker thread.

significantly reduced if the steps executed in each working phase can be compactly represented.

Note the difference in the stealing structure between the work-first and help-first schedulers in Figure 2. While continuations stolen in the work-first schedule seem to follow the same structure across all the levels shown, the help-first schedule can produce more complicated steal relationships. This distinction is the result of the difference in the scheduling actions of the two policies—especially when an `async` statement encountered, as shown in Figure 1. The challenge is to identify the key properties of help-first and work-first scheduling policies to compactly identify the the leafs of the tree of steps rooted at the initial step in each working phase.

5. Tracing Help-First Schedulers

Under help-first scheduling, spawned tasks are pushed onto the deque, while the current task is executed until the end or a finish scope is reached. Children of task spawned by `async` statements are in the same finish scope as the parent. Encountering a `finish` statement, the current task's continuation is enqueued onto the deque and the spawned task in the new finish scope is immediately executed. When all the tasks nested within the `finish` statement have been processed, the finish scope is exited and the execution of the parent task is continued, possibly spawning additional tasks. We refer the reader to Guo et al. [9] for the detailed algorithm.

Figure 2b shows a snapshot of the help-first scheduling of the program in Figure 2a. The steps in the outermost task, represented by `fn()`—`s1`, `s2`, `s3`, and `s4`—are processed before any other steps.

OBSERVATION 5.1. *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 5.2. *A task at a level is processed only after all its younger siblings in the same immediately enclosing finish scope are processed.*

Proof. A work-first scheduler enqueues the spawned tasks from the start of the task's lexical scope. The spawned tasks are enqueued onto the deque with the newer child tasks enqueued closer to the local-end than the older ones. The child tasks are enqueued until the executing task completes or a `finish` statement is encountered. Tasks under the encountered `finish` statement are immediately processed and the execution continues with enqueuing tasks spawned using the `async` statement. This property, combined with the fact that tasks popped from the local end are immediately processed, requires all younger siblings of a task to be processed before it can be processed. □

LEMMA 5.3. *A task is stolen at a level only after all its older siblings in the same immediately enclosing finish scope are stolen.*

Proof. Recall that all `async` statements in a task are in the same immediately enclosing finish scope, and `finish` statements introduce new finish scopes.

1. Consider the case when two `async` statements in a task have no intervening `finish` statement. In this case, the corresponding tasks are pushed onto the deque with the older sibling first. Thus the older sibling is stolen before the younger sibling.
2. Now consider the case where siblings in the same immediate finish scope are separated by one of more intervening `finish` statements. In particular, consider the execution of the following sequence of statements in a task: *async n; ... finish p; ... async m*. Tasks n and m are in the same immediately enclosing finish scope, while task p is not. Task n is first enqueued onto the deque. When the `finish` statement is processed, the current task's continuation c is then pushed onto the deque with the worker immediately processing statement p. At this point, if a steal occurs, n and c are at the steal end of the deque followed by tasks spawned from p. Hence, all older siblings (n and c) in the same immediately enclosing finish scope will be stolen before tasks generated from p. The execution proceeds past the `finish` statement only after all the tasks nested by p are complete. Once they are executed, the continuation of the current task c is dequeued (unless it was stolen) and task m is enqueued on top of n in the deque. The execution now devolves onto case 1 and the older sibling is stolen before the younger.

□

LEMMA 5.4. *At most one partial continuation is stolen at any level and it belongs to the last executed task at that level.*

Proof. A task's partial continuation is enqueued only when it encounters a `finish` statement. Consider such a task. When the task is being processed, its parent's continuation is in the deque, or it has completed processing. Based on lemma 5.2, all the parent's younger siblings have been processed. Hence, the queue has the parent's older siblings followed by a possible partial continuation of the parent followed by this task's older siblings, then this task. By lemma 5.3, this task's partial continuation will not be stolen until all the parent's older siblings are stolen. By lemma 5.2 all younger siblings of this task, if any, have been processed at this point. Thus the deque only has the partial continuation's children, all of which are at levels higher than this task. After this task's partial continuation is stolen, all subsequent tasks will be at higher levels, making this task the last one at this level. □

LEMMA 5.5. *All tasks and continuations stolen at a level l are immediate children of the last task processed at level $l-1$.*

Proof. We first prove by contradiction that the all tasks stolen at a given level are children of the same parent task. Let two stolen tasks at a given level be children of distinct parent tasks. Let t_a be the lowest common ancestor of of these tasks, and t_1 and t_2 be the immediate children of t_a that are ancestors of the two tasks of interest. t_1 and t_2 are thus sibling tasks. Without loss of generality, let t_1 be the older sibling. By lemma 5.2, t_2 is processed before t_1. By lemma 5.3, t_1 is stolen before any descendent of t_2 can be stolen. Thus no descendent of t_1 can be enqueued if a descendent of t_2 is stolen, resulting in a contradiction: either the steals will be at different levels because descendents of t_1 cannot be enqueued, or they will be children of the same parent task.

We now prove that the parent task q of all tasks stolen at level l is the last task processed at level $l-1$. Let t be any task at level $l-1$ that is not q. By lemma 5.2, task t must be an older sibling of the parent task q. From lemma 5.3, any task t must be stolen before q. By the above proof, any task with a higher level y must be a child of the same parent task. We now show by contradiction that y must be a child of q, the last task processed at level $l-1$. If y is a child of some task t, then t is currently being processed. By lemma 5.2, q is processed before t. q has not been processed, hence we have a contradiction. □

LEMMA 5.6. *The parent q of the tasks stolen at level $l+1$ is a sibling of the tasks stolen at level l.*

Proof. We prove this by contradiction. By lemma 5.5, q is last task processed on level l. By lemma 5.2, all younger tasks on level l have been processed. If q is not a sibling of the tasks stolen at level l, q could not have been processed. Thus the stolen tasks at level $l+1$ could not have been created, resulting in a contradiction. □

LEMMA 5.7. *The parent of the tasks stolen at level $l+1$ is either the immediate younger sibling of the last stolen task at level l, or the immediate younger sibling of the last stolen task at level l in the same immediately enclosing finish scope.*

Proof. From lemmas 5.4, 5.5, and 5.6, the last task in the same immediately enclosing finish scope that is executed is the closest younger sibling of stolen task that is in the same enclosing finish scope. Thus the last task executed at that level is either the immediate right sibling of the last stolen task, say t_1, or the closest right sibling in the immediate enclosing finish scope, say t_2. When both are identical, the outcome is clear. When they are distinct, the immediate younger sibling of the last stolen task is in a distinct finish scope. If the `finish` statement is stolen, no further tasks can be processed that are not descendents of this statement, making t_1 the last executed task. If the `finish` statement is not stolen, no descendent tasks of this statement can be stolen, making t_2 the last executed task. □

THEOREM 5.8. *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.*

Proof. The tasks stolen at a level can be determined from the number of tasks stolen at that level and the identification of these tasks' parent (by lemma 5.7) (transitively until level 0 which has just one task). The position of the partial continuation stolen at a level can be determined from the fact it is the last processed task at a given level (lemma 5.4) and from the number of tasks stolen at that level in the same finish scope as the parent. Together with the step information tracking, this uniquely identifies all stolen continuations. □

Illustration. A snapshot of execution under the help-first work stealing policy is shown in Figure 2b. Steps `s1`, `s2`, `s3`, `s4`, and `s11` have been executed. Because `s13` is encountered in a finish scope, `s12` spawns the task starting with step `s13` and continues recursive execution before continuing the execution past the finish scope. Meanwhile, the deque consists of tasks `s5`, `s7`, and the continuation past the finish scope, represented by `f`, and were stolen. Note that the help-first scheduler steals from left-to-right when stealing full tasks, and right-to-left (similar to a work-first scheduler) when stealing a partial continuation.

The subtrees executed in each working phase form a steal tree. The root of the tree is the subtree that includes the main continuation that began the program. Each child is a subtree stolen from the victim's subtree. Each node in the steal tree contains information about the continuations stolen from it, bounding the actual steps executed in that subtree. Each edge in the steal tree contains information about the position of the steal from the parent subtree.

6. Tracing Work-First Schedulers

Under work-first scheduling, spawning a task involves pushing the currently executing step's successor onto the deque, with the execution continuing with the first step in the spawned task.

The continuation that starts the working phase is at level 0, referred to as the root continuation. Tasks spawned by continuations at level l are at level $l+1$. The work-first scheduling policy results in exactly one continuation at levels $0, \ldots, l-1$, where l is the number of continuations in the deque. We observe the tasks spawned during a working phase and prove that there is at most one steal per level, and a steal at all levels $0, \ldots, l-1$ before a task can be stolen at level l. This allows us to represent all the steals for a given working phase as a contiguous vector of integers that identify the continuation step stolen at each level, starting at level 0.

OBSERVATION 6.1. *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 that precedes the continuation at level $i-1$.*

The execution of the work-first scheduler mirrors the sequential execution. The task executing at level l is pushed onto the deque before spawning a task at level $l+1$. Thus the deque corresponds to one path from the initial step to the currently executing task in terms of the spawner-spawnee relationship.

LEMMA 6.2. *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.*

Proof. The first part follows from observation 6.1 and the structure of the deque—because stealing starts from the steal-end tasks at levels 0 through $l-1$ must stolen before level l.

We prove the second part by induction. Consider the base case when the root of the subtree is the only continuation at level 0 in the deque. After the root of the subtree is stolen, no continuation exists at level 0 to create another task at level 1. Let the lemma be true for all levels $0, \ldots, l$. When a continuation at level l is stolen, no further tasks can be created at level l. Now consider the lemma for level $l+1$. After a steal at level l, no further tasks can be created at level $l+1$. Once the current continuation at level $l+1$ is stolen, no subsequent tasks are created at level $l+1$. □

THEOREM 6.3. *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.*

Proof. By lemma 6.2, steal operations on a victim are totally ordered, implying that each task stolen from a victim during distinct working phase is at a unique level. Because no additional tasks can be created at that level (again by lemma 6.2), the step of the continuation stolen at that level is sufficient to uniquely identify it. □

These observations allow the steal points to be maintained in a contiguous array. The size of the array corresponds to the number of steals in this working phase, and the value at position l in the array corresponds to the index of the continuation stolen at level l. The stolen continuations can be identified to absolute or relative indices. Absolute indices—counting the number of steps executed in this phase at each level—does not effectively support retentive stealing, as explained later. We employ relative indices, with the value at index l corresponding to the number of steps executed in the last task executed at this level. Note that the last task executed at level l is a child of a predecessor step of the continuation stolen at level $l-1$. Given there is only one task at level 0, we store the number of steps by this worker for this initial step.

Illustration. A snapshot of execution under the work stealing scheduler is shown in Figure 2c. The thread began execution from step `s1` and has completed execution of `s1`, `s2`, `s5`, `s6`, `s7`, `s9`, and `s10`. It is currently executing the task spawned by `s10` with the the deque consisting of steps `s3`, `s8`, and `s11`—bottom to top. These steps have been created but not yet processed. During some point in the execution, these tasks have been stolen by thieves. The steal order is `s3`, `s8`, followed by `s11`. Note that the execution in the work-first scheduling policy is *left-to-right*, while the steals are *right-to-left*.

7. Tracing and Replay Algorithms

We now present the algorithms for tracing and replay based on the properties identified for help-first and work-first schedulers. in Sections 5 and 6. These properties simplify the state to be managed to trace and replay both schedulers. The algorithms rely on some state in addition to the basic scheduler actions. This shared state is shown as C++-style pseudo-code in Figure 3.

Every continuation and task has an associated ContinuationHdr that stores the level and step. When a new working phase is started or an `async` or `finish` is encountered, the bookkeeping keeps track of the current level for each continuation. Each working phase that is executed by a worker maintains the state shown in WorkingPhaseInfo as part of the trace. A worker's trace is an ordered list of working phases it has executed so far, retained as one WorkerStateHdr object per worker. The root task of the entire computation is not stolen and its victim is set to -1. Each continuation's step is tracked as an integer and is updated on each `async` or `finish` statement.

7.1 Tracing

The tracing algorithms augment the steal operation to track the steal relationship and construct the steal tree. From the earlier discussion, we know that, unlike work-first schedulers, help-first schedulers allow multiple tasks to be stolen at each level. Thus for tracing a help-first scheduler, we store the number of tasks stolen at each level:

```
struct HelpFirstInfo : WorkingPhaseInfo {
  vector<int> nTasksStolen; //num. tasks stolen at each
        level, init to 0
};
```

When a continuation is stolen under help-first scheduling, the thief marks the steal in the victim's HelpFirstInfo. The HelpFirstInfo for the current working phase on the victim can only be accessed by a single thief, and hence requires no synchronization. myrank is the thief thread's rank. Note that just the number of stolen tasks at each level and the partial continuation's step is sufficient to reconstruct all information about the tasks executed in a given working phase.

```
@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);
```

When a continuation is stolen under work-first scheduling, the following marks the steal in the victim's WorkingPhaseInfo. For the work-first policy, the actions required are less complex because at most one task can be stolen per level.

```
@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);
```

Note that none of these actions require additional synchronizations, and all the tracing overhead incurred is on the steal path.

7.2 Replay

The collected traces include timing information, which allows the traces to be replayed. During replay, each thread executes the working phases assigned to it in order. Whenever a stolen task is spawned, rather than adding it to the deque, the corresponding thief is informed of its creation. Each thread executes its set of subtrees at the same time point as in the original run, after ensuring that the given subtree's root task has been spawned by the corresponding victim. The creation of the initial task in working phase indicates that all dependences before a task in that working phase have been satisfied.

During replay, each task tracks whether its children could have been stolen in the trace, i.e., the task is at the *frontier*, using the following additional field in the ContinuationHdr:

```
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;
```

Figure 3. Common data structures and level management for all algorithms

```
struct ReplayContinuationHeader : ContinuationHdr {
  bool atFrontier; // could any of its children have been
      stolen, initially false
};
```

When a worker encounters a task that was stolen, it marks it as stolen and notifies (in shared-memory) or sends (in distributed memory) the task to the thief. When a thread starts the execution of a working phase, it waits for the initial task to be created by the victim. The worker with the initial task for the entire computation begins execution, identifying the initial task as being at the frontier.

```
markStolen(Task t):
  // enqueue 't' for sending or directly send to the next thief
      in current working phase info's thieves
  // drop 't' from execution on this worker

@startWorkingPhase(WorkingPhaseInfo wpi):
  // get initial task from wpi.victim, if wpi.victim>=0

@init(Task initialTask, int myrank):
  if(wsh[myrank].wpi[0].victim == −1):
    initialTask.atFrontier = true;
```

The above actions are used to replay traces from both help-first and work-first schedulers.

When help-first traces are replayed, the number of child tasks spawned by each task in the same finish scope is tracked by augmenting the following to the HelpFirstInfo structure.

```
struct HelpFirstReplayInfo : HelpFirstInfo {
  vector<int> childCount; //num children for current
      executing task
};

// at the beginning of execution of a task
@taskEntry(Task this, int myrank):
  wsh[myrank].childCount[this.level + 1] = 0;
```

A task spawned by an `async` statement is marked as being at the frontier if it is the immediate younger sibling of the last child task stolen from this task. When the executing task is at the frontier and the child count is less than the number of tasks stolen at the next level, the spawned task is marked as stolen. A task spawned by a `finish` statement can mark the executing task as stolen as well. If the partial continuation of the `finish` statement is not stolen, none of its descendents is stolen either.

```
@async(Task t, Continuation this, int myrank,
    WorkingPhaseInfo current_wpi):
  t.level = this.level + 1; t.step=0; this.step+=1;
  if this.atFrontier:
    if wsh[myrank].childCount[t.level] < current_wpi.
        stealCount[t.level]:
      markStolen(t);
    else if wsh[myrank].childCount[t.level] == current_wpi.
        stealCount[t.level]:
      t.atFrontier = true;
  wsh[myrank].childCount[t.level] += 1;

@finish(Continuation t, Continuation this, int myrank,
    WorkingPhaseInfo current_wpi):
  // t does not contribute to calculation of childCount
  t.level = this.level+1;
  if this.atFrontier:
    if this.step == wsh[myrank].stepStolen[this.level]:
      markStolen(this); // continuation of this after spawning
          the finish
      t.atFrontier = true; // only child of stolen parent is
          −− also at frontier
      if wsh[myrank].childCount[t.level] < current_wpi.
          stealCount[t.level]:
        assert(wsh[myrank].childCount[t.level] ==
            current_wpi.stealCount[t.level]−1);
        markStolen(t);
```

When replaying work-first traces, the primary action is determining whether a task is at a frontier. When an `async` or `finish` statement is encountered, the following actions are executed:

```
@async(Task t, Cont this, int myrank):
  t.level = this.level+1; t.step=0; this.step += 1;
  if this.atFrontier:
    if this.step == wsh[myrank].stepStolen[this.level]:
      markStolen(this);
      t.atFrontier = true;

@finish(Task t, Cont this, int myrank):
  //same action as for async(t, this, myrank)
```

7.3 Space Utilization

The space overhead can be quickly computed from the data structures employed in the algorithms. In the following formulæ, b_h and b_w describe the total number of bytes required to trace help-first and work-first schedulers, respectively:

$$b_h = \sum_{i=0}^{n} v(1+s_i) + s_i(m+k) \qquad \text{(Total bytes for help-first)}$$

$$b_w = \sum_{i=0}^{n} v(1+s_i) + s_i m \qquad \text{(Total bytes for work-first)}$$

where n is the total number of working phases, v is the number of bytes required for a thread identifier, s_i is the number of steals in a working phase, m is the number of bytes required for a step

identifier, and k is number of bytes required to store the maximum number of tasks at a given level.

For Figures 5 and 7 that graph the storage required, we use integers to store the thread and step identifiers, and assume that the maximum number of tasks spawned at a given level does not exceed the size of an integer: $k = m = v = \text{sizeof(int)} = 4$ bytes.

8. Applications

8.1 Data-race Detection for Async-Finish Programs

Raman et al. [13] perform data race detection by building a dynamic program structure tree (DPST) at runtime that captures the relationships between `async` and `finish` instances. The `async` and `finish` nodes in the tree are internal and the leaves are `step` nodes that represent a step (same as the continuation step in this paper) in the program execution where data accesses may occur. The DPST is built dynamically at runtime by inserting nodes into the tree in parallel. Raman et al. present a DPST implementation that exploits the tree structure to insert nodes into the DPST in parallel without synchronization in $O(1)$ time. Here, we summarize the key cost factors in that implementation and refer readers to Raman et al. [13] for the full description.

To detect races, if an application performs a data access during a step, a reference to the step is stored in shadow memory so other steps that read/write this same address can reference it. If two steps access the same memory address, the structure of the tree is used to determine if these steps can possibly execute in parallel for any possible schedule. Conflicting access to the same memory location by concurrent steps is detected as a data race.

Two steps can execute concurrently, identified as satisfying the dynamically-may-happen-in-parallel relationship, if their lowest common ancestor (LCA) in the DPST tree is an `async` node. Each memory location is associated with two steps that read the location and one that last wrote that location. The two read operations form the extremal bounds of a DPST sub-tree that contains all concurrent reads to the memory location since the last synchronization point. Rather than comparing with every step performing a read, a step performing a write-operation might be flagged as causing a data race if it can execute concurrent with the three steps whose reads and writes are being tracked. Data-race detection for read operations is similar but also involves operations to track the extremal bounds of the sub-tree of concurrent reads.

It can be seen that computing the lowest common ancestor (LCA) of two steps in computing the dynamically-may-happen-in-parallel relation is a performance critical operation, invoked several times for each read and write operation. In particular, finding the LCA is among the most expensive parts of the data-race detection. The DPST creation takes constant time and requires no synchronization and updating the memory address locations atomically only happens when a memory address is written or read and the step is higher in the tree than the previous steps that read. On the other hand, computing the LCA between two steps involves walking the DPST from the two steps to their parent, with the cost proportional to the number of edges walked. Because the DPST contains the entire async-finish tree, it may be very deep if the application is fine-grained, leading to very expensive LCA calculations.

We observe that the steal tree can be used to partition the DPST based on the steal relationships. In particular, it can be seen that the LCA of two steps in distinct working phases is the LCA of the initial steps of the two working phases. Exploiting this additional structural information allows us to bypass potentially large portions of the tree when traversing upward to locate the LCA. We relate a step/async/finish in the DPST with the steal tree as we create the DPST. For each node (`step`, `async`, or `finish`) in the DPST a pointer is added to store the working phase it belongs to. If a step accesses data, a reference to that step may be stored in the shadow memory. We compute and track the absolute level of the initial task executed in each working phase, referred to as the depth of the subtree executed in that working phase. The algorithm below shows how we traverse the tree when two steps accessing the same memory address are executed in different working phases. If the depths of the subtrees are different, we traverse up the deeper one. If they are equal we traverse up both until the depths are different or the ancestors of both steps are in the same working phase subtrees. Once we are in the same subtree, we invoke the LCA method in [13] on the initial tasks of the two subtrees.

```
subtreeLCA(Cont s1, Cont s2):
  while(s1 and s2 are in different subtrees):
    while(s1 subtree depth != s2 subtree depth):
      if (s1 subtree depth < s2 subtree depth):
        s1 = initial task in s1's working phase
      else:
        s2 = initial task in s2's working phase
    if((s1 subtree depth == s2 subtree depth) &&
       (s1 and s2 in different subtrees)):
      s1 = initial task in s1's working phase
      s2 = initial task in s2's working phase
  return LCA(s1,s2) //Raman et al.'s algorithm
```

8.2 Retentive Stealing for Async-Finish Programs

Iterative applications that have identical or slowly evolving characteristics are said to exhibit the principle of persistence [18]. Such applications can be incrementally rebalanced based on performance profiles gathered from prior iterations. Lifflander et al. [12] exploited the property of persistence to improve the scalability of work stealing in iterative applications. In their approach to load balancing, each worker begins an iteration with the a collection of tasks executed by it in the previous iteration. This was shown to significantly improve parallel efficiency as the iterations progressed, in some cases from about 10% to about 90%.

Async-finish programs that exclusively exploit recursive parallelism have similar performance challenges. First, the slow propagation of work from the single initial task available at the beginning of the computation incurs significant ramp-up time. While help-first scheduling ameliorates some of the performance challenges, the challenges remains. Second, expanding from a single initial task for every iteration discards the application information on persistence.

While retentive stealing presented by Lifflander et al. [12] can address these issues, their approach was presented in the context of explicit enumeration of tasks to enable retentive stealing. This not only increases the storage overhead, but is also infeasible when intervening `finish` statements are involved. The key insight in enabling retentive stealing is to allow the execution in each worker to begin with the working phases in the previous iteration, while also allowing work stealing to improve load balance. We observe that retentive stealing can be applied to recursive parallel programs by building on the replay algorithms presented earlier. In particular, we explain the extensions to the replay algorithms to allow stealing of tasks from a working phase being replayed.

During normal execution, a worker can execute a stolen task to completion barring synchronization constraints imposed by the `finish` statement. However, a working phase being replayed completes execution when all the tasks in that phase are complete. In particular, the tasks at the frontier of a working phase need to be distinguished from other tasks. When stealing from a working phase, we therefore check whether the stolen task is at the frontier. In addition to the replay actions, a steal operation correctly identifies the steps represented by the stolen continuation, which can be extracted from the WorkingPhaseInfo structure. A task can be at the frontier,

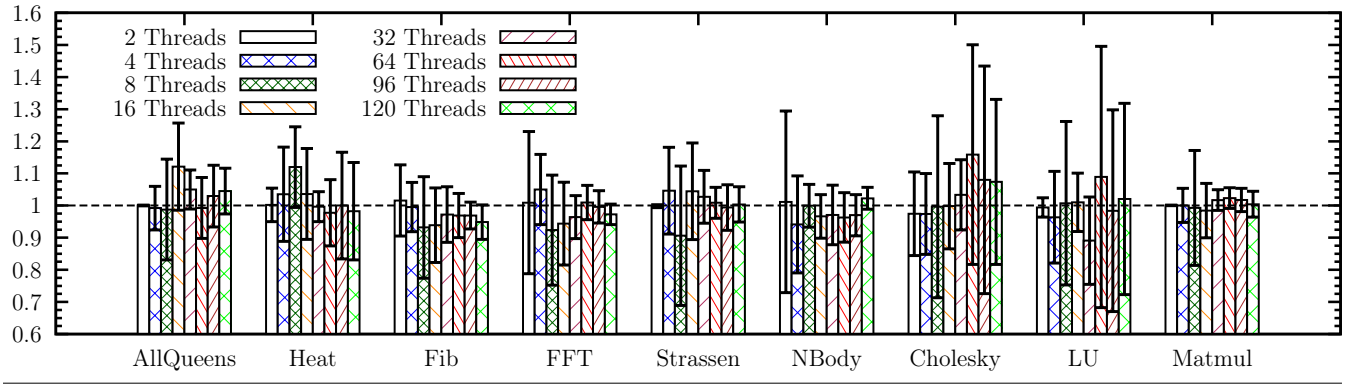


Figure 4. The ratio of mean execution time with tracing versus without tracing with a sample size of 15 on the POWER 7 architecture using the shared-memory Cilk runtime. The error bars represent the error in the difference of means at 99% confidence, using a Student's t-test.

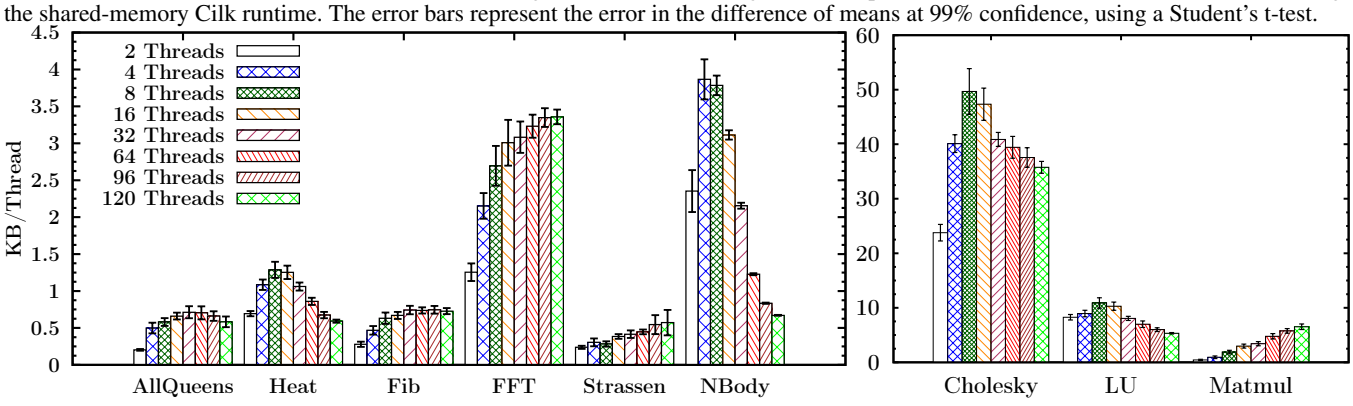


Figure 5. The storage overhead in KB/thread with our tracing scheme using the shared-memory Cilk runtime on the POWER 7 architecture. The error bars represent the standard deviation of storage size with a sample size of 15.

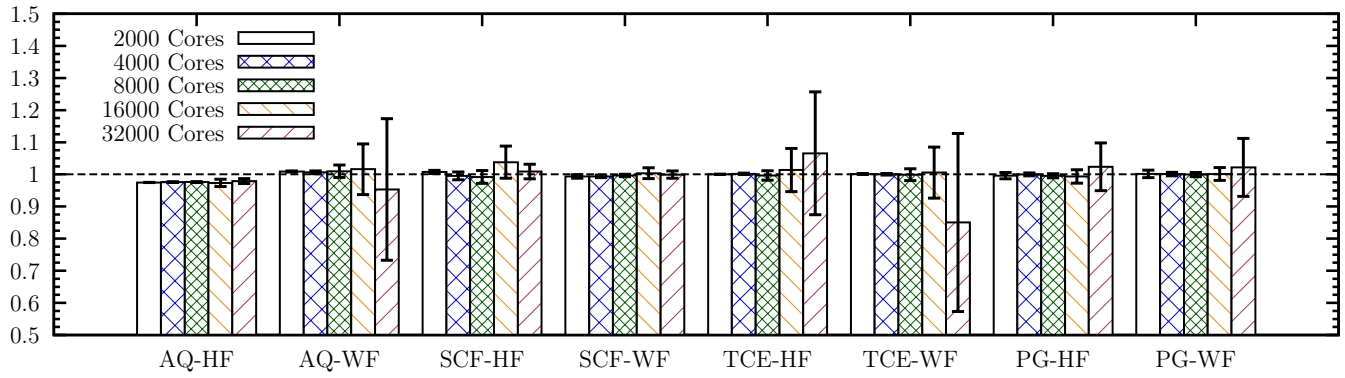


Figure 6. The ratio of mean execution time with tracing versus without tracing with a sample size of 15 on Cray XK6 Titan in distributed-memory. The error bars represent the error in the difference of means at 99% confidence, using a Student's t-test.

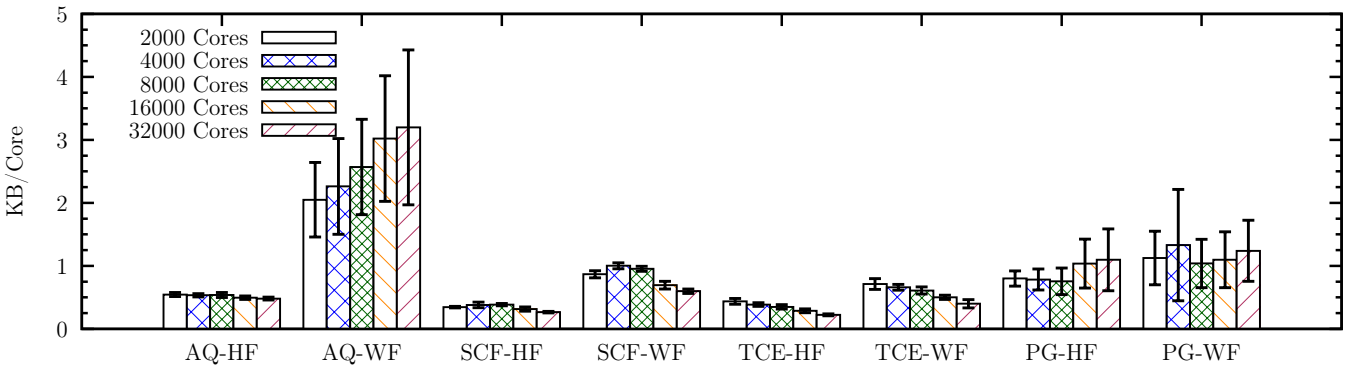


Figure 7. The storage overhead in KB/core with our tracing scheme for distributed-memory on Cray XK6 Titan. The error bars represent the standard deviation of storage size with a sample size of 15.

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

Table 1. Benchmark configurations.

i.e., its descendents stolen in the current working phase, only if its parent is at the frontier. Thus a thief stealing a step not at the frontier can execute all tasks reachable from that step. When a stolen task is at the frontier, the WorkingPhaseInfo structure associated with victim's current working phase is also copied. This allows the thief to ensure that it does not execute steps reachable from the stolen continuation past the work represented by the victim's working phase. The calculations that use the task's level to determine the frontier during replay are adapted to take into account the level of the stolen task in the victim's working phase.

In this approach, steals of tasks not at frontier just steal the step information and incur the same cost as a steal during normal execution. Steals at the frontier, also need to copy the frontier information from the victim's working phase, and thus incur greater data movement overhead on distributed-memory systems. Stealing from working phases to enable retentive stealing can also lead to working phase "fragmentation", potentially increasing the number of working phases and hence the storage overhead as the iterations progress. In section 9, we show that these overheads do not adversely impact performance and the storage required stays well below the amount required to explicitly enumerate all tasks.

9. Experimental Evaluation

The shared-memory experiments were performed on a POWER 7 system composed of 128 GB of memory and a quad-chip module with eight cores per chip running at 3.8 GHz and supporting four-way simultaneous multi-threading. The distributed-memory experiments were performed on the OLCF Cray XK6 'Titan', an 18688-node system with one AMD sixteen-core 2.2GHz 'Bulldozer' processor and 32GB DDR3 memory per node.

The implementations were compiled with GCC 4.4.6 on the POWER 7 and PGI 12.5.0 on Titan. The distributed-memory benchmarks used MPI (MPICH2 5.5.0) for communication. We implemented the tracing algorithms for shared-memory systems in Cilk 5.4.6, and distributed-memory versions using the implementation in [12]. We tried to reproduce the configurations used by Lifflander et al. [12] in their distributed-memory experiments.

9.1 Tracing on Shared-Memory Systems

We evaluated seven of the example benchmarks in the Cilk suite and have added two more—NBody and AllQueens; the configurations are shown in Table 1. AllQueens is a variant of the NQueens Cilk benchmark that searches for all valid board configurations rather than just one. The NBody benchmark is from the "Computer Language Benchmarks Game" suite [8]; we have parallelized the benchmark by spawning a task per nbody calculation and using synchronization between iterations for the n-body updates.

For these nine benchmarks, we graph the ratio of execution time with our tracing versus the execution time without tracing in Figure 4. Each bar is the ratio of the mean of 15 runs with and without tracing for each benchmark and the error bars are the standard error in the difference of means at 99% confidence, using a Student's t-test [7]. This figure shows that our tracing overhead is low and within the run-to-run variability on the machine. We performed these same comparisons on another shared-memory architecture (an AMD x86-64 system) and observed the same trend: low overhead but high variability between runs.

Figure 5 shows the storage overhead in KB/thread that was incurred with tracing as we strong-scale the nine Cilk benchmarks. The error bars represent the standard deviation from a sample of 15 runs. For all the runs the standard deviation is low, demonstrating that different random stealing schedules do not significantly impact storage overhead. To make the trends more visible, we graph six of the benchmarks that have less overhead on the left and three on the right that have more overhead with different y-axis. For the first few scaling points all the benchmarks increase in storage per thread, but this increase scales sub-linearly (note that the threads are doubled each time, except for the 96-thread point) with thread count. This graph demonstrates that our storage requirements are small, grow slowly with thread count, and have low variation even with differing schedules. The total storage overhead continues to increase with thread count, reflecting the fact that increasing thread counts increases the number of steals. Despite this increase, we observe that the total trace size, even on 120 threads, is small enough to be analyzed on a commodity desktop system.

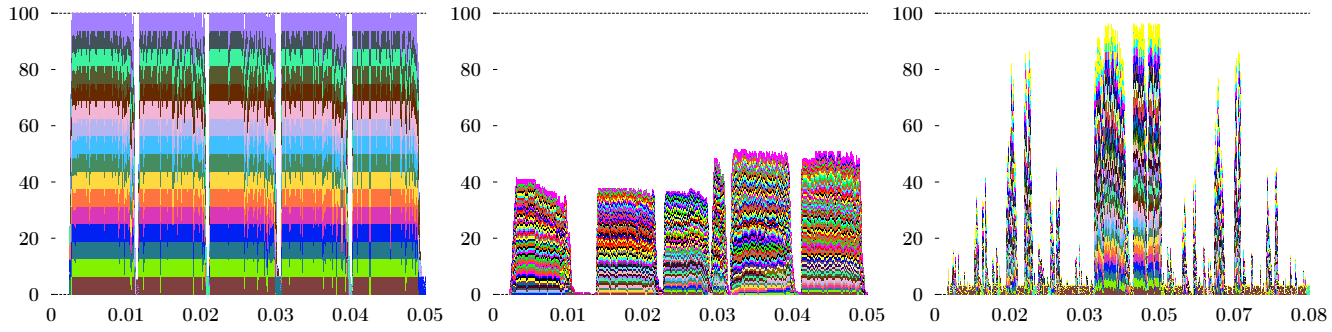
The traces were replayed to determine the utilization across time. Some of the results are shown in Figure 8. These plots, quickly computed from the traces, show the variation in executing identical iterations of Heat on 120 threads and the significant under-utilization when running LU even at moderate thread counts.

9.2 Tracing on Distributed-Memory Systems

For distributed-memory, we evaluate four benchmarks with two different scheduling strategies to measure the execution time and storage overhead that our tracing incurs (the configurations are shown in Table 1). The AllQueens (AQ) benchmark is a distributed-memory variant of the AllQueens benchmark. When the depth of recursion exceeds a threshold the benchmark executes an optimized sequential kernel. SCF and TCE are computational chemistry benchmarks designed by Lifflander et al. [12]. PG is a work stealing implementation of the pair-wise sequence alignment application designed by Wu et al. [17]. We refer the readers to the corresponding papers for details.

We execute the four benchmarks under both work-first and help-first scheduling policies. Figure 6 shows the ratio of execution time with tracing versus the execution time without tracing. For all the configurations, the overhead is low and mostly within the error bars, which represent the standard error in the difference of means at 99% confidence, using a Student's t-test. Some of the variation is due to obtaining a different job partition on the machine between runs (most likely the reason a couple points execute faster with tracing).

Figure 7 shows the storage overhead in KB/core that we incur with our tracing schemes. Note that for all the configurations except for AQ-WF, the overhead is less than 75 KB/core and is constant or decreases with core count. At 32,000 cores the total storage for the traces, assuming 75 KB/core, is 2.3 GB, which would



(a) 16 threads of the Heat benchmark (b) 120 threads of the Heat benchmark (c) 32 threads of the LU benchmark

Figure 8. Percent utilization (y-axis) over time in seconds (x-axis) using the steal tree traces, colored by a random color for each thread.

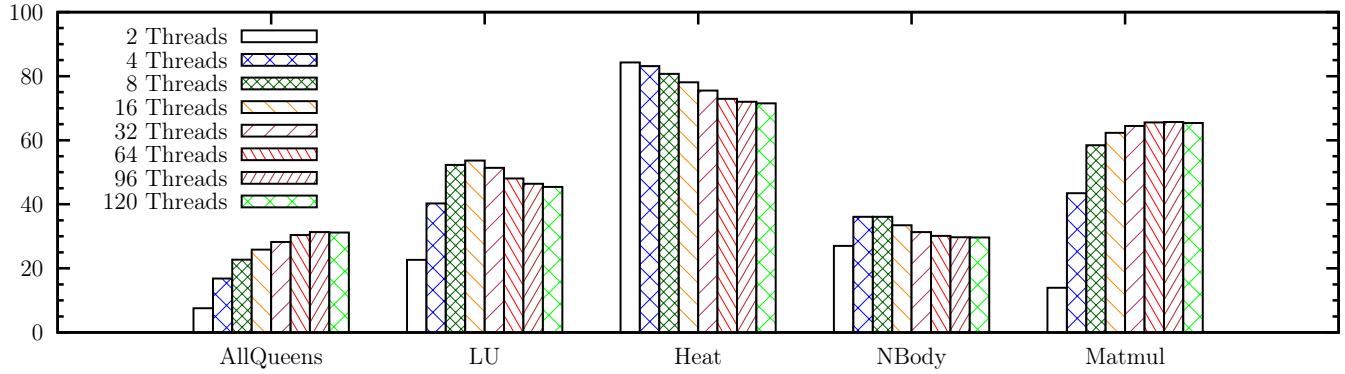
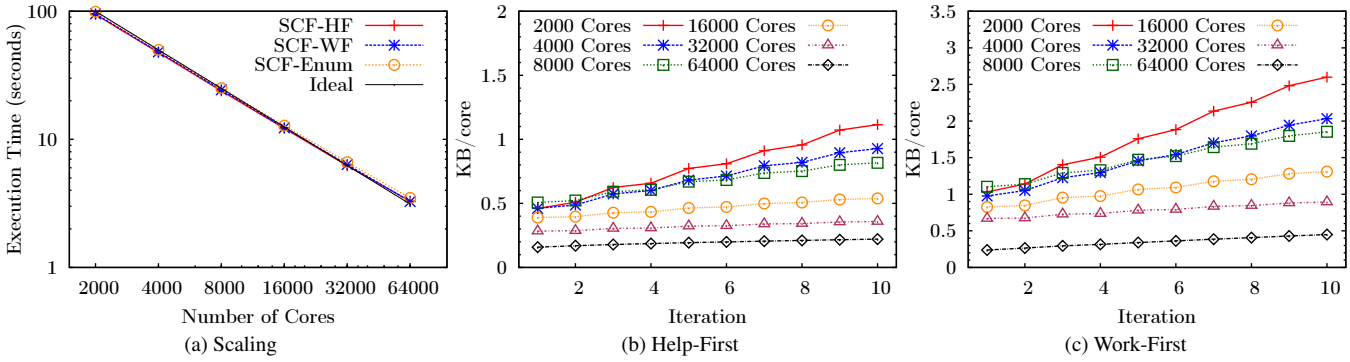
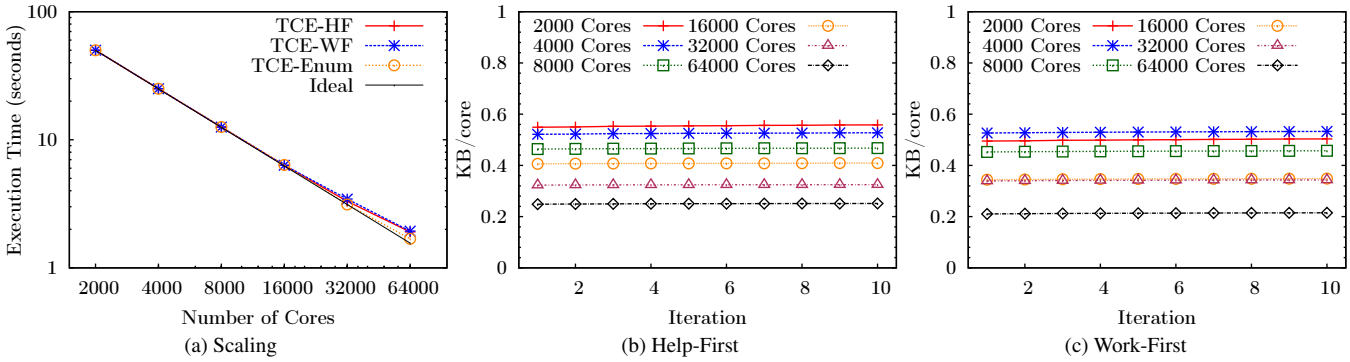


Figure 9. The percent reduction attained by exploiting our traces to reduce the tree traversals of the DPST (dynamic program structure tree) to detect races in a shared-memory application [13].



(a) Scaling (b) Help-First (c) Work-First

Figure 10. Retentive stealing using our tracing algorithms on recursive specification of the SCF benchmark on Cray XK6 Titan.



(a) Scaling (b) Help-First (c) Work-First

Figure 11. Retentive stealing using our tracing algorithms on recursive specification of the TCE benchmark on Cray XK6 Titan.

allow performance analysis to be performed easily on a shared-memory machine or a medium-scale work station. The AQ-WF configuration shows a somewhat different trend because the grain size for AQ is very fine and the WF scheme reduces parallel slack compared to the HF scheme.

9.3 Optimizing Data-Race Detection

For several benchmarks in Cilk, we show in Figure 9 the percent reduction of tree traversals achieved by exploiting the subtree information to bypass traversing regions of the tree. For the Heat benchmark we observe over a 70% reduction in tree traversals; for Matmul around 50–60% reduction; for NBody over 30% reduction; for LU around a 40–50% reduction; and for AllQueens around a 20–30% reduction. The general trend is that increasing the thread count partitions the tree more, causing a further reduction in the number of tree traversals required. Depending on the structure, further segmentation beyond some point may not be beneficial if the LCA is high up in the tree. Heat has this behavior where the best partitions are few and high up in the tree, with further segmentation causing an increase in tree traversals.

9.4 Retentive Stealing

We evaluated retentive stealing for the two iterative distributed-memory benchmarks: Self-Consistent Field (SCF) and Tensor Contraction Engine (TCE) (retentive stealing is not applicable for the others). In Figures 10a and 11a we show the scaling behavior of both benchmarks after several warm-up iterations and then running them 10 iterations to convergence with retentive stealing. The help-first and work-first schemes scale almost perfectly and the scaling results are comparable to the result in [12]. We graph the full task enumeration scheme used in that paper as TCE-Enum and SCF-Enum. We are able to reproduce this result without incurring the overhead of enumerating every task and storing them. For SCF, fully enumerating the tasks requires 20.7 KB/core on 2K cores; for TCE, full enumerating the tasks requires 8.3 KB/core on 2K cores. For both benchmarks, the steal tree uses significantly less storage per core compared to enumerating all the tasks. For instance, on 2K cores of SCF with help-first scheduling we only require 0.34 KB/core to store the traces; this decreases to 0.26 KB/core on 32K cores.

Also, our queue size is bounded by d (where d is the depth of the computational tree) for the work-first scheme or bd (where b is the branching factor) for the help-first scheme. With explicit enumeration the queue size is bounded by the number of tasks, which may require signification memory overhead.

In Figures 10b, 10c, 11b and 11c we graph the amount of storage per core required over time for retentive stealing because steals in subsequent iterations cause more partitioning of the work. We observe that the convergence rate is application- and scale-dependent. For the SCF benchmark, the convergence rate increases with scale under strong scaling as the benchmark approaches the limits of its concurrency. We thus anticipate the storage overhead to remain constant or increase very slowly for such an iterative application at large scales. TCE appears to be very well-behaved, with subsequent iterations causing almost no increase in storage overhead.

10. Conclusions

The widespread use of work stealing necessitates the development of mechanisms to study the behavior of individual executions. The algorithms presented in this paper to efficiently construct steal trees enable low overhead tracing and replay of async-finish programs scheduled using work-first or help-first work stealing schedulers.

We demonstrated the broader applicability of this work, beyond replay-based performance analysis, by demonstrating its usefulness in optimizing data race detection and extending the class of programs that can employ retentive stealing. As future work, we are considering the use of these tracing algorithms to construct skeletons for dynamic task-parallel programs that help study the stealing and execution structure of complex applications without requiring an understanding of the detailed domain-specific operations.

Acknowledgments

This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award number 59193, and by Pacific Northwest National Laboratory under the Extreme Scale Computing Initiative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science under Contract No. DE-AC05-00OR22725. We thank Vivek Sarkar and Raghavan Raman for discussions on their data-race detection algorithm [13].

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002. ISSN 1432-4350.

[2] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 163–174, 2008.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3): 404–418, 2009.

[4] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 133–144, 2004.

[5] R. D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Massachusetts Institute of Technology, 1995.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, 1995.

[7] J. F. Box. Guinness, Gosset, Fisher, and Small Samples. *Statistical Science*, 2(1):45–52, Feb. 1987. ISSN 0883-4237. doi: 10.1214/ss/1177013437.

[8] B. Fulgham. Computer language benchmarks game, August 2012. URL `http://shootout.alioth.debian.org/`.

[9] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–12. IEEE, 2009.

[10] T. Karunaratna. *Nondeterminator-3: a provably good data-race detector that runs in parallel*. PhD thesis, Massachusetts Institute of Technology, 2005.

[11] D. Lea et al. Java specification request 166: Concurrency utilities, 2004.

[12] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 137–148, 2012.

[13] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[14] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[15] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 271–271, 2007.

[16] N. R. Tallent and J. M. Mellor-Crummey. Identifying performance bottlenecks in work-stealing computations. *IEEE Computer*, 42(12): 44–50, 2009.

[17] C. Wu, A. Kalyanaraman, and W. R. Cannon. PGraph: efficient parallel construction of large-scale protein sequence homology graphs. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1923–1933, 2012.

[18] G. Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.